Designing a temporal graph management system for IoT application domains

Maria Massri

▶ To cite this version:

Maria Massri. Designing a temporal graph management system for IoT application domains. Computer Science [cs]. Université de rennes, 2022. English. NNT : . tel-04071498v1

HAL Id: tel-04071498
https://inria.hal.science/tel-04071498v1

Submitted on 17 Apr 2023 (v1), last revised 3 May 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE Nº 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Maria MASSRI

Designing a temporal graph management system for IoT application domains

Thèse présentée et soutenue à Rennes, le 20 Décembre 2022
Unité de recherche : IRISA (UMR 6074), Rennes

Rapporteurs avant soutenance :

Evvagelia PITOURA	Professor	Ioannina University
Julia STOYANOVICH	Associate professor	New York University

Composition du Jury :

Président :	Francois GOASDOUE	Professeur des universités	Université de Rennes 1
Examinateurs :	Angela BONIFATI	Professeur des universités	Université de Lyon 1
	Dan VODISLAV	Professeur des universités	Université CY Cergy Paris
	Gábor SZÁRNYAS	Researcher	CWI Amsterdam
	Philippe RAIPIN	Ingénieur chercheur	Orange Labs
Dir. de thèse :	Zoltán MIKLÓS	Maître de conférences, HDR	Univeristé de Rennes 1

Invité :

Pierre MEYE	Ingénieur	Rakuten

Acknowledgement

First and foremost, I would like to thank my supervisors, Zoltan Miklos and Philippe Raipin, who guided me throughout this dissertation and brought these efforts to fruition.

Besides his immense knowledge, Zoltan is the most professional, kind, and humble supervisor a student can hope for. I particularly appreciate his enthusiasm to assist in any way he could throughout these three years. Through this journey, he has helped me thoroughly with his motivating words, thoughtful comments, and always constructive feedback. Our discussions were vital in inspiring me to think outside the box and to have a broader perspective on my work. He has tremendously guided me to ameliorate my writing skills. We all know that the path of a Ph.D. student is never smooth; however, Zoltan always encouraged me through the inevitable rejections and setbacks.

Philippe is incredibly brilliant; each discussion with him turns into a deep brainstorming session. I started working with Philippe during my internship, and since then, he has been a source of inspiration for me. His intelligence, gentle critique, good leadership, and humility make working with him a pleasant experience that one can learn from. I was never hesitant about sharing my blurred thoughts during our discussions, knowing that Philippe would give the right directions. I particularly appreciate how much he believed in my potential by pushing me over the edge and relying on me for tasks I would never think I would be able to complete.

Thanks also go to Pierre Meye, who helped me throughout this journey. Pierre did not only give me feedback on my research work but also assisted with many technical issues that required professional engineering skills. Besides, Pierre is very organized and detail-oriented, as reflected in his comments and recommendations to make my research or technical presentations and writings more pedagogic and concise.

Next, my thanks go to the members of the defense jury for the thesis who did me the honor of agreeing to review it: Evaggelia Pitoura, Julia Stoyanovich, Francois Goasdoué, Dan Vodislav, and Gabor Szárnyas for kindly evaluating this work. I would also like to thank my CSID members, Angela Bonifati and Thomas Guyet, for their yearly progress reviews and evaluations.

The DDSD team was my home. Every member is highly professional, extremely caring,

and friendly, which fosters a pleasant work environment. I would like to particularly thank Romuld Cimia, the team's manager, for his efforts in assuring a cooperative atmosphere for undergraduate students who joined the group. I also thank him for being extremely patient with me, especially in administrative procedures, which I had little knowledge about. I also thank David Crosson, who is impressive on every personal and professional scale. He is most credited with developing my half-mature engineering skills by teaching me how to avoid technical pitfalls. I have gained a lot of experience from his engineering and communication skills which I highly appreciate and will carry into my future career. I am also grateful for working with the other team members, Sylvie Derrien, Alain Dechorgnat, Thomas Hassan, Cyprien Gottstein, and Amaury Bouchra Pilet.

I want to thank the DRUID team for their supportive work environment. Participating in the team seminars helped me discover new research areas. Their constructive feedbacks on my work were also formative. I want to particularly thank David Gross-Amblard for his support and insightful suggestions.

I want to express my gratefulness and appreciation to my lovely family. To my mother, Alice, for her invaluable support. Her unconditional love, devotion to her family, and strength inspired me to become a better version of myself. Indeed, words are not powerful to express my gratitude for her kindness to our family. To Mahmoud, for being the most wonderful father. Not letting him down was one of the main motivations to keep going after each rejection. He taught me that the key to every success is to be consistent, a lesson I have carried out throughout the past three years. To my wonderful sister Nadine, who had not the slightest doubt about my skills. Even though she was going through hard times, she was eager to support me and make me feel optimistic. To my amazing husband, Abdulkader, for bearing my absence and most brutal setbacks with utmost love and support. I am very grateful for his constant encouragement, which was essential for me to endure the journey. To the cutest nieces, Tia and Thalya who brought joy into our lives. I genuinely hope my journey will inspire them to pursue their dreams no matter how difficult they seem.

Being a Ph.D. student confined during a global pandemic overturned my daily habits. Overcoming this constraint would be unbearable without the presence of my friends Israa, Sana, Sahar, Yara, Maryam, Iman, Asma, Aicha, Judy, and Lara.

Finally, I thank my close friends Sabah, Roula, Bouchra, and Sally, and my work colleagues Maya, Dan, and Minh for the good times and unforgettable memories.

UNIVERSITÉ DE RENNES 1

Titre : Développement d'un système de gestion des graphes temporelles pour le domaine d'objets connectés

Mot clés : Bases de données orientées graphe, bases de données temporelles, langage de requêtes, stockage

Résumé : Les graphes sont fréquemment utilisés pour modéliser les interactions du monde réel comme une collection de nœuds et de relations fournissant, généralement, un modèle simple et intuitif pour analyser les domaines centrés sur les relations. Il y a eu un développement substantiel dans la conception de bases de données facilitant la modélisation et l'interrogation des données sous forme de graphes. Ces efforts ont conduit à la conception de bases de données orientées graphes qui intègrent des techniques spéciales de matérialisation et d'évaluation de requêtes optimisées pour les modèles de données de graphes.

Compte tenu de leur grande expressivité dans la représentation de relations complexes, les modèles de graphes ont été utilisés dans de nombreuses applications. Cependant, un grand nombre de ces graphes subissent des changements continus ou sporadiques. Dans de nombreuses applications, des informations plus importantes peuvent être extraites de l'analyse de l'historique de ces graphes plutôt qu'un seul état statique (c'est-à-dire non temporel). En d'autres termes, le suivi de l'historique des graphes ouvre un large éventail de capacités analytiques telles que la détection d'anomalies, la prévention des pannes ou la prévision du comportement futur. Ces capacités ont favorisé l'intégration de la dimension temporelle dans de nombreuses applications.

Dans cette thèse, nous nous concentrons principalement sur un cas d'usage particulier qui est celui de Thing'in, une plateforme de recherche innovante, et in vivo gérant un graphe d'objets, où les nœuds sont des objets (principalement des objets IoT) et les arêtes sont des relations entre eux. Les utilisateurs de Thing'in sont des entreprises et des administrations publiques développant des services autour des villes intelligentes, des bâtiments ou des usines intelligentes, ainsi que des propriétaires d'objets privés et des développeurs construisant des applications IoT. La majorité des nœuds du graphe Thing'in représentent des appareils IoT (par exemple, des machines, des détecteurs de mouvement ou des caméras). Alors que les autres nœuds sont liés à l'environnement de ces appareils et fournissent une description structurelle et sémantique de leur environnement (par exemple, des villes, des bâtiments ou des pièces).

Lorsque la plateforme a été initiée en 2017, le graphe de Thing'in contenait 50 milliers d'objets connectés. Ce nombre a régulièrement augmenté pour atteindre 50 millions, un nombre qui continuera à augmenter dans le proche avenir. Le graphe de Thing'in étant non statique, des questions importantes sur les états passés du graphe peuvent être posées ce qui a motivé la gestion de la dimension temporelle dans la plateforme. L'un des nombreux cas d'usage de Thing'in qui peut grandement bénéficier d'un support de version temporelle est Mo.Di.Flu, un projet dont l'objectif principal cible l'Industrie 4.0 et BIM2TWIN[1] projet axé sur l'intégration de Digital Twins dans des bâtiments intelligents (par exemple, des usines intelligentes). La plateforme Thing'in est utilisée dans ce projet pour

1. https ://bim2twin.eu/

analyser l'historique de ces objets comme suivre les différentes positions d'un produit tout au long du pipeline de fabrication, détecter les causes des retards de fabrication ou des pertes de produit et reconstituer l'état du graphe avant une défaillance du système.

Raisonner sur le passé et répondre aux requêtes temporelles des cas d'usage de Thign'in tels que Mo.Di.Flu n'est possible que si l'historique du graphe est géré. Au début de cette thèse, la plateforme Thing'in n'était pas conçue pour supporter la dimension temporelle et seul le dernier état du graphe était conservé. Ainsi, l'objectif de cette thèse est d'intégrer la dimension temporelle dans la plateforme Thing'in. L'une des solutions possibles est de développer une couche temporelle audessus d'une base de données de graphes non temporels. Bien que cette implémentation soit pratique en raison de sa simplicité, nous supposons qu'une base de données de graphes temporels devrait être construite avec un support temporel natif. Cela nous a motivés à construire un système de graphe temporel à partir de zéro en abordant les différentes caractéristiques de conception telles qu'un modèle de données de graphe temporel, l'algèbre, le langage de requête et d'autres fonctionnalités de conception du système (par exemple, l'optimisation du stockage et des requêtes). Pour cela, nous avons défini les objectifs suivants :

- Définition d'un langage de requête de graphe temporel pouvant répondre aux besoins des cas d'utilisation de Thing'in.
- Implémentation d'un moteur de stockage de graphes temporels qui offre un usage compact de l'espace de stockage secondaire tout en maintenant la facilité d'extraction des données.
- Conception d'un processeur de requêtes pour évaluer les requêtes temporelles proposées.

Bien que de nombreux systèmes existants soient conçus avec un support temporel, aucun de ces systèmes ne répond complètement à nos exigences. Plus précisément, les systèmes disponibles n'adressent pas plusieurs aspects dans la conception du système comme le langage de requêtes, stockage et optimisation des requêtes. Par exemple, il existe des systèmes qui se concentrent sur le stockage de données mais présentent des requêtes temporelles simples ou basiques ou des requêtes analytiques qui ne sont pas dans le cadre de cette thèse. Alors que les systèmes offrant un langage permettant l'écriture de requêtes de *Pattern match* n'abordent pas les fonctionnalités de conception telles que la technique de stockage ou les techniques d'optimisation des requêtes. Pour pallier ces limitations, nous avons conçu Clock-G, un système de gestion des graphes temporels, tel que le but ultime est d'intégrer ce système dans la plateforme Thing'in. Les principales caractéristiques de Clock-G, ainsi les principales contributions de cette thèse sont :

- La définition du langage de requête de graphe temporel T-Cypher qui permet l'écriture des requêtes temporelles non verbeuses en ajoutant des règles temporelles à la grammaire existante du langage Cypher. La nouvelle syntaxe permet d'exprimer des requêtes de pattern matching ou de parcours de graphes temporelles qui ne peuvent pas être exprimé par les langages existants.
- La proposition et implémentation d'une stratégie de stockage qui est basée sur les concepts de matérialisation des états du graphe à des instants temporels pour pouvoir y accéder facilement lors du requêtage. La méthode proposée offre un équilibre entre l'espace de stockage et le temps de réponse des requêtes ce qui ne peut pas être atteint avec les méthodes de stockage traditionnelles.
- La proposition et implémentation d'un processeur de requêtes T-Cypher qui choisit de façon *greedy* le meilleur plan d'évaluation d'une requête temporelle en tenant compte de l'intervalle temporel des requêtes.

TABLE OF CONTENTS

1 Introduction **11**

2 Related work **25**

2.1 Graph management 25
2.1.1 Graph models and querying 26
2.1.2 System design 31
2.2 Temporal graph management 36
2.2.1 Temporal graph models and querying 36
2.2.2 System design 45
2.3 Graph generators 57
2.4 Temporal graph generators 61
2.5 Conclusion 64

3 Clock-G: A temporal graph management system **65**

3.1 Overview 65
3.2 Conclusion 68

4 Temporal graph query language **69**

4.1 Motivations and contributions 69
4.2 Preliminaries 70
4.2.1 Time domain 70
4.2.2 Temporal property graph model 71
4.3 Temporal graph relation 72
4.4 Temporal query constructs of T-Cypher 76
4.4.1 Temporal slicing clause 76
4.4.2 Temporal functions and operators 77
4.4.3 Temporal paths 78
4.4.4 Temporal aggregation 82
4.5 Syntax of T-Cypher 83

4.5.1 Temporal values 84
4.5.2 Expressions 85
4.5.3 Patterns 86
4.5.4 Queries 87
4.6 Industrial integration of T-Cypher 88
4.6.1 Model translation rules 91
4.6.2 Query translation rules 92
4.7 Conclusion 95

5 Temporal graph storage technique **97**
5.1 Preliminaries 98
5.1.1 Temporal property graph model 98
5.2 δ-Copy+Log 100
5.2.1 Space and time complexities 106
5.3 Implementation 109
5.3.1 System components 109
5.3.2 Querying 116
5.4 Experimental evaluation 119
5.4.1 Experimental setup 120
5.4.2 Space usage and query execution time 121
5.5 Conclusion 128

6 Temporal graph query processing **129**
6.1 General overview 130
6.2 Temporal graph algebra 131
6.2.1 Operators 132
6.3 Cost model 135
6.4 Greedy plan selection algorithm 137
6.5 Temporal Histograms 139
6.5.1 Segment trees 139
6.5.2 Compression of temporal histograms 142
6.6 Evaluation 144
6.6.1 Experimental setup 145
6.6.2 Queries 146
6.6.3 Plan selection 148

6.6.4 Comparison with Neo4j 151
6.7 Conclusion 152

7 Temporal graph generation 155
7.1 Overview 156
7.2 Generation with degree distribution 157
7.3 Community-aware generation with degree distribution 159
7.3.1 Graph community 159
7.3.2 Stochastic block model 160
7.3.3 Stochastic block model with degree distribution 161
7.3.4 Hierarchical community structure 162
7.4 Relative graph generation 163
7.4.1 Earth mover's distance 164
7.4.2 Baseline relative graph generation 164
7.4.3 Relative community-aware graph generation 167
7.4.4 Accuracy of the generation procedure 170
7.5 Generating evolving properties 171
7.5.1 Model 171
7.5.2 Implementation 174
7.6 Experimental evaluation 175
7.6.1 Controlling the evolution of the degree distribution 177
7.6.2 Controlling the community structure 178
7.6.3 Generating graphs with deletions 179
7.6.4 Accuracy of the generation procedure 180
7.7 Conclusion 181

8 Conclusions and Future work 183
8.1 Summary of contributions 183
8.2 Future directions 185
8.2.1 Directions in querying temporal graphs 186
8.2.2 Directions in storing temporal graphs 188
8.2.3 Directions in processing temporal queries 189
8.2.4 Directions in generating temporal graphs 190
8.3 Summary of contributions and directions in the industrial integration . . 191
8.3.1 Summary of contributions in the industrial integration 191

8.3.2 Directions in the industrial integration 191

Bibliography **193**

Appendix **221**

A Description of temporal functions and operators 221

Chapter 1

Introduction

Graphs are frequently used to model real-world interactions as a collection of nodes and relationships, providing a fertile ground to analyze relationship-centered domains. To keep pace with this demand, there has been a substantial development in the design of adequate databases that facilitate storing and querying graph-oriented data. These efforts led to the design of graph databases that integrate special storage and query evaluation techniques optimized for graph data models.

