Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Lukas Nicholas Tveiten

Frameworks for Distributed Processing of Temporal Graph Queries

Master's thesis in MSIT
Supervisor: Kjetil Nørvåg
June 2023

NTNU
Norwegian University of
Science and Technology

Lukas Nicholas Tveiten

Frameworks for Distributed Processing of Temporal Graph Queries

Master's thesis in MSIT
Supervisor: Kjetil Nørvåg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

The graph data model is one of the fundamental data structures we have and is playing a crucial role in domains such as social networks, traffic networks and sensor networks. As data is continuously collected and generated, the time of different events in these domains are becoming increasingly more important for analysing and understanding them. In addition, as the data volume only grows, horizontal scaling facilitating distributed processing with multiple machines is a necessity. With this context in mind, there have been an increasing interest in the research literature for frameworks that extend this graph data model. In our literature review, we find that these framework extensions can be organized in temporal graph models, temporal graph queries and distributed programming models. Furthermore, we find that there is a lacking of query functionality that allow for the new type of temporal path queries with the addition of weighted path queries. We therefore made a prototype where we incorporated this functionality as an extension. Our solution shows varying results, but for most queries it is able to scale with larger datasets, more expensive parameters and tendencies of speedups with increasing worker nodes. Thus, this work shows the feasibility of processing temporal graph queries in distributed settings, and opens up for further research directions for these queries, which include database or stream oriented systems.

Sammendrag

Grafdatamodellen er en av de fundamentale datastrukturene vi har og spiller en sentral rolle i domener som sosiale nettverk, trafikknettverk og sensornettverk. Nå som data genereres og samles kontinuerlig, blir tidspunktene av forskjellig hendelser i disse domenene viktigere for å forstå og analysere dem. Siden datavolumet bare vokser er horisontal skalering som tillater distribuert prosessering med flere maskiner en nødvendighet. Med utgangspunkt i dette har det vært en økende interesse i forskningslitteraturen for rammeverk som utvider denne grafdatamodellen. I vår litteraturstudie organiserer vi disse utvidelsene i temporale grafmodeller, temporale grafspørringer, og distribuerte programmeringsmodeller. Dessuten har vi oppdaget at det er en mangel av spørringsfunksjonalitet som tillater de nye temporal grafspørringene med vektede spørringer. Derfor har vi laget en prototype hvor vi har innarbeidet denne funksjonaliteten som en utvidelse. Vår løsning viser varierende resultater, men for de fleste spørringer er den i stand til å skalere med større datasett, dyrere parametere og tendenser til økning i ytelse med flere arbeidsnoder. Dermed viser arbeidet vårt at det er mulig å prosessere temporale grafspørringer distribuert, og åpner for videre forskningsmuligheter for disse spørringene, som inkluderer database- eller strøm orienterte systemer.

Contents

1 Introduction

1.1 Background and Motivation

The graph data model has long been used to model data for networks and domains where relationships between entities are the most essential component, such as for social networks, sensor networks, traffic networks, biological networks etc. As data is continuously collected and generated, the time of different events in these domains are becoming increasingly more important for analysing and understanding them. This leads to the need for an extended graph data model that integrates this temporal aspect into the core of the model. In addition, as this data foundation is collected in large quantities over time, it requires substantial amount of processing and storage to support our analysis needs. This can only be done if we allow for horizontal scaling with multiple machines facilitating distributed processing of the data.

Analytics and processing frameworks such as Hadoop[1], Spark[2], Flink [3] etc. enables programming abstractions for general purpose analytical workloads to be executed in data-parallel and distributed settings. Furthermore, these and similar frameworks have been extended to work for processing of static graphs, and there are several well established systems and libraries for this such as Pregel [22], GraphX[4], Gelly[5], Giraph[6] etc. Recently, there have been an increasing interest surrounding analysis of dynamic and temporal graphs. Thus, there is an increasing amount of work attempting to extend and build new similar frameworks that are especially suited for processing of temporal graphs.

As part of the work towards temporal graph processing frameworks, there have been created new algorithms and programming interfaces specifically for temporal graphs. Particular emphasis has been placed on creating query interfaces that enables temporal graph queries. These queries are extensions of already existing graph query languages that can be found in graph frameworks and database systems. We see a growing body of work in this direction, but limited work in trying to structure and organize all of this in one study.

There are several studies that give an overview of the graph processing landscape as a whole [8] [10] in a structured manner. However, the studies only categorizes the temporal graph processing field and slightly describes some of the existing frameworks. The mentioned papers do not go in depth on comparing and evaluating the models, methods and techniques used for distributed temporal graph processing so far. There have especially been limited work on comparing temporal graph queries and languages that enable them.

In this work, we will compare and evaluate both the qualitative and quantitative aspects of the existing distributed temporal graph frameworks, especially with relation to temporal graph queries, which there have been done limited work on previously. In addition, we will do this by going further and experiment with novel temporal graph algorithms and applications that have not been extensively experimented with in previous studies.

[1] https://hadoop.apache.org/
[2] https://spark.apache.org/
[3] https://flink.apache.org/
[4] https://spark.apache.org/graphx/
[5] https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/
[6] https://giraph.apache.org/

1.2 Research Questions

In this work we will attempt to answer the following research questions:

- What are the existing distributed temporal graph processing frameworks/systems and how they compare?
- How can novel temporal graph algorithms and operators be implemented to extend these systems?

For both of these questions we will attempt to compare and evaluate based on qualitative and quantitative aspects of such systems.

- Qualitative aspects such as temporal graph models, programming models, query languages and operations allowed by the systems.
- Quantitative aspects such as runtime, memory and scalability performance specifically with relation to the distributed part (parallelism, partitioning etc.).

1.3 Research Method

Our method of answering these research questions will be divided into two main bodies of work, which will be done in order. We will answer the first research question "What are the existing distributed temporal graph processing frameworks/systems and how do they compare?" as a review of existing literature in the field. Here we will study both the qualitative and quantitative aspects as initially described, but our focus will particularly be on the qualitative comparisons. After having done a literature review and getting an understanding of limitations and possible further extensions of these frameworks, we will answer our second research question "How can novel temporal graph algorithms and operators be implemented to extend these systems?". This work will be focusing on extending the qualitative aspects while also considering the quantitative properties as previously described. This will be done by creating a technical prototype with said extensions and running experiments where we make the necessary measurements to answer these questions.

1.4 Contributions

Our specific contributions will be in the form of the two bodies of work we have outlined. We will contribute with a literature review of temporal graph frameworks, which have not been done in a comprehensive manner already. In addition, we contribute with the experimentation and exploration of new extensions that have not been tried out before.

Our general contribution to the field is further understanding of graphs and processing of graph data as one of the most applicable and fundamental data structures to represent real world data. Furthermore, we incorporate arguably the most fundamental physical quantity we can represent in a computer system, namely the temporal dimension. We generally explore this overlap and contribute to the understanding of it, both in terms of methods of analysing this data and of efficient processing of it.

1.5 Thesis Structure

The thesis is divided into two main parts in accordance with our research method: Literature Review and Extension. The Literature Review part is dedicated to answering the first research question, and the Extension part is dedicated to answering the second research question. The introduction part of the thesis includes this introduction section and background theory section which explains some preliminary background knowledge. The overall structure of the thesis is outlined in the following list containing the main subsections of the thesis:

- Introduction
- Background Theory
- Literature Review
  - Temporal Graph Models
  - Temporal Graph Queries
  - Distributed Programming Models
  - Temporal Graph Frameworks
- Framework Extensions
  - Temporal Weighted Path Queries
  - Algorithms and Data Structures
  - Experimental Results
- Conclusion and Future Work

2 Background Theory

2.1 Distributed Data Processing

In this section we will introduce the characteristics of distributed data processing and the field of big data processing in general. We will explain what type of processing workloads this entails and specify the intended focus in this study. Furthermore, we will introduce some influential and popular open-source frameworks. These are often the basis for the special purpose temporal graph frameworks that we will evaluate closer in the following chapters.

Following the widely known Flynns taxnomoy [14] of parallell processing architectures, where we have the two dimensions of instruction streams and data streams, distributed computer architectures would be classified as Multiple Instruction, Multiple Data (MIMD). MIMD would also include single computer multi-core, multi-processor and NUMA architectures, but with the major difference being that memory is shared among the processors here. Distributed computing architectures on the other hand, especially apparent from the programmers perspective, is that memory is *not* shared among the processes/nodes. This would be called a *distributed memory* or sometimes *shared-nothing architecture*, and is the type of systems and programming we will be focusing on. Figure 2.1 shows how a distributed shared-nothing architecture is structured.

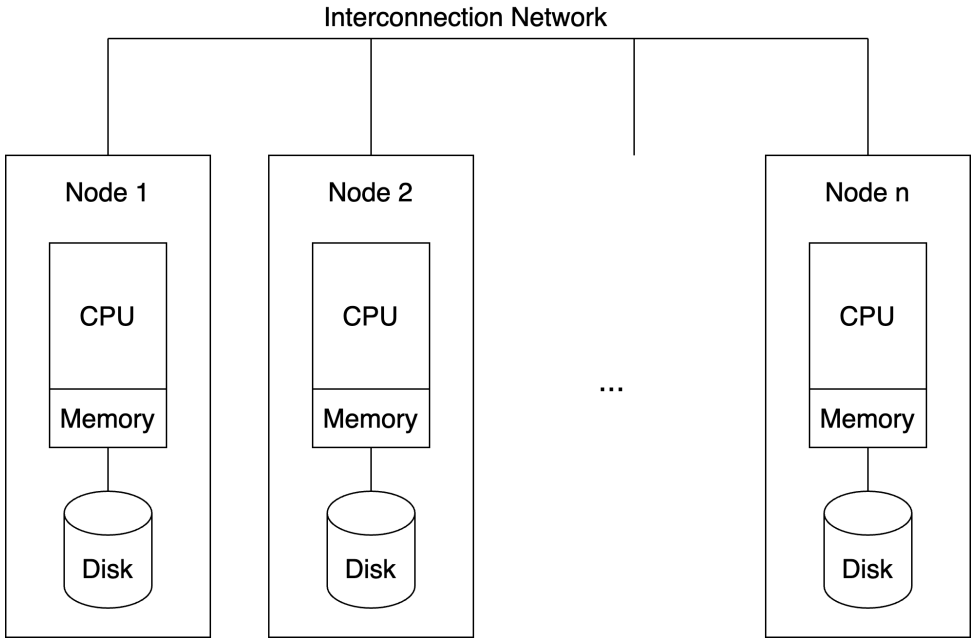


Figure 2.1: Shows how a distributed shared-nothing architecture is structured, which is the type of underlying architecture we will be assuming for the frameworks we will study.

In the era of big data where we want to process and analyze data at arbitrary large scales, it requires us to scale the amount of memory, parallelism and storage accordingly. A distributed architecture as shown in Figure 2.1 allows us to scale dynamically and horizontally as we desire, but as a result we need ways to create programs to process data distributed on multiple machines. The first major breakthrough in programming frameworks for big data processing was Googles introduction of the MapReduce framework [12]. MapReduce set the characteristic principles and golden standards of what a general-purpose distrib-

uted data processing frameworks should at least support. The following 5 principles and characteristics summarize this [31]:

- **Expressive Programming Model** that abstracts away the messy details of the underlying parallelization, fault-tolerance, data partitioning and load balancing. This enables the programmer to focus on creating data processing- and analytics applications without having a deep understanding of parallel and distributed programming.
- **Scalability** with large clusters of commodity PCs where you can dynamically add more nodes to the cluster, instead of having special made expensive hardware systems.
- **Fault-Tolerance** as the hardware is unreliable the framework will handle failure of a node without stopping the ongoing processing.
- **Parallelism** automatically support the parallelization of tasks by distributing the work as evenly as possible to each node on the cluster.
- **Data locality** Take advantage of data locality by partitioning the data to the machines where the executions resides thereby reducing the data transfer between the nodes and increasing performance.

Data Processing vs. Data Storage

Managing and processing big data requires both distributed storage and processing of different types of data and workloads. [6], [31] and [36] makes a clear distinction between components that are tailored towards storage and others for processing. Storage systems are particularly NoSQL databases or other distributed file-systems such a HDFS that have great ability to scale horizontally. Distributed data processing frameworks on the other hand, primarily give programming interfaces to process and analyze data satisfying the general principles listed above. Some are general-purpose others are specially made for some operations or type of data, which we will explain further in this section. In this work, we will focus on distributed data processing exclusively, mostly tailored towards temporal graph data, and not give any particular detail to the underlying distributed storage that is used.

Frameworks

[6], [31] and [36] gives further distinction between the type of processing categories, with the following categories:

- **General-purpose** processing gives the capability to create user-defined methods of processing the data given an overarching programming paradigm. Examples of these are the traditional MapReduce framework with its paradigm, Spark, Flink with functional and dataflow programming paradigms etc. We will give an example of Spark as such a framework later in this Section.
- **Graph** processing frameworks are especially suited for graph data with for instance the vertex-centric Pregel [22] style programming, which we will explain further in Section 2.4.

- **Structured** processing enables the programmer to process and analyze the data using SQL like languages. There also exists overlaps with graph processing, enabling processing and analytics with Graph Query Languages (GQL), which we will explore further in Section 2.3
- **Stream** processing allows processing on a continuous stream of data focusing on processing with lower latency as opposed to batch oriented workloads that focus on high throughput on longer running operations. In our work we have not focused on stream processing in particular, but most of the principles can be transferred and applied to stream processing as well.
- **Machine learning** processing frameworks focus on integrating the data so it is ready for training for a certain machine learning model. We have not focused on machine learning frameworks in our work.

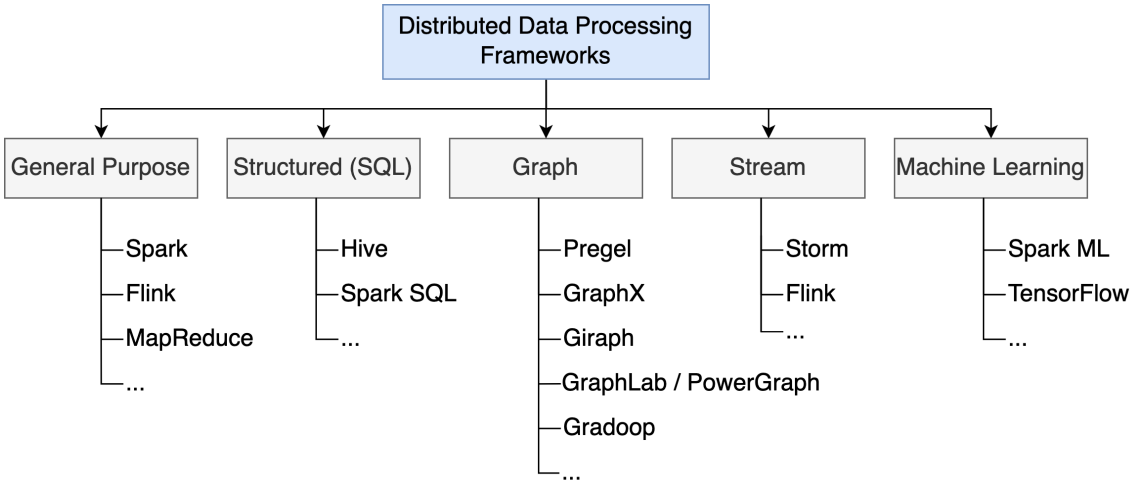


Figure 2.2: Shows a diagram of some common open-source projects in each category, inspired from similar diagrams from [6], [31] and [36].

Distrbuted temporal graph frameworks that we are going to evaluate in this work are often implemented on top of any one of these categories of processing frameworks. They usually focus on extending the programming model and capabilities of the framework to give support for temporal graph models, operators, querying and algorithms while optimizing the scalability, parallelism and data locality. In the following chapters of this thesis we will generally consider all of the characteristics of distributed data processing listed, but in particular focus expressive programming models where we specifically are interested in temporal graph queries.

Functional and Dataflow

Functional and dataflow programming models common to see in more general-purpose processing frameworks. The essence of such programming model is that it should be flexible to create flow of transformations and operations on the data without having to worry about the underlying execution, whether it is executed in parallel distributed to multiple machines or simply in sequence on a single machine. Figure 2.3 shows an example of a common data processing scenario using a functional programming model. In addition, as the principle behind functional programming is to have immutable data objects, this

also makes it easier to run in parallel, as mutability is often a source of pain when dealing with concurrency.

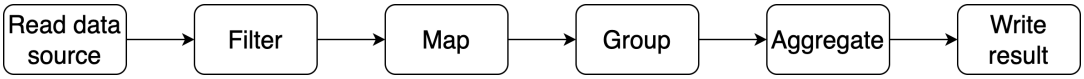


Figure 2.3: Shows the dataflow of a common data processing and analysis scenario using a functional programming model.

Example: Apache Spark

The most notable open-source project that are based on the functional and dataflow paradigm is the Apache Spark[7] framework. As we can see in Figure 2.4 a data processing application written for Spark goes through various layers in the execution process. This process ends with the application being processed on multiple machines, called workers, in parallel. The first layer of this process is the user program compilation or interpretation. Spark is written in and interfaced by the programming language Scala, among other languages. The main layer of abstraction in this programming interface is the RDD (Resilient Distributed Dataset) and a set of general purpose operators such as the one we saw in Figure 2.3. An RDD is simply an immutable distributed collection of data objects, which is split into multiple partitions and can be computed on different nodes of the cluster. When the programmer applies operators to an RDD, the framework will create an operator graph, which is further submitted to the DAG (Directed Asyclic Graph) Scheduler. This scheduler will split the operator into map and reduce stages with directed edges between the stages indicating the order of the execution. When a stage is ready, a set of tasks will be created and sent to the Task Scheduler, which will launch tasks via a cluster manager. The cluster manager distributes and assigns the tasks to the individual worker nodes, which then executes the tasks. The general structure of this process can be seen in Figure 2.4.

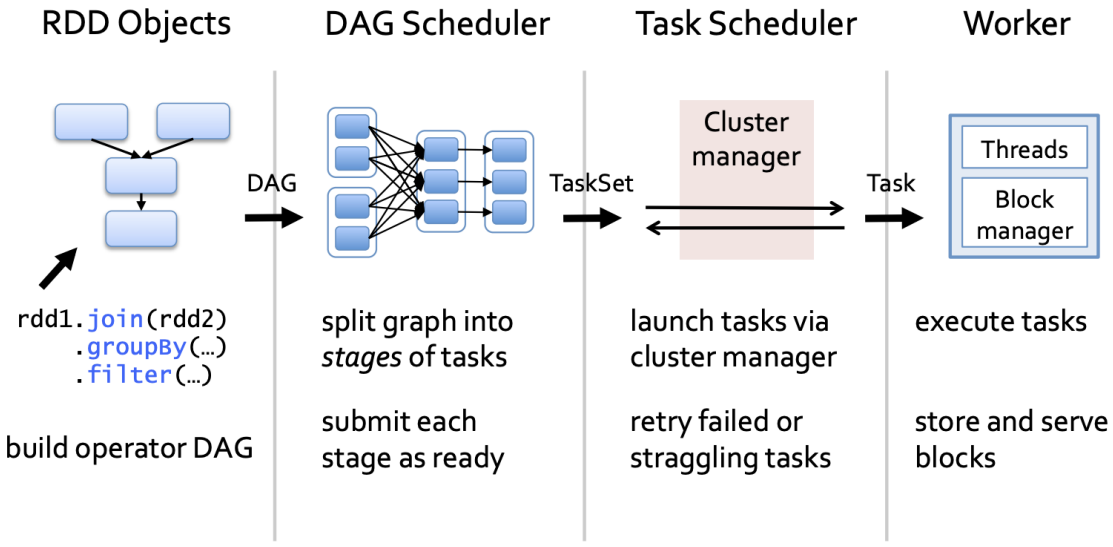


Figure 2.4: Shows the general structure of the process of creating an application program and ending up with tasks distributed and executed on worker nodes. Figure taken from [35]

[7] https://spark.apache.org/

2.2 Graph Definitions

Before going further into the different definition for temporal graph models in Section 3, this section will introduce some definitions for static graphs and time domains that are useful for further defining temporal graphs.

Property Graph Model

Here we are going to define the graph in a slightly modified way compared to the standard definition of $G = (V, E)$. Here, the set of edges is going to be its own set where we have an incidence function that associates an edge with two vertices. This is because it is more in line with the Property Graph Model (Definition 5)

Definition 1 (Graph)**.** A Graph is a 3-tuple $G = (V, E, \rho)$ where V is set the set of nodes, E is the set of edges and $\rho : E \to (V \times V)$ is a total function called the incidence function.

Definition 2 (Multigraph)**.** Is a Graph where you can have multiple distinct edges between any two pair of nodes. This allows ρ to have multiples edges $eid \in E$ that map to the same (u, v) pair.

Definition 3 (Labelled Graph)**.** Is a Graph where each node or edge can have one or multiple labels mapped to it. It is further extended with having a label set L and a label function $\lambda : (V \cup E) \to \mathcal{P}(L)$.

Definition 4 (Directed Graph)**.** Sometimes called Digraph, is a Graph where each edge have a direction, meaning that it has a distinct from-vertex and an end-vertex. This means that an edge $(u, v) \neq (v, u) \mid v \neq u$ where $u, v \in V$.

The Property Graph Data Model [2] is the graph model that is widely used among existing graph systems such as JanusGraph[8], Neo4j[9] and Oracle Database[10]. In the following definition we define this model by extending the previous definitions.

Definition 5 (The Property Graph Model)**.** Is a directed, labelled multigraph $G = (V, E, L, P, \rho, \lambda, \sigma)$ where we have the following:

- V is the set of nodes, E is the set of edges, L is the set of labels and P is the set of property names/keys.
- $\rho : E \to (V \times V)$ is a total function that associates an edge with a pair of nodes.
- $\lambda : (V \cup E) \to \mathcal{P}(L)$ is a partial function that associates a node or an edge to any combination of labels.
- $\sigma : (V \cup E) \times P \to val$ is a partial function that maps the property name/key of a particular node or edge to a value.

Some papers only use a property set without associating the nodes or edges with labels, for instance [26], but the complete definition of a property graph data model is to both have labels and properties, how it is used in practice and following [2]. Some [30], also use

[8] https://janusgraph.org/
[9] https://neo4j.com/
[10] https://docs.oracle.com/en/database/oracle/property-graph/

a separate source and target function instead of the single incidence function ρ, however these two are equivalent and the former therefore redundant.

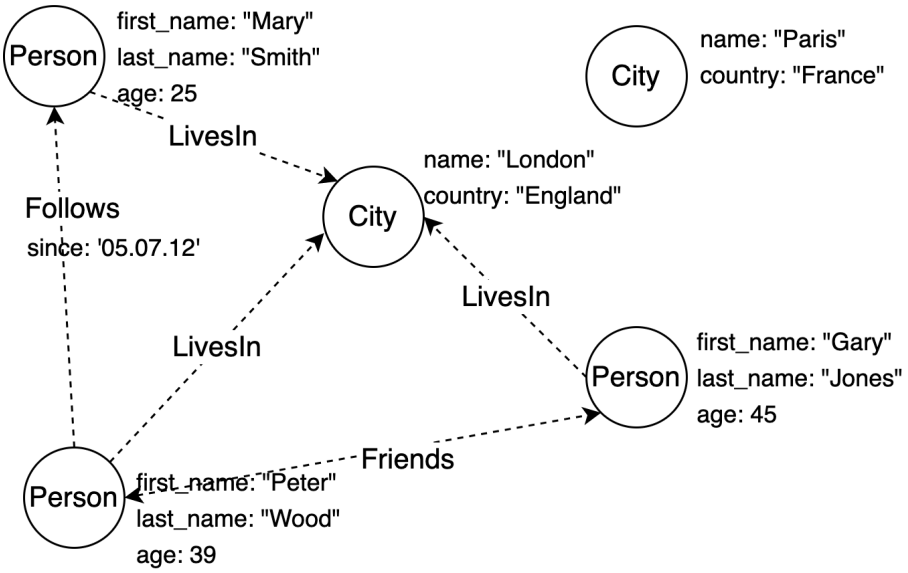


Figure 2.5: An example of the property graph model representing a simple social network graph

An example of a graph using the property graph model could be a simple social network graph where we have the following (visualized in Figure 2.5):

- $V = \{v1, v2, v3, v4, v5\}$
- $E = \{e1, e2, e3, e4, e5\}$
- $L = \{Person, City, Follows, Friends, LivesIn\}$
- $P = \{first\_name, last\_name, age, name, country, since\}$
- $\rho = \{(e1, v2, v1), (e2, v1, v3), ...\}$
- $\lambda = \{(v1, \{Person\}), (e1, \{Follows\}), ...\}$
- $\sigma = \{(v1, first\_name, "Mary"), (e1, since, '05.07.12'), ...\}$

For brevity, the function representations are not fully written out.

Path Definition

Here we are going to give a definition of a path, which is a term that we are going to utilize without any additional explanation in several of the future sections. In the graph theory and algorithms literature there exists some different definitions of a path, however we will stick to a similar definitions as could be found in [11].

Definition 6 (Path)**.** A *path* of length k from source vertex v to a destination vertex u in a graph $G = (V, E)$ is a sequence $< v_0, v_1, v_2, ..., v_k >$ of vertices such that $v = v_0$, $u = v_k$,

and the edges $(v_{i-1}, v_i) \in E$. The length of the path is the number of edges in the path, which is 1 less than the number of vertices in the path.

An example of a path given the graph in Figure 2.5 could be:

- $<$"$Peter$","$Mary$","$London$"$>$ as the vertex sequence mapped to the $first\_name$ or $name$ properties.
- The length of the path is 2 as there are 2 edges in the path.

Time and Interval Definitions

In this section we are going define some preliminaries for the time domain so that we have a clear understanding of what it entails and ensure the soundness of the temporal graph definitions later.

Definition 7 (Discrete Time Domain)**.** A discrete time domain Ω encompasses the set of all non-negative whole numbers $\mathbb{N}$, where an instant in time $\omega_i \in \Omega$ is called a *timepoint*. A time *interval* is defined as a closed-open interval $\tau = [\omega_{start}, \omega_{end})$, which represents a discrete contiguous set of timepoints: $\{\omega \mid \omega \in \Omega \wedge \omega_{start} \leq \omega < \omega_{end}\}$.

Definition 8 (Linear Ordering)**.** Timepoints are linearly ordered, meaning that if $\omega_i < \omega_{i+1}$ it implies that the timepoint ω_i happened before the timepoint ω_{i+1}.

We can consequently have boolean interval relations, which are further defined in [1], such as an interval happened $fully\_before$, $started\_before$, $during$, $equals$, $overlaps$ etc. another interval.

The following shows an example of timepoints and time intervals in a linearly ordered discrete time domain:

- Given the two timepoints $\omega_1 = 121$, $\omega_2 = 126$, timepoint ω_1 happened before timepoint ω_2
- the time interval $\tau = [\omega_1, \omega_2)$ consists of the discrete contigious set of timepoints: $\tau = \{121, 122, 123, 124, 125\}$

2.3 Graph Query Language

Graph query languages have been extensively used for graph databases for quite some time already. They model the graph according to the Property Graph Model (See Section 2.2) and give expressive query capabilities with this data model, similar to what is possible with SQL for relational-table data. The well known International Standards Organization (ISO), which among other things have standards for SQL, have now recently voted on establishing a standard specification for a set of general purpose Graph Query Langugages (GQL). The most notable implementation of this standard is the Cypher [16] language which was created for Neo4j[11], and have multiple implementations outside of the initial creation which go under the openCypher[12] project. Other notable languages are Oracle's PGQL [28], G-Core [3] specified by the LDBC and Gremlin [29] in the Apache Tinkerpop framework.

The most notable feature of GQL compared to the more standard relational SQL-like languages is that you can easily express graph pattern and path queries. Paths and edge relationships are treated as first-class citizens and can further be handled with the familiar predicates and selection features as we can with table-style relations in SQL. For instance you could use it to get all the common friends among two people in a social network graph. Or to get friends and friend-of-friends of a person. We could also find the 3 friends in the social network that have communicated the most with each other. These three queries expressed in Cypher are shown in Figure 2.9, 2.10, and 2.13 respectively with the given graph in Figure 2.8.

We will summarize the types of edge relationships and path queries that are possible with GQLs in the three following categories: constant length, variable length and weighted paths.