Graph models have spurred interest in many applications for their high expressiveness in representing complex relationships. However, many of these graphs are subject to continuous or sporadic changes. In diverse applications, deeper insights can be extracted from analyzing the history of these graphs rather than a single static (i.e., non-temporal) state. Tracking the history of graphs unlocks a breadth of querying capabilities, such as the reconstruction of the state of the graph before a system failure to analyze and prevent the causes of malfunctioning. These capabilities have fostered the integration of the temporal dimension in many applications. For example, in transportation networks, traffic is a dynamic property of roads that should be considered when finding the shortest paths. Including temporal information on roads leads to more accurate route planning, especially in critical situations such as evacuation from disastrous regions to safer ones [70, 94]. Another application of temporal graphs relates to environmental sciences, as exemplified by sensor networks deployed in water treatment plants. Monitoring and tracking anomalies in such systems can predict and thus prevent outgrowing hot spots such as harmful pathogen spills [92]. In social networks, queries retrieving the most long-lasting relationships between individuals lead to community detection [217]. Such information can be beneficial for organizing social or professional events. Another social network application is the detection of frequent co-occurrence of words between community members [219].

In this dissertation, we mainly focus on the particular use case of Thing'in [1], an

1. https://tech2.thinginthefuture.com/

innovative, open, and in-vivo platform initiated in 2017 by Orange Labs (The R&D (Research and Development) center of the French telecommunication company Orange). This platform manages a graph where nodes are objects (mostly IoT objects), and relationships are the connections among them. Currently, an R&D team of research engineers, project managers, and senior developers is working in tandem to develop this platform. Besides, doctoral students and post-doctoral fellows are annually appointed to help bridge the platform's industrial needs and the latest research work in the area. The clients of Thing'in are companies and public administrations developing services around smart cities, buildings, or factories, as well as private object owners and developers building analytical IoT applications. Indeed, IoT devices have received wide use due to their capability of sensing, actuating, and interacting among themselves, similarly with their environment. Most of the nodes in the Thing'in graph represent IoT devices (e.g., machines, motion detectors, or cameras), Whereas the rest of the nodes are related to the environment of these devices, and provide a thorough structural and semantic description of their surroundings (e.g., cities, buildings, or rooms). The graph of Thing'in is akin to a social graph where human-to-human interactions are replaced by object-to-object ones [229], as seen in Figure 1.1. The nodes and edges of this graph are labelled and can have a set of properties. For instance, the labels of the nodes in the graph of Figure 1.1 are *Office*, *Corridor*, *Sensor*, and *Lamp*. The labels of the edges are *Links* to and *Monitors*. The nodes representing an office or corridor have the property *Name*, whereas the nodes representing a sensor or lamp have the property *Model*.

Thing'in offers end-users an API to insert, update, delete, query, and visualize the graph objects. To illustrate the use of the Thign'in portal, we present in Figure 1.2 a real 2-D visualization of the result of one of the most basic Thing'in queries. As shown in the figure, this query returns the street lights near the Eiffel Tower in Paris. These nodes are distributed on the map based on their real geographic locations.

When the platform was first developed in 2017, the graph of Thing'in contained 50 thousand connected objects. This number has steadily risen to 50 million. As the graph of Thing'in evolves over time, important questions about its past states can be posited, motivating temporal analysis. One of the many use cases of Thing'in that can highly benefit from time-version support is Mo.Di.Flu, a project whose main goal targets the Industry 4.0 and BIM2TWIN [2] project focusing on the integration of Digital Twins in

2. https://bim2twin.eu/

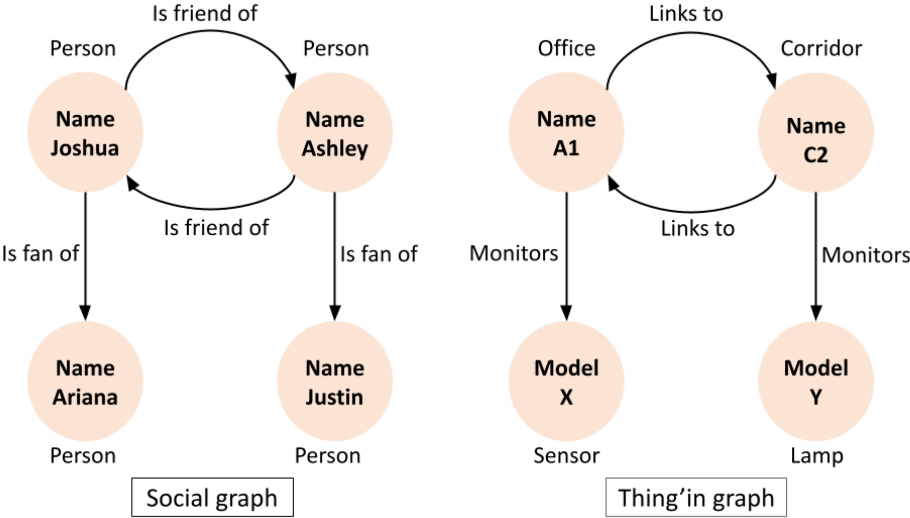


Figure 1.1 – Comparison between a social graph and the Thing'in graph

smart buildings (e.g., smart factories). The Thing'in platform is used in this project to analyze the history of these objects, such as keeping track of the different positions of a product throughout the manufacturing pipeline, detecting the causes of manufacturing delays or product losses, and reconstructing the state of the graph right before a system failure.

A smart factory graph example modeling such a graph is presented in Figure 1.3. The nodes of this graph refer to machines, sensors, employees, alert systems, and power sources. The relationships represent the maintenance of a machine by an employee, the transfer of objects between a pair of machines, the connection between a sensor and a machine, and the power supply between a power source and a machine. Now, each relationship is bound with a time interval during which the relationship was valid, whereas the time intervals of nodes are omitted for the clarity of the figure.

Agropole [3] is another example of a company using the Thing'in platform. Agropole is specialized in the agri-food industry in that it analyzes the conditions for creating, developing, and setting up agri-food projects. Besides, Thing'in is currently establishing a promising collaboration with one of the leading (German) companies in digital automation and energy management. One of the prerequisites of these collaborations is managing the history of the underlying systems, which accentuates the need for time-version support.

3. https://www.agropole.com/

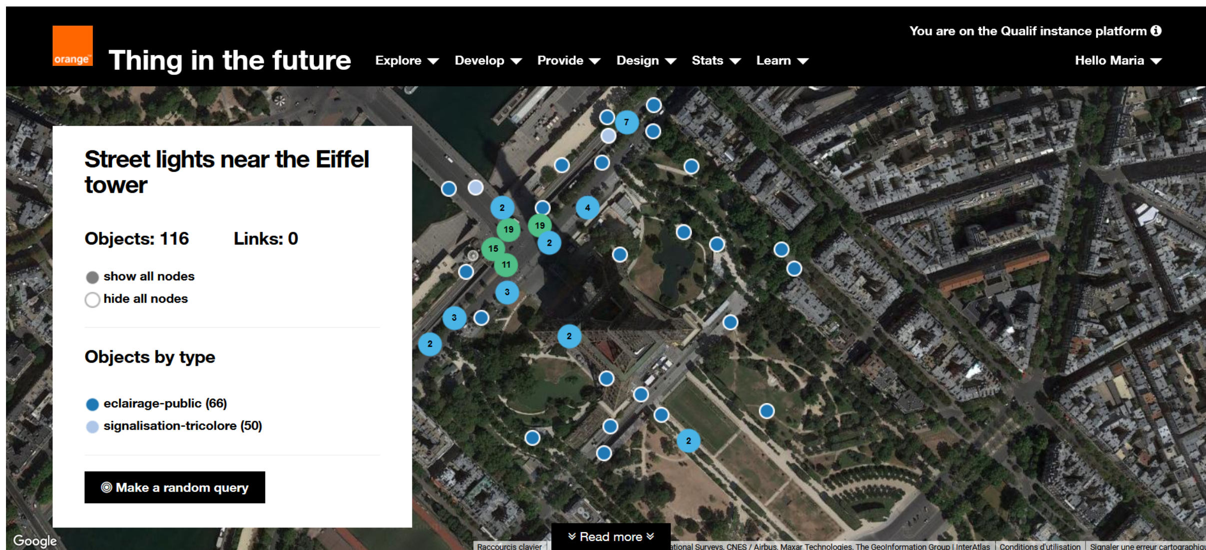


Figure 1.2 – A Thing'in 2-D visualization of the street lights near the Eiffel Tower

From the versatile applications mentioned above, we can derive that managing the history of Thing'in is becoming growingly urgent. Reasoning about the past and answering the temporal requirements of the use cases of Thign'in is only possible if the history of the graph is managed. When this dissertation started, the Thing'in platform was not designed to support the temporal dimension, and only the last updated state of the graph was maintained. Hence, our main goal is to design a temporal graph management system and integrate it into the Thing'in platform. In the following, we summarize and analyze the previous research on the management of temporal graphs and outline the key advances in this field of study that we have build on to design our own system.

Temporal graph management systems

Extensive work has been carried out on temporal databases. These efforts started in the 1980s by extending relational databases with time such as modelling temporal tuples [59], temporal relational models [34], temporal algebra [69] and other system design considerations such as storage and access techniques [208].

There is a growing recognition of the importance of managing temporal graphs, which is reflected in the technical literature on temporal graph systems. Some of these solutions are based on developing a temporal layer on top of an existing non-temporal

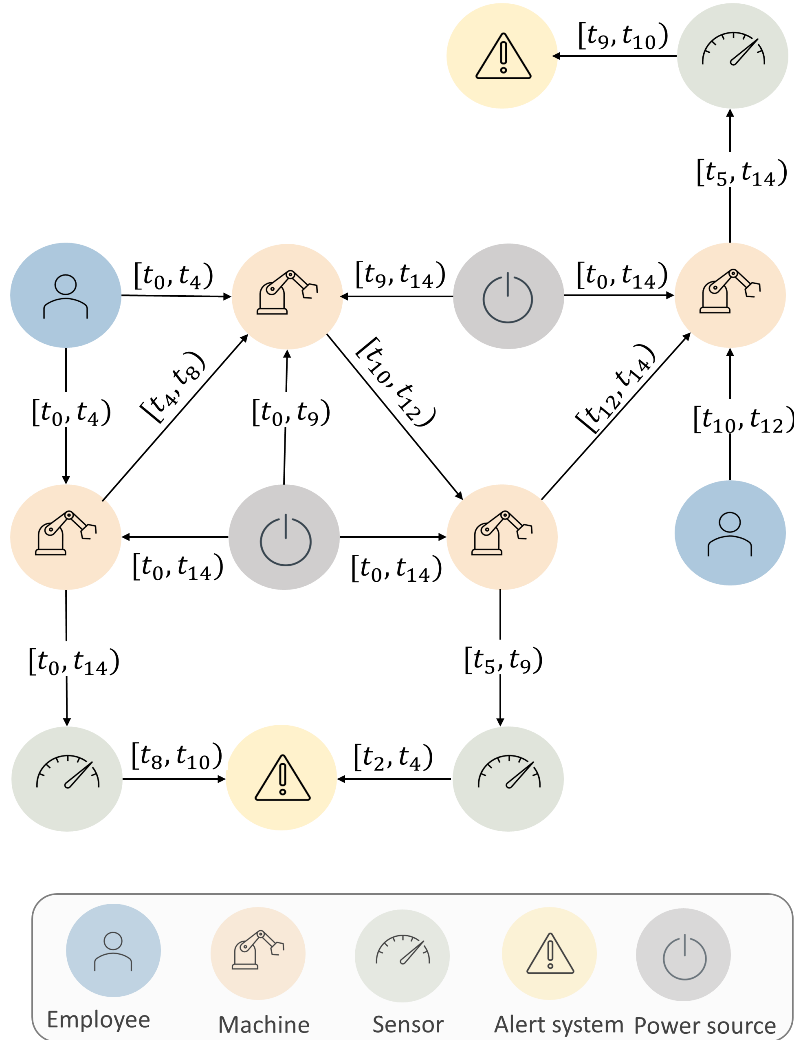


Figure 1.3 – A graph modeling the connections in a smart factory

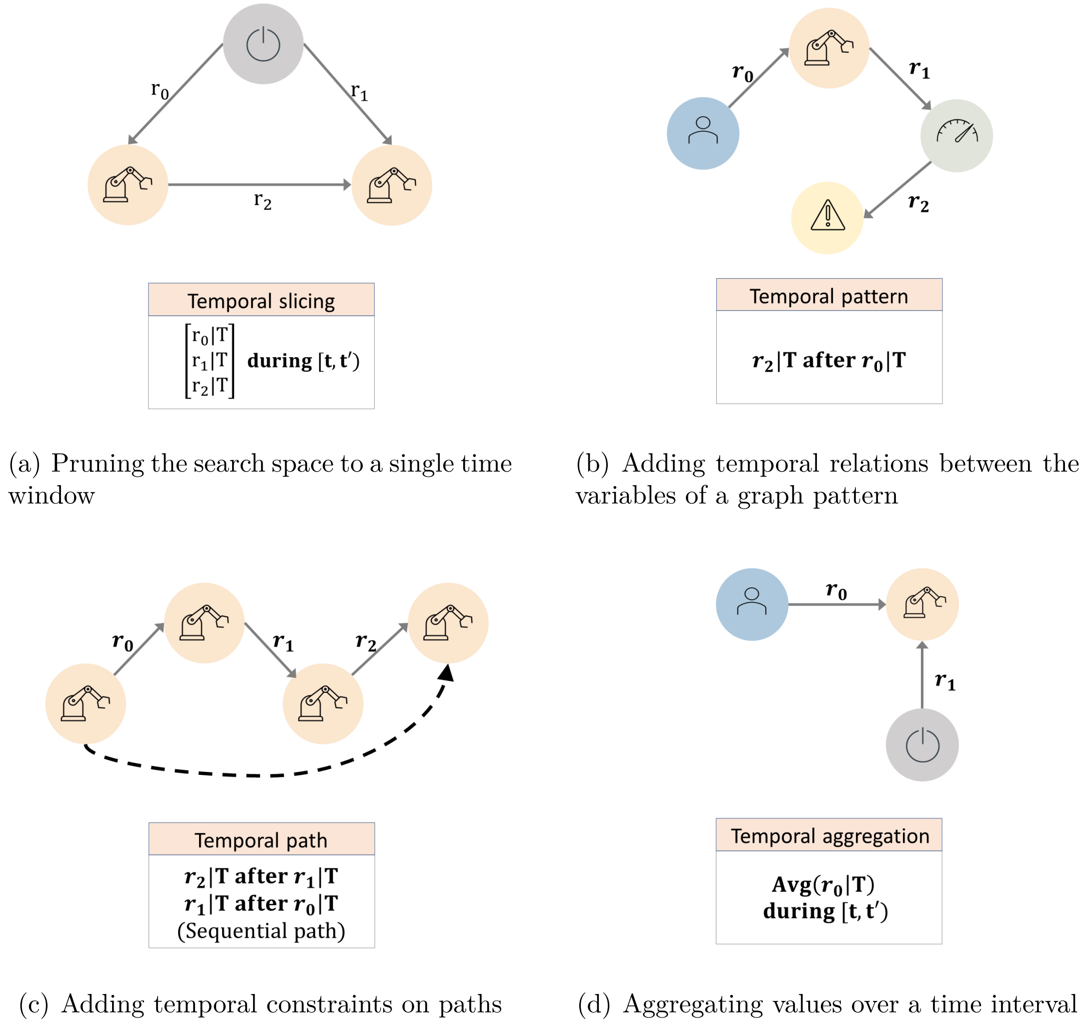
graph database [141, 46, 48, 124, 116], whereas other solutions build a system from the ground up [219, 112, 139, 165, 142, 111, 223]. These systems either offer temporal extensions of OLAP queries [219, 165, 223] such as iterative graph computing queries (e.g., shortest paths, diameter, and page rank) or OLTP queries [139, 142, 111, 223] such as constructing the state of the graph or the local neighborhood of a node at a time instant or during a time interval. These systems focus on optimizing the storage of temporal graphs but do not offer complex temporal graph queries such as time-extended graph patterns or paths or a language to express these queries. In this dissertation, however, we not only consider the system design characteristics but also

require expressive temporal graph queries extending traditional graph pattern matching and navigational queries with time which can be very useful for the users of Thing'in. This motivated us to build our own temporal graph system Clock-G with the goal of integrating it into the Thing'in platform. We present the different aspects of the design of Clock-G in the following.

Temporal graph querying

Many temporal graph querying solutions were proposed in the literature, which extends traditional queries with the temporal dimension. Some of these solutions extend OLAP queries with time such as finding most durable connected components [217, 169], temporal shortest paths [125] and temporal centrality [183], whereas others extend OLTP queries such as temporal graph pattern matching [65] and temporal navigational queries [194, 16]. As the Thing'in platform offers OLTP queries, we limit our research to this querying category and keep OLAP queries for future work. To illustrate the key temporal graph querying functionalities that we found in literature, we present examples of temporal graph queries in Figure 1.4 that can be applied to the smart factory graph illustrated in Figure 1.3. The **temporal slicing** queries (Figure 1.4(a)) return all the subgraphs satisfying a given pattern during a selected time interval, such as returning all the pairs of machines supplied by the same power source during time interval $[t, t')$. The **temporal graph pattern matching** queries (Figure 1.4(b)) return the subgraphs which satisfy a pattern with temporal predicates between the graph variables (nodes and relationship variables), such as the sensors connected to a machine that signals a system failure after the maintenance of that machine. The **temporal path queries** (Figure 1.4(c)) return all paths satisfying given temporal relations between the relationships, such as the sequential occurrence of consecutive relationships. For example, a path returning a sequence of product transferring actions between the machines is a sequential path. We also define further types of temporal paths that we present later in Chapter 4. The **temporal aggregation** queries (Figure 1.4(d)) return a value that is computed over a time interval by aggregating the information related to a node, its relationships, or neighboring nodes. As presented in the figure, the temporal aggregation computes the average duration of maintenance of a machine connected to a power source in the range of time $[t, t')$.

We list the illustrated key querying functionalities $\boldsymbol{R_1}$, $\boldsymbol{R_2}$, $\boldsymbol{R_3}$, and $\boldsymbol{R_3}$ as follows:



(a) Pruning the search space to a single time window

(b) Adding temporal relations between the variables of a graph pattern

(c) Adding temporal constraints on paths

(d) Aggregating values over a time interval

Figure 1.4 – Temporal graph querying

— R_1 (Temporal slicing): Returning a pattern valid during a given time interval (Figure 1.4(a)).

— R_2 (Temporal graph pattern matching): Returning a pattern with temporal predicates between the graph variables (Figure 1.4(b)).

— R_3 (Temporal path): Returning the pairs of nodes connected via temporal paths, which includes temporal constraints on the relationships of the path (Figure 1.4(c)).

— R_4 (Temporal aggregation): Returning an aggregated value of a node based on the information (e.g., property values) on the node itself, its relationships or neighbors in a time interval (Figure 1.4(d)).

Regarding the importance of such queries for our use case, we propose the tempo-

ral graph query language **T-Cypher** (presented in Chapter 4). We integrate T-Cypher into Clock-G by proposing a query processor (presented in Chapter 6). Our proposed query language extends the existing and well-known query language Cypher [86] with temporal constructs tailored to allow the expression of temporal pattern matching and navigational queries. Our extension is conservative, meaning that we add new grammatical rules without modifying existing ones, making it easier for practitioners who are already familiar with the language to reason about and learn the new syntax. Furthermore, T-Cypher is declarative in that it does not make any assumption about the underlying temporal graph system, which has the advantage of being portable on different temporal graph systems.

Despite the efforts initiated in the GQL [4] manifesto of May 2018 to define a standalone query language for graph databases, this project has not come to fruition by the time of writing this thesis manuscript (October 2022). Hence, we chose to extend Cypher because it is one of the languages that GQL is inspired by. Besides, Cypher has gained wide popularity in the graph database community because of its user-friendly syntax and expressive queries. It should be noted that T-Cypher is already being used in production such that Thing'in enables the use of the T-Cypher language to express temporal queries.

Storage of temporal graphs

Besides querying languages, we also address the challenge of managing the storage of temporal graphs. Many existing temporal graph management systems focus on this design characteristic which conducted many storage methods. Within this, we define a taxonomy to categorize the available storage methods.

The *Log* approach [94, 93] consists of preserving all the graph updates as a series of timestamped logs. The *Copy+Log* [138] approach consists of storing the graph updates in time windows, such that each time window contains a fixed number of graph operations. These time windows are stored along with snapshots representing the state of the graph at the start of each time window and are used as starting points for query evaluation. We illustrate the *Copy+Log* technique in Figure 1.5 where the graph updates are stored in time windows $\{\omega^1, \ldots, \omega^5\}$ and copies of the state of the graph are stored in snapshots $\{S^0, \ldots, S^6\}$. Each time window contains the same number of

4. https://www.gqlstandards.org/

graph updates ($N = 3$) and starts with a full graph snapshot that represents the state of the graph at the starting time instant of the time window. Despite the advantage of the *Copy+Log* method in pruning the search space, storing full graph snapshots is space-consuming, particularly in the case of growth-mostly graphs. The limitation of this method is that unchangeable graph entities will be copied across snapshots, causing a large volume of redundant graph entities. Whereas the *Log* approach has a detrimental impact on the query evaluation time. Hence, a compromise between both methods is needed, which is mainly addressed in this dissertation. To overcome this, we propose the δ-*Copy+Log*, a new storage method for temporal graphs that we detail in Chapter 5.

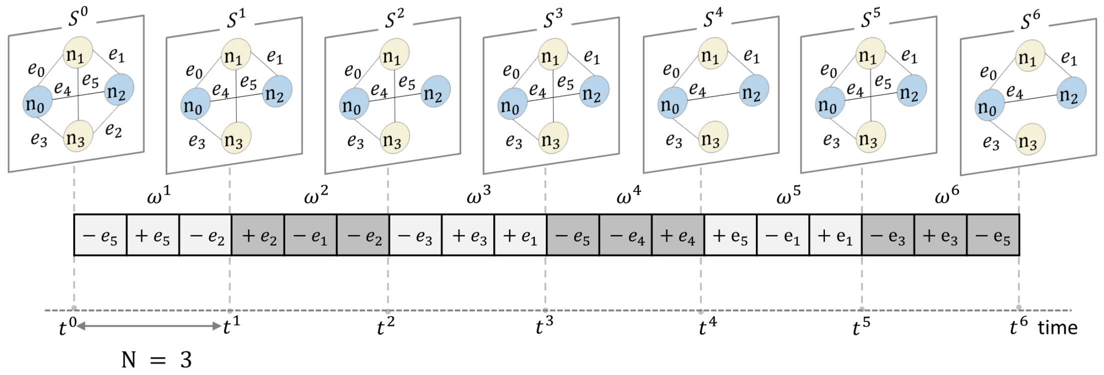


Figure 1.5 – Illustration of the *Copy+Log* technique representing the storage of a temporal graph using snapshots $\{S^0, \ldots, S^6\}$ and time windows $\{\omega^0, \ldots, \omega^6\}$

The δ-*Copy+Log* method stores all graph updates in time windows. Instead of storing full snapshots along with these time windows, it stores deltas which contain only the difference between two successive snapshots. To clarify, a delta contains all the graph updates included in a time window and not canceled by another graph update. For instance, an addition of a graph element is canceled by a deletion of the same graph element, thus, not stored in a delta. Besides deltas, a snapshot is stored for each M time window that is considered a starting point for query evaluation. That is, M represents a configurable parameter that can tune the performance of the δ-*Copy+Log* method. We evaluated this storage technique with basic temporal queries such as retrieving the graph's state or the node's neighborhood during a time interval or at a time instant. Then, we compared the results with those obtained from adopting the *Log* and *Copy+Log*. The results prove the efficiency of our proposal as they demonstrate that it can significantly reduce the storage cost compared to the *Copy+Log* tech-

nique whereas adding a slight query execution time overhead as compared to the *Log* method.

Processing of temporal queries

After defining the query language and storage technique, we steered our interest toward the processing of temporal queries. Our processing pipeline consists of parsing a T-Cypher query into a recognizable query object. This query object is then converted into an evaluation plan composed of algebraic operators. Each operator will then be executed against the backend store to find the final result. The best evaluation plan is greedily selected based on a cost model. Our cost model relies on cardinalities given by the backend store. The fundamental difference between our query processor and traditional ones is that we consider the optimal plan to change within the requested time interval. Meaning that a plan that is optimal for a time interval might not be optimal for another one since the cardinalities change over time which changes the cost of evaluation plans. To capture this, the system should keep track of the evolution of the cardinalities instead of a single state. Hence, we propose to preserve the history of the cardinalities in adequate data structures that return the cardinalities of each queried graph entity based on the requested time interval.

We implemented this query processor in Clock-G (as presented in Chapter 6) to enable the evaluation of T-Cypher queries. To evaluate our implementation, we executed several queries ranging from simple single-hop queries to complex pattern-matching ones using large-scale synthetic datasets. Besides, we implemented an alternative solution based on a non-temporal graph database by developing a temporal layer on top of it. Then, we compared the performance of Clock-G and the alternative technique by executing the same queries using the same temporal datasets. The obtained results validate the efficiency of our cost model and the superior performance of Clock-G compared to the alternative solution. This motivates our choice of building a temporal graph system from the ground up instead of relying on an existing non-temporal one.

Temporal graph generation

Synthetically generated graphs are usually used to evaluate different performance metrics of graph systems. In this dissertation, we are interested in the generation of temporal graphs that can be used to assess the performance of temporal graph systems. Available graph generation techniques try to mirror the characteristics of real graphs such as controlling the degree distribution [9, 49, 150, 140, 72, 19] or community structure [120, 134, 132]. None of the temporal graph generators controls the evolution of the degree distribution while maintaining a community structure. We argue that these two characteristics give practitioners the flexibility to model their temporal graphs based on their application demands. For instance, a power law degree distribution of a graph can tend more slowly to zero as time elapses since more nodes with high degrees are created. To capture this evolution, we propose to extend the generative model named Chung-Lu [56] that was originally posited to produce static graphs with a given degree distribution, as presented in Chapter 7. This extension produces temporal graphs such that the degree distribution of each graph snapshot approximates the degree distribution constructed from input parameters.