- Constant length paths
- Variable length paths
- Weighted paths

[11] https://neo4j.com/
[12] https://opencypher.org/

Constant Length Paths

Constant length paths, often just called relationship patterns, is the most basic GQL-specific query where we can express individual node-edge-relationships. The first row in Figure 2.6 shows the general syntax of relationship patterns in Cypher. The fundamental syntax of a query construct begins with the *MATCH*-symbol. Further, a and b indicate node variables, and r indicate edge-relationship-variable, where REL_TYPE is a placeholder indicating a user-defined relationship-type in accordance with the property graph model. We can further apply filter-predicates and attribute-projections with the *WHERE* and *RETURN* syntax respectively, shown in Figure 2.7. When creating these user-defined functions we can make use of a, b, r, and potentailly more user-defined variables, in the function procedures. The result of these queries are that we get all the nodes and edges that are connected with the given user-defined pattern. Figure 2.9 shows an constant-length path query that finds all common friends of "Becky" and "Sara".

GQLs have additional capabilities and syntax, which could also vary from the specific implementation language. For instance, as property graphs are directed graphs we can specify the required direction of the path by direction symbols. In Cypher you can use the arrow symbols <- and ->, and omitting them means that direction is irrelevant. Notice how in Figure 2.9 arrows in both direction in the *KNOWS*-relation means a mutual knowledge bond, which we interpret as a friendship. An additional important property for path queries, is that you can chain multiple paths together to from more complex path navigations. Figure 2.9 shows an example of this for a constant-length path query, however it can be applied to any of the length paths. You can further explore the Cypher capabilities and syntax on [16], but note that other GQLs have naturally some variations in syntax.

Path Query Type	Query Syntax
Constant-length	`MATCH (a)-[r:REL_TYPE]-(b)` `...`
Variable-length	`MATCH (a)-[r:REL_TYPE*n..m]-(b)` `...`

Weighted Path	`MATCH shortestPath(<path_expression>)`
	`...`
	`COST <cost_expression>`
	`LIMIT <top_k>`

Figure 2.6: Shows the general syntax for constant length, variable length and weigthed paths query.

```
...
WHERE <filter_predicate>
RETURN <attribute_projection>
```

Figure 2.7: Shows the general form of the *WHERE* and *RETURN* syntax in Cypher

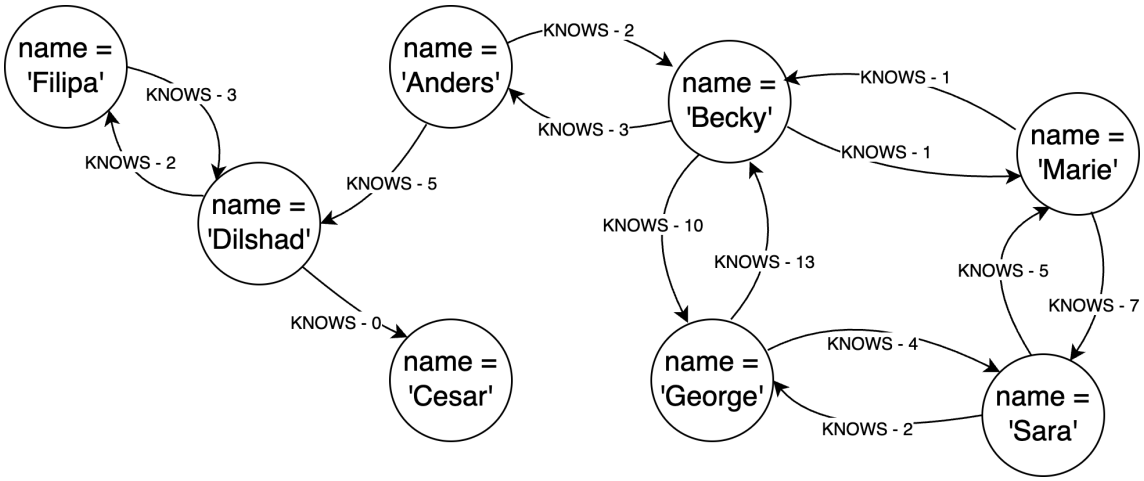


Figure 2.8: Shows a static property graph of a social network, where nodes have a Person-label with name-property and the edges are labeled with KNOWS-relationship. The numbers on the edges represent the edge-property *nr_messages*.

Query Expression	Query Results

`MATCH (b)<-[:KNOWS]->(common)`
`(common)<-[:KNOWS]->(s)`
`WHERE b.name = "Becky"`
`AND s.name = "Sara"`
`RETURN common.name`
	`+-------------+`
	`| common.name |`
	`+-------------+`
	`| "Marie" |`
	`| "George" |`
	`+-------------+`

Figure 2.9: Shows a chained Constant-length path query to find all common friends of "Becky" and "Sara".

Variable Length Paths

Variable-length paths essentially works in the same way as constant length paths, but with the addition that you can specify the minimum and maximum length of the paths. This means that you can specify how many intermediate nodes that are going to be apart of the returned path. The second row in Figure 2.6 shows the general syntax of a variable-length path pattern, where n is the minimum length and m is the maximum length of the path. The *-symbol here means that we want to return all the possible paths in the length span and not only the ones with n and m lengths. Figure 2.10 shows an example of a 1 to 2 length query to get all the friends and friends-of-friends for "Anders". However, you could possibly have 5-length or 100-length paths, which significantly distinguishes their capabilities from a constant-length path query. As we also can see, variable-length path queries can express all constant-length queries if we specify length of 1.

Query Expression	Query Results

`MATCH (me)<-[:KNOWS*1..2]->(remote_friend)`
`WHERE me.name = "Anders"`
`RETURN remote_friend.name`
	`+--------------------+`
	`| remote_friend.name |`
	`+--------------------+`
	`| "Becky" |`
	`| "Geroge" |`
	`| "Marie" |`
	`+--------------------+`

Figure 2.10: Shows a variable length path query in Cypher to retrieve all names of the friends and friends-of-friends of "Anders". Notice how edge directions matter for the result set, as we are just interested in friends-relationships.

However, with variable length paths there are also some additional isomorphism properties that needs to be considered, which are of particular importance for longer length paths. Path queries can create large size result-sets with the possibility of some repeating patterns, or potentially infinite traversals as well. There are three main classes of isomorphism-types that we are going consider, from the least constraining to the most constraining, which are the following:

- **Homomorphism**: No constrains for path matching.
- **Relationship isomorphism**: The same relationship cannot be returned more than once for each path matching record.
- **Node isomorphism**: The same node cannot be returned more than once for each path record.

In most cases, queries give most meaningful results with either node isomorphism or relationship isomorphism. For instance Neo4j uses node isomorphism as the default isomorphism property in their Cypher implementation.

Weighted Paths

With path queries of any length we can in addition have properties that applies for the path as a whole. In Cypher for instance, there is a way to treat the path as a variable and give it further properties or apply functions to it. The most notable use-case of this would be the find the shortest path, where we have some property of what the *shortest* would mean. Many GQLs give the capabilities of expressing user-defined cost-functions, such that the resulting paths give minimum cost-score among all the possible paths satisfying the query. Figure 2.11 shows an example in the G-CORE language of defining a cost-function based on the edge property *nr_messages*, where the higher number gives lower cost-score, which results in paths representing the people that have communicated the most with each other.

```
PATH wKnows = (x)-[e:knows]->(y)
COST 1 / (1 + e.nr_messages)
```

Figure 2.11: Shows an example of defining a path-cost-function based on the edge-property *nr_messages* in the G-CORE language.

In addition, we can specify how many results we want in the result set, where the results are also sorted in the order of the cost-score. This number is usually referred to as *limit-k* or *top-k* where k represents the number of results we are interested . Figure 2.12 shows an example of how this can be achieved in the G-CORE language. The complete query can be seen in Table 2.13 accompanied with the query results. Be aware that the syntax between different GQLs such as Cypher or G-CORE can vary slightly, however the semantics of the query still remains the same across the languages.

```
MATCH shortestPath((n:Person)-/p<wKnows*>/->(m:Person))
...
LIMIT 3
```

Figure 2.12: Given the path cost definition in Figure 2.11 we can retrieve the three shortest paths where the defined cost-property is minimized. Here shown in the G-CORE language.

Query Expression	Query Results

`PATH wknows = (x)-[e:KNOWS]-(y)`
`COST 1 / (1 + e.nr_messages)`
 	
`MATCH shortestPath((a)-[wKnows2..2]-(b))`
`RETURN path`
`LIMIT 3`
	`+---------------------------+`
	`| path |`
	`+---------------------------+`
	`| "Becky"-"George"-"Sarah" |`
	`| "Anders"-"Becky"-"George" |`
	`| "George"-"Becky"-"Marie" |`
	`+---------------------------+`

Figure 2.13: Shows a weighted path query in Cypher to retrieve the top three results of the chain of three people that have communicated with each other the most in the entire social network graph. Here we also follow node-isomorphism.

2.4 Vertex-Centric Programming

The Vertex-Centric programming model was first introduced by Google in their Pregel paper [22]. Pregel computations consist of a sequence of iterations, called *supersteps*. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The user-defined function can define the behaviour at single vertex V for a certain superstep S. It can further get information from the previous superstep $S-1$ from all neighboring vertices. Consequently, it means that vertex V needs to propagate information through messages to the proceeding superstep $S+1$ to a user-defined selected set of neighbouring vertices. This user-defined function is often called the *Compute* function in Vertex-Centric programming frameworks.

Given this high-level organization, we can define a set of functions that make up the needed primitives to construct graph algorithms that can implicitly run in parallel and in a distributed fashion. As part of these primitives, there would need to be a way to get and set the value of the vertex V: for instance called *GetVertexValue* and *SetVertexValue*. A way to iterate through neighbouring vertices: for instance called *GetEdgeIterator*. And a way to send messages to vertices for the proceeding superstep: for instance called *SendMessageTo*. There is an additional detail that also needs to be pointed out, that is when the computation should complete, so-called *halt*. This function is often called *VoteToHalt* and is part of this set of primitives. The computation stops when all vertices have made call to *VoteToHalt* or there are no more messages being sent and received.

Given the following Vertex class written in pseudocode in Figure 2.14, we can implement the Compute-function with the help of the given variables and functions to produce graph algorithms and computations that implicitly can run in parallel. Intuitively, we could make every vertex have its own machine and let the message passing function be the communication gateway. However, there are of course more vertices than machines and thus [22] explores ways partitioning and messaging passing can be optimized, but as we can see this model is agnostic to the layout and partitioning.

```
class Vertex {
    String vertexId;

    int getSuperstep();
    VertexValue getVertexValue();
    EdgeIterator getEdgeIterator();

    void setVertexValue(VertexValue value);
    void sendMessageTo(String destVertex, MessageValue message);
    void voteToHalt();

    abstract void compute(MessageIterator messages);
}
```

Figure 2.14: Shows the set of function primitives and the unimplemented *Compute* function that is part of the Vertex-Centric programming interface.

Classical Graph Algorithms

The Vertex-Centric Programming model can be used to implement iterative graph algorithms. Some classical examples of this could for instance be PageRank, Shortest Paths, Connected Components, or Semi-Clustering etc. An example implementation of the Single-Source Shortes Path (SSSP) can be seen in Figure 2.15

```
void compute(MessageIterator messages) {
    // Init the minimum length to 0 if its the source vertex
    if (getSuperstep() == 1 && isSource(vertexId)) {
        setVertexValue(0);
        return;
    }

    // Update the minimum length from source vertex to this vertex
    minLength = MAX_INT
    for (MessageValue m : messages) {
        minLength = min(m.value, minLength);
    }
    if (minLength < getVertexValue())
        setVertexValue(minLength);

    // Propagate minimum length to neighbouring vertices
    for (Edge edge : getEdgeIterator()) {
        message = new MessageValue(minLength + edge.value);
        sendMessageTo(edge.dstVertex, message);
    }
}
```

Figure 2.15: Shows an implementation of the *Compute* function to create the Single-Source Shortest Path (SSSP) algorithm using the Vertex-Centric programming interface.

Part I

Literature Review: Temporal Graph Frameworks

In this part we will review the current literature that exist on temporal graphs and frameworks for processing of temporal graphs. In Section 3 we will look at the concept of a temporal graph and the different representations that exist. Section 4 will go further into temporal graph queries that extends graph queries found in GQLs. Section 5 will go into distributed programming models that are especially tailored towards temporal graphs. Section 6 will give an overview of the temporal graph frameworks we have studied and discuss limitations and further extensions.

3 Temporal Graph Models

Most the systems that we have reviewed give some sort of definition of a temporal graph model. These definitions are slightly different and are extensions of existing models from different starting places. However, as we will see these models are conceptually and logically equivalent to each other, but can physically and visually be represented in different ways. 3.1 will first clarify some distinction between terms and concepts found in the graph literature, then the rest will present how the models can be defined and derived, and the different representations that exists for temporal graphs.

3.1 Terms and Concepts

In the literature surrounding graphs and networks that have some sort of change related to them, there have been used several terms to confusedly refer to both the same and different concepts. We will follow the general outline given by [8] [10] surrounding the terms static, dynamic, temporal, [time]-evolving and streaming graphs, and specify what sort of graphs we will study further.

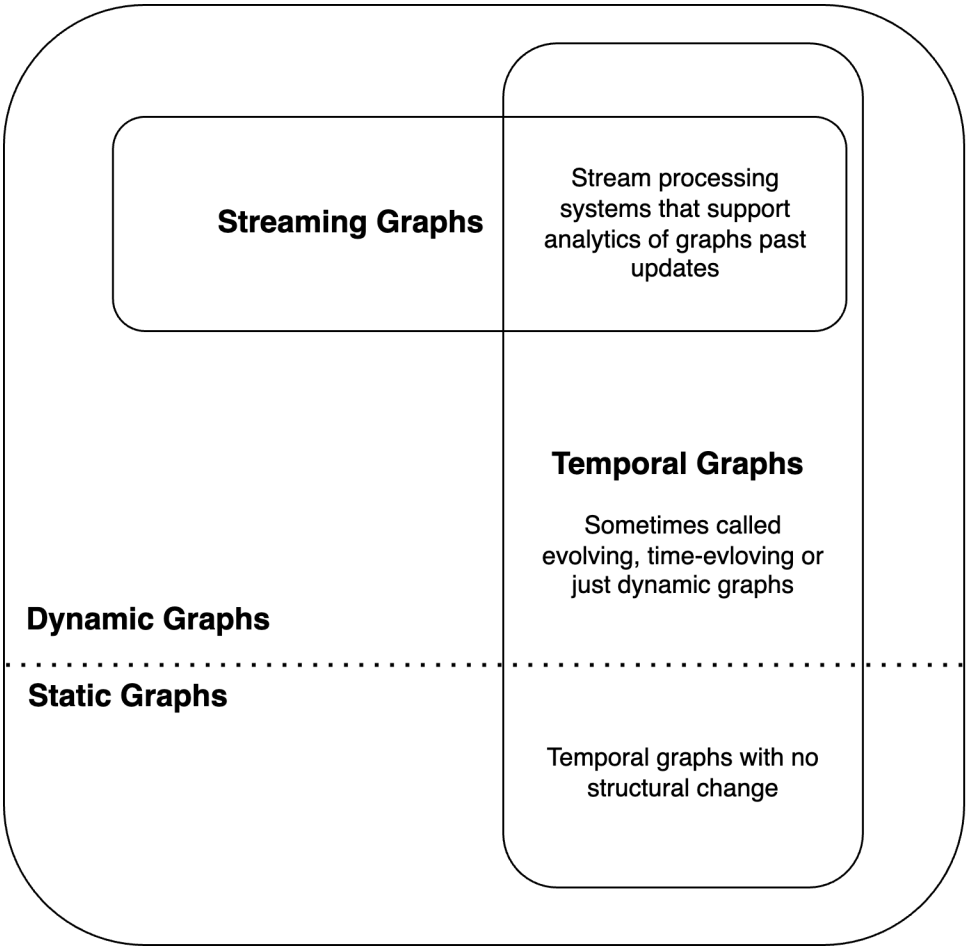


Figure 3.1: Shows a Venn diagram illustrating relations between the concepts, inspired simplified version of the taxonomy introduced in [8]

The first distinction is between *static* and *dynamic* graphs. *Static* graphs is the most simple form of graph model, where we only have a set of vertices and a set of edges. This can be defined as $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. In the domain of static graphs we also have further definitions such as directed, acyclic, property, attributed etc. which could also apply to the dynamic graph or all the other terms. *Dynamic graph* is then the general term for graphs that first and foremost allows *structural changes* of the graph over time, such as vertex and edge insertions and deletions.

Temporal graphs is the general term for graphs where vertices, edges or subgraphs have attached some sort of *temporal information* such as a timestamp, thus having several versions of the same graph element over time. The term evolving, time-evolving or somewhat unspecified just dynamic has also been applied interchangeably with temporal graphs [25] [18], but has also been applied to the different concept of streaming graphs [5]. In addition, Temporal graphs does not necessarily have to be dynamic as the structure of the graph itself could be unchanged in the given time interval. However, most processing systems and frameworks that focuses on temporal graphs naturally allows for dynamic graphs, as structural change is most of the time part of the "temporality". These frameworks focuses on allowing processing on the *entire history* or an *interval* of the temporal graph and not necessarily just the *current state*. In this work, we will focus on systems and frameworks that allows distributed processing of temporal graphs, with the underlying assumption that the graphs are also dynamic.

Then, we also have the concept of *streaming graphs*, which is referred to systems and algorithms that supports processing of the graph in its *current state* where we have a continuous *stream of graph updates*. In most cases the streaming graph model assumes the underlying graph to be dynamic. Some systems allow for temporal graph analysis on streaming graphs by storing all the changes from the stream, [32] is an example of such an system. Therefore, the two main concepts in the area of dynamic graphs are *temporal* and *streaming* graphs, and the main difference is that the first looks at the *history* of the changes over time and the latter keeps the graph in the *current state*. And as pointed out, there is used different terms interchangeably and overlapping between these two concepts, but we will stick to terms described here. In this work, we will not focus on systems or frameworks that exclusively allows processing on the streaming graph model.

3.2 Property Graph Extension

The first temporal graph model that we are going to look at is the direct extension of the property graph data model (Definition 5) with a linearly ordered discrete time domain (Definition 7 and 8). The main way this is done in the literature is by labeling each graph element, i.e nodes and edges, with a valid time-interval. This means that the graph element only exists in the graph at the given valid time interval. There are also some that use start-time and duration labeling instead of an interval [13], but this is entirely practical as it is conceptually and logically equivalent as an interval, and is a matter for representation as we will look further into in Section 3.4. We are going to extend Definition 5 with a time aware property-value-function σ^T and an existence function ξ^T. The following defines the Temporal property graph model:

Definition 9 (Temporal Property Graph Model)**.** Is defined as a 8-tuple $G = (V, E, L, P, \rho, \lambda, \sigma^T, \xi^T)$

- The sets V, E, L and P, as well as the functions ρ and λ are defined in the same way as in Definition 5
- $\sigma^T : (V \cup E) \times P \times \Omega^T \times \Omega^T \rightarrow val$ is a partial function that maps the property name/key of particular node or edge at a certain time interval to a value.
- $\xi^T : (V \cup E) \times \Omega^T \times \Omega^T \rightarrow Bool$ is a total function that maps a node or an edge with a time interval to $Bool = \{True, False\}$ indicating whether the node or edge exits in that time interval.
- Ω^T is a linearly ordered discrete time domain (Definition 8).

With this we also need a set of constraints to ensure the validity of the definition. We have to make sure that all the sets and functions are in line with the existence function ξ^T. Firstly, we need ensure that edges do not exists outside of the range that time interval of the edges that they connect. We also need to ensure that property-value-function do not map from intervals where that graph element does not exists. The two following constrains will therefore be defined:

Definition 10 (Constraints for the Temporal Property Graph Model)**.**

- For every edge that exists in a certain time interval connecting two nodes, both the nodes also need to exist in the entirety of the interval.
- For every property-key value for a specific node or edge in an interval, the node or edge needs to exist in that interval

In Figure 3.2 we show an example of the same property graph as the property graph showed in Figure 2.5 in Section 2.2, but now accompanied with a timeline containing the intervals for each node and edge, indicating the time when these graph elements existed. This timeline essentially represents the two new functions ξ^T and σ^T, however σ^T is at the property-key level, which for this example remains unchanged with relation to ξ^T. If we further write out this function as set representation we would get the following, where we implicitly assume that we give them $True$-value and $False$-value as default for intervals not specified:

- For the nodes: { $(c1, 14{:}00, 23{:}59)$, $(c2, 14{:}00, 23{:}59)$, $(p1, 14{:}00, 23{:}59)$, $(p2, 16{:}00, 23{:}00)$, $(p3, 18{:}00, 22{:}00)$ }
- For the edges: { $(p1, c1, 14{:}00, 23{:}59)$, $(p2, c1, 16{:}00, 23{:}00)$, $(p3, c1, 18{:}00, 22{:}00)$, $(p2, p1, 17{:}00, 21{:}00)$, $(p2, p3, 19{:}00, 21{:}00)$, $(p3, p2, 19{:}00, 21{:}00)$ }

Notice how the constraints are satisfied in this example. Every edge that connects two nodes are entirely inside the interval of the nodes that connect them. For instance the edge $(p1, c1, 14{:}00, 23{:}59)$ both the nodes exist entirely in this interval: $(p1, 14{:}00, 23{:}59)$ and $(c1, 14{:}00, 23{:}59)$. The same needs to be satisfied for all other edge-node-pairs.

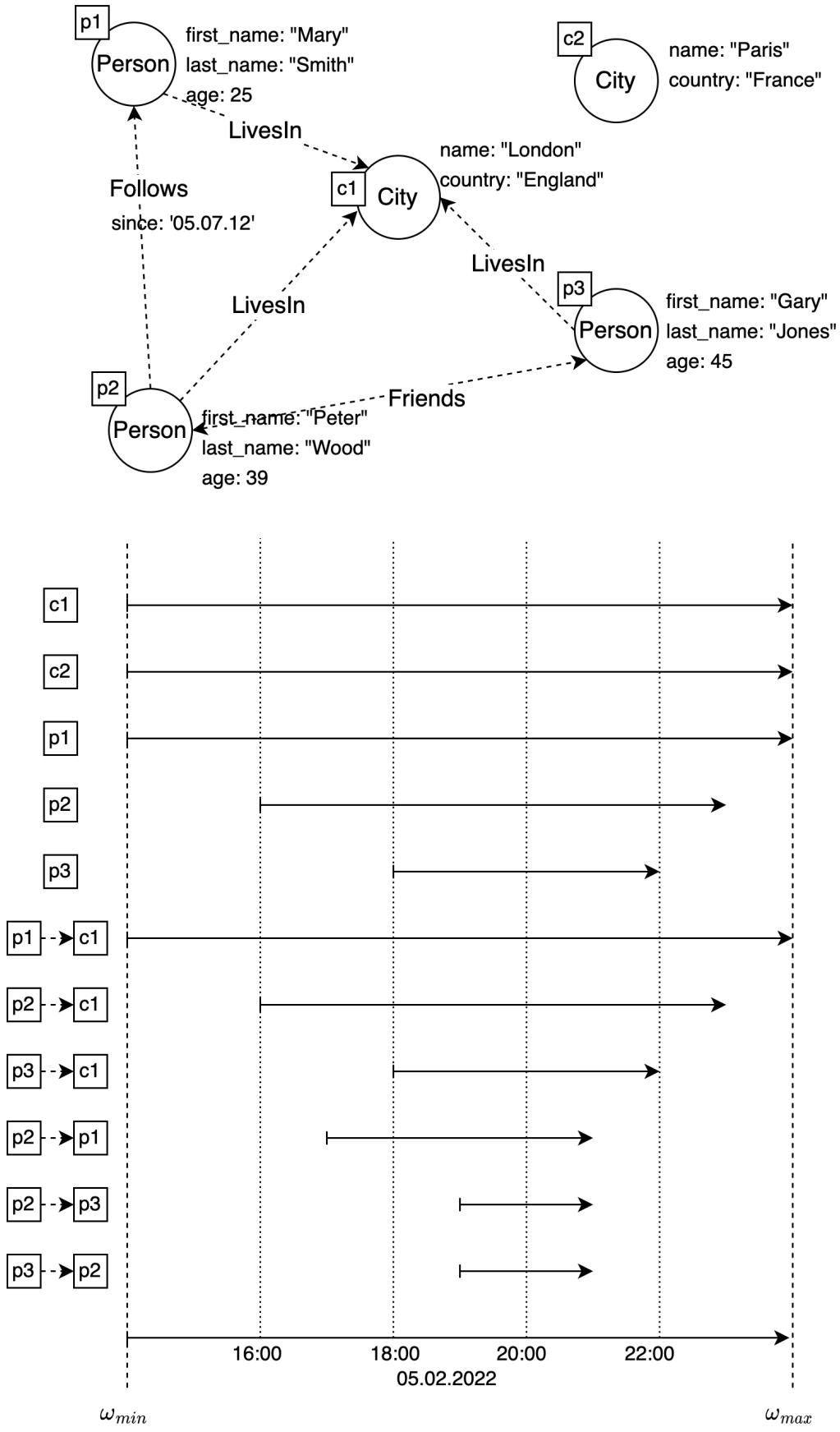


Figure 3.2: Shows a Temporal property graph based on the property graph in Figure 2.5 with a timeline indicating the existence of the graph elements.

Similar equivalent definitions are done by [17], [26] and [30] giving it names and abbreviations such as TGraph [26] or TPGM [30] or simply just Temporal Graph Model. There are slight differences in the definitions however, some map the valid time interval directly on the node set V and edge set E to the element without having a ξ function and leaving the property-key-value function σ unchanged. However, this means that a single node or edge technically cannot change value for a specific key during its lifetime, and for this to happened a new node or edge id have to be replaced along with a new property value, which in the end is a matter of practicality.

[30] also have a further extensions to this by having a collection of temporal graphs, called the TPGM database model. Similar collection model is defined in [13].

3.3 Alternative Definitions

From the literature we have found two additional ways of defining temporal graphs. The first one is based on temporal relations that are found in SQL and second is more in line with the streaming model where we have an update history set. As the first definition is very similar to our initial Definition 9, and have been logically proven to be equivalent by [26], we will not focus on it. Here we will outline the general foundation of *update history set* way of defining a temporal graph, though not rigorously define it as the previous definition.

Update History Set

This temporal graph definitions is made up of a single set that contains the entire update history of the temporal graph. This means that the graph starts at a complete empty state and a single entry in the update set indicate node, edge creations/deletions or property-key-value update at a certain time point. This model is more unifiable with the streaming model, and has for instance been used by [32] which supports both temporal and streaming graphs unified in one.

An example of such a set could for instance look something like this: $H = \{< t_i, created > , < t_j, deleted >, ..., < t_l, updated >\}$

[32] has more formalized and further definitions of update semantics. With this sort of set, we can further derive functions that get the similar information that we get from the other temporal graph models. For instance we can get all the nodes or edges by taking the union of all the creations. Or we get get the snapshot of the graph at a certain timepoint t by taking into account all creation, deletions and updates up to t. This means again that this model is logically equivalent to our two former definitions of temporal graphs

3.4 Temporal Graph Representations

In this section we are going to look at some physical and visual representations of temporal graphs, mainly through diagrams and figures for illustrations. In Section 6 we are going to summarize how these representations have been used by the frameworks we have studied.

Valid Intervals

The first representation is the one that follows the closest to our initial definition of temporal property graphs (Definition 9). This is essentially the same representation as the one in Figure 3.2, but here we indicate the interval of the nodes and edges directly inside the graph itself, instead of having the timeline separately below the graph. Figure 3.3 shows a simplified example of temporal property graph represented in the valid intervals representation, where we have omitted property-key-values for readability.

Snapshot Sequence

The snapshot sequence representation is likely the most intuitive, however also the most redundant in representation. Here, we simply have a sequence of snapshots of the temporal graph for each timepoint there is a change and where each snapshot represents the whole graph at that timepoint. We could have a snapshot for each single step of timepoint in the time domain, but that would be unnecessarily redundant even for the snapshot representation, however this is the worst case scenario in terms of redundancy. Figure 3.4 shows an example of the same graph in the interval representation (Figure 3.3) as a sequence of snapshots for each time there is a change in the temporal graph.

Delta Snapshots

The Delta Snapshots representation is essentially the same as the Snapshot Sequence representation, but without the redundancy of representing the whole graph at all the relevant timepoints. Here, we only represent the changes that happened at that timepoint from the last timpoint the graph changed. This representation is more inline with the update history set model. Figure 3.5 shows an example of the same graph as for the previous representation, but in the delta snapshots representation.

Transformed Graph

The Transformed Graph representation, sometimes also called Directed Acyclic Graph (DAG) form, of a temporal graph is one where we have each snapshot condensed into one single *"static"* graph. This representation has the same drawbacks of redundancy as the snapshot sequence representation, but can be considered for some time-dependent temporal graph operations. However, we will not focus on this representation too much further as it is generally considered as a *"cheap"* way of representing a temporal graph as a standard graph. Figure 3.6 shows the same graph as in the previous representation, but in the transformed graph representation.