Furthermore, we assume that most real-world graphs gradually evolve, which implies that successive graph snapshots share a lot of commonalities. We integrate an optimal transport solver into our graph generator to capture this characteristic. This optimal transport solver minimizes the number of graph updates needed to generate a new graph snapshot from a previous one. Our generator can also produce time-varying attributes and types on the nodes and relationships. Indeed, real-world graphs are usually enriched with various node and relationship attributes. Nodes and relationships may be of different *types* and have several *properties* with values that may change over time.

Contributions

The main contributions of this thesis are the following:

- Proposing T-Cypher, a temporal graph query language that allows writing complex temporal queries with a user-friendly syntax.
- Proposing δ-*Copy+Log*, a storage technique for temporal graphs that can mitigate the space and evaluation time trade-off induced by traditional storage tech-

niques *Log* and *Copy+Log*.

— Developing a query processing engine that can evaluate T-Cypher queries. This query engine is optimized with a query planner that can greedily select the best evaluation plan by setting the order of algebraic operators based on the requested time interval.

— Proposing RTGEN, a temporal graph generative tool that produces the evolution of the degree distribution while maintaining a community structure that cannot be achieved by any available graph generator.

Outline

The thesis is organized as follows. Chapter 2 discusses the related work on graph management systems to set the stage for introducing the management of time in these systems. We describe the prior work in temporal graph management, such as presenting the different temporal graph models, storage techniques, and query languages. We also survey the literature on the available generative models for static and temporal graphs. Throughout the chapter, we highlight the limitations of the related work and motivate our choice to propose new techniques to tackle them.

Chapter 3 briefly describes the architecture of our temporal graph management system Clock-G and highlights in which chapters each component of our system is presented.

Chapter 4 presents our novel query language T-Cypher that we integrate into Clock-G to enable temporal graph querying. This chapter describes the temporal query constructs and the syntax of the language by giving its grammar rules. It also presents the details of the industrial implementation of T-Cypher in the Thing'in platform.

Chapter 5 presents our space-efficient storage technique that we implement in Clock-G. We provide some preliminary concepts, such as the temporal graph model that we refer to throughout the chapter, to define key concepts of our proposal. We then present a detailed description of our storage technique, followed by a complexity analysis. We give the implementation details of our technique in Clock-G. Besides, we define a set of basic graph queries that we used to evaluate the performance of Clock-G. In this evaluation, we compare the performance of Clock-G with that of a non-temporal graph database to demonstrate that our technique outperforms the alternative solution with particular types of temporal queries.

In Chapter 6, we introduce the query processor that we integrate into Clock-G to evaluate T-Cypher queries. We first provide a general overview of the processing pipeline. We then define a temporal graph algebra in which we use special graph operators extended with the notion of time. We propose a cost model that allows the processor to estimate the cost of a query plan. We also provide the greedy algorithm we use to choose an optimal execution plan. Since the cost estimation of a T-Cypher query depends on the requested time interval, we maintain temporal histograms instead of single-state histograms. We also show the results of evaluating our query processor, which validates the efficiency of our greedy algorithm and cost model.

Chapter 7 describes our temporal graph generator RTGEN. We describe in this chapter our relative graph generation technique that can produce temporal graphs while approximating the given evolution of degree distributions and community structure. This technique also minimizes the effort of transforming a graph snapshot into another one. We define metrics to measure the distance between the generated graphs and the desired parameters. Finally, we evaluated our generator's results while varying the input parameters.

Finally, we conclude the thesis in Chapter 8 with a summary of achieved results and the directions for future work.

Chapter 2

Related work

Designing a data management system from the ground up implies studying different key elements: data model, query language, system considerations such as physical storage layouts, and query processing techniques. This chapter introduces the prior work on the essential elements in graph data management systems design and the graph generators used to benchmark these systems. This introduction gives the necessary background for the design of temporal graph data management systems and generators, which is also discussed in this chapter. We identify the limitations of the related work on temporal graph management and outline our proposals to overcome these limitations.

We organize this chapter as follows: we first set the baselines by introducing the different aspects of graph management, including conceptual modeling, defining graph models and query languages, and system design, such as storage techniques and query processing. We then discuss temporal graph management by presenting temporal graph models and query languages, and system design methods. We also discuss the existing static and temporal generative models. Throughout the chapter, we discuss the limitations of these methods addressed in this dissertation.

2.1 Graph management

In this section, we describe the building blocks of a graph management system. The graph-oriented data model has received wide acceptance due to its natural representation of relationships. Hence, many academic and commercial graph databases have been developed, namely Amazon Neptune [12], ArangoDB [172], Blazegraph [30], CosmosDB [18], DataStax Enterprise Graph [73], HANA Graph [209], JanusGraph [127], Neo4j [175], Oracle PGX [181], OrientDB [123], TigerGraph [102], etc. This diversity creates an interoperability challenge, resulting in multiple graph models,

query languages, and other design considerations. In this section, we present and summarize the core features of a graph database design and compare different available graph databases based on these features. Indeed, this study is essential to integrating the temporal dimension into graph management systems.

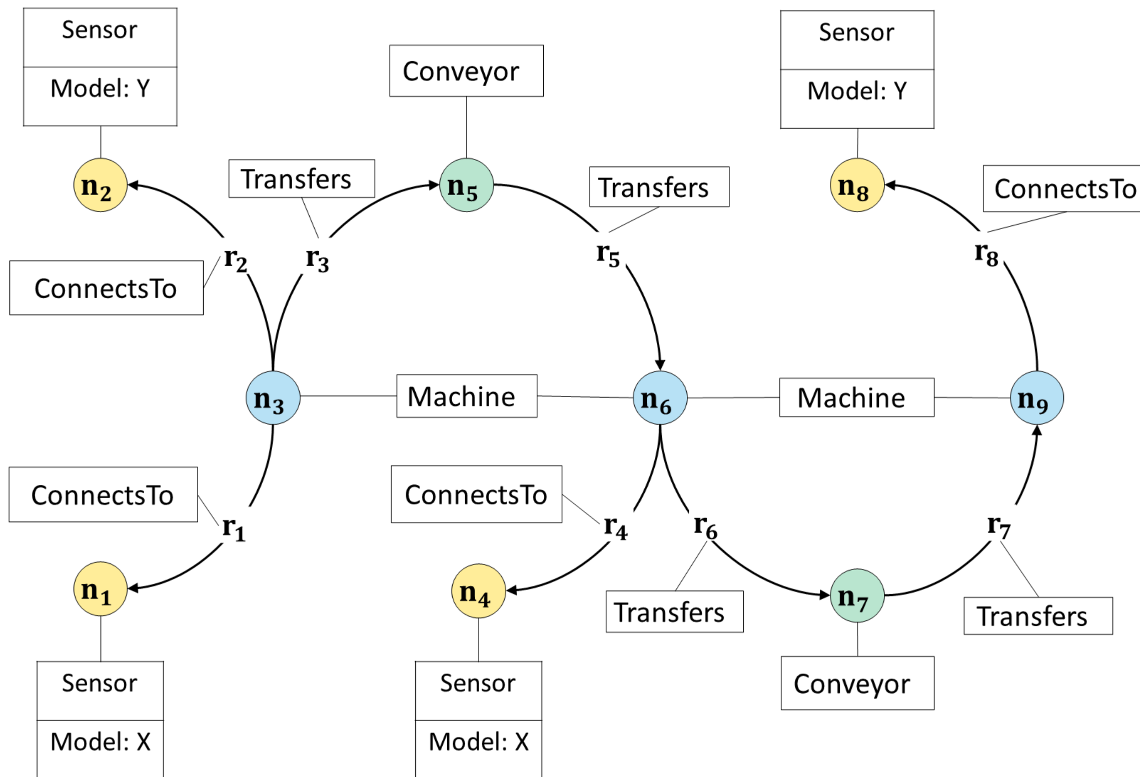


Figure 2.1 – Example illustrating a property graph.

2.1.1 Graph models and querying

Graph models

Multiple graph models were proposed in academia and industry. The Resource Description Framework [1] (RDF) and property graph data models are the two most commonly used models. This list is, of course, by no means exhaustive. For instance, some industrial solutions map graph datasets onto the relational model and store them in

1. https://www.w3.org/RDF/

relational database systems. However, we use two specific data models that gained wide acceptance in the community and are arguably more intuitive for graph-shaped datasets than the relational model.

RDF is a standard for describing the metadata of data models introduced by the World Wide Web Consortium (W3C) [149]. It is used to generalize the description of web resources by referring to an SPO (Subject; Predicate; Object) model where the labels of nodes represent a subject and an object of a fact, while the label on the relationship represents a predicate. The elements are uniquely identified using a **U**niform **R**esource **I**dentifier (URI) and can hence be used across different applications. Many languages have been proposed to query RDF graph stores such as SPARQL [113] and RQL [135].

Labeled directed property graphs In this dissertation, we are particularly interested in the labeled directed property graph model. This model underlies most commercial graph databases in the market [13] and many formal definitions of this model have been proposed [86, 15, 204, 233]. A transformation between this model and RDF graphs was proposed in [115, 114]. These efforts aim at reconciling both models and showing that property graphs could be queried using RDF query languages such as SPARQL. However, we mainly focus this work on property graphs and limit our research to this particular model.

We outline the property graph model based on [86]. Consider the following notations: Let V denote an infinite set of atomic values that can have any type from a finite set of data types D (e.g., string), L denote a finite set of strings, 2^X denote the set of all finite subsets of the domain X. Following this, we define a labeled property graph as the tuple:

$$G = \{N, R, P, \alpha, \rho, \tau, \theta\}$$

— N denotes the set of node identifiers.
— R denotes the set of relationship identifiers.
— P denotes the set of property keys that a node or relationship can have.
— α: $R \rightarrow (N \times N)$ assigns for each relationship its source and target identifiers.
— ρ: $N \rightarrow 2^L$ assigns a node with a finite set of labels from L.

— τ: $R \rightarrow L$ assigns a relationship with a label from L.
— θ: $(N \cup R) \times P \rightarrow V$ assigns a node or relationship identifier and a property key to a value.

It should be noted that its is more standard to use the terms “node“ and “edge“ instead of “node“ and “relationship“. However, this terminology is used in [86] which presents the query language and data model we extend with the temporal dimension.

Example 2.1.1. For illustration, we apply this formalism to the graph in Figure 2.1. This graph models the different interactions in a smart factory where nodes refer to machines, conveyors, or sensors. The relationships refer to the connections between sensors and machines or the transfers of products between machines and conveyors and vice versa. In the following, we show how, **for a sample of nodes and relationships**, this graph can be represented using the above definitions:

— $N : \{n_1, \dots, n_9\}$;
— $R : \{r_1, \dots, r_8\}$;
— $P : \{\text{Model}\}$;
— $\alpha : r_1 \rightarrow (n_1, n_3), r_2 \rightarrow (n_3, n_2), r_3 \rightarrow (n_3, n_5)$;
— $\rho(r) = \begin{cases} \{\text{Machine}\}, & \text{if } r \in \{n_3, n_6, n_9\} \\ \{\text{Sensor}\}, & \text{if } r \in \{n_1, n_2, n_4, n_8\} \\ \{\text{Conveyor}\}, & \text{if } r \in \{n_5, n_7\} \end{cases}$;
— $\tau(r) = \begin{cases} \text{ConnectsTo}, & \text{if } r \in \{r_1, r_2, r_4, r_8\} \\ \text{Transfers}, & \text{if } r \in \{r_3, r_5, r_6, r_7\} \end{cases}$;
— $\theta : (n_1, Model) \rightarrow X, (n_2, Model) \rightarrow Y, (n_8, Model) \rightarrow Y$;

Graph querying

In this section, we present the available query languages and core querying functionalities. We provide two types of graph queries: Navigational and graph patterns representing the core functionality of graph queries. Our goal in this dissertation is to extend these core functionalities with the temporal dimension.

Graph Query languages Despite their wide use, graph databases still lack a standard query language such as SQL [163] that represents the standard query language for relational databases. Many graph query languages have been proposed namely

Cypher [86], PGQL [200], G-Core [14] and Gremlin [203] developed by Neo4j [175], Oracle [180], Linked Data Benchmark Council (LDBC) [76] and Apache societies, respectively. Many attempts have been initiated to develop a new standard graph query language, such as OpenCypher and LDBC G-Core. However, none of these attempts has yet led to a standard query language. The GQL (Graph Query Language) [2] is an upcoming international standard language for property graph querying currently being developed. The idea of a standalone graph query language to complement SQL was raised by ISO SC32/ WG3 members in early 2017 and is echoed in the GQL manifesto of May 2018. However, this project is not fully accomplished by the time of writing this thesis manuscript (October 2022).

The available graph query languages differ in their syntax and semantics and do not share a common underlying formalism [15]. However, most graph query languages share the same core features: Navigational and Graph pattern matching. It is notable that graph querying also includes shortest paths [27, 67], graph clustering [210], and graph construction [128, 154]. However, navigational and graph pattern-matching queries are the most commonly used.

In the following, we describe the regular path and conjunctive regular path queries, which form the basis of navigational and pattern-matching queries. We refer to the toy graph presented in Figure 2.1 to explain the semantics of each of the queries.

Regular Path Queries A Regular Path query (RPQ) returns the pairs of nodes having at least a single path where the sequence of the relationships forms a word in the language of the regular expression defined over the set of relationship types Γ [24, 164].

A regular path query is recursively defined as follows:

— If $t \in \Gamma$, then $t \in$ RPQ.
— If $x \in$ RPQ, then $(x)^{-} \in$ RPQ.
— If $x, y \in$ RPQ, then $(x)|(y) \in$ RPQ.
— If $x, y \in$ RPQ, then $(x).(y) \in$ RPQ.
— If $x \in$ RPQ, then $(x)^{*} \in$ RPQ.

Consider the graph $G = \{N, R, P, \alpha, \rho, \theta\}$, the semantics of an expression $e \in$ RPQ denoted as $[\![e]\!]_G$ is recursively defined as follows. Note that all the following examples of RPQs are applied to the toy graph of Example 2.1.1 (Figure 2.1).

2. https://www.gqlstandards.org/

— If $e = a \in \Gamma$, then $[\![e]\!]_G = \{(u, v) \mid \exists r \in R$ s.t. $\alpha(e) = (u, v)$ and $\rho(e) = a\}$.
(e.g., $[\![$Transfers$]\!]_G$ = $\{(n_3, n_2), (n_5, n_6), (n_6, n_7), (n_7, n_9)\}$).
— If $e = (x)^-$, then $[\![e]\!]_G = \{(v, u) \mid (u, v) \in [\![x]\!]_G\}$.
(e.g., $[\![(Transfers)^-]\!]_G = \{(n_2, n_3), (n_6, n_5), (n_7, n_6), (n_9, n_7)\}$.
— If $e = (x)|(y)$, then $[\![e]\!]_G = [\![x]\!]_G \cup [\![y]\!]_G$.
(e.g., $[\![(: Transfers)|((: Transfers)^-)]\!]_G = \{(n_3, n_2), (n_5, n_6), (n_2, n_3), (n_6, n_5), (n_6, n_7), (n_7, n_9), (n_7, n_6), (n_9, n_7)\}$.
— If $e = (x).(y)$, then $[\![e]\!]_G = \{(u, v) \mid \exists z \in N(u, z) \in [\![x]\!]_G$ and $(z, v) \in [\![y]\!]_G\}$.
(e.g., $[\![(: ConnectsTo)^-.(: Transfers)]\!]_G = \{(n_2, n_5), (n_1, n_5), (n_4, n_7)\}$.
— If $e = (x)^*$, then $[\![e]\!]_G = \{(u, v) \mid \forall(u, v) \in TR([\![x]\!]_G)$ s.t. TR(R) denotes the transitive closure of binary relation R $\}$.
(e.g., $[\![(: Transfers)^*]\!]_G = \{(n_3, n_5), (n_3, n_6), (n_3, n_7), (n_3, n_9), (n_5, n_6), (n_5, n_7), (n_5, n_9), (n_6, n_7), (n_6, n_9), (n_7, n_9)\}$.

Conjunctive regular path queries Graph databases use RPQs to allow the expression of navigational queries. However, such queries cannot express graph pattern matching, such as finding the subgraphs with cycles or branching paths on intermediate nodes. Hence, CRPQ (Conjunctive Regular Path Query) has been proposed [22, 23, 25] to allow the expression of graph patterns. CRPQs, also known as pattern subgraph matching queries, identify substructures in a graph. In other terms, these queries ask for all subgraphs that match a given graph pattern. In CRPQs, conjunctions intersect the results between pairs of RPQs. A query pattern is expressed as a set of path predicates where a path predicate is a tuple consisting of an RPQ and a pair of node variables. A CRPQ is expressed with the following syntax:

$$(w_1) \dots (w_m) \leftarrow e_1(u_1, v_1), \dots, e_n(u_n, v_n)$$

where

— $m \geq 0, n \geq 0$,
— $u_1, v_1, \dots, u_n, v_n \in N$,
— $e_1, \dots, e_n \in RPQ$,
— $\forall 0 \leq i \leq m, w_i \in \{u_1, v_1, \dots, u_n, v_n\}$.

The following CRPQ query returns the machines connected to a conveyor and a sensor.

$$(m, c, s) \leftarrow: \text{Transfers}(m, c), : \text{ConnectsTo}(m, s)$$

This query, applied to the toy graph of Example 2.1.1 (Figure 2.1), returns: { (n_3, n_5, n_1), (n_3, n_5, n_2), (n_6, n_7, n_4)}. Note that extensions of CRPQs were proposed such as UCRPQs to allow the union of CRPQs.

Core querying functionalities in practice Concepts of conjunctive regular path queries continue to hold in modern graph querying languages such as Cypher [86], PGQL [200], G-Core [14], SPARQL 1.1 [113]. Besides, the design of a standardized query language for property graphs (GQL [3]) emphasizes the support of RPQs.

Some of the current query languages partially support CRPQs. Cypher [86] partially supports UCRPQs (union of CRPQs). However, the transitive closure operator is restricted to a single repeated relationship type. PGQL [200] partially supports CRPQs but does not allow their union (UCRRQ). Besides, Cypher and PGQL restrict the path length. Gremlin [203] supports CRPQs and allows specifying the number of times a traversal should be performed through the means of the **repeat** operator. SPARQL [113] supports CRPQs with negation and inverse. Besides, the design of a standard query language for property graphs, aka GQL, emphasizes the use of RPQs.

To achieve our goal of defining a temporal graph query language, we decided to extend Cypher among the existing non-temporal graph query languages discussed above. The rationale behind this choice is the user-friendliness of Cypher's syntax, making it very popular in the marketplace and familiar to a large fraction of practitioners. As previously mentioned, Cypher offers core graph querying functionalities. Besides, Cypher is a declarative query language that can be portable across different systems. It should also be noted that Cypher is built upon to define the syntax of the new standard language GQL.

2.1.2 System design

In this section, we discuss the implementation details of a graph system by addressing the storage and query processing methods. Although the main focus is on temporal graph system design, these methods form the basis of extending graph systems with temporal support.

3. https://www.gqlstandards.org/

Storage

In the following, we provide the most common storage layouts and backend stores that persist graphs.

Storage layout A static graph $G = (N, R)$ can be represented using an adjacency matrix. This is a $|N| \times |N|$ matrix, such that the element a_{ij} is defined as $a_{ij} = 1$ if $ij \in R$ and $a_{ij} = 0$, otherwise. This representation requires $O(n^2)$ memory where $n = |N|$. It can be seen that the storage required for this representation is independent of the number of relationships in the graph. Hence, there are no memory savings even for sparse graphs. However, there are numerous formats for storing sparse matrices in a compact form, some of them can be updated [41].

Another storage layout, referred to as the adjacency list layout, has been proposed to exploit the sparsity of real-world graphs. In this representation, an array of lists for each node is used to store a graph G. The list corresponding to a node $n \in N$ contains all nodes adjacent to n in G. For a directed graph, the space requirement for the lists is $O(m)$ where $m = |R|$. The total memory requirement is $O(n + m)$ where $n = |N|$. In the case of weighted graphs, the weight of each relationship uv is stored with the node v in u's adjacency list.

Backend store Some implementations rely on an existing data store, whereas others have chosen to rebuild their graph store from the ground up to perfectly adapt it to their specific needs. An example of DBMS building its own store is Neo4j. In many GDBMSs, NoSQL databases, namely Key-value stores, are used. Examples of such systems are: ArangoDB [172], DSE (DataStax Entreprise)[73], DGraph [68], CloudGraph [60], and Redis Graph [196].

Query processing

General pipeline The general pipeline of query processing, as presented in [36], starts with parsing the textual representation of a query, which also validates its syntactic correctness. Then, the parsed query is sent to the query optimizer, whose primary role is to compute, most of the time, heuristically, an approximation of the optimal query evaluation plan. The choice of the optimal plan is steered by statistics extracted from the database, such as the distribution of the node label values. These statistics help

estimate each plan's cardinality in the set of candidate plans. Besides, the cost of an execution plan is tightly coupled to the backend store, data layout, structure of indexes, etc. The query optimizer abstracts these access costs with variable access costs that only depend on the access methods of the database. After computing the execution plan, the query optimizer sends it to the database engine. The database engine first converts the execution plan into a set of physical operators and then executes each operator against the database.

In this dissertation, we follow the same query processing pipeline to evaluate temporal queries where we adapt each element to include the temporal dimension.

Graph algebra The evaluation of a query usually implies converting it into a set of operators defined by an Algebra. Several graph algebras were proposed in literature such as GraphQL [117] defined along the lines of the relational algebra and GRAD [96]. In this thesis, we will build on the algebra defined by Hölsch et al. in [126]. The main operators of this algebra are **getNodes** and **expand**, used to access the database nodes and relationships, respectively. Note that this algebra was originally proposed for Cypher queries. However, it can be generalized and used for other graph query languages. Hölsch et al. also discuss the equivalence rules that allow the optimization of queries. We present the operators of this algebra in the following.

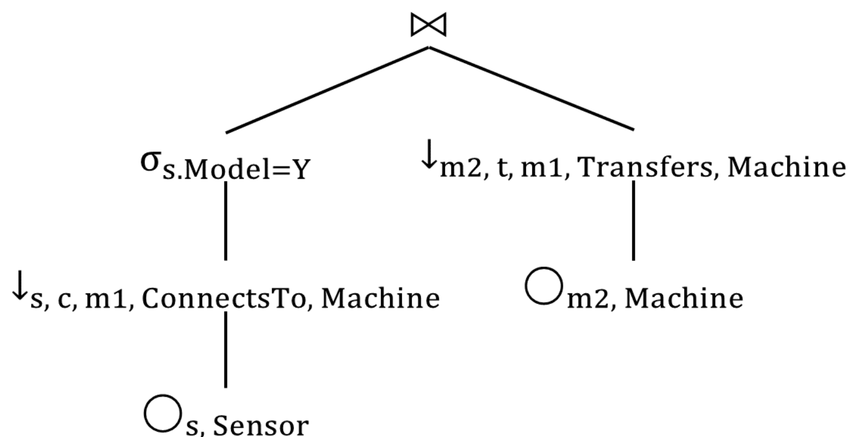


Figure 2.2 – Possible evaluation plan of the query **Q**

GetNodes operator $(\bigcirc_{a,\rho_a})$

This operator access all the nodes of a graph with a given label where a is the node variable, and ρ_a is the node label.

Expand operator $(\uparrow_{a,b,ab,\rho_b,\rho_{ab}}(E))$

This operator computes for the nodes given as input, the relationships that satisfy a given set of conditions where:

— a: is the name of the node in the input relation.
— b: is the name of the added target node.
— ab: is the name of the added relationship.
— ρ_b: is the label of the added target node b.
— ρ_{ab}: is the type of the added relationship ab.

This notation is used to denote expansion from node variable a to b. However, the notation $\downarrow_{a,b,ab,\rho_b,\rho_{ab}}(E)$ is used to denote expansion in the opposite direction.

Besides, this algebra also includes relational operators such as **join** or **select**. The join denoted $\bowtie$ is used to join the result of two algebraic expressions. The select operator denoted σ_ρ, is used to filter the result of an expression based on a condition ρ defined over the set of variables of the expression.

To illustrate, we present an example of an algebraic plan composed of the given operators. We consider the following query **Q** written in Cypher [86], which retrieves the sensors connected to a machine m_1 that transferred an object to another machine m_2. This query can be applied to the toy graph of Example 2.1.1 (Figure 2.1).

Query **Q**

```
MATCH (s:Sensor) <- [c:ConnectsTo] - (m1:Machine)
- [t:Transfers]-> (m2:Machine)
WHERE s.Model = Y
RETURN *
```

We present in Figure 2.2 one of the possible algebraic plans represented as a tree of operators used to evaluate this query. This plan is evaluated in a bottom-up fashion where leaf nodes are executed first. Each parent node receives the result of its child nodes and computes a new result.

We present in Figure 2.3, the result of evaluating each operator on the graph of Example 2.1.1 (Figure 2.1). Expression E_0 returns the sensor nodes of the graph. Expression E_1 expands the sensor node variable of E_0 with the relationship c and target

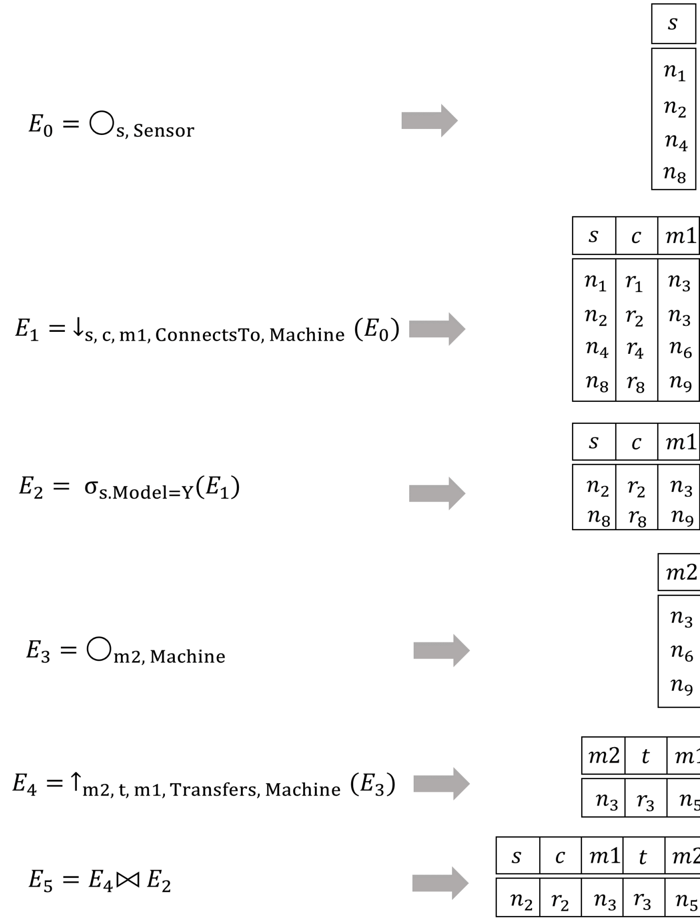


Figure 2.3 – The result of evaluating each expression in the plan of Figure 2.2

node m_1 variables. Expression E_2 filters E_1 based on the value of property Model. The returned result contains a single tuple satisfying the given condition. Similarly to E_0, the expression E_3 computes all the machine nodes of the graph and assigns them with the node variable m_2. Then, the expression E_4 expands the result of E_4 by adding to additional variables t and m_1, corresponding to the added relationship and target node, respectively. The final expression E_5 joins the results of E_4 and E_2 by matching the shared node variable m_1.

In principle, the expand operator is a possible way of expressing joins between the input relation and the underlying graph as presented in [158] where the graph operators defined in the graph algebra of Hölsch et al. [126] are transformed into relational operators. However, expressing paths (i.e., a succession of expansions) as a recursion of join operators forces an underlying relational model. Hence, defining an expansion

operator is more convenient since it does not restrict the underlying data model. In this dissertation, we extend these graph operators defined by Hölsch et al. with the temporal dimension as presented in Section 6.2.