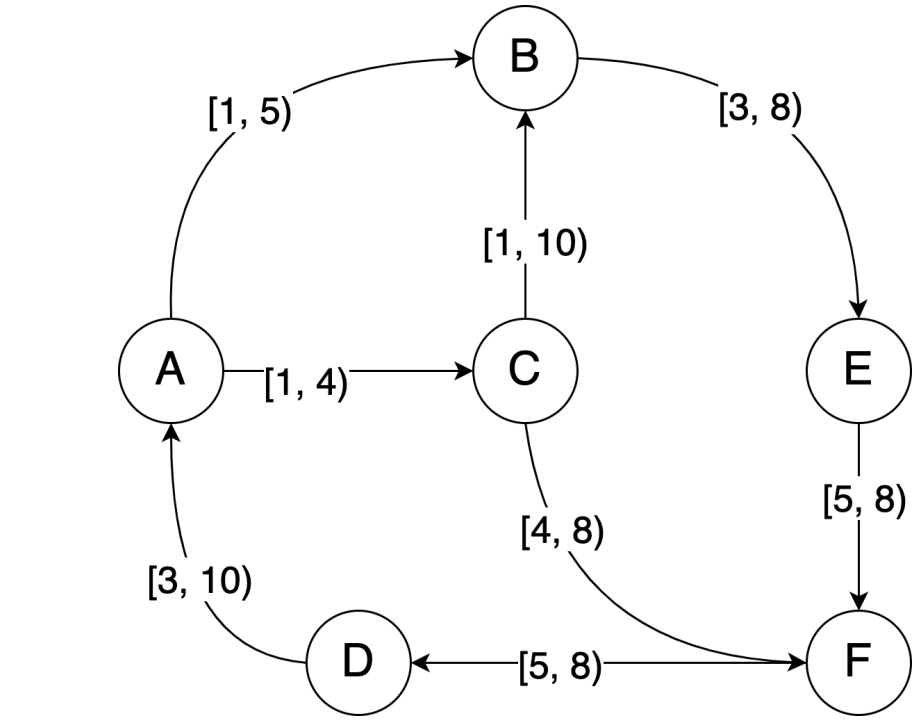


Figure 3.3: Simplified example of a temporal property graph in the valid intervals representation, where property-key-values and node intervals are ommited for readability.

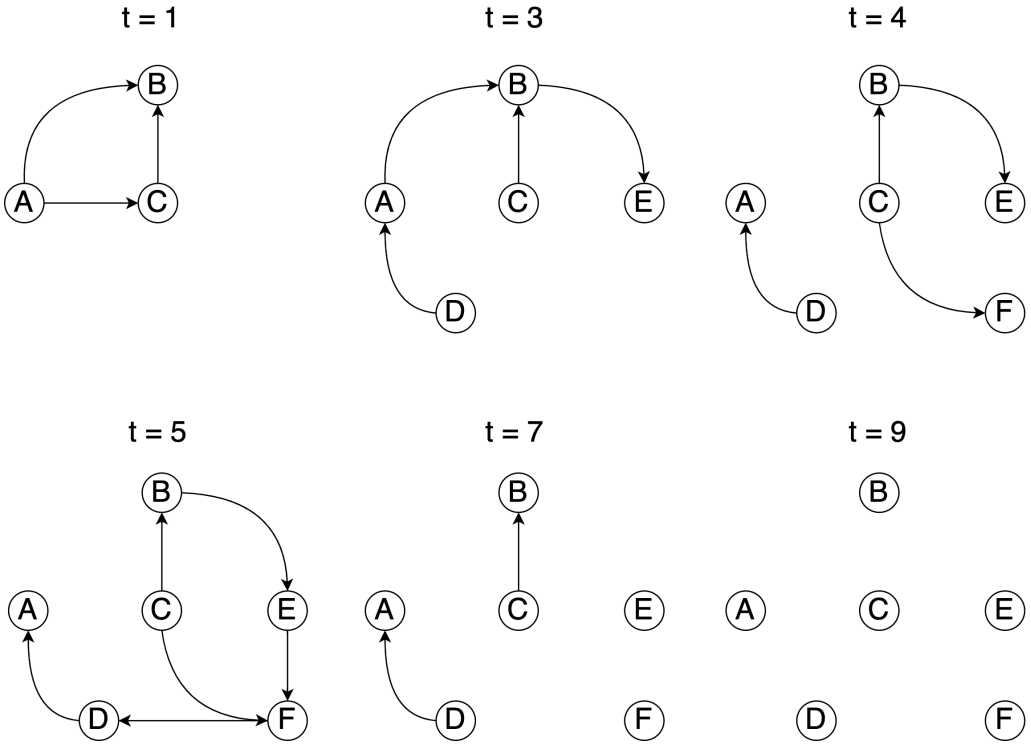


Figure 3.4: The same temporal graph as in Figure 3.3, but in the snapshot sequence representation.

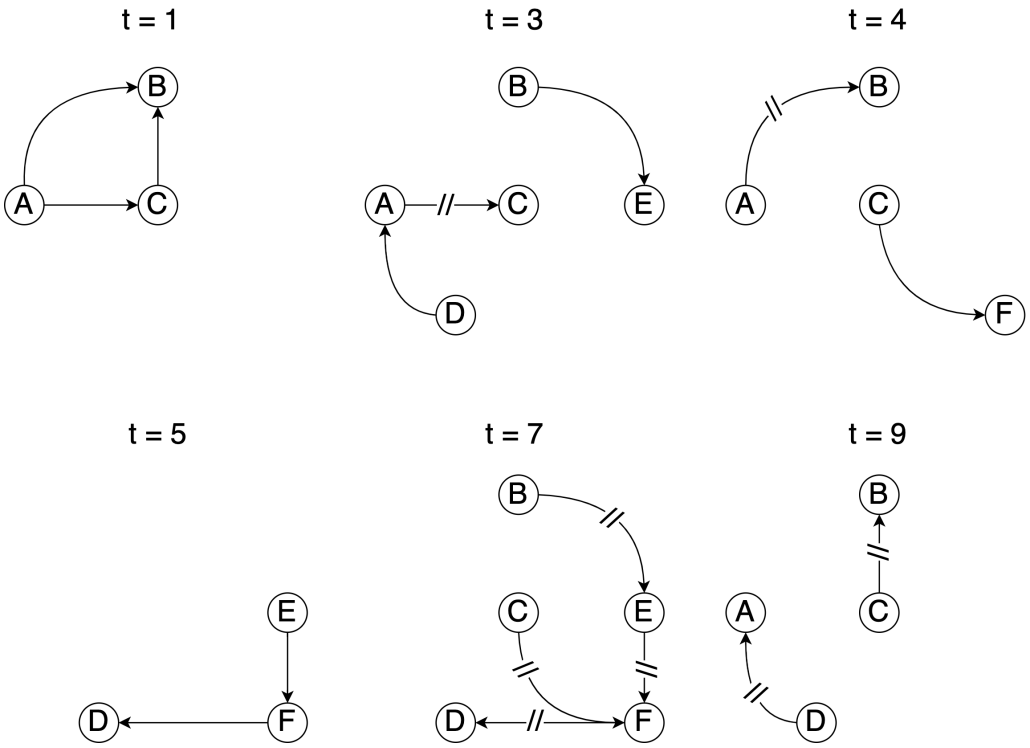


Figure 3.5: Shows the delta snapshots of the equivalent temporal graph in Figure 3.3 & 3.4, where the "//" indicate removal of the corresponding edge.

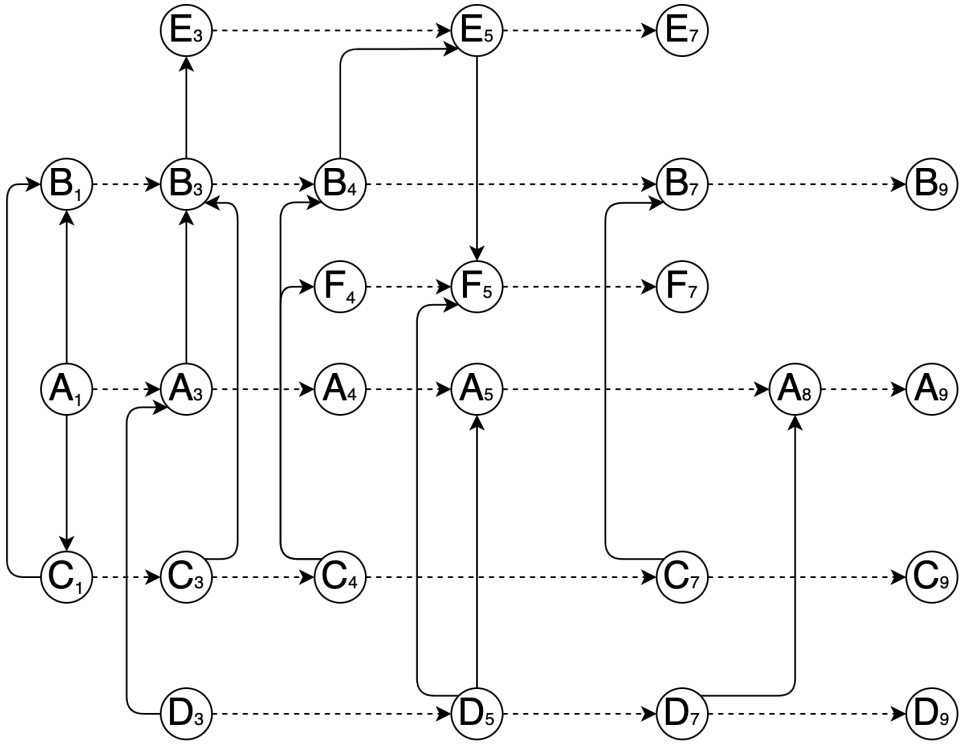


Figure 3.6: Directed Acyclic Graph representation of the same temporal graph as in Figure 3.3, 3.4, 3.5. Here striped lines means the temporal direction.

4 Temporal Graph Queries

In this section, we are going to present some special types of query operations that have received particular attention in a handful of the reviewed temporal graph frameworks and in the literature in general. These operations are extensions of the path queries found in GQLs that were introduced in Section 2.3. We will first introduce an extended form of predicates that adds temporal interval relations as new type of operator. Based on these predicates, we will further introduce temporal equivalents of constant length, variable length and weighted paths. Additionally, we will present some example queries based on recent languages specifically designed for expressing these types of queries, while at the same time pointing out some limitations in the current solutions.

4.1 Temporal Predicates

The queries enabled by GQLs from Section 2.3 could be useful for querying and pattern analysis on snapshots of temporal graphs. However, to be able to make time-dependent pattern analysis we need to be able to express predicates with temporal interval relations. For instance to be able to express patterns such as: Find all the pair of people that became friends before they entered the same university. Or more complex relationships such as: Find all common friends between two existing friends, where the common friends became common before the time the existing friends became friends. In the following sections we will give multiple examples of queries that can be made with this extended type of temporal predicates.

The standard GQLs have capabilities to express predicates on individual timepoints with time- and datetime-data types, however they are lacking in temporal *interval* relation operators. They also lack expressivity, as there is lacking of standard syntax for expressing such relations. This is further amplified by the fact that standard GQL does not assume that the underlying property graph is a temporal graph, making some queries impractical and some impossible.

Most of the temporal interval relation operators that make up a predicate can be summarized in Allen's classical paper [1] about managing time intervals. He introduces 7 main temporal interval relations and their inverses, making up a total of 13 relations (inverse of equals is equals). Figure 4.1 summarizes these with an accompanied timeline illustration. Notice that a time-interval is an open-closed interval $[\omega_{start}, \omega_{end})$, meaning the time-interval includes timepoints ω_{start} up to ω_{end}, but not including the timepoint ω_{end}.

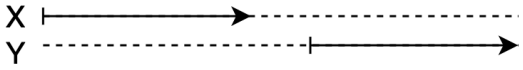
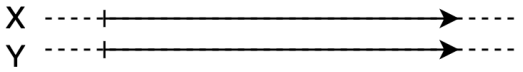
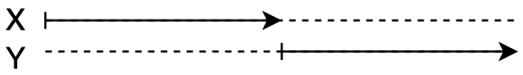
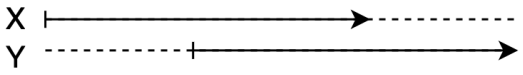
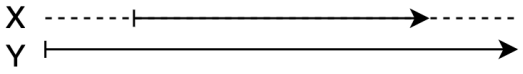
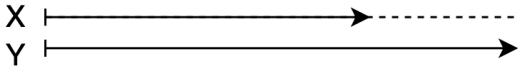
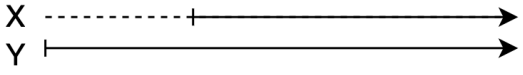
Relation	Inverse	Timeline Illustration
X *before* Y	Y *after* X	
X *equal* Y	Y *equal* X	
X *meets* Y	Y *met-by* X	
X *overlaps* Y	Y overlapped-by X	
X *during* Y	Y *contains* X	
X *starts* Y	Y *started-by* X	
X *finishes* Y	Y *finished-by* X	

Figure 4.1: Summarizes the temporal interval relations introduced in [1] accompanied with timeline illustrations.

These interval relations are the basis for the types of temporal paths that we will define, and are the following:

- **Temporal constant length paths:** Constant length path queries allowing predicates with temporal interval relation operators.
- **Temporal variable length paths:** Variable length path queries allowing temporal interval relations on each consecutive pairs of edges in the path.
- **Temporal weighted paths:** Temporal paths of any length that also allows user-defined cost-functions, which could be based on the intervals as well.

Furthermore, these definitions capture the capabilities that can be found in some of the novel temporal Graph Query Languages, which include: *TemporalGDL* [30], *Granite* [27], *Chronograph* [9], *TRPQ* [4] and *T-Cypher* [13], which we will also see examples of. For variable length paths and weighted paths we will give more generic definitions that allow even more query capabilities. More generic forms of temporal weighted paths in particular have not been given any attention in the literature, and is the basis for our framework extensions that we will see in Section 7.

4.2 Constant Length Paths

Lets say we want to find the all the paths of train trips that go from "Grand Central Station" and goes through a total of 3 stations. In addition, we need make sure that the the path of trips respects time, meaning that the time-interval of the first trips comes entirely before the next trip, and the next trip entirely before the last trip. Figure 4.2 illustrates how this time-dependent relationship would have to work.

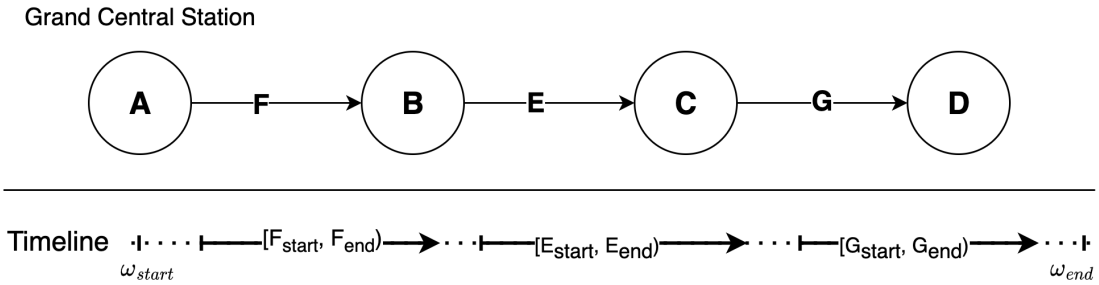


Figure 4.2: Illustration of a time-respecting path consisting of a total of 3 trips that begins in "Grand Central Station".

In Figure 4.3 we show an example query in *TemporalGDL* [30] that answers the path query illustrated in Figure 4.2. Here *.val* syntax indicates the valid time-interval of the graph-element (node or edge) of the corresponding variable. $before$- and $after$-functions is the temporal relations operators that can be found in Figure 4.1.

```
MATCH (a:Station)-[e:Trip]->(b:Station)
      (b:Station)-[f:Trip]->(c:Station)
      (c:Station)-[g:Trip]->(d:Station)
WHERE a.name = "Grand Central Station" AND
      e.val.before(f.val) AND
      g.val.after(f.val)
```

Figure 4.3: Shows a query in *TemporalGDL* [30] that gets the pattern of trips illustrated in Figure 4.2.

Notice further that the query in Figure 4.3 is a constant length path query with 3 edges, but these types of queries are possible with variable-length and arbitrary-length queries as long as we can create user-defined variables for the nodes or edges. However, this means that we can only create predicates including the start and the end node for longer length paths. Figure 4.4 shows a variable-length path query similar to the one in Figure 4.3. As we can see now, it is not possible to specify the interval relation between pair of edges so that the sequence of edges make up a valid time-respecting path. This is because the edge-variable e refers to all the edges in the path, and we require access to consecutive pair of edges. This is a major limitations of only extending GQLs with node/edge-variable predicates, and we are going to look at a more general solution to this in the next section, namely: variable length temporal paths.

```
MATCH (a:Station)-[e:Trip*3..10]->(b:Station)
WHERE a.name = "Grand Central Station" AND
      a.val.before(b.val)
```

Figure 4.4: Shows a variable-length path query with maximum length of the path as 10, not succeeding to answer the same query as in Figure 4.3.

4.3 Variable Length Paths

Here we will explore how to extend variable-length path queries such that the resulting paths are also time-dependent according to a interval relation. Firstly, we will define a generic temporal path and a property that allows us to specify the characteristic of the temporal path. Furthermore, we will examine some specific instances of our general approach, that have been studied extensively in the literature and have useful applications.

Generic Temporal Paths Definition

Firstly, we define a *temporal path*, which we interchangeably also call a *time-dependent path*, as a generic path with edges that also have time-intervals associated with them. Note that this definition does not impose any constraints on the time-intervals of the edges in the path.

Definition 11 (Temporal path)**.** Given a temporal graph from Definition 9, a *temporal path* is a sequence of edges from the temporal graph: $e_1(v_1, v_2, T_1)$, $e_2(v_2, v_3, T_2)$, ..., $e_k(v_k, v_{k+1}, T_k)$, where e_i's are the edges, v_i's are the nodes connecting the edges and T_i's are the time-intervals of the edges in the sequence.

Our goal is to impose a constraint on this path, such that any temporal relation from Table 4.1 can be specified to hold between each pair of consecutive temporal edge in the path. Note that this does not necessarily refer to the edge intervals of the edges in the originating graph, but the intervals of the edges constituting the resulting valid path. We formalize this constraint as a property of a temporal path below:

Definition 12 (Consecutive Edge Interval Relation)**.** Given a temporal path t_P, a *consecutive edge interval relation* is a temporal interval relation operator σ_t such that T_{i-1} σ_t T_i holds for all $k-1$ sequence of consecutive edge pairs in t_P.

There are some special instances of the general temporal paths that are of particular interest. They can be further specified based on the *Consecutive Edge Interval Relation* we defined above, as we shall explain. There are three main forms of temporal paths that are also defined in [13], [33] and [34] which are the following:

- **Continuous paths:** Paths that exists entirely in a single time-interval, all edge intervals of the path are equal to this time-interval. Essentially a generic temporal path with the resulting edge intervals having *equals*-relations between pairwise edges. Note that the interval is not user-specified and this type of path is usually constructed by taking the *intersection* between the edge-intervals from the originating temporal graph.

- **Pairwise Continuous paths:** A more relaxed variant of a continuous paths, defined as a generic temporal path that only requires *overlaps*-relations between pairwise edges.
- **Consecutive paths:** A generic temporal path that requires *before*-relation between pairwise edges of the path.

4.4 Weighted Paths

As we saw in Section 2.3 some GQLs have the capabilities of defining cost-functions associated with the paths for retrieving the top-k shortest paths. This could of course also be extended to apply for temporal paths of any length or any property. Temporal cost-functions also opens up the opportunity for new types of shortest paths, for instance where we can utilize start-time, duration and end-time of the interval in the cost-function. We will see examples how this could be integrated with temporal paths and the associated query results.

However, many of the proposed languages have not focused on giving this capability to the temporal graph query language. Therefore, we will not give examples of how this could be expressed in the query language, but rather look at example results given an equivalent non-weighted temporal path query. In Section 6.3 we will further discuss language limitations and extensions, and discuss temporal weighted paths in particular. In our extension work we have focused on implementing temporal weighted paths, which we will explain and discuss more of from Section 7 and onwards.

4.5 Example Queries

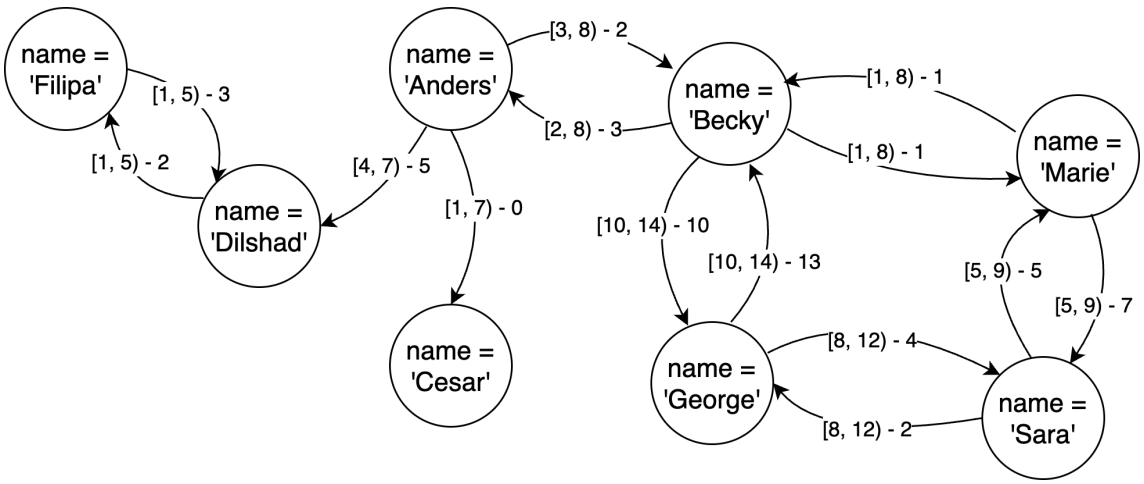


Figure 4.5: Shows a temporal graph equivalent to the one in Figure 2.8 representing a social network. The numbers alongside the intervals represent the edge-property *nr_messages*.

Continuous path

These paths can answer GQL path queries that require the entire path to have existed continuously in a valid time interval of the graph. For example, we could find all the friends of friends of friends (chain of 3) of each person and the time period when the

relationship existed along the entire path. Given the example graph in Figure 4.5, Figure 4.6 we can see the query to find all these paths originating from "Marie" expressed in the *T-Cypher*[13] language. Note how paths through "Sara" and "George" are not continuous as $[5, 9)$ and $[10, 14)$ do not overlap and $[1, 8)$ and $[8, 12)$ do not overlap. In addition, we show the result of a hypothetical shortest weighted path query given the cost-function being either interval-duration or interaction-weighted (i.e. $1/(1 + nr\_messages)$).

Query Expression	Query Results

`MATCH (a:Person), (b:Person),`
`path = cPath((a)<-[:Knows3..3]->(b))`
`WHERE a.name = "Marie"`
`RETURN a.name, b.name, interval`
	`+-------------------------------+`
	`| a.name | b.name | interval |`
	`+-------------------------------+`
	`| "Marie" | "Dilshad"| [2, 7) |`
	`| "Marie" | "Cesar" | [4, 7) |`
	`+-------------------------------+`

Shortest duration-weighted path	`+-------------------------------+`
	`| a.name | b.name | duration |`
	`+-------------------------------+`
	`| "Marie" | "Cesar" | 3 |`
	`+-------------------------------+`

Shortest interaction-weighted path	`+-------------------------------+`
	`| a.name | b.name |interaction|`
	`+-------------------------------+`
	`| "Marie" | "Dilshad"| 1 / 10 |`
	`+-------------------------------+`

Figure 4.6: Shows a continuous path query in *T-Cypher*[13] to retrieve the friends of friends of friends of "Marie" that existed in a continuous time-interval. Additionally, the shortest duration-weighted and interaction-weighted results are shown. Here we also follow node-isomorphism where applicable.

Pairwise Continuous path

These paths can be used to answer similar types of queries as with proper continuous paths, but where we allow the time-interval constraint of the path to be more relaxed. For instance in the last query, we could instead only want the immediate friend of friend relationship to necessarily be continuous, opening up for more relevant results than if we restricted ourselves to the whole path being continuous. In Figure 4.7 we see the equivalent query as in Figure 4.6. We now get more results, which also leads to the weighted path results to change. Note that the interval that is included in the result as a pairwise continuous path is the union of the intervals, representing the whole time-interval the path have existed with at least one edge.

Query Expression	Query Results

`MATCH (a:Person), (b:Person),`
`path = pairCPath((a)<-[:Knows3..3]->(b))`
`WHERE a.name = "Marie"`
`RETURN a.name, b.name, interval`
	`+-------------------------------+`
	`| a.name | b.name | interval |`
	`+-------------------------------+`
	`| "Marie" | "Dilshad"| [1, 8) |`
	`| "Marie" | "Cesar" | [1, 8) |`
	`| "Marie" | "Becky" | [5, 14) |`
	`| "Marie" | "Sara" | [1, 12) |`
	`+-------------------------------+`

Shortest duration-weighted path	`+-------------------------------+`
	`| a.name | b.name | duration |`
	`+-------------------------------+`
	`| "Marie" | "Cesar" | 7 |`
	`+-------------------------------+`

Shortest interaction-weighted path	`+-------------------------------+`
	`| a.name | b.name |interaction|`
	`+-------------------------------+`
	`| "Marie" | "Becky" | 1 / 23 |`
	`+-------------------------------+`

Figure 4.7: Shows a pairwise continuous path query in *T-Cypher*[13] to retrieve the friends of friends of "Marie" where pairwise time-interval of the relationships overlaps. Additionally, the shortest duration-weighted and interaction-weighted results are shown. Here we also follow node-isomorphism where applicable.

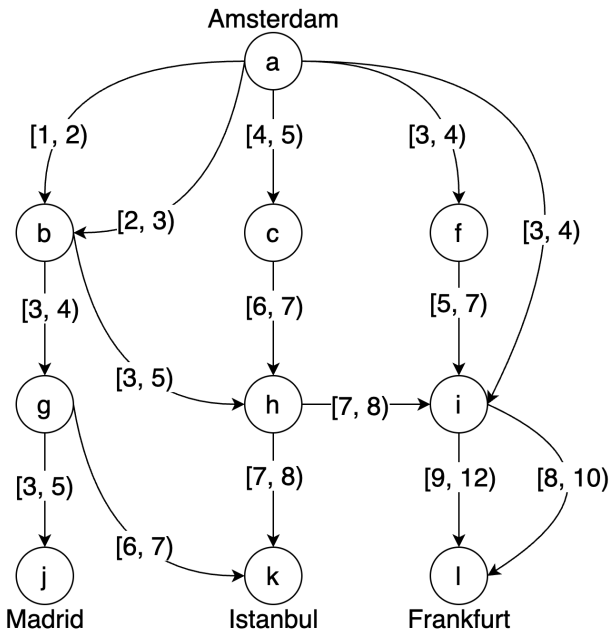


Figure 4.8: Shows a temporal graph representing flight network where nodes are airports and edges accompanied with intervals represent flight schedules.

Consecutive paths

Consecutive paths are more similar to the paths we have seen in the train trip example in Section 4.2 and Figure 4.2. These are more applicable when we want to find trips that are consecutive in time, and thus are useful for doing analysis on traffic networks such as flight networks. As the only consecutive paths we have in the social network graph in Figure 4.5 are "Marie"->"Becky"->"George and "Anders"->"Becky"->"George", we will use the the flight network in Figure 4.8 to make our example queries on. For consecutive paths we are also more interested in the duration and end-time of the trip, such that we can find the find the fastest trip and the earliest-time-arrival trip for our weighted path queries. We will go in further detail on additional use-cases for this in Section 7.3.