Query planning The evaluation of CRPQs or subgraph isomorphism is proved to be NP-complete [91]. Another challenge of processing such queries is the random storage access which induces significant query latency. Query planning is an optimization technique used to limit access to secondary storage by predicting, at runtime, the best execution plan between various candidate plans. This query optimization is usually based on reducing the size of intermediate results. Optimizing query plans is challenging since various plans with varying costs may exist, even for simple queries. Database engines use heuristics-based and greedy optimization techniques to accelerate the computation of a query plan. Query planning was extensively studied in the literature of relational databases [226]. The graph algebra proposed by Hölsch et al. [126] presented above was used in many proposals around graph-oriented query processing. It is used in [104, 220] to propose query planning techniques for graph-oriented queries.

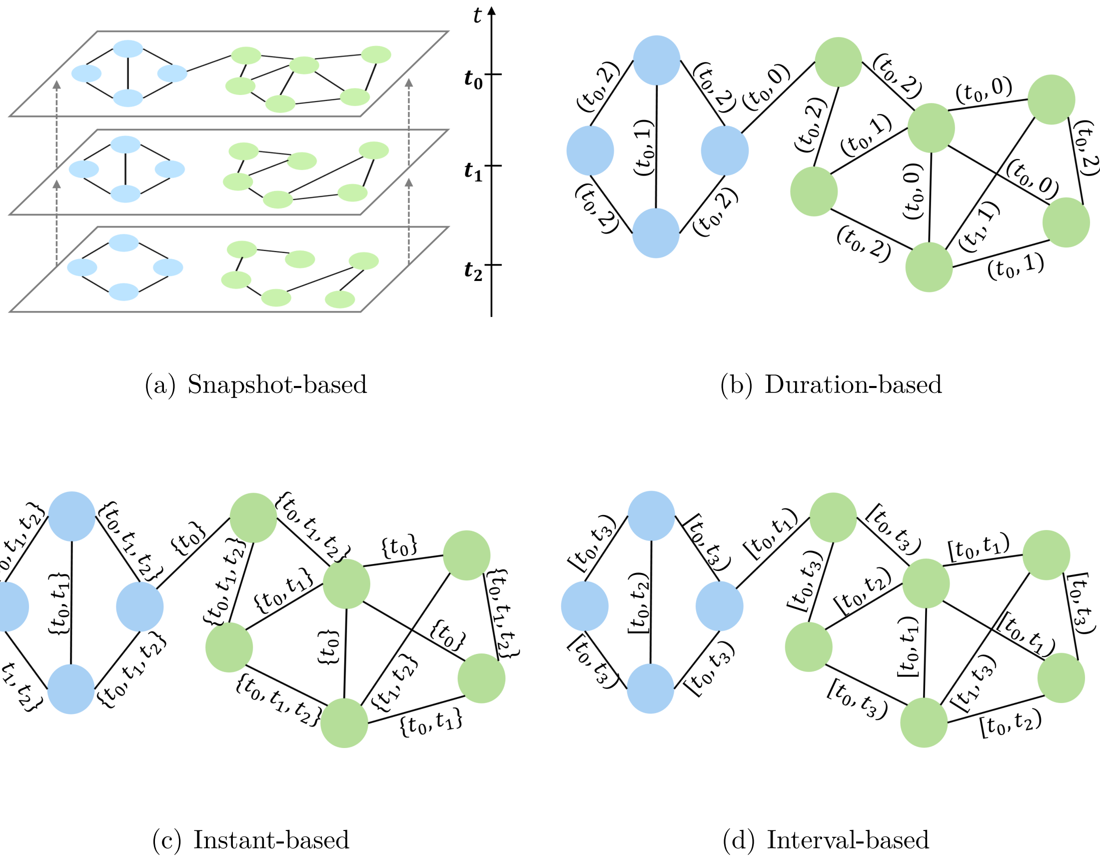
2.2 Temporal graph management

After introducing the fundamental concepts of graph management, we discuss the different approaches to extending these concepts with the temporal dimension in this section.

2.2.1 Temporal graph models and querying

Temporal graph models

Temporal graphs are not as intuitive as their static counterparts [121]. Hence many temporal graph models were proposed. These models differ in what temporal semantics they encode and what time representation they use (point-based or interval-based), what graph entities they annotate with temporal information (nodes, relationships, or property values), and whether they represent only structural evolution or also property evolution.



(a) Snapshot-based (b) Duration-based

(c) Instant-based (d) Interval-based

Figure 2.4 – Different temporal graph models

Snapshot-based temporal graphs Based on the Snapshot-based temporal graph model, a temporal graph is defined as the sequence of graph snapshots such that each graph snapshot represents the state of the graph that was valid at a time instant as seen in Figure 2.4(a). Each graph snapshot can be considered as a non-temporal graph. Regarding its intuitiveness and simplicity, this model has received wide acceptance in the research field [38, 146, 133, 199, 80, 138, 139, 111]. The models defined in [138, 139, 111] add properties to the nodes and relationships. Temporal queries can be easily evaluated if the temporal graph is modeled and stored as a sequence of timestamped snapshots. Despite being a widely accepted model, the fundamental disadvantage of using the snapshot-based model when defining temporal graph operators is that they cannot explicitly reference temporal information, as presented in the dissertation of Vera Moffitt in [166]. To overcome this, proposals have been made to assign nodes, relationships, or property values with time. These models will be presented

next.

Duration-based temporal graphs Based on the Duration-based temporal graph model, a graph entity is assigned with a starting time instant and a duration for which the entity persisted. For instance, this model can be used in telecommunication networks, where the duration of calls is the kind of temporal information needed to find paths with the minimum routing duration. It was defined in referred to in [237, 186] where only the relationships are annotated with time. An illustration of this model is given in Figure 2.4(b).

Instant-based temporal graphs Based on the Instant-based temporal graph model, a graph entity is assigned to a set of time instants. This model can be used when the temporal graph represents transactions in an e-commerce platform or a bank system. It was presented in [234] where only the relationships are annotated with time. In such use cases, we often have a graph in which each node represents a user account, and each relationship with a timestamp represents a money transaction between two user accounts. Hence, the connections (relationships) can be considered instantaneous (i.e., their duration can be neglected). An illustration of this model is given in Figure 2.4(c).

Interval-based temporal graphs Based on the Interval-based temporal graph model, each relationship is assigned to a time interval, a compact representation of a convex set of time instants. An illustration of this model is given in Figure 2.4(d). Note that, in this illustration we only annotate edges with time for simplicity. However, in this thesis we include temporal information on the nodes, edges and properties. An Interval-based model, including properties on nodes and relationships, was first proposed by Moffitt et al. in [167]. Authors of this work define a linearly ordered discrete time domain Ω^T where time instances have limited precision. Based on this model, a temporal property graph is defined as the tuple $G = (N, R, L, \rho, \psi^T, \lambda^T)$, where:

— N is a finite set of nodes, R is a finite set of relationships, $N \cap R = \emptyset$, and L is a finite set of property labels;

— $\rho : R \rightarrow (N \times N)$ is a total function that maps a relationship to its source and destination nodes;

— $\xi^T : (N \cup R) \times \Omega^T \times \Omega^T \rightarrow B$ is a total function that maps a node or a relationship and time interval to a Boolean, indicating existence of the node or relationship during that time interval; and
— $\lambda^T : (N \cup R) \times L \times \Omega^T \times \Omega^T \rightarrow val$ is a partial function that maps a node or a relationship, a property label, and a time interval to a value (val) of the property during that time interval.

We note that the duration and interval-based models are used when the relationships have a non-negligible duration. Both models are generalizations of the instant-based model where the duration is equal to zero in the former, and the start time instant is equal to the end time instant in the latter.

In this dissertation, we follow the interval-based time representation as proposed by Moffitt et al. in [167] to define our temporal property graph model as it represents graph evolution more compactly than storing each temporal state of the graph. Hence, we consider that each node and relationship existed during a set of validity intervals and has a collection of property names assigned each to a set of value/ validity interval pairs. Indeed, we consider that properties evolve over time which reflects a large fraction of real world graphs. In the toy graph of Example 2.1.1 (Figure 2.1), the type of connection (i.e. property of the relationship *ConnectsTo*) between a sensor and a machine can change over time from *Bluetooth* to *Zigbee*.

Temporal graph querying

This section presents the prior work on querying temporal graphs.

Local/Global-Point/Range Temporal graph queries can be divided into four quadrants: **Local-point**, **Local-Range**, **Global-Point**, and **Global-Range** queries [165]. For instance, local queries access a subset of the graph, whereas global queries access the whole graph. Point queries follow the instant-based time representation and retrieve either a local or global state at a given time instant. Range queries follow the interval-based time representation and retrieve either a local or global state valid during a time interval. These four types of queries are included in most temporal graph management systems [141, 138, 124, 139, 165, 142, 111]. However, each system defines different types of local and global queries. Most systems implement a simple form of a local query that represents the local or N-hop neighborhood of a node. Some systems also

refer to global queries as the ones retrieving the state of the whole graph, whereas others define such queries as analytical queries, retrieving, for example, the diameter or page rank of the graph.

To illustrate, we present the following temporal query examples that can be applied to the toy graph of Example 2.1.1 (Figure 2.1):

— **Local Point:** What was the state of a machine two days ago?
— **Local Range:** How many sensors were connected to a given machine during the last month?
— **Global Point:** What was the degree of all graph's nodes last year?
— **Global Range:** How was the connectivity evolving between the graph's nodes in the past five years?

Although these types of temporal queries have been widely adopted by available temporal graph management systems, we are interested in more complex graph queries such as graph pattern matching and navigational queries. Hence, we survey the research on extending such queries with the temporal dimension in the following. Recall that functionality $\boldsymbol{R_1}$ consists of pruning the search space of a query to given time instant(s) or time interval(s). Functionality $\boldsymbol{R_2}$ consists of expressing temporal predicates between the variables of a graph pattern. Functionality $\boldsymbol{R_3}$ consists of expressing temporal relations between the relationships of a path. Functionality $\boldsymbol{R_4}$ consists of expressing the temporal aggregations.

Temporal graph querying approaches As presented in Section 2.1.1, subgraph pattern matching or navigational queries form the core of graph querying. Hence, many proposals to extend these queries with the temporal dimension were posited. In the following, we will present these approaches and outline their limitations.

Non-decreasing time flow pattern In this temporal graph pattern, each path between two nodes follow a non-decreasing time flow [184, 197, 244]. This type of temporal pattern is particularly interesting for studying the spread of a disease or analyzing the flow of rumors in a social network. However, the applicability of such querying approaches is narrowed to a particular use case. Besides, it does not offer the flexibility of defining any temporal order between the variables of a pattern.

Most Durable Graph Pattern (MDGP) This type of temporal graph patterns, defined by Semertzidis et al. in [214, 217, 215], returns the most durable matches of a given non-temporal pattern. This querying functionality is very useful for analyzing the tightness of connectivity between the graph nodes. However, the supported graph pattern is non-temporal (i.e., it does not include temporal predicates).

Temporal reachability queries Semertzidis et al. introduce in [216] conjunctive, and disjunctive reachability queries on a sequence of graph snapshots. In a conjunctive reachability path, the intermediate relationships of the returned path should be valid during the entire requested time interval. In a disjunctive reachability path, the relationships should co-exist at a single time instant (in a single temporal graph snapshot). **TopChain** [237] is another approach offering temporal reachability queries. In this proposal, a node is considered reachable from another if a time-respecting path exists between these nodes. Besides, it also supports the earliest arrival path and fastest path queries. Despite the utility of both approaches, they do not support subgraph pattern matching queries.

Temporal-based language for Software Defined Networks (SDN) A time-based language for graph databases is proposed in [130]. The authors propose a querying language for temporal and layered property graphs motivated by SDN, where nodes are grouped into layers. The language permits horizontal navigation (with intra-layer relationships) and vertical navigation (with inter-layer relationships). Furthermore, they added a special clause that can limit the search of a query to a single time point or time interval. The main limitation of this query language is that its semantics is restricted to a particular use case and cannot be generalized to account for other applications.

Granite A temporal navigational query is proposed in [194] and implemented in a distributed query engine called Granite. The temporal predicates are integrated to add a set of comparisons between a node/relationship/property lifespan and a given time interval. Besides, temporal ordering constraints can be added between the incoming and outgoing relationship variables of a single node variable. Besides, Granite includes temporal aggregations by grouping all the paths starting with the same node and aggregating the values of the properties of the last node in the path. The main limitation of

this proposal is that it does not allow the expression of graph pattern matching queries.

Temporal regular path queries (TRPQ) Arenas et al. propose a temporal extension of regular path queries (RPQs) in [16, 7].To the best of our knowledge, no temporal extension of RPQs have been proposed so far, and so the proposed language is particularly novel. The proposed language includes two orthogonal navigational operators: structural and temporal. The former corresponds to topological movements over the graph, and the latter corresponds to temporal movements over the graph. The language's syntax is based on the **Match** clause used by the modern graph query languages PGQL [200], Cypher [86], and G-Core [14]. This language allows the expression of temporal navigational queries such as time-increasing paths. It also allows limiting the scope of a query to a time instant or interval of interest by adding a temporal constraint to the path expression. Besides, the authors plan to include temporal aggregation in a future work. However, TRPQ does not offer the ability to express graph patterns since it focuses on the semantics of RPQs.

Temporal graph algebra (TGA) Moffitt and Stoyanovich propose in [167] a Temporal Graph Algebra (TGA), which presents temporal generalizations based on temporal relational algebra for some graph operators. Besides, authors of this paper present the system Portal implementing the operators of the proposed algebra. This algebra includes the **Trim** operator, which returns a result in which each node and relationship should exist at a time instant or during a time interval. Another possible operator is the **Subgraph** operator, which results in subgraphs that are isomorphic to a given pattern during a given time instant or interval. It differs from non-temporal subgraph pattern-matching queries by expressing temporal predicates. Besides, TGA supports user-defined functions allowing the definition of any type of temporal predicate between the variables of a pattern (including Allen's temporal relations). TGA proposes temporal **Aggregation**, which is used to compute a new value for a node based on information available at the node itself, its relationships, or its neighbors. However, TGA does not support path queries since it is compositional (i.e., closed under its operations).