Query Expression	Query Results

`SELECT a.name, b.name, interval`
`MATCH (a:Airport)-`
`  (/NEXT/FWD/:flight/FWD)[2,3]`
`  -(b:Airport)`
`WHERE a.name = "Amsterdam" AND`
`  (b.name = "Frankfurt" OR`
`  b.name = "Istanbul")`
`ON flight_network`
	+-----------------------------+
	| a.name | b.name |interval|
	+-----------------------------+
	|"Amsterdam"|"Istanbul" |[4, 8) |
	|"Amsterdam"|"Istanbul" |[4, 8) |
	|"Amsterdam"|"Frankfurt"|[3, 10) |
	|"Amsterdam"|"Frankfurt"|[3, 10) |
	|"Amsterdam"|"Frankfurt"|[3, 12) |
	|"Amsterdam"|"Frankfurt"|[3, 12) |
	+-----------------------------+

Shortest duration-weighted path	+-----------------------------+ | a.name | b.name |duration| +-----------------------------+ |"Amsterdam"|"Istanbul"| 4 | +-----------------------------+
Shortest end-time-weighted path	+-----------------------------+ | a.name | b.name |end-time| +-----------------------------+ |"Amsterdam"|"Istanbul"| 8 | +-----------------------------+

Figure 4.9: Shows a consecutive path query in *TRPQ*[4] to retrieve the valid sequence of flights starting in "Amsterdam" and ending in either "Frankfurt" or "Istanbul". Additionally, the shortest duration-weighted and end-time-weighted results are shown. Here we also follow node-isomorphism where applicable.

4.6 Summary

Now that we have seen numerous definitions and examples of temporal graph queries, we will here briefly summarize and categories both the temporal and non-temporal graph queries. In relation to this, we will also summarize the capabilities of the newly suggested temporal graph query languages that we have studied.

Language / Properties	GQLs	Temporal-GDL	Granite	TRPQ	Chrono-graph	T-Cypher
Normal Graph Queries						
Constant Length	X	X	X	X	X	X
Variable Length	X	X		X	X	X
Weighted Paths	X					
Temporal Graph Queries						
Temporal Constant Length		X	X	X	X	X
Temporal Variable Length				X	X	X
Temporal Weighted Paths					*	**

Figure 4.10: Summarizes the capabilities of some of the newly suggested temporal graph query languages found in the literature.

Here the * and ** indicate that these languages have some operators and algorithms for some special purpose temporal shortest paths, but not capabilities for general purpose cost functions including intervals.

5 Distributed Programming Models

In this section we are going to go over some of the novel programming models and methods that are suggested in the literature for creating programs to process and analyze temporal graphs in a distributed setting. These programming models are based on existing models for distributed graph processing, such as the vertex-centric Pregel programming model from Section 2.4, which we will see two different extensions of in section 5.1 and 5.2 respectively. As we will see, some of these are more suited for programming the temporal graph queries that we have looked at in the previous section.

5.1 Snapshots Incremental

The snapshots incremental model is a framework for applying vertex-centric computations that we saw in Section 2.4 to a sequence of snapshots in a temporal graph. However, it is not really an extension of the programming model from the perspective of the programmer, as the programming interface is essentially unchanged from what we saw in Section 2.4. Frameworks [19], [23], [24] and [32] all support snapshots incremental computations, but with the major difference in how they implement the underlying parallel execution when taking multiple snapshots into consideration. However, the focus in this section is rather on what class of algorithms this model is capable of creating for temporal graphs.

Time-Independent Algorithms

Here, we are going to introduce a class of graph algorithms which often go under names such as *snapshot-reducible*, *time-independent* or just *static graph algorithms*. This means that they compute the graph at a specific state, i.e. snapshot, without accounting for the time element or any of the possible structural changes of the temporal graph. These algorithms are the same as the classical graph algorithms we saw in section 2.4, which were for instance SSSP and PageRank.

For temporal graphs these algorithms are still relevant as we could be interested in analyzing the temporal graph at certain individual snapshots in time. For instance we could use it to find the shortest path from London to Paris at 14:00 2nd of April 2016 in a road-traffic graph. Or process the PageRank state of a web graph at certain point in time, for instance to find the most influential page Monday morning. We could additionally take this analysis further by looking at how the PageRank state is developing over time, or see how the shortest path had evolved over a certain time interval. For instance, how he shortest path between Oslo and Stockholm have been developing over the last 5 years given that the conditions have changed over time. Then we simply would have to pick out a series of snapshots and then run the classical static graph algorithms on each snapshot in isolation.

These sort of computations are possible with a vertex-centric programming model with the addition of having snapshots incremental computations. This essentially means that we compute the same algorithm for each snapshot, but in an parallel and efficient manner which are explored further in [19] [23] [32]. However, as we will see in the next section, temporal graphs, particularly in the valid-interval representation, have an extended class of algorithms, often called *time-dependent* or *extended snapshot-reducible* algorithms. The algorithms required to compute the temporal graph queries that we have looked at so far

would go under this class of algorithms. The basic vertex-centric model with snapshot incremental computations is not sufficient for creating such algorithms.

5.2 Interval Centric

In Section 5.1 we looked at ways to apply time-independent graph algorithms to a series of snapshots in a time interval. Now we will look at a slightly altered programming model that enables programming of time-dependent algorithms. In this section we are first going explain time-dependent algorithms through an example, and then look at the interval-centric programming model which can implement this class of algorithms.

Time-Dependent Algorithms

Time-dependent algorithms, sometimes also called *extended snapshot-reducible* [26], is a class of graph algorithms that are specially tailored for temporal graphs, where the interval of the nodes and edges are specially accounted for. They differ particularly from time-independent algorithms, in that they take the whole temporal graph into account and do not look at specific snapshots in isolation.

The primary example of such algorithms, which give fundamentally different results from their time-independent equivalent, are so-called time-dependent path algorithms. For these paths the time interval of the edges in the path need to be taken into account for the validity of the path. It could for instance be that the interval of the edges need to overlap, be entirely before each other or all exists entirely in a user-specified interval.

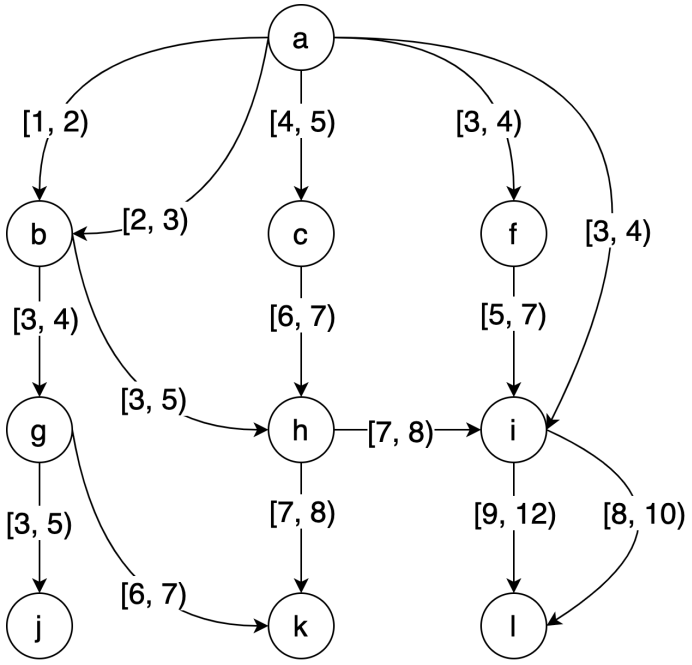


Figure 5.1: Shows a temporal graph representing flight network where nodes are airports and edges accompanied with intervals represent flight schedules.

To illustrate this further, lets say we have a temporal graph representing flight network where nodes are airports and the edges with accompanied intervals represent a flight from

start-hour to finish-hour. Such a temporal graph can be seen in Figure 5.1 in valid-interval representation. There are some paths in this network which would not be valid paths, for instance $< a, b, g, j >$ as we see that flight (g, j) have already started when we reach g. A time-independent reachability algorithm would not capture this as the mentioned path is valid when not accounting for the time-intervals. For another example, lets consider the shortest path from a to l, where for the time-dependent case want to reach l at the earliest time. We can see that the best path would be $< a, c, h, i, l >$, so that we can take the $(i, l, [8, 10))$ edge. A time-independent shortest-path would most likely find that $< f, i, l >$ is the best path as it has fewest edge-hops. Thus, we realize that static graphs with time-independent graph algorithms are not able to capture these paths and relationships. This illustrates that time-dependent algorithms represents a distinct class of algorithms compared to time-independent algorithms.

Other use cases for social networks could be time-dependent centrality measures, which could be used to estimate information propagation delays. For instance, a common centrality measure is to count how many paths that exists going out from a node to any other node. Here, we could reuse the case of time-dependent paths and apply it to this form of centrality measure with different time measures to extract useful information. Many of the traditional time-independent graph algorithms that we might be familiar with, have time-dependent equivalences, and many more are actively explored in literature as we speak.

Interval Programming

Similarly to the vertex-centric model introduced in Section 2.4, the interval-centric model follows the same pattern of having a user-implementable compute-function with message passing between neighbouring vertices in a iterative superstep manner. However, now we add an interval, which will be a part of identifying the vertex itself. In Figure 5.2 this modified programming interface is shown. This means that a vertex is identified with both its *vertexId* and its *interval*. This further means that in a single *superstep* we can have multiple *IntervalVertex*-instances with the same *vertexId* but with different non-overlapping *intervals* that holds a distinct *vertexValue*.

```
class IntervalVertex {
    String vertexId;
    Interval interval;

    int getSuperstep();
    VertexValue getVertexValue();
    IntervalEdgeIterator getEdgeIterator();

    void setVertexValue(VertexValue value, Interval interval);
    void sendMessageTo(String dstVertex, Interval interval, MessageValue msg);

    abstract void compute(MessageIterator messages);
}
```

Figure 5.2: Shows the modified vertex-centric programming interface with intervals for both vertices and edges.

Therefore, the execution of the *compute*-function is now split across the non-overlapping intervals for each *superstep*. If, we now want to utilize the *setVertexValue*- or *sendMessageTo*-functions we need to specify the *interval* as well. However, we are not required to pass in the *interval* which the current *IntervalVertex* is identified with, meaning that different interval relations can be expressed between neighbouring nodes. This further implies that in the next *superstep* we will have a new set of vertices for the same *vertexId* with different non-overlapping *intervals* that holds distinct *vertexValues*, compared to the previous *superstep*.

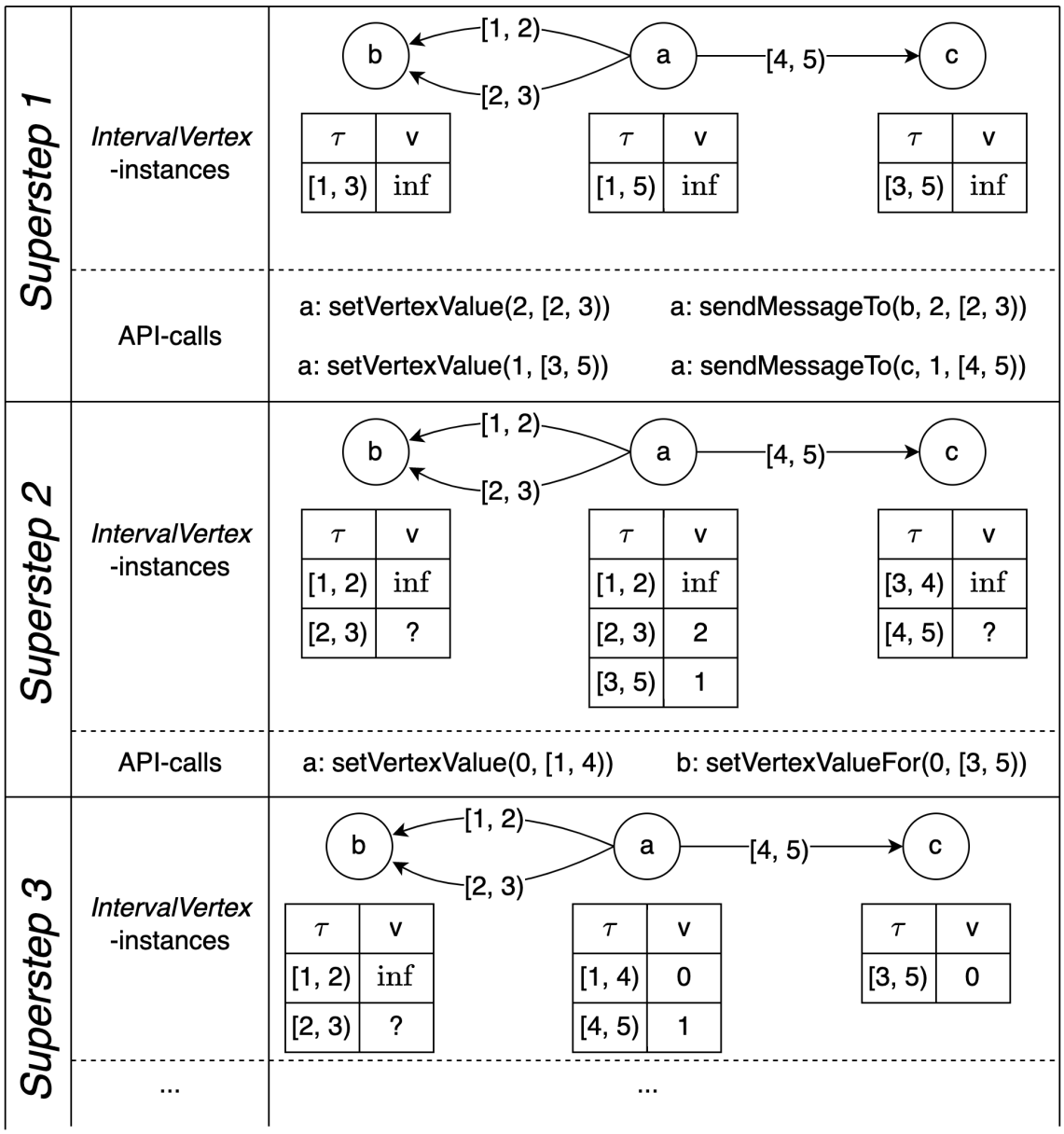


Figure 5.3: Shows how the new Interval centric API-calls impact the *IntervalVertex*-instances over multiple *supersteps*.

Figure 5.3 shows an example illustrating how this mechanism works over 3 supersteps with some arbitrary API-calls. Here we can see a table for each vertex, where a row represents the *interval* and the *vertexValue* corresponding to that *interval*. The values start with infinity just as an example. We are mostly looking at the perspective of the

a-vertex as it is the only one that has access to its neighbors, because we have directed edges here. Notice how using the *setVertexValue* can split and merge the intervals for the next *superstep* depending what interval we pass in. Furthermore, *sendMessageTo* with a given interval causes a new interval to be created for the other vertex so that the *compute*-function only receives the messages in their *interval*. The *?* indicate that this value could be anything depending on how the *compute*-function handles the incoming messages and uses the *setVertexValue*-function.

However, from the programmers perspective of just needing to implement the *compute*-function, we can isolated look at a single *IntervalVertex*-instance without thinking about the underlying relationship between all the intervals. This should be handled by the underlying framework, hopefully making it easier for the programmer to implement time-dependent algorithms where the intervals are integrated part of the nodes and edges. In Figure 5.4 we can see an implementation of Temporal Single Source Shortest Path (tSSSP). Here, we have also added a weight-attribute for the edges to further illustrate how this programming model can be useful when working with intervals. The algorithm essentially finds the cheapest path that also respects time, meaning that the consecutive edge intervals come before each other in the path.

```
void compute(MessageIterator messages) {
    // Init the minimum length to 0 if its the source vertex
    if (superstep == 1 && isSource(vertexId)) {
        setVertexValue(interval, 0);
        return;
    }

    // Update the minimum length from source vertex to this vertex
    int minLength = MAX_INT;
    for (MessageValue m : messages) {
        minLength = min(m.value, minLength);
    }
    if (minLength < getVertexValue())
        setVertexValue(minLength, interval);

    // Propagate minimum length to neighbouring vertices
    // We need to take into account the interval and the weight of the edges
    for (Edge edge : getEdgeIterator()) {
        int travelTime = edge.interval.end - edge.interval.start;
        int travelCost = edge.value;
        MessageValue value = new MessageValue(minLength + travelCost);
        Interval dstInteval = new Interval(interval.start + travelTime, MAX_INT);
        sendMessageTo(edge.dstVertex, dstInterval, message);
    }
}
```

Figure 5.4: Shows an implementation of Temporal SSSP with interval-centric programming.

In Section 8 we will implement a similar algorithm to tSSSP, but where we take into account multiple paths according to a top-k parameter, as well as having to deal with multiple source vertices and destination vertices based on user-predicates. In addition to

dealing with constraints based on interval relations, minimum and maximum length.

6 Temporal Graph Frameworks

6.1 Framework Characteristics

In this section we will present the categories and characteristics that we will be looking at when comparing and discussing the existing frameworks for processing of temporal graph queries. We have also included some frameworks that focus on general purpose processing of temporal graph data as well.

Temporal Graph Models

As we have seen in Section 3 there are some conceptually different models and representations of temporal graphs. Logically they can be seen as equivalent and can represent the same type of data and operations, however in practice they can operate quite differently. For instance the snapshot sequence representation requires different implementation than a valid-interval representation. These representations could either be stored in a database or directly in memory so that it is ready to be processed. The main categories that we will be distinguishing from when classifying the frameworks are the following:

- **Valid-interval**: Intervals indicate the existence of the graph element, the vertices and edges, in the application domain. These time intervals are usually stored along with the property data of the graph elements.
- **Snapshot representation**: Normal graph snapshots at individual timepoints or a function for retrieving the snapshots at arbitrary timepoints. These frameworks usually store the entire snapshots, where snapshots are laid out either in the temporal dimension (by timepoints) or in the structural dimension (by graph elements).
- **Delta Snapshots**: Conceptually similar to snapshot representation, but where we only store the changes that happened between each snapshot. Will always have a function for retrieving the snapshots at arbitrary timepoints.

Temporal Graph Queries

As we have seen in Section 4 there have been some new temporal graph query languages that gives the capabilities of expressing queries with temporal predicates with interval relations. This extends the constant-length, variable-length and weighted path queries of the standard graph query languages. When comparing the frameworks we will thus use the same categories as we did in Section 4, which summarized are the following:

- **Temporal Constant Length**: Constant length path queries with interval relation operators as part of the predicates.
- **Temporal Variable Length**: Variable length path queries where you can specify the interval relation between each pairwise edge interval of the path.
- **Temporal Weighted Paths**: Path query of any length where we can have a cost-function, which might include the interval as part of the expression, such that we can find the shortest paths.

- **Ad-hoc Algorithms**: Some frameworks provide some of the similar temporal graph query capabilities as we have seen, but they are specialized algorithms that does not allow for customizable queries with a proper query language.

Distributed Programming Models

As we have seen in Section 5 there have been work targeted towards creating programming models for programming distributed algorithms on temporal graphs. These have been mainly extensions of the existing vertex-centric programming model, originating from Pregel [22]. In addition, in Section 2.1 we introduced the functional and dataflow programming model that allows general purpose distributed data processing. Some of the frameworks that we have analyzed are based on this paradigm instead. However, as we will discuss there are performance and scalability differences between these two approaches, especially when implementing temporal graph queries that includes longer length paths. The main categories we will be distinguishing between are the following:

- **Snapshots Vertex-Centric**: The vertex-centric programming model that enables programming of graph iterative algorithms distributed and in parallel. However, with the addition of easily being able to apply them to a series of snapshots over a time interval. The underlying framework usually have performance optimizations for computing each snapshots efficiently and in parallel.
- **Interval-Centric**: Extension of the vertex-centric programming model, which allows for easier ways to implement time-dependent algorithms on temporal graphs. Time-dependent algorithms would also include algorithms for computing temporal graph queries.
- **Functional and Dataflow**: Allows for functional data operators for graphs and temporal graphs. These could for instance be existing or newly suggested operators such as *temporal-zoom*, *temporal-trim*, *temporal-grouping*, *snapshot-difference*, *subgraph* etc.
- **Temporal Graph Query Language (TGQL)**: Systems that primarily give an interface through a temporal graph query language, that usually is an extension of a normal graph query language such as Cypher.

Implementation Framework

Most of the frameworks we have been looking at are built on top of an already existing system or framework, usually a graph framework or a general purpose framework. We will categories the implementation framework based on the distributed programming model that it enables, as well as what type of system and workload it is targeted towards, where we also distinguish between: processing and storage (database). This is essential for if the system focuses on enabling distributed scalability and parallelism on processing of the queries. There are also slight differences between the actual frameworks, some are shown to perform and scale better, while other are known to have better programming interfaces and integration with other frameworks and ecosystems. We have summarized the implementation frameworks that were used for the systems that we have looked at:

- **Spark and GraphX**: Spark[13] is a dataflow framework as we explained Section 2.1. GraphX[14] is a main component of Spark and provide similar dataflow operators for Graph data as well as a Pregel API. We will explain GraphX in more detail for our own implementation in Section 8.4. The primary benefit of using Spark and GraphX is that they have good integration with other big data frameworks and ecosystems, making it easier to put the framework to use in practice.

- **Flink and Gelly**: Flink [15] is a dataflow framework quite similar to the Spark framework, but is more focused towards unifying normal batch processing with stream processing. Gelly [16] is again the equivalent of what GraphX is to Spark. Some have only used the dataflow operators provided by Flink, and some have used the Pregel API provided by Gelly in their implementation. In Table 6.1 we have put in parenthesis whether they use the dataflow operators or the Pregel interface in their implementation, this goes also for those that have used Spark and GraphX as well.

- **Giraph**: Giraph[17] is a framework that is primarily focused on implementing and providing a Pregel based programming interface. It usually shows great performance and distributed scalability with this.

- **Neo4j**: Neo4j[18] is a graph database that is focused towards transactional database workloads. It does have ability to run in distributed cluster mode. However, this is primary used for (Geo)-replication and does not support parallel processing where we partition the graph evenly among the nodes for high parallel performance.

- **Tinkerpop/TinkerGraph**: Tinkerpop[19] is graph framework/interface that is tightly coupled with the Gremlin[20] graph traversal language. There exists some different graph system implementations, but most are tailored towards transactional database workloads such as for Neo4j. TinkerGraph[21] is the standard such implementation which is an in-memory graph database.

- **Custom (No Framework)**: Some frameworks that we have looked at have built their solution from the ground up without starting from an existing graph framework. These implementations often have good performance, but usually lack in the temporal graph queries functionality that we are interested in.

Distributed Scalability

This category is essentially just a category that is dependant on the implementation framework and the distributed programming model. Here we are classify the framework as being able to scale and perform distributed or not, such that we can easily evaluate further limitations. Thus the categories that we distinguish between here are simply the following:

- Yes, Not Focus or Slightly

[13] https://spark.apache.org/
[14] https://spark.apache.org/graphx/
[15] https://flink.apache.org/
[16] https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/
[17] https://giraph.apache.org/
[18] https://neo4j.com/
[19] https://tinkerpop.apache.org/
[20] https://tinkerpop.apache.org/gremlin.html
[21] https://github.com/tinkerpop/blueprints/wiki/TinkerGraph

6.2 Existing Frameworks

Framework	Temporal Graph Models	Programming Models	Temporal Graph Queries	Implementatio Framework	Distributed Scalability
Chronos	Snapshot Sequence	Snapshots Vertex-Centric	Not Focus	Custom	Yes
Gradoop	Valid Interval (Bitemporal)	Dataflow, TGQL	Temporal Variable Length	Flink (Dataflow)	Yes
Portal	Valid Intervals	Dataflow	Not Focus	Spark and GraphX (Dataflow)	Yes
Raphtory	Delta Snapshots	Snapshots Vertex-Centric	Not Focus	Custom	Yes
Graphite	Valid Intervals	Interval-Centric	Ad-hoc Algorithms	Giraph (Pregel)	Yes
Granite	Valid Intervals	Interval-Centric, TGQL	Temporal Constant Length	Giraph (Pregel)	Yes
Chronograph	Valid Intervals	TGQL*	Temporal Variable Length + Ad-hoc Weighted Paths	Tinkerpop (Database)	Slightly
T-Cypher	Valid Intervals	TGQL	Temporal Variable Length + Ad-hoc Weigthed Paths	Neo4j (Database)	Not Focus
Tink	Valid Intervals	Dataflow	Ad-hoc Algorithms	Flink and Gelly (Pregel)	Yes

Figure 6.1: Shows a table summarizing the properties and capabilities of the different existing frameworks for distributed processing of temporal graph data.

GRADOOP

GRADOOP [30] is a graph dataflow system for scalable, distributed analytics of temporal property graphs. The complete framework has been continuously developed since 2005, primarily by the Database Group of Leipzig University. It is an an open-source research project with available source code repository [22] and is licensed under the Apache License 2.0. The latest GRADOOP paper [30] introduces TPGM temporal graph model, that is a valid-intervals based model, along with several dataflow operators specifically tailored for this data model. This includes a temporal graph query language (TGQL) called *TemporalGDL*, which is an extension of a subset of features found in Cypher, where they include temporal constant length path queries. They use Flink as the implementation framework and employs a table-based graph representation.

GRADOOP has a dataflow programming model with operators that work on the TPGM data model. These operators can work on a collection of logical graphs or a single logical graph. They have general purpose operators such as *transform*, *groupBy*, *aggreate* , as well as more specially tailored temporal graph operators such as *snapshot*, *subgraph*, *diff*, *query*. The *query*-operator is where they implement their temporal graph query language. It allows expressing constant-length and variable-length paths with node- and edge-variables that allows for interval relations on them. From all the frameworks we have reviewed GRADOOP has the most comprehensive set of operators and analytics functionality, and is closest to being a production-ready framework.

Portal

Portal [25] [26] is an earlier work in similar vein as GRADOOP in that it uses a dataflow programming model with operators that work on a temporal graph data model. They introduce a model that is called *TGraph*, which is essentially the same as our valid-time interval definition found in Definition 9. Their work focus more on creating an algebra that is unifiable with temporal relation algebra found in SQL-like systems. They describe an prototype-implementation of their model and algebra built on top of Spark and GraphX as the underlying distributed execution engine. They experiment with several common partitioning schemes, such as vertex-cut, edge-cut, label and hybrid approaches, and found that there is none that works best for all workloads. However, that label based approaches seem to perform overall the best. Further, their experiments shows that their prototype generally scales well on larger datasets.

ChronoGraph

ChronoGraph [9] is a system that extends the functionality of the Gremlin path query language with some time-dependent functionality. In their paper they aim to bridge the gap between point-based semantics (i.e. snapshots) and interval-based graph traversals (i.e. time-dependent algorithms) in one system. The system provides a syntax for both of these semantics, and presents a method for converting between them. In addition, they implement three so-called Gremelin recipes to further achieve time-dependent path functionality: temporal Breadth-First Search (tBFS), temporal Depth-First Search (tDFS), and temporal Single Source Shortest Path (tSSSP). They focus on applying the temporal syntax and path functionality to the TinkerGraph-engine, which is the existing database

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

engine coupled with the Gremlin language. This means that they did not apply their system in a distributed data processing context, but focused on transactional database workloads.

T-Cypher

T-Cypher [13] is an extension of the Cypher language giving temporal query functionality on the temporal property graph data model, where they use valid-time intervals. The paper [13] also presents algorithms for computing various types of temporal paths and query operators for different temporal semantics. They further demonstrates a proof-of-concept implementation using Neo4j, a widely used graph database system. They evaluate their performance on synthetic and real-world datasets, with initially insufficient scalability to complete some of the queries, however adding indexing capabilities showing that the longer consecutive path queries are possible in their prototype. Similarly to Chronograph, they did not apply their system in a distributed data processing context, but focused on the transactional database aspects.

Tink

Tink [20] [21] is a library for Flink providing a selected set of time-dependent path algorithms and temporal graph operators. Tink extends the property graph model to support time intervals on edges (not on nodes), and further provides some general purpose operators for transforming and analyzing this data model. The paper [20] also describes the implementation of the algorithms based on the pregel-based API provided by Gelly, the graph library for Flink. In particular they have focused on the single-source shortest path earliest-time-arrival algorithm. As Tink is a library for Flink, which is a general purpose distributed data processing framework, it provides distributed exeeution and scalability out of the box, but the papers have not focused on optimizing this any further. They report experimental results on some synthetic and real datasets, but not in any large scale. In addition, they also report some poor parallel scalability when increasing the node worker counts.

Graphite

Graphite [17] is the first paper to present the interval-centric model of computing (ICM), which we discussed in Section 5.2. The Graphite platform implements ICM using Apache Giraph, a graph computation framework especially made with the vertex-centric model in mind. They have further optimized message passing by experimenting with a novel time-warp operator that partitions and groups messages based on time intervals, which reduces the number of user logic calls and messages sent. Optimizations have been further explored in [7]. They use ICM to implement 12 algorithms for both time-independent and time-dependent graph analysis, including temporal single-source- shortest path and temporal reachability. They test their performance on various large datasets using a 10-node commodity cluster setup. They report of achieving up to an order of magnitude speedup over the baseline platforms and showing general parallel weak-scalability, which is a desirable scalability property.

Language / Properties	GQLs	Temporal-GDL	Granite	TRPQ	Chrono-graph	T-Cypher
Normal Graph Queries						
Constant Length	X	X	X	X	X	X
Variable Length	X	X		X	X	X
Weighted Paths	X					
Temporal Graph Queries						
Temporal Constant Length		X	X	X	X	X
Temporal Variable Length				X	X	X
Temporal Weighted Paths					*	**

Figure 6.2: Summarizes the capabilities of some of the newly suggested temporal graph query languages found in the literature.

6.3 Limitations and Further Extensions

By further analysing the frameworks and looking at Table 6.1 and Table 6.2, we notice some limitations especially with the overlap of some features. In general, we notice a gap in the overlap between temporal path query functionality and the focus on scalable distributed execution in a data processing context. Furthermore, we notice a lacking of languages that focus on combining general temporal paths with user-defined weighted shortest paths. In this section, we are going to discuss these limitations and further extensions, and introduce what we are going to focus on in our prototype.

Distributed Scalability Limitations

To run the queries enabled by TGQLs (Section 4) in a distributed and in parallel, they have to be implemented using either dataflow operators that we saw in (Section 2.1) or a variation of the vertex-centric programming model (Section 5.1, 5.2). In this section we will review the current efforts in this direction and identify some weaknesses and limitations of the current solutions.

TemporalGDL was implemented using a set of dataflow operators in Flink, generally showing good scalability on larger datasets. However, some dataflow operators have been shown to be inefficient for path querying, specifically join operators are expensive for longer length paths. In their paper [30], they have show the usage of join operators to construct the paths and the negative performance impact, and explore options for creating query planners, with both promising and of varying results. Also, other more general purpose operators which we are familiar from MapReduce [12] are known to not scale for specialized graph algorithms, in particular for path algorithms. This has been extensively explored in the past by [17], [22], which was the reason for the emergence of vertex-centric computations. This means that utilizing a vertex-centric pregel type of programming model might boost scalability and performance.

TRPQ described an additional syntax based on regular grammar that makes it easier to transform the language into expressions-form that are more suited for execution and processing. They proposed some algorithms for transforming the syntax into an execution tree that could be processed by a set of dataflow operators, however they did not focusing on actual distribtued execution. As executing these expressions would require execution of dataflow operators, it could have some of same limitations with longer length paths.

Chronograph's implementation is tightly coupled with the Gremlin language and the TinkerGraph engine that comes with it. This engine is more tailored towards sequential transactional database workloads, meaning it is not optimized for parallel and distributed execution on larger scales. Their experiments also shows rather poor scalability with increasing dataset sizes and particularly longer path queries. Their largest experimental datasets would not count for a large enough scalability test for distributed data processing targeted workloads.

Granite [27] introduce a distributed query planner and execution engine utilizing the interval-vertex-centric programming model. They implement a temporal graph query language with temporal constant length paths, where their engine shows great performance and scalability on larger datasets with many different types of workloads. It generally shows that a vertex-centric based implementation could be well suited for implementing these types of queries. It makes this case focus more on performance and scalability,

however lacking in path functionality.

Language Limitations

TemporalGDL is a language that support many of the TGQL functionalities, but as we saw in Figure 6.2 they lack support for temporal variable length paths and is also missing weighted shortest path functionality. Chronograph [9] have extended some functionality of the Gremlin language with ability for some time-dependent path functionality. However, as Gremlin is not not a declarative query language it is not seen as an official GQL, which comes with some limitations in general aspects of the language. In addition, they have also lack for user-defined weighting of the nodes and edges of the path. Granite [27] Have implemented their own prototype syntax that supports constant-length paths with node-temporal predicates. This language however is limited to constant-length path queries, which is an obvious limitation. The TRPQ-language [4] supports temporal variable-length queries with generally good support for temporal predicates with interval relations. They have some additional functionality where you can explicitly navigate individual steps in the temporal or structural dimensions as well. However, they do lack some functionality when it comes to expressing cost-functions and temporal weighted paths.

We notice that there is a lack of path query models that can express temporal paths in combination with lengths and weights associated with the path. As we have seen, weights and costs can be used to compute user defined shortest paths. In addition, weighting allows you to get the top-k result in accordance to some weight-score, which could also apply to the top-k shortest paths. This functionality is available in some GQLs as we saw in Section 2.3, however it is not been explored to do this for temporal paths that have interval relations between edges of the path.

A general path query model that incorporates temporal paths with lengths, and weighting in one model has as far as we seen not been explored. The nearest is some specialized cases of consecutive paths, such as fastest paths, foremost, reverse-foremost and shortest-length paths that have been seen in papers such as [9], [13]ß, [21] and [34]. However, these are specialized path algorithms that that have limited parameters and no further customization options. For instance, they only focus on *single-source* shortest paths where the source node and most of the times also the destination nodes are given. They do not have additional query capabilities such as the length of the path, the specific interval relation between edges of the path, additional tests on the nodes or custom weight functions. The following list of features summarizes this:

- Constraints on the length of the path, minimum and maximum length, meaning variable length paths.
- Options for interval relation operator between consecutive pair of edges.
- Arbitrary filter predicate on start, end and intermediate nodes, meaning that *multiple sources* and *multiple destinations* are considered for shortest paths.
- Custom weight functions defined on nodes and edges to be used to calculate the shortest path.
- Specifying a value k for retrieving the top-k shortest paths.

Further Extensions

We would like to explore a general model that can express queries for such shortest paths where we allow the combination of the user-specifiable features listed above. Realizing that creating a complete functioning path query language is comprehensive, we only want to focus on the intersection of functionality we listed above. That is why we want to keep the model as simple as possible, focusing on these combined aspects only. However, our model should be relatively easily added or incorporated in any other language or system.

At the same time, we are focusing on distributed data processing of temporal graphs. Thus, we ultimately want retrieve and compute the shortest(s) paths from the temporal graph in an efficient and scalable way. This means that we need to use existing programming models and operators that scale well, such as the vertex centric computing models that we have seen in Section 2.4 and 5.2. Therefore, when we create a query model we need to keep this processing context in mind so that it can be executed in a reasonably efficient way, while also be technically feasible to implement.

We will first introduce a simple query model that can take in all of the features we mentioned in the language limitations as parameters to the model. This model will then be able to express shortest temporal weighted paths. We will then look at two application areas where we can apply our model, namely: Social network analysis and transportation network analysis, where we will go more in detail on how this could be useful for practical analysis. We will look into some specific use cases based on existing analysis use-cases found in the literature, but with slightly added features showing the extended possibilities of our model. For social network analysis we will particularly look at two cases: Interaction paths and Recruitment recommendations. For transportation networks we will look at three types of shortest duration paths: Earliest arrival time, latest depature time, and fastest path.

Furthermore, we will introduce the algorithms and data structures required for distributed processing of our query model. This will be based on the ideas behind the interval-centric model where we make some extensions and extra data structures required for our implementation. Then, we will implement a prototype of this with a graph processing framework utilizing a Pregel-API, showing the capabilities of the model and the algorithms. We will test and analyze the performance and scalability with experiments primarily on synthetic datasets of various sizes, where we will try to test the scalability with as large as possible datasets. The list on the next page summarizes the work we will focus on and attempt to achieve.

Outline of Our Extensions

We want to answer the following questions with our extensions:

- Can we implement the temporal graph queries we have discussed so far and seen in Section 4, where we make the previously unexplored extension of having weighted paths?
- Is it possible to implement these queries so that it is able to run distributed on multiple machines in parallel, and can scale with larger datasets and worker counts?

Thus, we will attempt to create a prototype with the following capabilities and properties:

- Create and implement a query model with a set of parameters that can solve the sort of temporal weighted paths we explained above.
- The focus is to make it run in parallel, distributed and in-memory following the properties of distributed data processing frameworks (see Section 2.1).
- Achieve this by utilizing a variant of the interval centric programming model (Section 5.2).
- Further we will focus on expressing novel queries on domain-specific datasets.
- Run experiments on these datasets, where we will experiment with different model parameters, graph sizes, node worker counts and partitioning schemes.

As part of the implementation of this prototype, we will do the following technical background work:

- Choose an underlying distributed graph processing framework that allows a Pregel-API .
- Extend the Pregel-interface of the framework to work with intervals, however not focusing on optimizing it as it is already been focused on in [7] and [17].
- Implement a suitable temporal graph representation that is makes it possible to create queries and run the algorithms on.
- Choose a programming language that makes it practical and reasonably efficient to implement the prototype with the chosen framework.

Part II

Framework Extensions

In this part we will explore and experiment with extensions of the concepts we have studied in our literature review. In Section 7 we will introduce our query model that enables temporal weighted path queries, accompanied by two application areas with example use cases. In Section 8 we will construct the necessary algorithms and data structures to implement the query model in practice. In Section 9 we will go over our experimental methodology and further details of our experiments and discussions around our results.

7 Temporal Weighted Path Queries

In this section we are going to present our query model and the set of parameters that users can supply to create the desired temporal weighted path queries. We are then going to show some application areas and use-cases where these queries and resulting shortest paths can create valuable analysis. The two general areas we are going to focus on are: Social networks and transport networks. For both areas we are going to look at two to three use cases where we can apply our model.

7.1 Query Model

The query model should have most of the capabilities that we have seen from GQLs and temporal paths in previous sections. We are not going to give in-depth examples and explanations, as we have seen this in previous sections, and this is essentially an aggregation of that information. For the parameters we do not define a rigorous language syntax, but instead give simple function definitions like we are familiar from traditional programming languages. For instance $\rightarrow$ would indicate the return type here, which could be *boolean*, *number*, *interval* etc.

Node Predicates

The first query capability that we want to support is the ability to select what source, intermediate and destination nodes that is allowed to be a part of the path. Thus we are going to define three parameters that can be specified with user-defined predicate functions given an arbitrary node or edge. Remember that we are working with a temporal property graph, meaning that every node have a type-label, property-keys with values, and an interval. Thus, we define the user-defined function as an arbitrary function that takes in a node with label, properties and interval and computes a boolean value (True or False) indicating whether this node will be included. This is similar to what we have seen with GQLs and the WHERE-clause. The following list summarizes these three functions:

- $sourcePredicate(node) \rightarrow boolean$
- $intermediatePredicate(edge) \rightarrow boolean$
- $destinationPredicate(node) \rightarrow boolean$

Consecutive Interval Relation

Since, we want to retrieve general temporal paths, we want to support the ability to select a consecutive interval relation. The most relevant cases will be to supply *overlaps* or *before* for getting continuous, pairwise continuous or consecutive paths. We will also need to give the possibility of defining what is the next valid interval of the path. For instance for *continuous paths* the next valid interval is the intersection of the current interval with the next edge interval. For *pairwise continuous path* or *consecutive path* this would simply just be the next edge interval. We will call this interval function the *nextInterval*. The following following parameter functions summarizes this:

- $intervalRelation(interval, interval) \rightarrow boolean$

- $nextInterval(interval, interval) \rightarrow interval$

As we in practice are mostly interested in *continuous*, *pairwise continuous* and *consecutive* paths we will also have the option to specify this directly as a preset without having to define the two functions each time.

- $temporalPath \rightarrow \{Continuous, PairwiseContinuous, Consecutive\}$

Weight Functions

For the weight functions we simply want to give the option for a user-defined function that maps each edge to a numbered weight that will be used for scoring the paths. This weighted function which returns a number can be based on the type-label, properties or the interval of the edge, for instance a numerical formula that uses a combination of these as parameters. It could also be useful for the user to define the weight based on other entities or relationships or a combination. However, we have left this option out of the query capabilities as more comprehensive weight mapping is something that should left to a separate pre-processing step. In summary, the user-defined function we provide is the following:

- $weightMap(currentWeight, edge) \rightarrow number$

Lengths

The lengths parameters are essentially the minimum and maximum length of the path, similar to what we have seen from the capabilities from GQLs. The minimum length is at least 1 and can be further specified. The maximum length can be specified as a value that is larger or equal to the minimum length. The following parameters summarize this:

- $minLength \rightarrow integer$
- $maxLength \rightarrow integer$

Top-K Parameter

This parameter is just a simple number k indicating the number of paths that we want to have in the result, which is sorted by the weights of the shortest paths:

- $k \rightarrow integer$

7.2 Social Network Analysis

Here we are going to base our example on a social network data model and accompanied example queries coming from the Linked Data Benchmark Council's (LDBC) Social Network Benchmark (SNB) [15]. In the experiments we will also use their comprehensive data generator for generating large synthetic datasets used for both validation testing and scalability testing.

The Social Network Data Model that is provided by LDBC is grouped into two parts: dynamic and static. The dynamic part is where we have a lifespan, meaning valid interval, on the entities and relationships. The dynamic part consists of entity-classes such as Person, Forum, Message, Post and Comment and the natural relationship-classes which occur between most of these. The static part is entities and relationships that always exists in the domain, such as City, Country, Continent, University and Company. The details of the data model schema can be seen in the UML-based diagram in Figure 7.1, which is taken directly from the benchmark report [15].

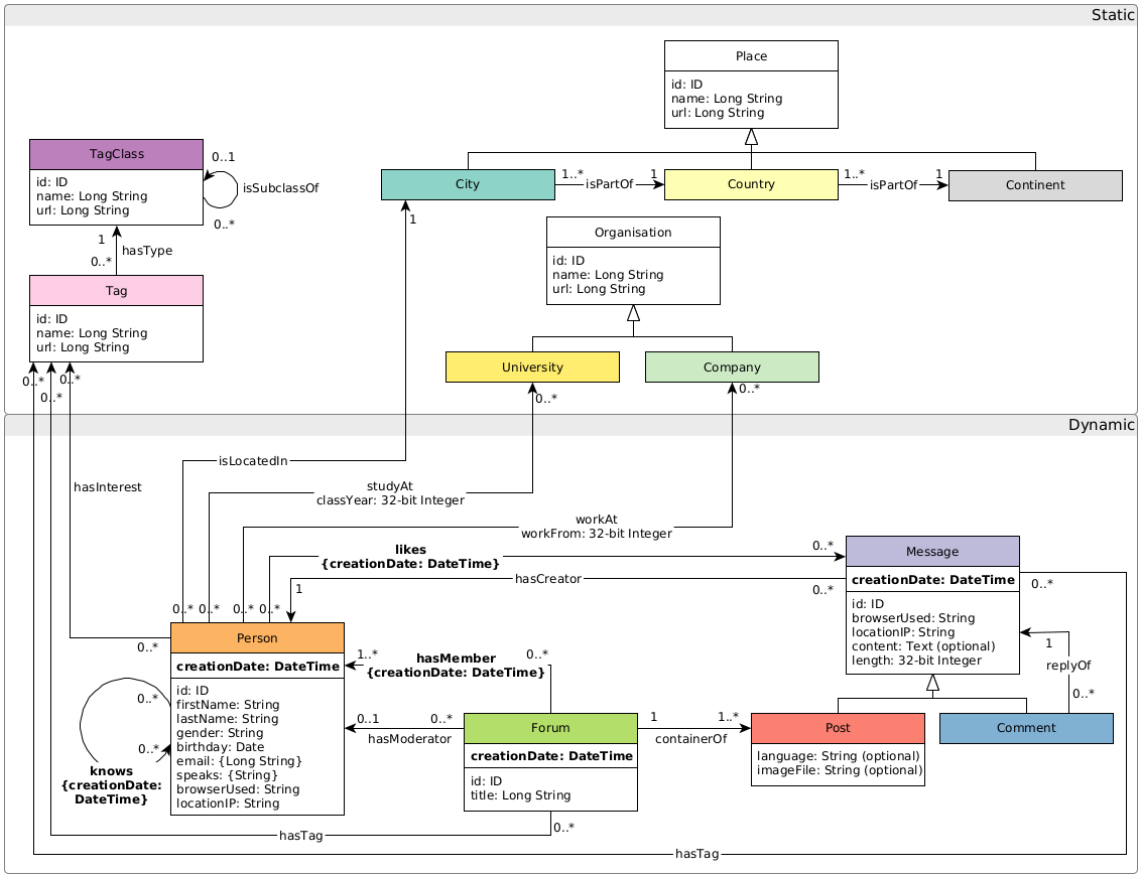


Figure 7.1: Shows UML-based Entity-Relationship schema of the social network data model provided by the LDBC SNB Benchmark [15]. Taken from [15].

Based on this type of social network data model, we can create numerous useful analysis use cases using our query model. We will be looking at two in particular: interaction paths and recruitment recommendation paths, where we have taken inspiration from the queries found in the benchmark. For these use cases we will mostly be using the dynamic part and the Person and the Person-knows-Person relationship in particular, however many of the other classes have to be used to preprocess the temporal graph to the correct format.

In the analysis use-cases we will use two terms to explain and define the queries, which we will here distinguish between, namely: *analysis parameters* and *query model arguments*. The query model arguments are the arguments we give into the query model we defined in the previous section, which means node predicates, weight function, length, interval relation and top-k. The analysis parameters are the parameters we further use to get different variations of the specific analysis use-case. However, the analysis parameters will most of the times be used as part of an expression which ends up as arguments to the query model.

7.2.1 Interaction Paths

The first type of analysis we will focus on is shortest interaction paths between two given groups (sometimes called clusters) of Persons in a social network. This is based on the knows-relationships between Persons in the data model, and the number of interactions those two pair of Persons have had between each other. From the original data model from SNB, this number comes from the number of replies they have had between posts and comments between each other on Forums they have been active on. However, here we are going to assume that this number is projected down to the knows-relationship as a property on the edge. Thus, we have a graph with Persons as nodes and knows-relationships as edges with *numInteractions* as numbered property on the edge. An interaction path is then a (pairwise) continuous temporal weigthed path between two arbitrary Person-nodes. The analysis we want to answer is then: What is the shortest interaction paths between two groups of Persons in the social network, where we now will go in more detail what specifically this means for analysis parameters and query arguments. Figure 7.2 shows an illustration of the sort of shortest interaction paths that we are interested in.

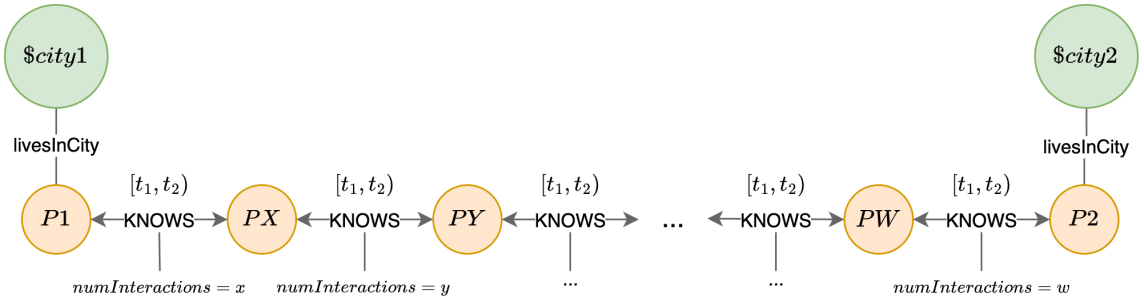


Figure 7.2: Shows the general structure of an interaction path where the Person groups are identified with which city they live in. Similar to diagram found in [15].

For the specific query we will base some of our analysis parameters and query arguments from the SNB, first and foremost the weight function. SNB defines the weight function of an edge in the interaction path with the following formula, and we will use this formula as the weight function query argument:

- $max(round(40 - \sqrt{numInteractions}), 1)$

This means that the higher number of interactions will give a lower number of weight for the path, which means that the shortest interaction path is the one where there is high number of interactions between the persons in the path.

At the same time we also see that that longer the length of path is, i.e. number of edges in the path, the more expensive the path also becomes even though there is a high number of interaction. That is why it would also be interesting to set a minimum length constraint so that we can retrieve more interesting interaction chains, rather than just the 1 or 2-length paths, which most of the time would be the shortest. We will define this as a one of the analysis parameters, which directly will also be one of the query arguments:

- $\$minLength$: The minimal length of the interaction path.

Furthermore, we need to specify what criteria that classifies a group/cluster of Persons . In general, this could be anything, for instance age, country, name or any other similarity measures. For our case we are not that interested in computing clusters or similarity measures, but rather that this is a part of the query. Therefore, we are will base ourselves on the SNB so that we get the most immediate meaningful outputs. They define their group/clusters of Person based on the city they live in, where they define two of their parameters based on this. We will therefore use the same two set of parameters as parameters for our query:

- $\$city1$: The city to classify the first group of Persons.
- $\$city2$: The city to classify the second group of Persons.
- $\$city1 \neq \$city2$, and no Person should live in both $\$city1$ and $\$city2$.

In addition, as we are working with temporal graphs an interaction path will only be valid if it is a continuous or a pairwise continuous path, where a continuous one would result in a stronger interaction path. If there is a gap in time between the knows-relationships in the path, we would from an analysis point of view not count this as proper interaction path. The time-interval of the interaction path will then also become a part of the result. The query argument to our model will then be a continuous path:

- Continuous path or Pairwise Continuous path

Lastly, we are also interested in specifying a value for limiting the result to the k shortest paths. We will simply have this as a parameter for the query, but a good starting point would be to just get the 5 best interaction paths. Thus, we have the last parameter as the following:

- $\$k$: Value for the number of shortest interaction paths we are interested in.

To summarize the analysis parameters and the arguments we send in to our query model, we have the following list of arguments, seen in Figure 7.3:

Parameter	Query Argument
Source Nodes	$livesInCity = \$city1$
Destination Nodes	$livesInCity = \$city2$
Intermediate Nodes/Edges	$edgeLabel = KNOWS$
Temporal Path	$ContinuousPath$
Weight Function	$max(round(40 - \sqrt{numInteractions}), 1)$
Lengths	$minLength = \$minLength$
Top-K	$\$k$

Analysis Parameters
$\$city1$
$\$city2$
$\$minLength$
$\$k$

Figure 7.3: Summarizes the analysis parameters and the query arguments to our query model for an interaction path query.

7.2.2 Recruitment Recommendations

This is an analysis case for companies that wants to find relevant candidates for their company through study- and knows-relationships originating from current employees at their company. This could for instance be used on professional networking and career development social networks, where LinkedIn is an real world example of such a network. The main characteristic difference from a interaction path is that we have potentially many destination vertices, but fewer intermediate edges that satisfies our predicate. Figure 7.4 shows a figure of the general structure of a recruitment recommendation path.

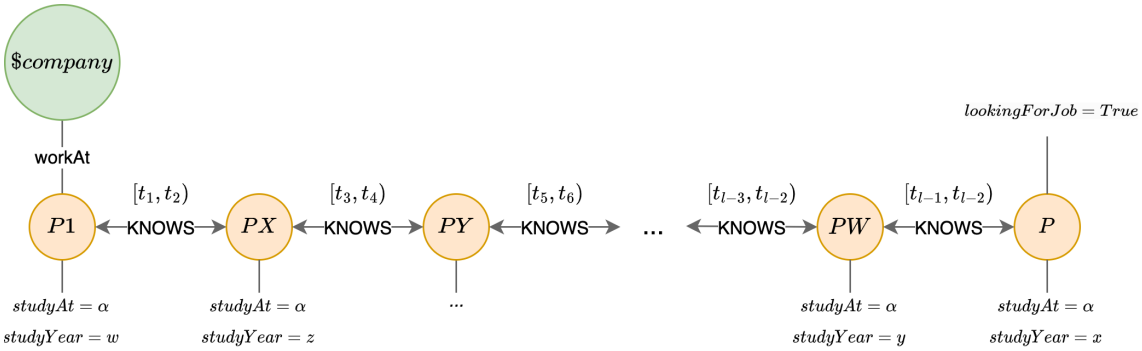


Figure 7.4: Shows the general structure of an recruitment recommendation path where the source Person P1 is working for a user-given company and the destination person P is looking for a job. Similar to diagram found in [15].

We will not go in depth of explaining the query arguments and analysis parameters as in the last example, as most of the explanation is similar and we follow the same structure. Figure

7.5 shows query arguments we give into our query model and the analysis parameters that can be varied for each use case of the query.

Parameter	Query Argument
Source Nodes	$lookingForJob = True$
Destination Nodes	$workAt = \$company$
Intermediate Nodes/Edges	$src.studyAt = dst.studyAt$
Temporal Path	$PariwiseContinuousPath$
Weight Function	$\|src.studyYear - dst.studyYear\| + 1$
Lengths	$minLength = 2$
Top-K	$\$numCandidates$

Analysis Parameters
$\$company$
$\$numCandidates$

Figure 7.5: Summarizes the analysis parameters and the query arguments to our query model for a recruitment recommendation path query

7.3 Transport Network Analysis

For transport network we are going to assume a generic transport network data model that can be applied to multiple types of transport and traffic data models and datasets.

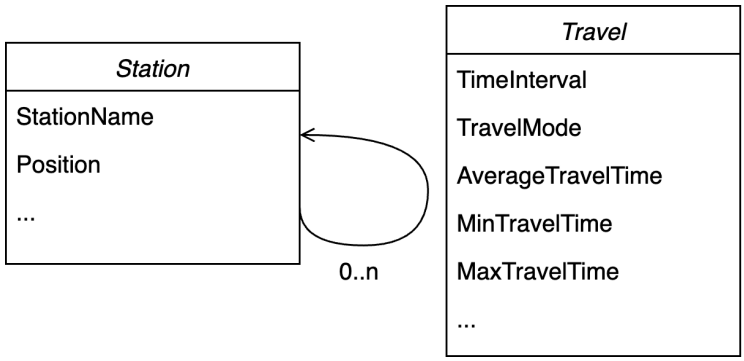


Figure 7.6: Shows a generic transport network data model with a single entity called Station and a relationship towards itself which represents the Travel between two stations in certain interval.

We will in particular use an aggregated NYC City Bike dataset that is based on data from bike riders that have measured their bike rides between different stations in New York over a 5 year period. We will preprocess this dataset such that we get the average, min, max and standard deviation durations between two stations split by certain time-windows (hours or half-hours) of the entire week. As the durations might change based on the time of the day and the day of the week because of traffic conditions, this would be a meaningful dataset to apply our query model to. We will also add some additional property data that can be used for querying the source and destination nodes, such as position coordinates.

We will define four interesting shortest duration paths (which can also be found in [13] [33] and [34]) that we will look further into. These paths are consecutive paths, meaning that consecutive edge interval relation is *before*. In addition, to further define the specific shortest duration paths we need to define three properties of the temporal path, namely: *duration*, *start* and *end*. To formalize the three shortest paths that we are interested in, we have the set of possible paths P, and the following shortest paths definitions:

- *Earliest-arrival path* (also called *Foremost Path*): $min\{end(P') : P' \in P\}$
- *Latest-depature path* (also called *Reverse-Foremost Path*): $max\{start(P') : P' \in P\}$
- *Fastest path*: $min\{dur(P') : P' \in P\}$

Here we will see how we can supply these definition as arguments to our query model, showing the possibilities of the model. The following list could be potential query arguments.

- *Earliest-arrival*: $min(currentWeight, edgeInterval.end)$
- *Latest-departure*: $max(currentWeight, edgeInterval.start)$
- *Fastest-path*: $currentWeight + edge.travelTime$

As we have mentioned in previous sections, prior work have only looked at computing *single-source* variants of these paths without any additional query capabilities such as source, length or interval constraints, weight mapping functions or top-k possibilities. Therefore, we will use our query model as we have done for the previous use-cases to define some additional parameters that are of particular interest.

In Figure 7.7 we can see an example of a full set of query arguments creating the query of finding the top 3 fastest paths from a user-defined coordinate (this could be their home or work), to a destination that is in walking distance to a coffee shop. Coffee shop is here a constant coordinate value, but could be a collection as well, but we do not focus on indexing the coordinates as that is entire work on is own. We further define walking distance as less than 1 km.

Parameter	Query Argument
Source Nodes	$distance(source.pos, \$sourcePos) <= 1km$
Destination Nodes	$distance(destination.pos, coffeeShop) <= 1km$
Temporal Path	$ConsecutivePath$
Weight Function	$averageTravelTime$
Lengths	$minLength = 1, maxLength = 10$
Top-K	3

Analysis Parameters
$\$sourcePos$

Figure 7.7: Shows the full set of arguments to our query model to create a fastest path query.

Thus, we have shown that doing analysis on transport network datasets is a real possibility with our query model. However, in our experiments we did not focus on implementing these queries because of time and resource limitations.

8 Algorithms and Data Structures

8.1 Main Ideas

Interval-Weight Centric Vertex

For computing the shortest paths in a distributed manner we will use the vertex-centric programming model, where we extend the vertex state to incorporate intervals, weights and lengths combined. We take inspiration from the interval-centric model where we split the message passing between the verticies into non-overlapping intervals. For each vertex we then need to keep track of the shortest path weight that end in this vertex, but we will have to do it for all the possible lengths that are under the maximum length of the path. In addition, we also need to specify which interval these values are valid for so that we satisfy the temporal path constraints. This is a general description of what we are trying to achieve, we are further going to explain how this structure will find the shortest path satisfying our constraints.