T-GQL is a querying language for temporal property graphs proposed in [65]. Indeed, the language is based on GQL[4]. The core feature of T-GQL is the ability to express different types of temporal paths (Requirement $\boldsymbol{R_3}$). In a **continuous path**, all relationships should have an intersection between their time intervals. In a **pairwise path**, there must be an intersection between the time intervals of every pair of consecutive relationships. In a **consecutive path**, an outgoing relationship of each node should be valid after incoming relationships. Following this definition, different types of consecutive paths are defined, namely: earliest arrival time paths and fastest paths. Besides paths, the query language uses a temporal slicing operator to limit the search space of a query. The syntax of the T-GQL language is based on GQL such that it is SQL-like with a **Select-Match-Where** structure. The main limitation of this language is that it does not allow the expression of temporal predicates between the variables of a pattern. Besides, T-GQL includes temporal aggregations in the computation of the fastest or earliest arrival time paths. However, it does not allow explicitly expressing aggregations.

GRALA Gradoop [205] is a graph system that offers temporal querying functionalities using a declarative analytical language GRALA. The authors propose temporal operators to determine graph snapshots, the difference between two snapshots, and the subgraphs satisfying a given time-dependent graph pattern. GRALA supports temporal slicing (Requirement $\boldsymbol{R_1}$). It also offers the option of extracting the difference between two snapshots. This solution also offers the construction of a time-dependent graph pattern. The temporal extension of a graph pattern consists of defining temporal relations between the variables (nodes and relationships) of the pattern by partly using Allen's temporal algebra. We limit this description to the most relevant operators. However, Gradoop also offers temporal extensions of graph processing operators. This query is expressed using the temporalGDL, a temporal extension of the Graph Definition Language (GDL) [131] inspired by the syntax of Cypher.

Other approaches also use time as an additional semantic to graph queries. For instance, Aghasadeghi et al. define a temporal generalization of the node creation operator in [5, 7]. Indeed, this operator is present in several existing graph query languages [81, 118] and consists of creating nodes that represent property values. That is, the property value of a node in the input graph will result in the addition of a new

4. https://www.gqlstandards.org/

property node such that the original node will be connected to the newly created property node. The temporal generalization of this operator takes part of the previously presented temporal graph algebra (TGA) [167]. This generalization consists of creating nodes for each property value and binding this node with temporal information, such as the number of nodes having this value and the time interval during which this number was valid. It also consists of adding nodes to change the temporal resolution of the graph data. Such temporal quantifiers allow observation of strong connections over a volatile temporal graph. A similar operator is included in the temporal graph system Gradoop [205]. This so-called zooming functionality is worth mentioning. However, it falls outside the scope of this thesis and will be no further discussed in this survey.

Querying temporal RDFs Although this thesis mainly focuses on the property graph data model, we should not neglect the efforts to extend the RDF data model and query languages with time. Many solutions were proposed targeting the extension of the RDF query language SPARQL with time. The following solutions share common core features, such as temporal slicing. Besides, they allow the expression of temporal variables corresponding to the time interval of triple variables. Temporal relations were proposed using these temporal variables. Some solutions suggest using the operators of Allen's interval algebra [11], whereas others recommend using a subset of these operators. The key feature is the ability to express temporal relations between a pattern's temporal variables, such as comparing the time of occurrence of two triples in a query pattern. SPARQL-ST [189] extends SPARQL with query constructs allowing the expression of the spatial and temporal dimensions. Since the graph of Thing'in contains spatial information about the location of IoT objects and their surroundings, helping end-users track their objects, we plan to include the spatial dimension in T-Cypher in future work.

T-SPARQL [100] is a temporal graph query language for RDF that embeds the features of TSQL2 [221] (query language designed for temporal relational databases). To express temporal predicates over timestamp variables, the authors propose using a subset of Allen's temporal operators: Precedes, Equals, Overlaps, Meets, and Contains. Authors of [227] present **τ-SPARQL** which is another query language for temporal RDF stores. The expression of temporal relations between two temporal variables is enabled by using the operators of Allen's interval algebra. The authors present a technique to translate τ-SPARQL queries to SPARQL queries, enabling the execution

of temporal queries using any non-temporal RDF store that uses SPARQL. **SPARQL**^{T} is another query language for temporal RDF stores. Although this language does not offer dedicated temporal operators to express temporal relations, it is possible to use temporal functions that extract the starting and ending time instants of a tuple to express any of Allen's temporal relations.

Motivation Although a formal comparison of the different temporal graph querying solutions allows us to position our solution with respect to the state-of-art, such a comparison is out of the scope for this work. This thesis presents an alternative approach to querying temporal graphs that can express all the temporal functionalities $\boldsymbol{R_1}$, $\boldsymbol{R_2}$, $\boldsymbol{R_3}$, and $\boldsymbol{R_4}$. We choose to extend a well known graph query language that is familiar for a large fraction of practitioners. Our main goal is to propose a succinct and user-friendly syntax that is easy-to learn and makes reasoning about and writing temporal graph queries intuitive. This motivation lead us to the definition of T-Cypher (presented in Chapter 4), a temporal graph query language that extends the well known query language Cypher [86]

2.2.2 System design

Storage

Data locality Data locality is exploited by data management systems to store related data sequentially on disk or on the same server. Knowing that related data are likely to be retrieved together, the data locality reduces the query latency by reducing the number of reading operations from secondary storage or message exchanges between multiple servers in distributed architectures. We are particularly interested in the data locality applied to secondary storage in this work.

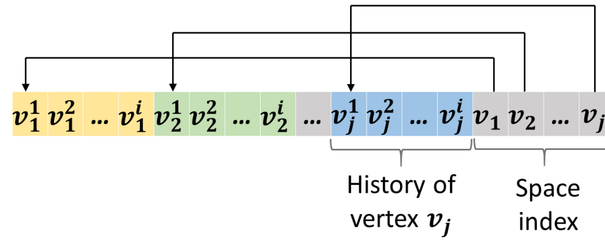
Graph management systems also exploit data locality in that they store the neighbors of a node, edge, or property sequentially on disk or in memory. When managing temporal graphs, the data locality is not only restricted to the structure of the graph. Hence, two possible data locality layouts derive: the structural and temporal localities [112, 165, 142]. The structural locality, referred to as structural or time-centric locality, consists of contiguously storing related nodes, relationships, or properties of a single snapshot sequentially on disk as previously described for non-temporal graphs. The

second, referred to as the temporal or entity-centric locality, consists of storing the consecutive states of a single node sequentially on disk or in memory. While the temporal locality can naturally fit concerning the linearity of time progression, the former, referred to as structural locality, is more challenging. Still, several approaches aim at approximating the physical closeness of nodes forming a neighborhood or a community. Figure 2.5(a) shows that each data block following the temporal locality stores the consecutive graph updates related to a single node sequentially. However, Figure 2.5(b) shows that each data block following the structural locality stores the consecutive graph updates related to the full graph sequentially.

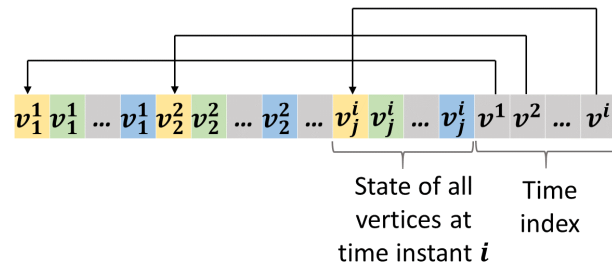
Indexing these structures consists of using a space index with a temporal locality file layout where each block is referred to by its node identifier and a time index with a structure locality where each block is referred to by its relative time range. ImmortalGraph [165] replicates the temporal graph such that one replica is stored following a structural locality, whereas another replica is stored following a temporal locality. When querying the graph, the system chooses which replica to read from based on the query type. For instance, a global query directs the search to the structural locality-aware replica, whereas a range local point directs the search to the temporal locality replica.

Underlying storage Some implementations of a temporal graph system rely on an existing backend storage system, whereas others (e.g., Immortal Graph [165]) build their own system from the ground up to adapt it to their needs. Some systems rely on a NoSQL database as a backend store. Some solutions are based on a commercial graph database such as Neo4j [175]. The following presents different types of backend stores used by available temporal graph systems.

Although a native graph store has been advocated in the graph community, a significant fraction of graph management systems uses a key/value store as a backend. Hence, it is unsurprising that some existing systems also build upon NoSQL databases as backend stores. However, the integration of time is not an obvious task which leads to several storage layouts. Authors of [138] have used, in their first proposition, the hierarchical index structure DeltaGraph, the key/value store Kyoto Cabinet [144] for the reasons of flexibility, fast retrieval times, and scalability. In their recent work TGI, they use the column-oriented database Apache Cassandra [147]. Hinode [142], and Rapthory [223] also rely on Cassandra as a backend store for the ease of graph representation, scaling capabilities with large datasets, and engineering maturity. Portal



(a) Temporal locality



(b) Structural locality

Figure 2.5 – Physical layouts of different data localities in temporal graph databases (v_j^i is the state of a node v_j at time instant t_i, v_j is a space index corresponding to the history of v_j and v^i is a time index corresponding to time instant t_i)

[166] is a temporal graph framework on top of Apache Spark [5] that uses Apache Parquet [6] as a storage backend.

Some implementations include building a temporal layer on top of an existing non-temporal graph database where the temporal information is represented as properties attached to each node and relationship. For instance, the starting and ending time instants can be simply defined as two properties. An example of such an implementation is presented in [1] which build a temporal translation layer on top of DSE [73]. Other solutions [48, 124] build a layer above Neo4j to include the notion of time. The system design of TGraph [124] is based upon the fact that real-world graphs, exemplified by transportation networks, tend to have a static structural topology and a fraction of dynamic properties. Hence, the proposed solution separates the storage of the graph

5. https://spark.apache.org/
6. https://parquet.apache.org/

(nodes and relationships) from the storage of dynamic properties.

We believe relying on an existing non-temporal graph database can lead to performance deterioration since it is not designed with native temporal support. Hence, we choose to build our temporal graph management system Clock-G by relying on Cassandra, which can be more flexibly adapted to support the temporal dimension than existing graph databases.

Storage methods We describe in the following the different storage approaches of temporal graphs we categorize as follows: Copy, Log, Copy+Log, Copy-on-write, and Referencing.

Method Copy The Copy technique stores full graph snapshots representing each materialization of the state of the graph at a given time instant. The advantage of this solution is that it facilitates the evaluation of queries since it already keeps the state of the graph without the need to recompute it. The apparent limitation of this technique is the space consumption of the snapshots that can explode the secondary storage for large-scale graphs (e.g., the graph of Thign'in).

Method Log The Log approach consists of storing an initial snapshot representing the initial state of the graph and the following graph updates (e.g., the addition of a relationship or update of the property of a node).

Time Aggregated graph (**TAG**) proposed in [95, 92, 219] is a log-based storage approach. This approach consists of using adjacency lists (explained in Section 2.1.2) where each node points to a linked list in which each element is a direct neighbor holding a time series representing the evolution of the relationship weights taken at periodic time points. The periodic representation of relationship weights forces the redundancy of values that remain constant. Another drawback of such a model is the unfaithful representation of versions since the periodicity of the data storage might not be able to follow finer change rates. The Log approach is also implemented by **DeltaGraph** [138], **Hinode** [142], and **Raphtory** [223].

Method Copy+Log The Copy+Log technique is a hybrid between the Copy and Log. As previously mentioned, the Copy approach favors reducing the execution time at the expense of increasing the storage space. Whereas the Log approach minimizes

the storage space at the expense of evaluating a query by reading all graph updates whose time instant is lower than the requested one.

The core idea of the Copy+Log is to store graph updates in time windows, each containing a sequence of graph updates and starting with a graph snapshot. The timestamps of the graph updates in a single time window fall within the time interval of the time window, and the graph snapshot represents the graph's state at the starting time instant of the time window. Now, querying the graph implies reading from the closest snapshot to the requested time instant and all graph updates stored in a single time window, whose timestamps are lower than or equal to the requested time instant. The following describes the different implementations based on the Copy+Log technique.

Temporal Graph Index (**TGI**) [139], the descendant work of DeltaGraph [138], follows a Copy+Log approach. TGI creates spatial partitions and temporal partitions. Thus, each partition is represented by a single TGI and corresponds to one spatial partition, plus the graph updates falling within a single time interval. The Copy+Log technique is also followed by **ImmortalGraph** [165]. Since the size of a time window is strongly bonded to the capacity of each application in handling redundancy, the authors of ImmortalGraph propose a redundancy ratio threshold should be set by the user who decides better, based on the application resources, how many full copied snapshots can be handled. **Auxo** [111] is a temporal graph system implementing a variant of the Copy+Log technique. The difference resides in defining different policies for splitting time windows: Time and Graph splits. The former strategy split the time window such that each newly created time window contains the graph updates of the sub-time interval of the one bounding the original time window. A graph split, however, splits a time window such that each newly created time window contains the graph updates of a part of the nodes of the original time window.

Snapshot trigger of the Copy+Log technique Time windows can be limited by the number of graph updates or the time interval. To clarify, one can fix a threshold for the number of graph updates such that the time window will be closed after reaching the specified threshold, and no more graph updates will be directed to this time window. Alternatively, one can fix a duration such that whenever the duration of the time windows reaches a threshold, the time window will be closed. The latter approach will lead to unbalanced time windows and should be avoided. A special strategy for choosing the timestamps of snapshots was proposed in [238]. This method analyzes the querying workload where each query is applied to a given time instant. Knowing that the closer

the snapshot is to the requested time instant, the lower the query execution time, the time instant of a snapshot is chosen as the centroid of the queried time instants. The main limitation of this technique is that the querying workload cannot be predicted a priory which delays the partitioning of time windows. Besides, query workloads change over time which might induce a costly re-partitioning task.

Method Copy-On-Write This technique consists of creating a new version of a modified node that can be considered as a copy of the last version of the node (right before the modification) plus the modified property values or relationships. In the temporal graph system **G*** [145], creating a new version of a node does not only affects the corresponding node but its neighboring nodes also. To clarify, if a node is modified, all its existing neighboring nodes will also be considered modified. Hence a new version of each of the adjacent nodes will be created. The resulting domino effect of neighbor modifications can reach a significant fraction of the graph, especially if the modified node is a star node (i.e., a node with a relatively large number of incident relationships) which induces a considerable space usage overhead. A similar Copy-On-Write storage technique is also presented in [116].