Multiple Phases

Since we have multiple parameters to our query model, the order in which we apply each parameter is essential for the correctness and performance of the executions. For instance we require to have all the weights mapped before we can proceed to calculate the shortest paths. And for performance we always want to filter the graph as much as possible as the first step, such that we can consider the smallest possible graph, increasing the performance of the proceeding phases. Therefore, we will structure our execution into multiple phases where each phase focuses on a particular task and gets the input from the previous phase and outputs their result to the next phase. The following phases that we will consider are therefore:

- Subgraph (filter) Phase
- Weight mapping Phase
- Pregel (shortest path computation) Phase
- Path Construction Phase

8.2 Phase Executions

Subgraph Phase

Before getting the the graph to the form where we can compute the shortest paths, we need to filter and preprocess the graphs such that we only consider those sources, destinations and intermediate vertices that satisfies the user-defined tests. We want to mark which vertices are sources, which vertices are destinations and which are intermediate vertices, this can be summarized in three different overlapping states.

- Source vertex test success
- Intermediate vertex test success
- Destination vertex test success

Thus after this phase every vertex in the graph will be marked with at least one of three states, and every vertex that does not get marked will be filtered away. This further means that a vertex can be a source and a intermediate vertex at the same time, also a intermediate vertex and destination vertex at the same time. In theory, we also allow vertices to be both a source and destination vertex at the same time, however this is an issue of path isomorphism which we will further discuss in a another section. If we get such a case in the first place, most of the time it would mean a semantically poor user-defined query or something unexpected with the query.

In Figure 8.1 we show an illustration of how we imagine the preprocessed graph look like after the subgraph phase. The green vertices are marked as the sources, the red vertices are marked as the destinations and the white vertices will be marked as having passed the intermediate vertices tests. The one vertex that is sort of orange have passed as both source and destination, showing that this case is indeed possible in our model. Furthermore, note that in the figure we have left out properties and intervals of the temporal graph, but the user-defined tests could be any function that works on both properties and intervals on the vertices and edges, for instance a specific time interval for the graph.

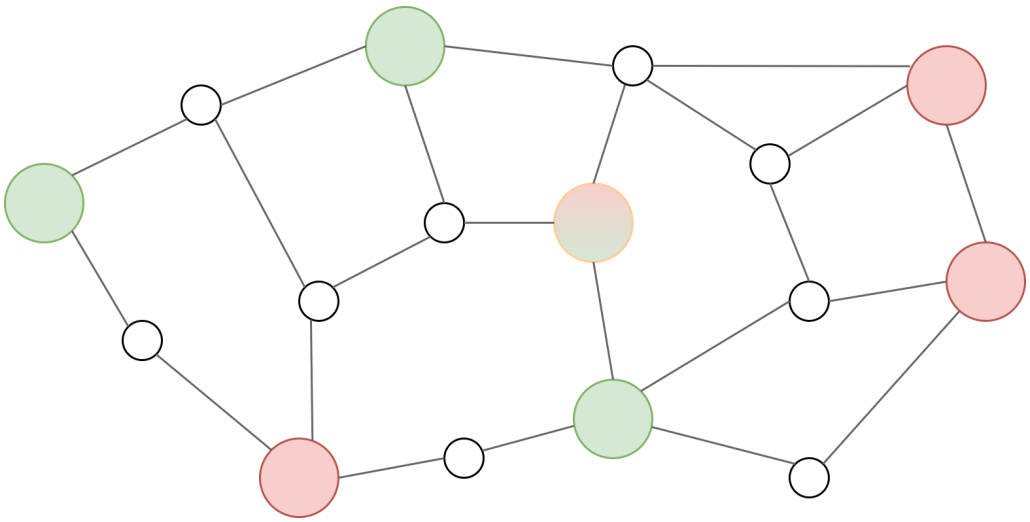


Figure 8.1: Shows an illustration of a subgraph where green vertices are sources and red vertices are destinations, while the white vertices are intermediate vertices.

As this phase is essentially a filter-stage, it will not account for that much of the computations compared to some of the following phases. In addition, filtering is a rather trivial problem to execute distributed as each worker-node can independently from each other apply the same filter condition without any need for communication between the nodes. Many of the underlying distributed frameworks have operators both for general purpose filtering and subgraph-operators working on vertices and edges of a graph. This means that this phase will not be that challenging to implement and we will consider this as a preprocessing step of the Pregel phase.

Weight Mapping Phase

As the next and second step of the graph preprocessing before preceeding to the pregel phase we execute the weight mapping phase. To further allow the user to define weighting functions of the vertices and edges as part of the path, we need to map the graph to a weighted graph where we apply these functions. Similarly to the subgraph phase, the weight-mapping phase is rather trivial to implement in a distributed manner as it is essentially just a map-operator. This can be done on each worker-node independently of each other similarly to filtering.

In Figure 8.2 we show the same graph as in Figure 8.1, but now as a weighted graph where we have added weights to the edges. Note that we still retain the intervals of vertices and edges as they are needed in the pregel phase when we are going to apply the interval-weight centric computations to create proper temporal paths.

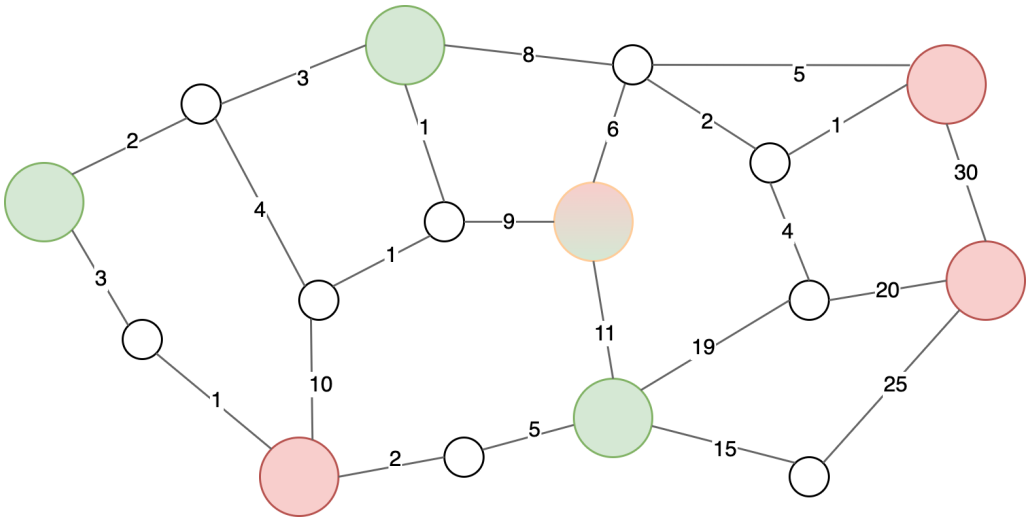


Figure 8.2: Shows an illustration of the weighted graph after the weight mapping phase.

Pregel Computation Phase

This is phase is the most computationally expensive phase where most of the processing will be happening, with a single superstep for incrementally longer lengths of the paths. The vertex-state here represents the weight of the shortest path ending in the vertex that is in focus, where the path ultimately started from a vertex classified as a source vertex. Here, we are going to explain the vertex-state data structure and message passing and how this relates to the time intervals.

First, assume that we have a graph that have gone through the subgraph and weight mapping phases, Figure 8.3 shows an example of such a graph. Further, assume that we have a query with the following given parameters:

- Length between 3 and 4
- A *Continuous temporal path*
- k = 2 for retrieving the 2 shortest paths

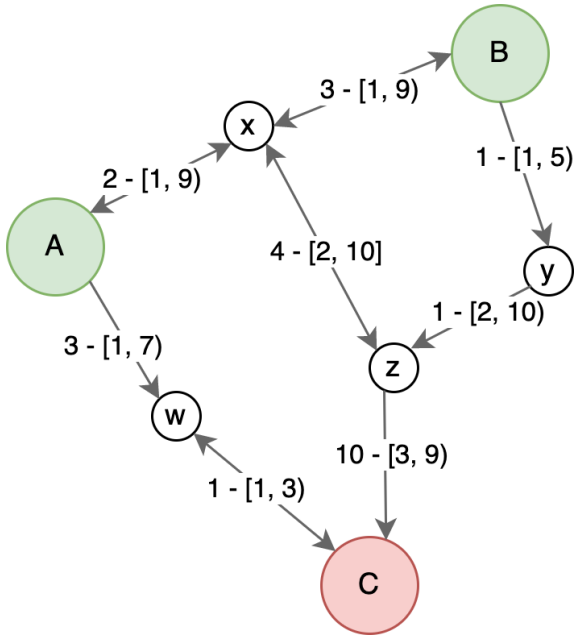


Figure 8.3: Shows a temporal graph that have gone through the subgraph and weight mapping phases.

Focusing on a single vertex for a single interval we need to keep track of the two shortest paths for all possible lengths, and also keep track of which of the neighbouring vertices was the parent source vertex for that path. We essentially need to have enough information so that we can reconstruct this path if we wished to backtrack from this vertex to the source vertex, which we ultimately want to do from all the destination vertices, however this will happen in the path construction phase. Therefore, we will create a table with the following properties that represents this information:

- Columns: weight (w), length (l), source (s)
- Each entry represents a path ending in the vertex in focus with the final length (l), weight (w) and originating from neighbour vertex (s).
- For each length we will only keep the top-k minimum weights, meaning that the table will have a maximum size of max_length * k
- The table should preferably be sorted according to length and weight so that it can be easily updated.

Figure 8.4 shows the general structure and outline of such a table.

l	w	s
1	2	a
1	4	b
...		
2	8	c
2	12	b
...		
max_len_1	39	f
...		
max_len_k	68	g

Figure 8.4: Shows the general structure and outline of the table that represents the state for each interval-vertex.

In addition, we need to remember that we want to retrieve the valid temporal paths that respects the consecutive interval relations, so we also need to keep track of which intervals this table or entries in the table are valid for. Here, we take inspiration from the inteval-centric model where we split the vertex into non-overlapping intervals that have different vertex-states, where each interval gets its own table of this sort.

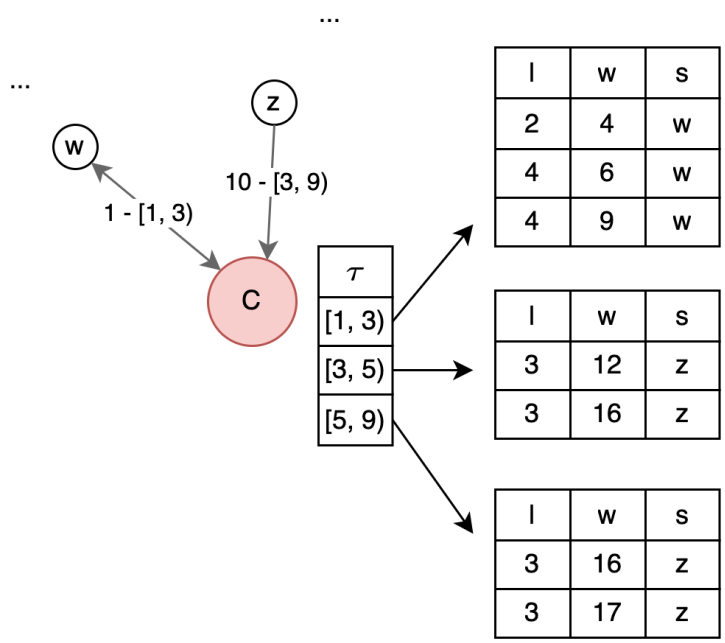


Figure 8.5: Shows an example of the resulting intervals and tables after 4 supersteps have computed for the destination vertex C.

But then is the question how do we go from the initialization phase of having an empty vertex-state, meaning an empty table, to constructing this sort of table. We will start with the initial messages as part of the first superstep, where only the vertices that are classified

as sources will send out messages to their neighbouring vertices. The logic for each vertex is then to update their table according to the messages they receive and then propagate these updates to their neighbours in the next superstep. Figure 8.6 shows pseudocode for this message passing logic from a vertex-centric perspective.

```
void init() {
    // Check if this vertex is a source vertex
    //  - Then send out the initial messages
}

void compute(MessageIterator messages) {
    // Take all the incoming messages and split them according to intervals
    // For each interval do the following:
    //  - Find the top k minimum messages for the current l (superstep)
    //  - Add these to the state table
}

void sendMessages() {
    // Select the latest entries of the table where l == superstep
    // From these entries we create messages where we do the following:
    //  - Increment the length with 1
    //  - Increase the weight with the outgoing edge weight
    //  - Set the source to our vertex id
    //  - Update the interval by applying the consecutive interval relation operator
    // Send these messages along the edges that satisfies the intervals
}
```

Figure 8.6: Shows pseudocode for the init, compute and message passing logic from a vertex-centric perspective

Note that the current superstep (iteration) corresponds to the current length from any of the source-nodes we are in the path. Also note that the same vertex can be repeated in the path, as we do not account for node isomorphism.

Path Construction Phase

Now that we have computed the state tables we want to construct the shortest weighted temporal paths according to our query, such that the results can be retrieved and viewed on the driver node. The main idea of the path construction is to backtrack from the destination nodes based on the state tables. This can be done recursively until we reach a source vertex and stand with a path satisfying our requirements. The first step of this process is to collect all the destination vertices and then find the table-entries that satisfy our length requirements and collect the k minimum weights entries. Then we will recursively collect the neighbor vertex that has been marked as parent-vertex in the table until we stand with paths of at least minimal lengths. These paths will be the k shortest paths because we backtracked from the k minimal entries from all the destination vertices. This is general outline of the main idea, Figure 8.8 shows the a more detailed description of this process in pseudocode.

First we will introduce a simple table data structure that in practice will help us execute this process, which we will call PathTable. The PathTable contains some additional information from the length-weight tables we saw in the previous section, to make it easier to construct the paths in a backtracking manner. During the path construction phase we will have a single PathTable with k rows, representing the k-best paths. We will incrementally expand the path based on the information we have in the columns, which should be sufficient to collect the appropriate table-entries from the correct neighbouring vertex. Figure 8.7 shows an example of such a table.

Interval	Length	Weight	Parent-Vertex	Path
[1, 20)	6	28.9	d	c -> b -> a
[1, 10)	6	31.2	g	h -> i -> j
[1, 20)	6	40.9	l	m -> n -> o
[5, 10)	7	102.3	q	r -> s -> t
[5, 20)	7	150.2	w	x -> y -> z

Figure 8.7: Shows an example of a PathTable where k=5 and paths of length 2 have already been constructed.

The PathTable has five columns: Interval, Length, Weight, Parent-Vertex and Path. Interval represents the interval of the latest edge we backtracked to in the path, such that we can use it to backtrack to the correct interval in the proceeding step. Length represents the remaining length of the unfinished constructed path, which should be 0 at end of construction. Weight is weight of finished path used for sorting and keeping same sorted order in PathTable, but remains unchanged during path construction. Parent-Vertex is next vertex we want to backtrack to in proceeding step. Path is the current constructed path so far in the construction phase, and will be completed at end of the phase.

As a side note, remember that k is a rather small number as it a number that is chosen such that the results can be viewed in the most user-friendly and informative way. Therefore, we consider this value to be more or less a constant in terms of algorithmic complexity.

The following pseudocode shows a bit more detailed description of the path construction phase we have introduced so far by utilizing the PathTable data structure.

```
PathTable constructPaths(PregelResultGraph graph, Query query) {
    // From the graph collect all vertices classified as destination-vertex
    //  - Accompanied with their intervals-state-tables
    // Merge all these interval-state-tables together
    //  - By mapping each state-table to a PathTable with the createTable-method
    //  - While only keeping the minimal k entries
    //  - At the end we end up with a single PathTable
    // While there exists entires in the table that have length greater than 0
    //  - Use the extendPaths-method to extend path entires in the table
    // Return the resulting table
}
```

Figure 8.8: Shows a more detailed pseudocode of the path construction phase utilizing the PathTable data structure

The main procedure is the *constructPaths*-procedure which takes in a temporal graph from the previous phase and outputs the PathTable where all the entries have completed constructing the paths according to the input query. The input graph is the temporal graph that went through the pregel computation phase where each vertex have an accompanied intervals-state-table, which we saw in Figure 8.4 and 8.3. In addition, we also take in the original query because we require some information such as k, minLength, maxLength and temporalPath. As mentioned the output will be in the form of a PathTable where each entry have completed constructing the path, meaning that all entries have 0 in the length-column and have a completed path in the Path-column.

We additionally have two helper procedures: *createTable* and *extendPaths*. Figure 8.9 shows more detailed pseudocode of these two procedures. The *createTable* procedure takes in a single intervals-state-table coming from a vertex in the pregel-input-graph and creates a new PathTable based on the table data and the query parameters. In the first step of *constructPaths* after collecting all the destination vertices, we will create a single PathTable by using *createTable* and merging them together such that we only keep the top-k entries sorted by minimal weight. After this step have completed in *constructPaths*, the paths will be further extended by using *extendPaths*. The *extendPaths* procedure takes in a PathTable and expands this table by collecting the appropriate intervals-state-tables from the graph based on the currently created path, interval and length data for each entry. These interval-state-tables will be used to extend the path with one edge, and a new PathTable with updated entries for all the columns will be returned. In *constructPaths*, the *extendPaths* procedure will be repeatedly run until all the entries have completed constructing their paths, which will happen when all the length-values in the PathTable is 0.

```
PathTable createTable(IntervalStates intervalStates, Query query) {
    // Create a new empty PathTable
    // Take all tables from intervalStates merged together such that:
    //  - We filter on minLength and maxLength from query
    //  - We stand with the top-k minimal weight entries where k is from query
    //  - We add an interval column where we project down the interval
    //  - We add a new column to the table that will represent the full path
    //      - Initially as a single edge(parent-vertex -> state-table.vertex)
    // Take all the entries in the merged table and insert them in our new PathTable
    // Return this table
}

PathTable extendPaths(PathTable table) {
    // Group the entries in the path-table by key: (parent-vertex, path, interval, length)
    //
    //  - For each group collect the interval-state-tables from the graph:
    //      - By finding the vertex that have an edge that satisfies both:
    //          - source_id == parent-vertex
    //          - destination_id == path.first_id
    //      - Take only tables that satisfies interval-relation with interval and edge.inte
    //      - Take only entries that satisfies length - 1
    //      - Take the top-m minimal entries by weight where m is rows in the group
    //      - We end up with two filtered tables (with m as row counts)
    //          - The grouped path-table entries
    //          - The filtered state-table entries
    //
    //  - Take the grouped entries and the filtered state-table for each group
    //      - Both sorted by weight and of same length m
    //      - For the same rows in both tables
    //          - Create a new edge based on parent-vertex in both tables
    //          - Enqueue this edge to the path for this entry
    //      - Return the updated entires for this group
    //
    // Flatten entries for all groups into a single updated PathTable
    // Return this table
}
```

Figure 8.9: Shows more detail pseudocode of the main helper procedures needed for the path construction.

Note that when we use the word *collect* we mean retrieving all the vertices from all the worker-nodes/executors in the cluster to the single driver node in the cluster. The driver node is the machine/node where results are collected and viewed from the user and the programmers perspective. This further means that this operation could be considered a potential bottleneck as it essentially is a synchronization of all the worker-nodes where we require message passing and consequently waiting for these messages on the driver. As we are doing this for each edge in the path, total number of times equals to the length of longest path, the path construction phase could end up being an expensive phase if the message passing is slow, but this still remains to be seen in practice.

8.3 Algorithms Analysis

In this section we will discuss and illustrate ideas of how the sequence of phases and algorithms lead to the correct result as well as what types of runtime and memory characteristics we would expect. These are not rigorous proofs, but based on our initial intuitive expectations. Further real world testing would be done in the implementation and experimentation in further sections.

Correctness

The main idea behind the correctness of the algorithm, is based on the invariant which we have touched upon in the previous sections. That is, that the entry in the interval-tables represent one path that ends in the vertex that is in focus, where it is valid under the specific interval. As long as we hold this invariant true under the whole execution, we should end up with states on the destination-vertices that hold the top-k paths ending in that vertex. The path construction-phase should be able to backtrack from the destination-vertices to get to the sources for all possible paths. Then, we simply take all the top-k best paths as the result.

In the Implementation section we have also outlined a reference algorithms based on Breadth-First-Search that is used as a validation test. As our reference algorithm gives the same expected results it makes us additionally confident that our algorithm is more or less correct. Either way, small bugs in the implementation does not stop us from answering our question whether the prototype is able to scale and that the query model is feasible in practice.

Runtime and Memory

To intuitively analyse the runtime and memory growth of the phases and algorithms, we will base ourselves on the interval-state data structure in the pregel phase. As runtime is entirely dependent on how many table entries we generate, it is thus the same as the memory size growth, and we assume the same growth for both runtime and memory size. Firstly, if we look at the case without intervals, a single table will maximum take $max\_length \cdot top\_k$ size for each vertex in the graph. This means that in the worst case will get $O(|V| \cdot max\_length \cdot top\_k)$ memory size without intervals. Intuitively, if we add the intervals we would multiply the number of intervals: $O(|V| \cdot num\_intervals \cdot max\_length \cdot top\_k)$.

However, depending on the *nextInterval*-function we defined in Section 7, we might get a unique interval for each separate path. We have to remember that one single entry in a table represent a path of a particular length. Therefore, the worst-case performance is all possible paths that is possible to create, that is the average vertex degree multiplied for each increment of length, meaning $avg\_deg^{max\_length} = (2 \cdot |E|/|V|)^{max\_length}$, where $|E| = (|V|^2)$ is for dense graphs and $|E| = O(|V|)$ for sparse graphs. The resulting worst case memory size growth is exponential: $O(|V|^{max\_length})$ for dense graphs and $O(2^{max\_length})$ for sparse graphs. However, we have to remember this is entirely dependent on the *nextInterval*-function. In the case where we have limited amount of intervals, for instance an interval for each edge, which is the case for pairwise continuous and consecutive paths, we would get $O(|V| \cdot avg\_deg \cdot max\_length \cdot top\_k)$ worst case performance.

8.4 Implementation

In this section we will go over the technical implementation of our prototype, which includes the implementation of our query model from Section 7 and the algorithms and data structures we have discussed in Section 8.

Implementation Framework: Spark and GraphX

In Section 6.3 we outlined some specifications for the implementation of our prototype. Summarized, this was choosing an underlying graph processing framework such that we could utilize and extend the Pregel-API to implement our interval centric ideas. In addition, this framework should facilitate the creation of a temporal graph representation that enables distributed processing. Here, we will explain and discuss the underlying framework, GraphX, that we have used to implement our prototype. In addition, we will give arguments for why we have chosen it as the framework instead of other alternatives.

GraphX is one of the main components of the Spark framework and extends the RDD API we initially introduced in Section 2.1 with abstractions for representing graph-oriented data. Special purpose graph operators and transformations are provided by GraphX, in addition to the standard operations that are available on RDDs. In addition, there are provided common graph algorithms and graph builders which makes it easier to test and develop applications with the framework. However, first and foremost it includes a variant of the Pregel API, which we can utilize to implement our algorithms.

One of the main reasons for choosing Spark and GraphX over similar graph processing systems is that it provides better tools and utilities for simplifying the design and implementation of complex graph algorithms and pipelines. As Spark is one of the most widely used open-source big data processing framework, it is well documented and integrated with other tools and utilities. For instance HDFS (Hadoop Distributed File System) is a widely used distributed file system for storing large files, and something that we have been needed to use when loading large datasets for our larger scale experiments. There are multiple other examples such as interactive development and visualizations through notebooks, integration with programming languages, customized deployment etc. In addition, Spark and GraphX is open-source with well maintained documentation and guides to rapidly get started with the development. As we are limited with time and resources, and our main goal is to create a functioning prototype, the ease of development have been a major factor when choosing implementation framework.

When it comes to performance, GraphX is known to generally perform well, but slightly worse than other competing open-source graph processing frameworks. However, it still provides the characteristics of a distributed data processing framework that we require to produce the necessary results to answer our research questions. As this is our goal, the constant factor performance differences are not essentially important.

Phases and Pseudocode Implementation

In Section 8.1 we outlined the main ideas behind the algorithms and data structures necessary for implementing our query model. This included the multiple phases of subgraph, weight-map, pregel, and path construction. Here we will present the operators and transformations from GraphX that we have used to implement this in a scalable way. As well

as overall description of the way we implemented the pseudocode we have presented in Section 8. We will also refer to the implementation details in our code repository, which can be found in https://gitlab.stud.idi.ntnu.no/lukasnt/it3920-lukasnt (however this requires NTNU access to view).

Starting with the subgraph phase, which essentially is just a filter phase, and as the name suggest utilizing the already existing *subgraph*-operator in GraphX would be a natural choice. The *subgraph*-operator takes in a vertex-predicate and an edge-predicate, where we have passed in the user-defined predicates as part of the parameters in the query model. In addition to filtering the graph we want to mark the nodes that pass the test of being a source, destination and/or intermediate such as we illustrated in earlier in Figure 8.1. To be able to mark each node in the graph with any of these labels we have to map all the vertices. Here we have used the *mapVertices*-operator in GraphX, which maps every vertex given the mapping function. We have simply passed in the user-defined predicates as part of the query parameters to this map function. The source code implementation can be viewed in the repository in file */executors/ParameterSubgraph.scala*

Then we have the weight-mapping phase, which will work in a similar way as for the subgraph-phase, where we utilize an already existing operator and pass in the user-defined mapping-function. As you might guess this operator would be a *map*-operator, but specifically the *mapEdges*-operator as we are only interesting in mapping the edges for getting a weighted temporal graph, as we saw in Figure 8.2. The source code implementation can be viewed in the repository in file */executors/ParameterWeightMap.scala*

Figure 8.10: Shows the dataflow of operators used in preprocessing the graph before the pregel-phase.

Then we have the pregel-phase, where we utilize the pregel-API provided by GraphX. Here we have implemented the provided *initMessage*, *vertexProgram*, *sendMessage* and *mergeMessages* according to our pseudocode in Figure 8.6. Further implementation details can be seen in the code repository at file */executors/ParameterPregel.scala*. We have also implemented the interval states consisting of length-weight tables which we saw in Figure 8.4 and 8.5. These data structures are implemented as JVM objects using the Scala standard language library, which includes *Map*, *ListBuffer*, *Interval* among other data structure and utilities classes. These also follow the pseudocode that we have described, and further implementation details can be seen in the repository under the folder */utils*.

Then we lastly have the path-construction phase implementation, which was the most involved and complicated to get to work correctly. Here we have mostly followed our pseudocode that we outlined in previous section, and further implementation details can be seen in the repository at file */executors/ParameterPathsConstruction.scala*

Reference Algorithm Implementation

Additionally, we have made a sequential and rather brute-force reference algorithm meant to return the same results as the main implementation. This was used for testing and further validation of the expected result from the main implementation. It was also used

as a rough reference benchmark when measuring performance on both smaller and larger datasets, where we expected similar performance on smaller ones, but where our main implementation would outscale the reference for larger datasets. Here, we will give a small pseudocode and implementation description of this reference algorithm.

The reference algorithm we have used was a simple Breadth-First-Search (BFS) algorithm that brute-force searches a path from the source-nodes to the destination-nodes, given that the interval and length conditions are met. A rough pseudocode of this algorithm can be seen in Figure 8.11. In the implementations there are more details and some extra optimizations done. The implementation can be found in the code repository at file */executors/SerialQueryExecutor.scala*.

```
PathTable execute(Query query) {
    // Collect source and destination vertices from the temporal graph
    // For each source vertex, runBFS with the destination vertices
    // and take the top-k paths sorted by weight-score
}

PathQueue initQueue(Query query, VertexID srcId) {
    // Initialize empty queue
    // for each out-edge for this srcId enqueue the edge as a path of length 1
    //  - Using the query weightMap-function as initial weight
}

PathTable runBFS(Query query, VertexID srcId, VertexID[] destinationIds) {
    // Initialize queue for this srcId
    // Initialize empty resultList
    // While queue is not empty
    //  - Dequeue the queue to get the next path
    //  - if the following conditions hold:
    //      - end-node of path is in the destinationIds
    //      - the path satifies query length constraints
    //          - Add this path to the resultList
    //  - Extend the current path by all of the out-going edges for the end-node
    //      - Making sure the edges satisfies the interval-relation
    //      - The new path will have the updated weight according to the weight-function
    //  - Add all the resulting extended paths to the queue
}
```

Figure 8.11: Shows pseudocode of the BFS-based algorithm used as a reference for validation and performance testing.

Code and Repository Outline

As we have pointed towards already we have a code repository containing the code for our prototype implementation. Here we will go over the overall structure and outline of the part of the code repository that is associated with the implementations.

- *./*: The root folder consists of several configuration files that are used for development and setups, which includes *README.md*, *.env*, *Dockerfile* etc. There is also several

default folders that are empty, where files and data that will be generated during development and execution. This includes *logs*, *hdfs* and *target*.

- *zeppelin-notebooks/*: This is a sub repository where all the notebooks, which consists of some test runs and visualizations. In the next sections we will provide some examples of this.. This notebook can be run and viewed by following the *README.md*
- *src/*: Consits of the source code of the core implementation, following the pseudocode and algorithms we have explained so far.
  - *main/scala/*
    - *executors/*: Consists of implementation of all the phases and the classes that execute the queries as a whole and return results. Here, we also have interface for each phase such that multiple implementations of a phase could potentially be created.
    - *experiments/*: Consists of the code related to the execution of the experiments, this includes the classes and command line interfaces for varying the parameters, datasets and configurations of the experiments.
    - *io/*: Consists of implementation of classes for loading and writing datasets and results to files and other destinations. This includes code for formatting and preprocessing of the datasets to get it to the desired format.
    - *models/*: Consists of logical data model classes that work as an interface to the data and query models. This inlcudes temporal graph, temporal interval and temporal graph query and result classes.
    - *util/*: Consists of data structures and other utilities classes used for the underlying implementation. Here we find query-state, interval-state, length-weight-table, path-table etc. that we have touched upon in Section 8.
    - *visualizers/*: This is code related to visualizing the temporal graph and resulting path. The most notable piece of code here is the HTMLGenerator-class which generates an HTML representation accompanied with JavaScript that utilizes the d3.js library for creating dynamic graph visualizations and animations.
  - *main/resources/*: Consists of some small validation and testing datasets that can be used by default when no other data sources are specified, these resources will come with the resulting Jar executable when the source code is compiled and built.
  - *test/*: Consists of unit tests for some of the implementation classes, particularly the util classes and model classes.

Visualizing Temporal Graphs and Results

The implementation also includes ways to visualize temporal graphs and the resulting paths from the query result.

The following example shows a graph of person nodes and their relationships to other person nodes, where an edge is a Knows-relationship. Edge directions are not included in the visualization.

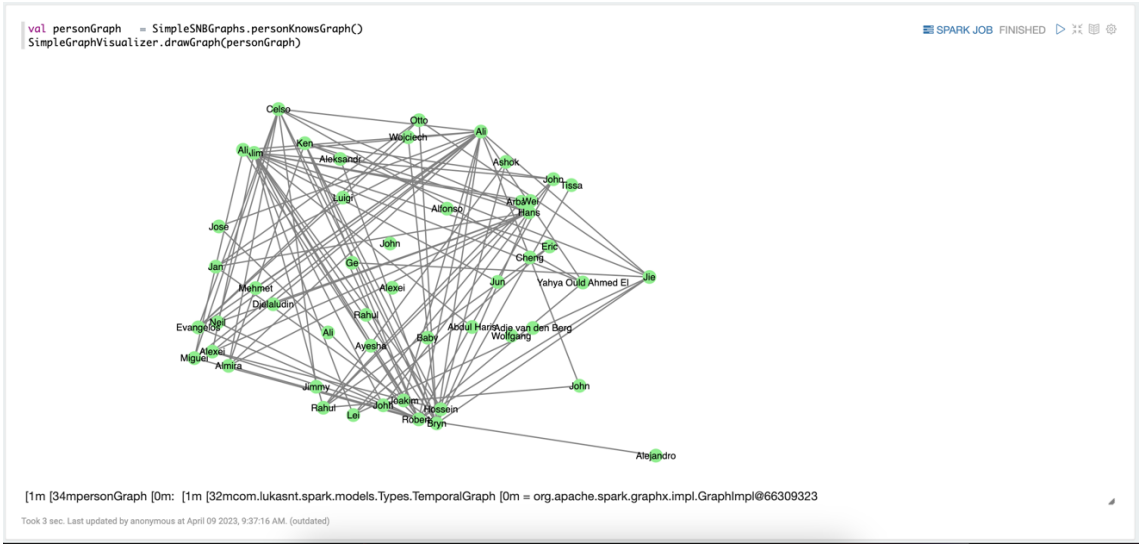


Figure 8.12: Shows a visualization of the smallest possible graph generated by the synthetic graph generator we are using.

The following example shows a query that finds the top-5 shortest continuous paths originating from a male ending in a female, where the path length is at least 3 and at most 5. The weight is defined as the duration of the interval of the edge, meaning how long the persons have known each other. Therefore, the these paths are persons who have known each other the least amount of time.

Figure 8.13: Shows an example query using our created interface.

This shows the result of the query as a table. The columns are the weight of the path, the source-id of the path, the destination-id of the path, the start-time of the continuous interval, the end-time of the continuous interval and the path as sequence of ids.

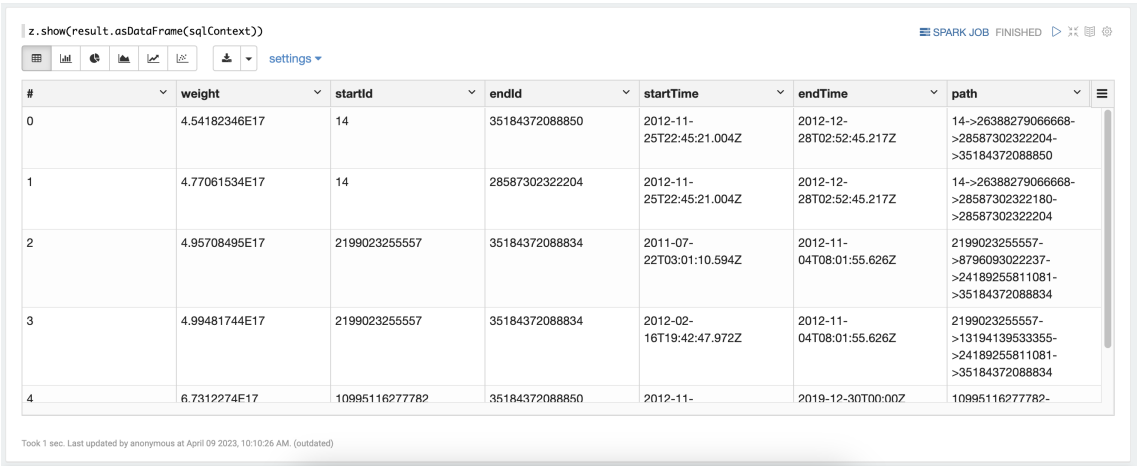


Figure 8.14: Shows the query result of the query in table-format.

Given the person graph from above, this shows the result of the query as a graph visualization, where the nodes are the persons with their names and the edges are the Knows-relationships.

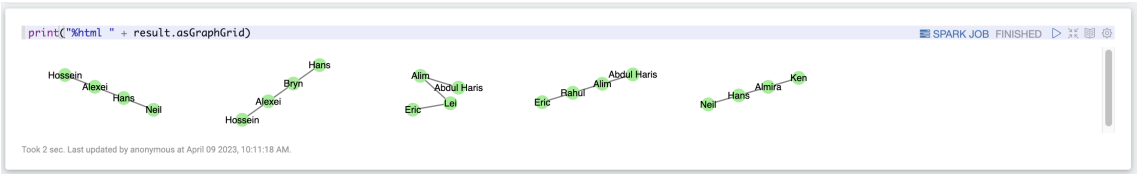


Figure 8.15: Shows the query result of the query visualized as paths.

9 Experiments

9.1 Methodology

Here we will go over our experimental methodology and further details of our experiments and how we have received our results.

Dataset Characteristics

As we have mentioned in Section 7.2 we have used the LDBC Social Network Benchmark [15] and the provided dataset generator to generate our experimental datasets. Here we will summarize some of the different characteristics and parameters of the datasets that we have used in our experiments.

The graph generated by the generator follows general characteristics and distributions of social networks. This includes for instance degree distribution similar to found in Facebook, high presence of triangles in the network and similar persons in the network have high probability of being connected. The graph generated is a temporal property graph, meaning that there are multiple labels other than just persons and friendships. For instance messages, comments, forums, locations, cities etc. For our experiments, we have first and foremost used the Person-entities as vertices and the Knows-relationships as edges, and the necessary extra attributes we have preprocessed to be projected down to these labels. More details of distributions, parameters, quirks and other considerations can found further in the original paper [15].

The benchmark comes with three types of formats and modes depending on the targeted workload. The three main different modes are Interactive, BI (Business Intelligence) and Raw. They differ in how much of the dataset is divided into initial dataset in one part and inserts and update streams as the other part. For instance interactive mode, which is targeted towards transactional database workloads, have smaller initial dataset, but more inserts and updates. The BI mode have large initial dataset and a few large bulk inserts. The raw mode contains the full dataset from the start, as a proper temporal graph, without any insert or update streams. For us, we are first and foremost interested in generating temporal graph data, where we have time intervals for the lifespan of the entities and relationships. The only one of the modes that provide this without any preprocessing is the raw mode. In addition, as our workload is targeted towards more complex long running queries, having the full data generated is more suited for our experiments. That is why we have generated our experiment datasets using the provided raw mode.

The generator can generate datasets of different sizes, which they call scale factors. They provide scale factors of 0.003, 0.3, 0.1, 1, 3, 10 up to a maximum of 10 000, but we have only used up to 3. A scale factor of 1 approximately account for a gigabyte (GB) of domain data, but the actual raw file sizes are however quite larger, at least 2 to 3 times. The scale factor affects the number of vertices and the number of edges that are in the generated graph. For each vertex there are 13 properties and for each edge there are 6 properties, meaning the total unrolled properties is quite larger than just number of vertices and edges. The following table summarizes all these characteristics for the different sizes.

Scale Factor	$\|V\|$ **- Person**	$\|E\|$ **- Knows**	**Total Unrolled Properties**
0.003	50	88	650 + 528 = 1178
0.1	1 700	18 135	22 100 + 108 810 = 130 910
0.3	3 900	57 095	50 700 + 342 570 = 393 270
1	10 620	219 450	138 060 + 1 316 700 = 1 454 760
3	25 870	668 431	336 310 + 4 010 586 = 4 346 896

Figure 9.1: Shows the graph characteristics of the graph generated with the different scale factors.

We wished to use dataset sizes up to at least 10, however we got some issues of transferring and preprosessing the large amount of large files over to the cluster where we ran our experiments. However, we feel that the dataset sizes we got was sufficient enough to show the general scalability of our implementation and still gave us valuable results.

Baseline Query and Parameters

Here we outline the type of query that we have used and the parameters that we have used as the baseline for the experiments. We have focused on interaction paths as the experiment query as it generally gives the most balanced and well rounded workload to demonstrate the performance characteristics of our prototype and the influence of the parameters in our query model. For instance we can make sensical queries of a certain length as well as having a decent amount of source- and destination vertices. Also varying the topK parameter returns enough results to show the performance impact and generally make sense for the query. Both continuous and pairwise continuous paths are natural temporal path queries to make, and also shows very different performance characteristics.

In Section 7.2.1 we explained the parameters to pass in to our query model to make interaction path queries, and for the experiments we have used these values as default values in the experiments. Additionally, we have set some parameters to baseline values, and for each experiment we have varied one parameter at a time deviating from this baseline. The parameters that we vary one at a time are length, topK, dataset size, worker counts and partition strategy provided by GraphX, as all of these might have impact on the performance and scalability. In Figure 9.2 we have summarized these baseline parameter values. These values are based on the execution time of initial test runs made on the experimental setup, where we did not want the experiments to take too long for practical reasons.

Parameter	**Baseline value**
Length (Min to Max)	3 to 4
Top K	10
Worker Count	4
Temporal Path Type	Pairwise Continuous or Continuous
Partition Strategy	Random Vertex Cut
Graph Size	Scale Factor 1

Figure 9.2: Shows the baseline parameter values for the experiments. Each experiment deviates one value at a time from this baseline where each experiment focuses on varying a single variable.

When it comes to preprocessing the dataset for the experiments, we need to get the data to a temporal graph format which has the necessary attributes to make interaction path queries. As the *nr_interactions* attribute is not present by default on the edges in the temporal graph generated, we have preprocessed the raw temporal graph to incorporate *nr_interactions* as an attribute on the Knows-edges. We have done this by following the query specification for interaction paths in [15]. The *nr_interactions* are defined as the number of comments to posts or other comments that two person have between each other. The details of the preprocessing can further be viewed in the code repository, specifically under *src/main/io/IntercationLoader.scala*.

Experimental Setup

The experiments have been run on a cluster with 24 worker nodes. Each worker consists of a processors with 24 CPU cores, 128 GB of RAM, 4 to 8 TB of disk storage that runs on Ubuntu 18.04. The cluster is run and managed by Cloudera CDH 6.3, which as a an open source platform distribution of Apache Hadoop and other Big Data workflow technologies. The prototype is written in Scala 2.11 and is made for Spark 2.4.x with the GraphX version that comes with it, where the specific version of Spark for the experiments is 2.4.0+cdh6.3.2. More details about all versions of the packages used can be found in the the code repository, specifically in the *./pom.xml* file.

The prototype have been set up by compiling the source code and packaging the result in a .jar-file with Maven. This jar have then been run through the cluster with specific arguments on our created command line interface. In addition, the jar contains all the necessary dependencies such as GraphX and Spark Core inside the jar itself, sometimes referred to as a "fat" jar, which is a recommended and common way of running spark programs through the spark-submit command. The command line interface we have created lets us specify the query, as well as multiple variables for each specific parameter, where the query is run for all variable combinations specified, where we also can specify the number of runs for each combination. We have provided a set of arguments to the jar for each experiment, meaning for each single parameter we are varying one at a time, which is also equivalent to one graph in the Result (9.2) Section. When it comes to the datasets, they have been stored on the HDFS provided by the cluster, and is loaded in before each experiment run, which is not counted towards the execution time.

Note that not all of the computing and storage resources have been utilized to the fullest in the actual experiments. In addition, as the cluster is used by multiple people, potentially at the same time, and in general have varying amount of load at different times of day, it has had effect on the experimental results. We have tried our best to alleviate this by running each experiment with a sample size of at least 10 and running the variables in random order for each run. Some of the experiments that we wished to be run together in one program execution, specifically the worker count experiments, had to be run separately for each worker count variable because of some practical limitations with the cluster and spark arguments. This resulted in them being run at different times of the day, with different existing load on the cluster, meaning we got significant different and unexpected results for those experiments specifically.

9.2 Results

9.2.1 Measurements

For the experiments we have measured both the runtime and the maximum memory usage during the run for each sample. In addition, we have divided both the runtime and memory usage in subcategories so that it summed together account for the total. For runtime we have divided it into the phases we have outlined in the previous sections. This is subgraph, pregel and path construction phase runtimes. We have omitted the weight-map phase as it accounted for too small amount of the total runtime. We believe this is because the phase is actually executed along with the subgraph phase, as we know that Spark creates an internal operator DAG, meaning that it is possible that an internal map-operator does the mapping for the subgraph and weight-map in one operator. In addition, Spark is an lazy evaluated system, meaning that it only computes what it needs when it needs it. For the memory we have divided it into driver memory size, meaning the memory that is used at the master node, and the RDD memory size, which is the total size of the data that is distributed among the worker nodes. We also use the maximum memory usage that was measured for the run sample. These measurements was done from the prototype executable during the experiment runs. The raw and formatted results can also be found in the code repository under the folder *./results*

9.2.2 Interaction Paths: Pairwise Continuous

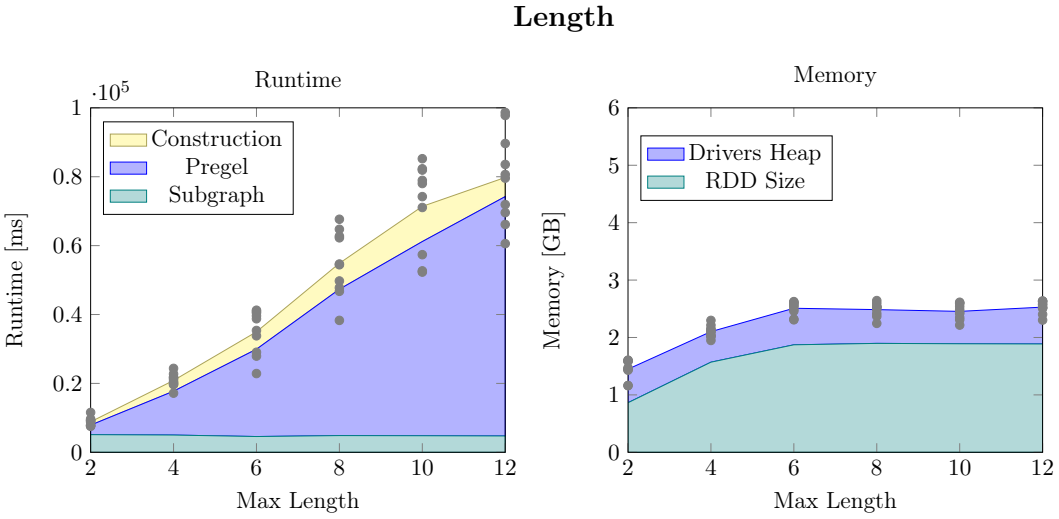


Figure 9.3: Shows the runtime and maximum memory consumption for increasing max-lengths. The smallest max-length of 2 gives an average runtime of 8.86s, where the subgraph, pregel and construction phases account for 57.8%, 31.4% and 10.8% respectively. The corresponding average maximum memory consumption is 1448MB, 585MB for drivers heap and 863MB for RDD size. The largest max-length of 12 gives an average runtime of 79.8s, where the corresponding phase distribution is 5.9%, 87.1% and 7.0% respectively. Corresponding average maximum memory consumption is 2529MB, 642MB for drivers heap and 1887MB for RDD size. For runtime we generally see a linear increasing trend and for memory consumption we see an initial increase that quickly flats out.

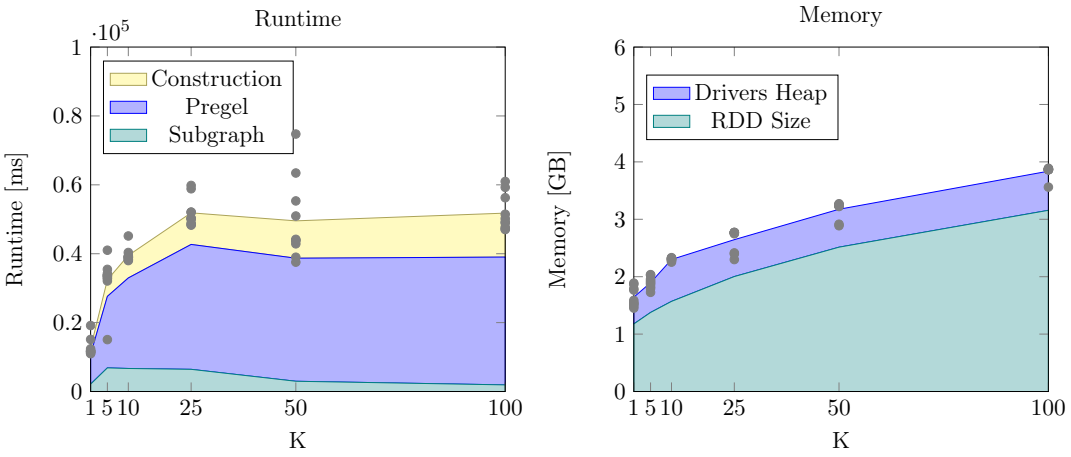


Figure 9.4: Shows the runtime and maximum memory consumption for increasing top-ks. The smallest top-k of 1 gives an average runtime of 12.7s, where the subgraph, pregel and construction phases account for 16.8%, 68.1% and 15.1% respectively. The corresponding average maximum memory consumption is 1644MB, 467MB for drivers heap and 1177MB for RDD size. The largest top-k of 100 gives an average runtime of 51.8s, where the corresponding phase distribution is 3.7%, 71.7% and 24.6% respectively. Corresponding average maximum memory consumption is 3838MB, 682MB for drivers heap and 3156MB for RDD size. For runtime we generally see an initial steep increase that quickly flats out and for memory consumption a slow linear increasing trend.

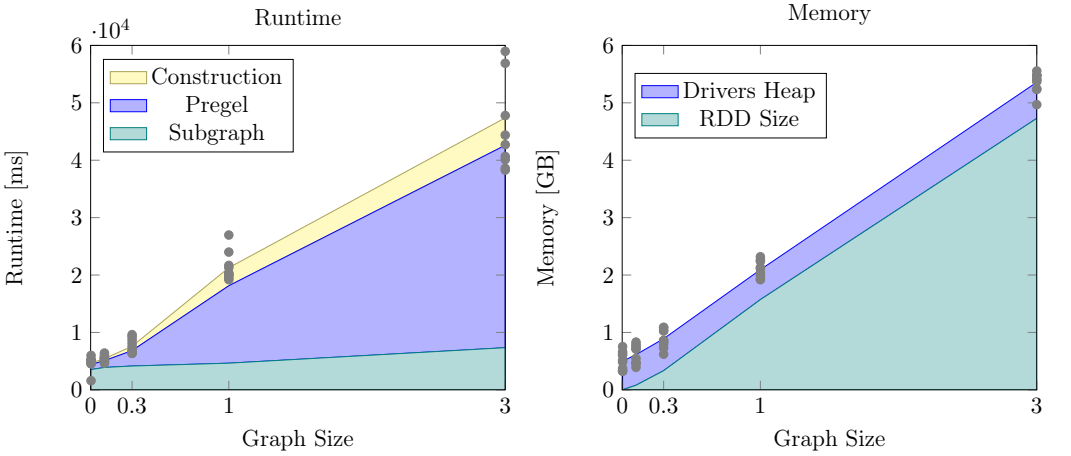


Figure 9.5: Shows the runtime and maximum memory consumption for increasing graph sizes. The second smallest graph size with scale factor of 0.1 gives an average runtime of 5.44s, where the subgraph, pregel and construction phases account for 71.6%, 21.8% and 6.6% respectively. The corresponding average maximum memory consumption is 613MB, 534MB for drivers heap and 79MB for RDD size. The largest scale factor of 3 gives an average runtime of 47.4s, where the corresponding phase distribution is 15.5%, 74.4% and 10.1% respectively. Corresponding average maximum memory consumption is 5352MB, 629MB for drivers heap and 4723MB for RDD size. For both runtime and memory consumption we generally see a very consistent linear increasing trend.

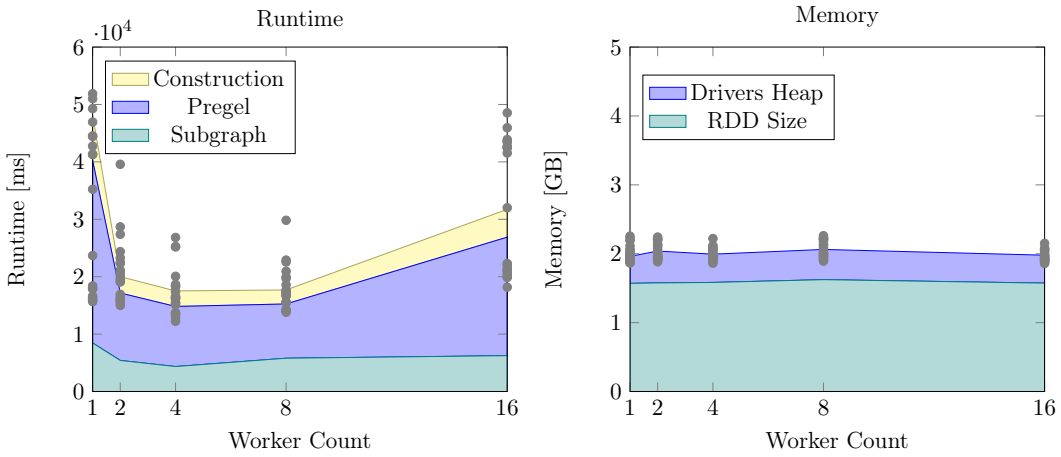


Figure 9.6: Shows the runtime and maximum memory consumption for increasing the number of work counts. Worker count of 1 gives an average runtime of 48.1s, where the subgraph, pregel and construction phases account for 17.6%, 66.1% and 16.3% respectively. The corresponding average maximum memory consumption is 1961MB, 392MB for drivers heap and 1569MB for RDD size. Worker count of 8 gives an average runtime of 17.7s, where the corresponding phase distribution is 32.8%, 53.4% and 13.8% respectively. Corresponding average maximum memory consumption is 2062MB, 439MB for drivers heap and 1623MB for RDD size. For runtime we see an initial decrease which quickly flats out and for memory consumption we see no changes.

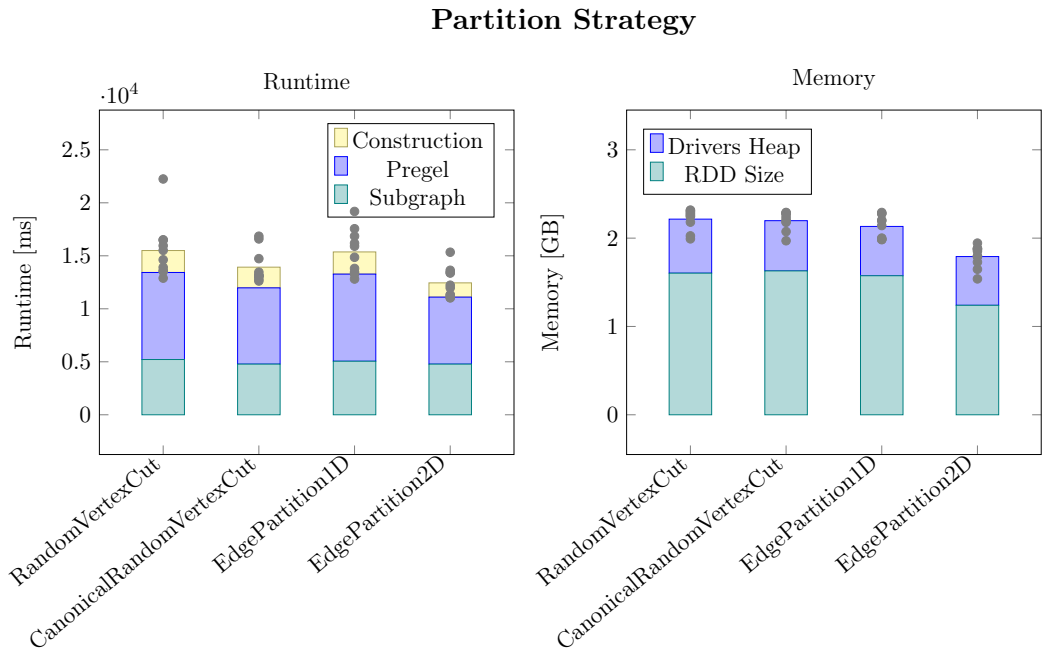


Figure 9.7: Shows the runtime and maximum memory consumption for different graph partitioning strategies found in GraphX. We see very similar runtimes and memory consumptions among all approaches. *RandomVertexCount* as worst performing gives an average runtime of 15.4s. The corresponding average maximum memory consumption is 2215MB, 612MB for drivers heap and 1603MB for RDD size. *EdgePartition2D* as best performing gives an average runtime of 12.5s. Corresponding average maximum memory consumption is 1791MB, 552MB for drivers heap and 1239MB for RDD size.

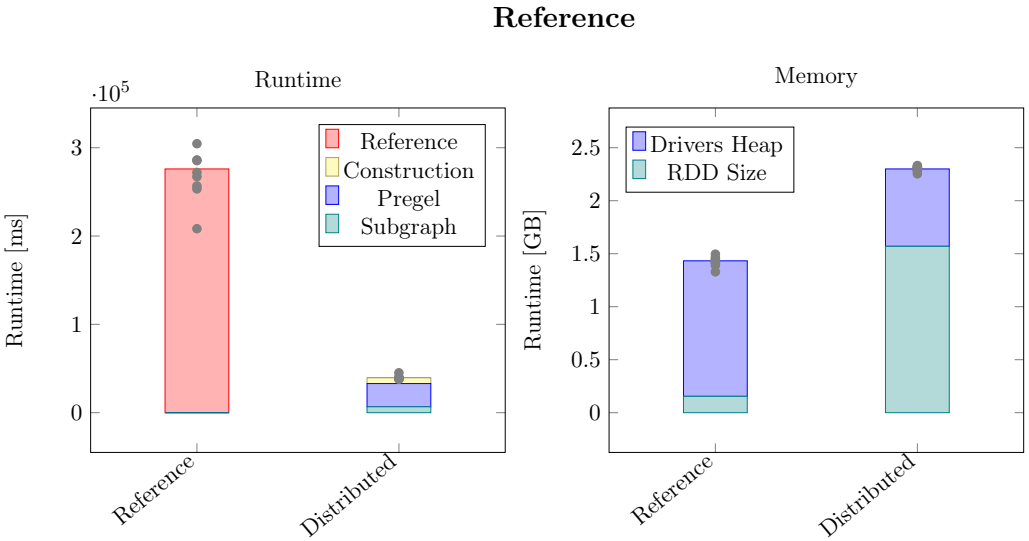


Figure 9.8: Shows the runtime and maximum memory consumption of the reference solution compared the distributed solution on the baseline experiment. The reference gives an average runtime of 276s compared to the average runtime of 39.5s for the distributed. Corresponding memory consumption is 1432MB for the reference, where 1278MB is drivers memory and 154MB is RDD size, and 2300MB for the distributed, where 730MB is drivers memory and 1570MB is RDD size. We see that the distributed solution outperforms the reference by a factor of 6.9 for runtime, but uses 1.46 times more memory in total.

Pairwise Continuous Path Observations

We see that for pairwise continuous paths we get a linear increase with increasing max-lengths, top-ks and graph sizes. Even if the constant runtimes can be relatively high, a linear increasing runtime is generally a desirable property of an algorithm of this sort. From the baseline experiment we see the runtimes at around 20 seconds, where the pregel phase accounts for around two thirds and the rest is the subgraph and path construction phases. For the baseline experiment we believe this is a reasonable constant runtime. However, as we have only created a prototype it has not been extensively optimized. Therefore, this number could become lower making the runtime reasonable in practice, and in essence showing that these type of queries could be realistic in practical use.

We generally see that the pregel phase is the dominating phase both for the constant baseline and with increasingly expensive parameter inputs. This can be seen for both continuous and pairwise continuous paths for the max-length, top-k and graph size experiments. This is expected as the more interval-states and paths that we are generating for each vertex and interval, the more computations have to be made to generate these.

The subgraph phase is as expected only dependent on the graph sizes. That is the reason for the sugraph phase being essentially constant in runtime for most of the experiments expect of the graph size experiments. However, even for the graph size experiments the increase is marginal compared to the pregel phase. The eventual difference we see in the experiments or across different experiments, we believe is caused by the load differences on cluster at different time of the experiments, as we briefly pointed out in the previous section.

The path construction phase generally accounted for much less of the runtime than we initially expected. We know that this phase is more dependent on collecting the graph elements to the driver node, meaning that the network throughput and delay would be the major source of a possible bottleneck for this phase. We believe that this was not an issue as the cluster had adequate network infrastructure and resources, as is generally common for compute clusters as the network is usually not the most expensive. In contrary to the CPU computations (i.e. the pregel phase), which also took larger part than maybe initially expected. However, we generally see that the path construction phase still increases with increasingly expensive parameter inputs for max-length, top-k and graph size. This increase is not significant compared to the pregel phase and can barley be visible on the resulting graphs.

We see that the length and graph size experiments follow a similar linear increasing trend, while the top-k experiments show an increase to a certain point where it starts to flat out. We believe the flat out is because of it not being more than that amount of results for each indivdual interval-vertex state, meaning that a $k = 25$ and $k = 100$ ends up sorting the same 25 results. However, we still get 100 paths for the final query result for the $k = 100$ because we can have multiple destinations vertices to take from. The linear increasing trend for length and graph size generally shows that each increment in length or for each increase in graph element accounts for a certain constant runtime increase.

For the worker count experiments we did not get the results we hoped for. We can see an initial decrease in runtime of going from 1 to 2 workers, seeing that there is indeed benefit of doing the computations in parallel. Disappointingly we did not see any performance benefit of going beyond that. We had to run each individual worker count number separately at different times of the day, unlike the other experiments, and we were seeing wildly different runtimes at different times of the day. This is the major reason why it was hard to

get measurements that was only affected by the worker count. In addition, we believe that larger datasets would generally see more benefit from more parallelism as there would be longer running operations. This would cause less relative overhead because of synchronization and communication between the workers and the driver. We can already see for our experiments that increasing to 16 workers actually decreases performance significantly.

For memory consumption we see that the driver memory holds itself rather constant across all experiments and parameters, while the RDD size is the most affected by the different variations. The driver memory can be explained by the fact that the path construction phase is only dependent on the top-k and the number of destination vertices, where the top-k is relatively small and the number of destination vertices are constant. The RDD size is dependent on the actual interval-states and the paths that are generated during the pregel phase. The number of paths we get for each vertex and interval naturally depends on the max-length, graph size and the top-k. In addition, we generally see that memory consumption follows the runtime relatively closely. This can be explained by the more paths that we have generated the more CPU computation have to be made, in the pregel phase in particular.

9.2.3 Interaction Paths: Continuous

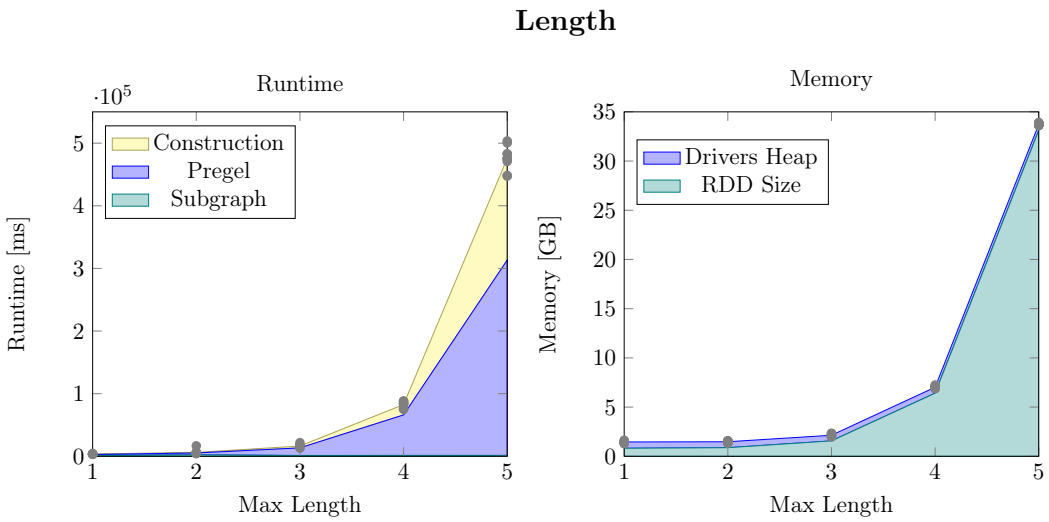


Figure 9.9: Shows the runtime and maximum memory consumption for increasing the max-lengths. The smallest max-length of 1 gives an average runtime of 35.7s, where the subgraph, pregel and construction phases account for 37.7%, 44.5% and 17.8% respectively. The corresponding average maximum memory consumption is 1455MB, 638MB for drivers heap and 817MB for RDD size. The largest max-length of 5 gives an average runtime of 476s, where the corresponding phase distribution is 0.3%, 65.6% and 34.1% respectively. Corresponding average maximum memory consumption is 33738MB, 627MB for drivers heap and 33111MB for RDD size. For both runtime and memory consumption we see an exponential increasing trend.

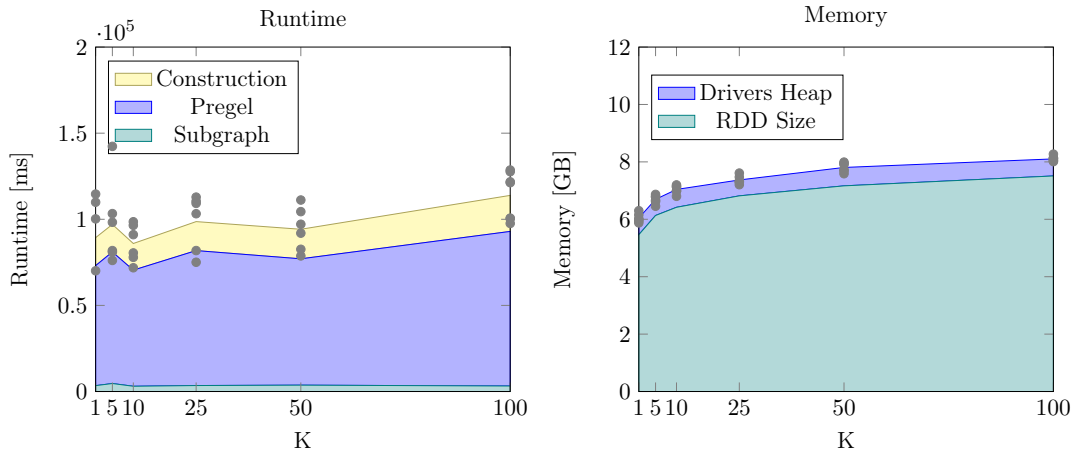


Figure 9.10: Shows the runtime and maximum memory consumption for increasing the top-ks. Smallest top-k of 1 gives an average runtime of 89.3s, where the subgraph, pregel and construction phases account for 3.8%, 77.9% and 18.3% respectively. The corresponding average maximum memory consumption is 6044MB, 581MB for drivers heap and 5463MB for RDD size. Largest top-k of 100 gives an average runtime of 114s, where the corresponding phase distribution is 2.8%, 78.8% and 18.4% respectively. Corresponding average maximum memory consumption is 8104MB, 594MB for drivers heap and 7510MB for RDD size. For both runtime and memory consumption we generally see small changes.

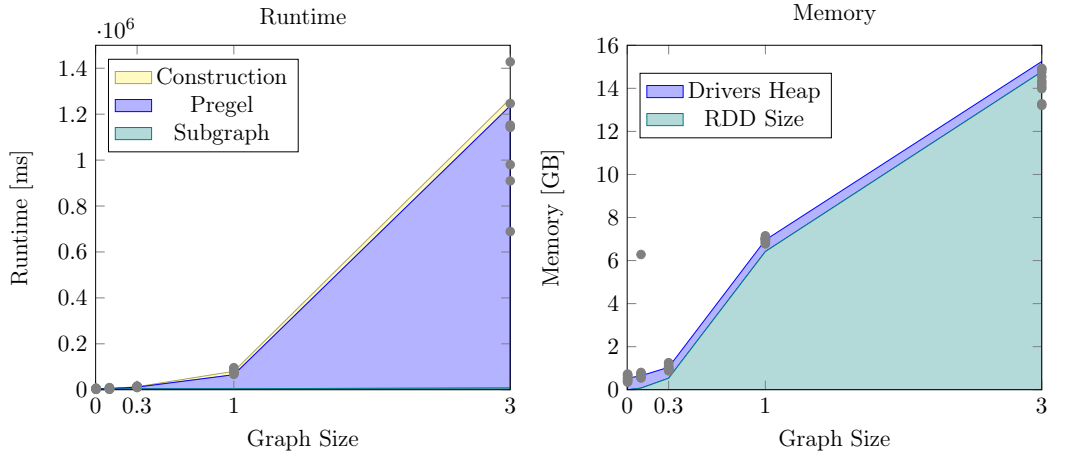


Figure 9.11: Shows the runtime and maximum memory consumption for increasing scale factors of the graph. Second smallest scale factor of 0.1 gives an average runtime of 5.4s, where the subgraph, pregel and construction phases account for 62.1%, 29.4% and 8.5% respectively. The corresponding average maximum memory consumption is 650MB, 576MB for drivers heap and 74MB for RDD size. Scale factor of 3 gives an average runtime of 1270s, where the corresponding phase distribution is 0.5%, 96.9% and 2.6% respectively. Corresponding average maximum memory consumption is 15248MB, 477MB for drivers heap and 14771MB for RDD size. For both runtime and memory consumption we see an exponentially increasing trend.

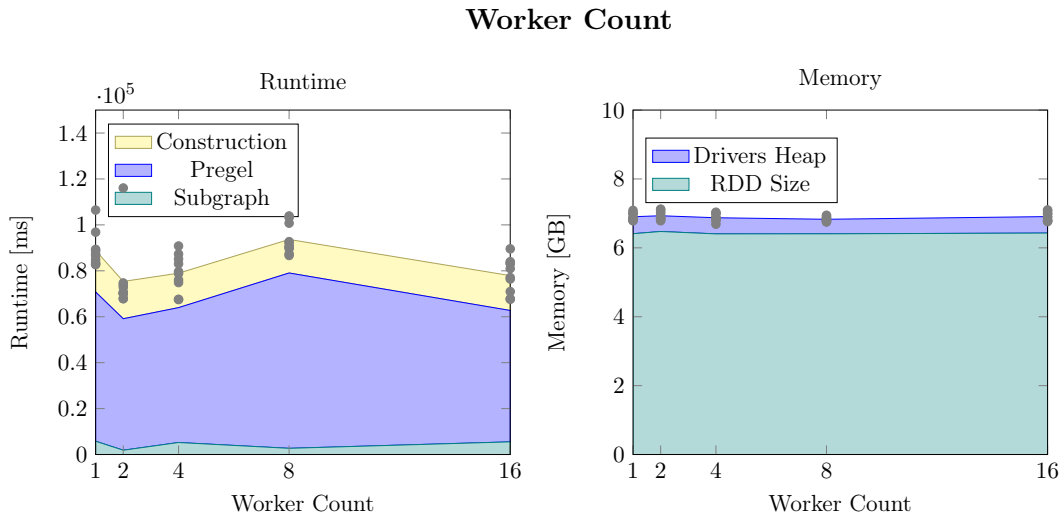


Figure 9.12: Shows the runtime and maximum memory consumption for increasing the number of work counts. Worker count of 1 gives an average runtime of 889s, where the subgraph, pregel and construction phases account for 6.5%, 73.1% and 20.4% respectively. The corresponding average maximum memory consumption is 6903MB, 493MB for drivers heap and 6410MB for RDD size. Worker count of 16 gives an average runtime of 779s, where the corresponding phase distribution is 7.1%, 73.4% and 19.5% respectively. Corresponding average maximum memory consumption is 6906MB, 478MB for drivers heap and 6428MB for RDD size. For both runtime and memory consumption we generally see small changes.

9.2.4 General Observations

The most evident observation we can make about the runtime is that there is a clear difference between the pairwise continuous and continuous paths. We see that pairwise continuous paths increase rather linearly with both increasing max-lengths, top-ks and graph sizes. The continuous paths results on the other hand shows something that is closer to exponentially increasing runtimes. This is especially apparent for the max-length and graph size experiments. This shows that pairwise continuous paths are generally feasible with our prototype implementation, while continuous paths are not. The reason for this, as we touched upon in Section 8, is that continuous paths create unique intervals for most paths, meaning that we are not able to prune away results in each interval, such that we end up generating all possible paths.

Generally, we see that our solution, that is for pairwise continuous paths, are able to scale with larger datasets and more expensive parameters such as length and top-k. For parallelism we see tendencies of speedups with increasing worker count, but there is limited amount of scalability. The other general observation we can conclude with is that continuous paths are generally not feasible in practice with our solution as runtime and memory grows exponentially with input. Here there needs to be other solutions to deal with the problem of having unique intervals which leads to generating near all possible paths.

10 Conclusion and Future Work

In this section we will give a conclusion of our work and attempt to answer our initial research questions. In addition, we will discuss issues and limitations with our solution and possible future work that can be explored. We will further discuss other directions in the field of temporal graph queries and distributed processing of temporal graph data.

The distributed temporal graph frameworks we have studied can generally be categorised by the following characteristics and properties we have explored in our literature review: Temporal graph models, temporal graph queries and distributed programming models. We have found that the temporal graph models are conceptually equivalent, but can be represented in different ways, where we have found valid-time, snapshot sequence, delta snapshots and transformed graph representations. Temporal graph queries utilizes this temporal graph model and is an extension of the queries found in GQLs. In essence, they have added predicate capabilities with interval relation operators, which create the extensions of temporal constant length, temporal variable length and temporal weighted path queries. We see that there are newly suggested languages that have incorporated these query functionalities, but where we have found that temporal weighted paths in particular have not been generally explored.

Furthermore, there have been frameworks that have focused more on lower level distributed programming interfaces that can implement these temporal graph queries or other temporal graph algorithms. We have distinguished the programming models capabilities in whether they are able to implement time-independent or time-dependent algorithms, where we have the snapshots incremental and the interval-centric models respectively, which both are extensions of the vertex-centric programming model. In addition, we have the more general purpose dataflow programming model that also have been applied to graphs and temporal graphs. We have seen existing attempts to implement some form of temporal graph queries with either of these approaches, both showing working solutions, but the vertex-centric based models seems to be able to scale better.

With this we have essentially summarized our answer to the first research question, which was "What are the existing temporal graph framworks/systems and how do they compare". With this knowledge and literature review we were able to answer our second research question which was "How can novel temporal graph algorithms and operators be implemented to extend these systems?". As we concluded, there is a lacking of languages and query models that were able to express general forms of temporal weighted path queries. In addition, we saw that there were generally a lacking of overlap between temporal graph query functionality and distributed scalability.

We therefore made a prototype where we incorporated temporal weighted path queries which included parameters such as length, top-k, node predicates, consecutive interval relations and weight functions in one query model. We looked at two applications areas, namely social network analysis and transport network analysis, where we showed the applicability of the model with some example use cases. We implemented the query model with a vertex-centric based programming model where we took inspiration from the interval centric model as it had showed promising results. To the best of our abilities, we created our own algorithms which included multiple phases and data structures to support the necessary computations. Our solution showed that we got the correct and expected results from test and experimental queries. For pairwise continuous paths we see that our solution are able to scale with larger datasets and more expensive parameters as well as showing tendencies of speedups with increasing worker counts. With this query model, prototype

and experimental results, we have essentially answered our second research question.

However, looking at our experimental results further we find that there are room for improvements and directions for future work. We find that continuous paths in particular show exponential runtime and memory increases. The essential problem that we find is that each path exists in its own unique interval, which leads to generating near all possible paths. This leads to future work in ways to improve upon the interval centric model where we can deal with this problem, either as part of the programming interface or internally hidden from the programmer. This also leads to possible work towards the interval state representation in general. As we realize that there is quite a lot of redundant state, there is possibilities of finding efficient ways of representing or compressing this interval state.

As we have only focused on processing and not storage of temporal graphs, another direction is to focus more on the database and storage aspect as well. This also opens up the opportunity of creating indexes for efficient retrieval of the path queries. As the type of temporal weighted path queries we have explored have not been experimented with before, looking at ways of speeding up the queries with indexes is a possible future work direction.

Taking a step back and looking at the field in general, we see quite a bit work towards unifying and merging the interfaces for batch oriented processing and stream processing. As we have noted towards in our literature review, there are already some work in this direction for temporal graphs. However, these systems have not focused on the type of temporal graph queries we have looked at here. Therefore, possible directions for future work could be to look at these types of temporal graph queries done for stream processing systems.

References

[1] J. F. Allen. 'Maintaining Knowledge about Temporal Intervals'. In: *Communications of the ACM*, vol. 26, no. 11, 1983, pp. 832–843.

[2] R. Angles. 'The Property Graph Database Model'. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2018.

[3] R. Angles, M. Arenas, P. Barcelo et al. 'G-CORE A Core for the Future Graph Query Languages'. In: *SIGMOD '18: Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1421–1432.

[4] M. Arenas, P. Bahamondes, A. Aghasadeghi et al. 'Temporal Regular Path Queries'. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2412–2425.

[5] S. Aridhi, A. Montresor and Y. Velegrakis. 'BLYDAG: A Graph Processing Framework for Large Dynamic Graphs'. In: *Big Data Research*, vol. 9, 2017, pp. 9–17.

[6] F. Bajaber, R. Elshawi, O. Batarfi et al. 'Big Data 2.0 Processing Systems: Taxonomy and Open Challenges'. In: *Journal of Grid Computing*, vol. 14, 2016, pp. 379–405.

[7] A. Baranawal and Y. Simmhan. 'Optimizing the Interval-centric Distributed Computing Model for Temporal Graph Algorithms'. In: *EuroSys '22: Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 541–558.

[8] M. Besta, M. Fischer, V. Kalavri et al. 'Practice of Streaming and Dynamic Graphs: Concepts, Models, Systems, and Parallelism'. In: *arXiv*, no. 1912.12740, 2021.

[9] J. Byun, S. Woo and D. Kim. 'ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time'. In: *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 3, 2020, pp. 424–437.

[10] M. E. Coimbra, A. P. Francisco and L. Veiga. 'An analysis of the graph processing landscape'. In: *Journal of Big Data*, vol. 8, no. 55, 2021.

[11] TH. Cormen, CE. Leiserson, RL. Rivest et al. *Introduction to Algorithms - Fourth Edition*. The MIT Press, 2022.

[12] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *Communications of the ACM*, vol. 51, no. 1, 2008, pp. 107–113.

[13] A. Debrouvier, E. Parodi, M. Perazzo et al. 'A model and query language for temporal graph databases'. In: *The VLDB Journal*, vol. 30, 2021, pp. 825–858.

[14] M. Flynn. 'Fynn's Taxonomy'. In: *Encyclopedia of Parallel Computing*, 2011, pp. 689–697.

[15] LDBC Social Network Benchmark task force and contributors. *The LDBC Social Network Benchmark (version 2.2.1)*. 2022.

[16] N. Francis, A. Green, P. Guagliardo et al. 'Cypher: An Evolving Query Language for Property Graphs'. In: *SIGMOD'18: Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1433–1445.

[17] S. Gandhi and Y. Simmhan. 'An Interval-Centric Model for Distributed Computing over Temporal Graphs'. In: *IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1129–1140.

[18] A. P. Iyer, T. Das, L. E. Li et al. 'Time-evolving graph processing at scale'. In: *GRADES '16: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

[19] A. P. Iyer, Q. Pu, K. Patel et al. ‘TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs’. In: *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*, 2021, pp. 337–355.

[20] W. Lightenberg, G. Fletcher and M. Pechenizkiy. *Tink, a temporal graph analytics library for Apache Flink*. 2018.

[21] W. Lightenberg, Y. Pei, G. Fletcher et al. ‘Tink: A Temporal Graph Analytics Library for Apache Flink’. In: *WWW ’18: Companion Proceedings of the The Web Conference 2018*, 2018, pp. 71–72.

[22] G. Malewicz, M. H. Austern, A. J. C. Bik et al. ‘Pregel: A System for Large-Scale Graph Processing’. In: *SIGMOD ’10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[23] Y. Miao, W. Han, K. Li et al. ‘Chronos: A Graph Engine for Temporal Graph Analysis’. In: *EuroSys ’14: Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[24] Y. Miao, W. Han, K. Li et al. ‘ImmortalGraph: A System for Storage and Analysis of Temporal Graphs’. In: *ACM Transactions on Storage*, vol. 11, no. 3, 2015, pp. 1–34.

[25] V. Z. Moffitt and J. Stoyanovich. ‘Querying Evolving Graphs with Portal’. In: *arXiv*, no. 1602.00773, 2016.

[26] V. Z. Moffitt and J. Stoyanovich. ‘Temporal Graph Algebra’. In: *DBPL ’17: Proceedings of The 16th International Symposium on Database Programming Languages*, 2017, pp. 1–12.

[27] S. Ramesh, A. Baranawal and Y. Simmhan. ‘Granite: A distributed engine for scalable path queries over temporal property graphs’. In: *Journal of Parallel and Distributed computing*, vol. 151, 2021, pp. 94–111.

[28] O. v. Rest, S. Hong, J. Kim et al. ‘PGQL: a Property Graph Query Language’. In: *GRADES’16: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

[29] M. A. Rodriguez. ‘The Gremlin Graph Traversal Machine and Language (Invited Talk)’. In: *DBPL 2015: Proceedings of the 15th Symposium on Database Programming Languages*, 2015, pp. 1–10.

[30] C. Rost, K. Gomez, M. Täschner et al. ‘Distributed temporal graph analytics with GRADOOP’. In: *The VLDB Journal*, vol. 31, 2022, pp. 375–401.

[31] S. Sakr. *Big Data 2.0 Processing Systems - A Systems Overview (Second Edition)*. 2020.

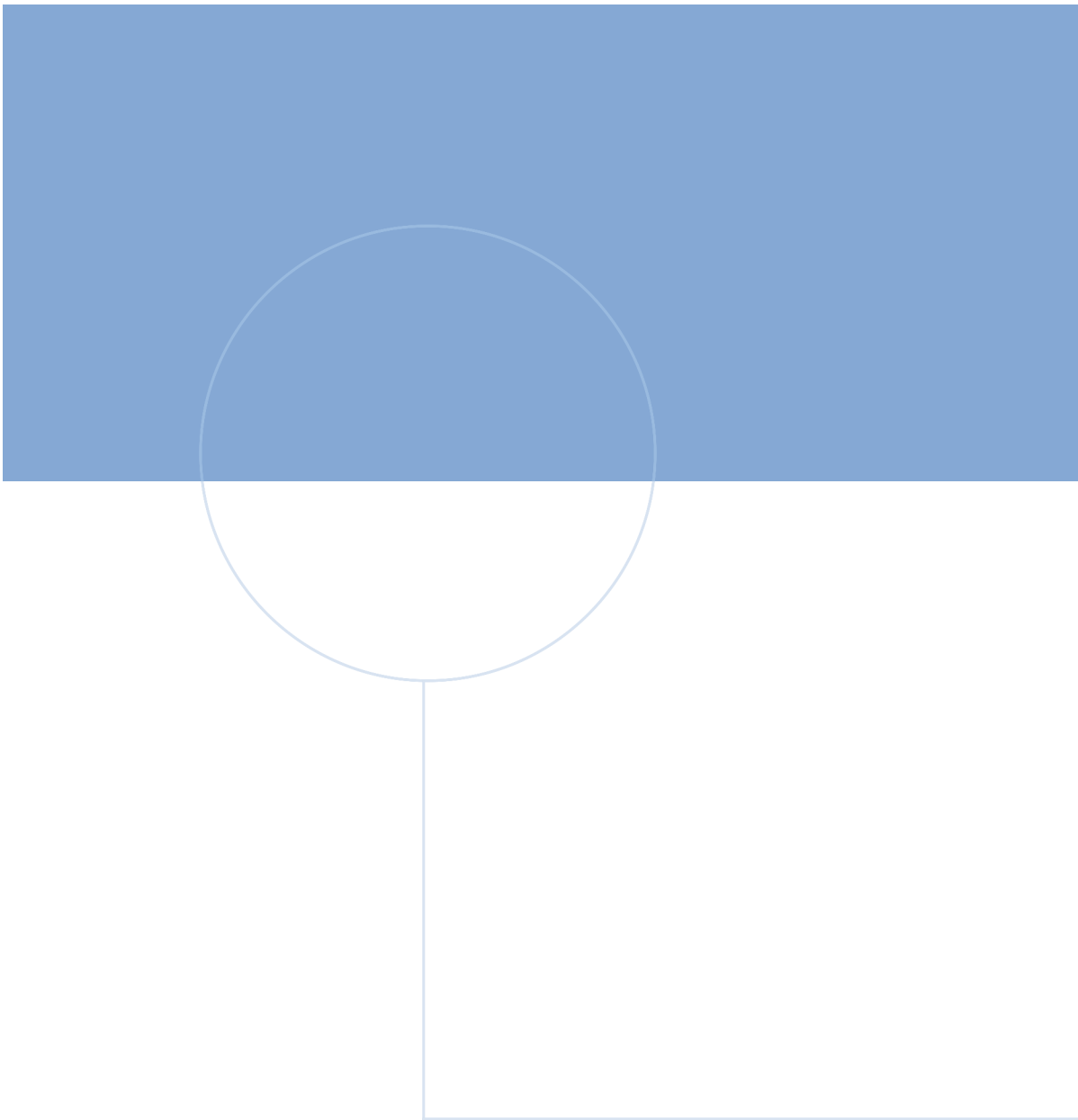
[32] B. Steer, F. Cuadrado and R. Clegg. ‘Raphtory: Streaming analysis of distributed temporal graphs’. In: *Future Generation Computer Systems*, vol. 102, 2020, pp. 453–464.

[33] Y. Wang, Y. Yuan, Y. Ma et al. ‘Time-Dependent Graphs: Definitions, Applications, and Algorithms’. In: *Data Science and Engineering*, vol. 4, 2019, pp. 352–366.

[34] H. Wu, J. Cheng, Y. Ke et al. ‘Efficient Algorithms for Temporal Path Computation’. In: *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, 2016, pp. 2927–2942.

[35] R. Zadeh. ‘Communication Patterns’. In: *Spark Summit 2015*, 2015.

[36] Y. Zhang, T. Cao, S. Li et al. ‘Parallel Processing Systems for Big Data: A Survey’. In: *Proceedings of the IEEE*, vol. 104, no. 11, 2016, pp. 2114–2136.



NTNU

Norwegian University of
Science and Technology