Method Referencing This technique consists of explicitly representing time by nodes where each timestamp or time interval, depending on the choice of a discrete or continuous temporal flow, is a node (called **Frame node**) containing references to all existing entities during the relative time interval. The referencing technique can be considered a Copy-On-Write approach since all modified nodes are explicitly stored and pointed to by their corresponding frame node. Still, it is worth considering as a separate category since the storage layout adds an overhead for holding references outweighed by faster query computation time. In the following, we will describe two implementations of the referencing storage technique. The systems implementing the Referencing technique are [48, 46].

Comparison of storage methods In this section, we provide a comparative analysis of the previously described storage techniques. We show the advantages and drawbacks of each solution in Table 2.1. Besides, we compare the temporal graph systems in adopting these solutions in Table 2.2.

The **Copy** approach is used as a comparative baseline but is not adopted by any temporal graph management system. This method facilitates the evaluation of temporal queries especially point queries; however, the frequent storage of full graph copies is space-consuming.

The **Log** storage approach offers optimal space usage since it only keeps graph updates without copying any graph or node states. However, it adds a significant query latency from reading every graph update recorded into the database whose timestamp is lower than the requested time instant.

The **Copy+Log** technique has the advantage of reducing the query execution time as compared to the Log approach since it requires adding full graph snapshots valid at different time instants. Despite the utility of these snapshots, they are full representations of the graph, hence inducing a high storage cost.

The advantage of the **Copy-On-Write** technique is the fast retrieval of the state of nodes since the entire state of the node is created upon modification which avoids the reconstruction of its state by reading irrelevant data. However, it induces a significant storage overhead caused by duplicating unmodified properties or relationships that did not change in multiple node copies. Besides, the copying frequency is affected by the modification frequency, which is cumbersome for the high updating rates.

The **Referencing** technique, a variant of the Copy-On-Write, mitigates the time needed for verifying if the time interval of a graph entity falls within the requested boundaries since these entities will be referenced (through a relationship) to a frame node indicating a single time interval. The frame nodes, though, form large nodes (nodes with many incident relationships), which are hard to manage and induce additional storage costs.

Each of the proposed solutions presents some advantages and drawbacks. They either favor space usage at the expense of query execution time or vice-versa. By comparing the Log, Copy and Copy+Log, we notice that the Copy+Log is the most suitable solution for storing temporal graphs as it mitigates the space and query execution time tradeoff presented by the other extreme methods. By comparing the Copy-on-write with the Copy+Log, we conclude that the latter is a more convenient strategy since it does not copy complete graph entities after each modification. Instead, it copies the entire graph state after a given, typically sufficiently large, number of graph modifications. Besides, the Copy+Log prunes the search space to a single time window when evaluating a query since the closest time window must be searched with the subsequent

Physical storage solution	Advantages	Drawbacks
Copy	Fast point queries	Cost-prohibitive space usage
Copy-On-Write [145, 116]	Reduction of the latency resulting from reconstructing the state of a node	Forcing data redundancy caused the duplication of unmodified entities
Log [95, 92, 219, 142, 138, 223]	Optimal space usage	Significant query latency resulting from reading irrelevant records to evaluate a temporal query
Referencing [48, 46]	Mitigation of the query execution time overhead resulting from verifying the existence of an entity in a snapshot	Creation of large nodes representing the time frames and a space overhead caused by the additional relationships connecting a time frame node to each graph entity
Copy+Log [141, 139, 165, 111, 238]	Mitigation of the query evaluation by adding full graph snapshots	Space overhead resulting from the duplication of unmodified graph entities across snapshots.

Table 2.1 – Advantages and drawbacks of different storage techniques of temporal graphs

System	Storage technique	Backend store	Temporal query types
Tag [95, 92, 219]	Log	Dedicated (not mentioned)	Shortest paths
[141]	Copy+Log	Append-only files (log) + Neo4j (snapshots)	Local/global point/range + Differential
VersionGraph [46]	Referencing	Neo4j (Proposition)	-
[48]	Referencing	Neo4j	Graph pattern (Cypher) + temporal constraints
DeltaGraph [138]	Copy+Log	Kyoto Cabinet	Global point/range
Chronos [112]	Copy+Log	Dedicated	Iterative graph computation
TGraph [124]	Log	Neo4j + Key/value (properties)	Local point/range queries + shortest paths
TGI [139]	Copy+Log	Cassandra	Local/global point/range
Immortal graph [165]	Copy+Log	Dedicated	Global/Local Point/Range + Iterative graph computation
Hinode [142]	Log	Cassandra	Global/Local Point/Range
Chrono graph [110]	Log	ChronoDB [109]	Gremlin (Snapshot)
Auxo [111]	Copy+Log	Dedicated	Global/Local point
Chrono graph [42]	Log	MongoDB	Gremlin temporal traversal (+Allen relation)
Greycat [116]	Copy-On-Write	Neo4j	-
Raphtory [223]	Log	Cassandra	Iterative graph computation

Table 2.2 – Comparison between different temporal graph management systems.

graph updates occurring before the requested time instant, a facility not offered by the Copy-On-Write technique. Following the same reasoning, the Copy+Log is more convenient than the referencing technique. In addition, the frame nodes represent large nodes whose presence in any graph database affects the overall performance.

From the analysis of these methods, we conclude that the Copy+Log is the most suitable technique for storing temporal graphs. However, full graph snapshots can be space-consuming and contain a considerable fraction of duplicates because of the similarities between consecutive snapshots. As mentioned in [165], there is an 80% or larger similarity between consecutive snapshots in many real-world evolving graphs. In this dissertation, we propose the δ-*Copy+Log* (described in Chapter 5) storage approach to reduce the space cost induced by full graph snapshots.

Evaluation of temporal graph queries

Since our goal in this thesis is to design a temporal graph system (Clock-G) that can efficiently process temporal graph queries (T-Cypher), we present the processing techniques of temporal queries posited in the literature in the following.

Temporal graph algebra To convert a query into a set of algebraic operators, the query processor uses algebra. In the context of a temporal graph management system, a temporal graph algebra (TGA) is proposed by Moffitt and Stoyanovich in [167], which includes graph operators that are extended with the temporal dimension (e.g., trim, subgraph). In this dissertation, we define a temporal graph algebra (in Chapter 6) that extends the graph algebra defined by Hölsch et al. for Cypher queries in [126]. Indeed, we choose to extend this algebra instead of other alternative graph algebras (such as GraphQL [117] and GRAD [96]) because of its compatibility with our proposed query language, which extends Cypher. For instance, TGA includes operators that cannot be expressed with T-Cypher, such as the node creation operator and the intersection operator. Besides, alternative (non-temporal) graph algebras also include operators that cannot be expressed with Cypher and T-Cypher, such as GraphQL, including the Cartesian product operator.

Plan selection and cost model Plan selection for temporal graph queries was proposed in the system Granite [194]. This temporal graph query engine evaluates nav-

igational queries with temporal predicates defined between successive relationships. Evaluating a query can be done using one path segment or two sub-path segments created after a split. Selecting an evaluation plan consists of defining a split point that might divide the path at the node with the highest selectivity, reducing the cardinality of the two path segments. The sub-plans are joined to keep the matching tuples. Granite also uses a cost model to estimate the plan cost based on cardinality estimation. These estimations build upon statistics extracted from temporal histograms that return, for each property value, the number of nodes or relationships having the property value and the average in and out degrees of these nodes. The main restriction of this query engine is that it only evaluates path queries (i.e., navigational queries). However, in this dissertation, temporal graph pattern matching is one of our critical requirements (Requirement $\boldsymbol{R_2}$). This type of querying needs a more sophisticated plan selection algorithm (as presented in Chapter 6).

Temporal histograms As presented in Section 2.1.2, query planners are used to estimate a query plan's cost to choose the best plan that reduces the overall query execution cost. These query planners build upon histograms collected from the database, which associate a given node label, relationship type, or property value with its cardinality in the graph. However, when querying temporal graphs, the cardinalities of the given elements evolve as time elapses, which implies that temporal histograms should be maintained to provide accurate estimates of the cardinalities during a requested time interval. We provide, in the following, the approaches that use temporal histograms for estimating the cost of temporal queries. It should be noted that some of the solutions are not presented mainly for the property graph model, yet they can be generalized to cope with any data model.

A special implementation of histograms called **A**daptive **M**ulti-dimensional **H**istograms (AMH) is proposed in [225] in order to evaluate spatio-temporal SPARQL [113] queries (SPARQL-ST). As outlined in this work, a spatio-temporal query asks for the total number of objects located in a given spatial region at a time instant. The histogram is partitioned into buckets where each bucket covers several cells, and each cell contains the total number of objects in a location and time interval. The partitioning strategy reduces a variance metric, ensuring that each bucket covers similar cells (i.e., cells with close frequencies). The resulting histogram is then indexed with a binary tree whose leave nodes correspond to a bucket. The main limitation of this solution is that a single

update can vary the variance metric, which then implies a reorganization task of the buckets. This task might induce costly merges and splits of the rectangular histograms and updates to the structure of the binary trees indexing these histograms.

RDF-TX is a temporal RDF graph store implementing temporal histograms [89] to help the query processor choose the order of joins in temporal queries. To manage these histograms, they use a compressed version of **M**ulti-**V**ersion **SB** Trees (MVSBT) [241]. The MVSBT combines the features of **M**ulti-**V**ersion **B+** Trees MVBT and SBT. Hence, instead of a single tree, this structure creates a tree of SBT trees corresponding to a time partition. The nodes of these trees correspond to a rectangle on the key/time histogram such that the value of each node corresponds to the aggregate value (total number of keys between k_{min} and K_{max} occurring during t_{min} and t_{max}). The index is dynamically updated by splitting the tree nodes and compressed to cope with the outgrowing size of the histograms.

Granite [194] is a temporal graph query engine that leverages the functionality of Apache Giraph [207] with graph querying capabilities. Besides offering a temporal graph query language, Granite also provides a query planner based on a cost model to estimate the selectivity of the best execution plan. These selectivity estimates are based on temporal information collected from the underlying graphs and maintained in temporal histograms. The query planning is based on a splitting method that divides paths into two sub-paths query segments computed in parallel. The choice of a split point is based on the assumption that a good plan should first evaluate operators, including predicates with low selectivity. The special technique of compressing temporal histograms is based on *hierarchical tilling* [174], which consists of partitioning a given rectangular histogram in a way that minimizes the number of tiles (partitions) concerning a given threshold. Histograms are then represented by interval trees, s.t., each node represents a partition. Once constructed, the temporal histograms are used to recursively compute an estimation of the number of active and matched nodes and relationships per super-step. A temporal histogram is created for each property key, node label, and relationship type.

To conclude, the solutions presented in [225, 89, 194] are similar in that they partition the rectangular key/time histograms to reduce the number of partitions and limit the footprint of the data structure used to maintain the histograms. However, updating a single value in the temporal histogram might lead to costly re-partitioning, including merges and splits of the partitions. Besides, the temporal histogram (MVSBT) structure

presented in [89] is not suitable for range queries. Using MVSBT, one can only estimate the number of keys less than k with timestamps less than t. Hence, two MVSBTs must be constructed such that the first estimates the total number of keys less than k and whose starting time instants less than t, whereas the second estimates the total number of keys that are higher than k whose ending time instants that are higher than t. This, indeed, adds a space overhead resulting in maintaining two MVSBTs.

Motivated by the above observations, we propose to use temporal histograms in the design of Clock-G to help the query processor estimate the cardinalities of sub-results according to the requested time interval. Hence, we propose using a compact data structure that can efficiently answer range queries and reduces space usage by summarizing the data. We describe our proposed query processor and its implementation in our system Clock-G in Chapter 6.

2.3 Graph generators

Large graphs are needed for evaluating the performance of graph management systems. However, real datasets are inaccessible or do not fit the scale requirements. Hence, practitioners tend to generate synthetic graphs by implementing graph generators. These graph generators attempt to fill the gap between real and synthetic graphs by controlling one or more graph characteristics: degree distribution, community structure, clustering coefficient, the similarity between the properties of connected nodes, etc. In this dissertation, we are interested in generating temporal graphs that can be used in benchmarking temporal graph management systems. Hence, we discuss some critical characteristics of real-world graphs and the graph generation models proposed in the literature in the following.

Graph characteristics

Different properties, such as degree distribution and community structure, can characterize real-world graphs. We describe these characteristics in the following.

Degree distribution The degree of a node corresponds to the total number of incident relationships to that node. The degree distribution of the graph corresponds to the distribution of the node degrees over the entire graph. Many real-world graphs, such as

social graphs, obey a power law distribution. That is, the degree distribution of a graph is power law if the number of nodes n_k having a degree k is given by $n_k \propto k^{-\gamma}$ where γ is known as the power-law exponent.

Community structure The modular structure of real-world graphs has received attention after the introduction of the modularity metric by Girvan and Newman [97]. That is, it has been shown that many graphs from diverse domains such as sociology, biology, and computer science exhibit a modular structure [84], in the sense that their nodes can be partitioned into groups characterized by their connectivity. Yet, a graph community cannot be quantitatively well defined. Instead, one can intuitively define a community as a group of nodes more densely connected with each other than they are with the rest of the graph. Many community detection algorithms have been proposed. Many of them optimize a quality function that assigns a score to a node partition [97, 57]. A common metric used to control the community structure of graphs is the modularity defined in [177, 176]. The modularity is, up to a multiplicative constant, equal to the difference between the number of edges connecting nodes in the same group and the number of edges connecting nodes in the same group in a graph where the edges are randomly placed and the same nodes preserving their degrees as in the original graph. Hence, the value of modularity can be either positive or negative, such that positive values indicate a possible presence of community structure. Based on this, one can identify a community structure by finding the blocks in a graph with positive, preferably large, values of modularity.

Graph generative models

Erdős Rényi The Erdős Rényi model [74] generates directed, undirected, and multipartite random graphs with and without self-loops. It takes two input parameters: the number of nodes N and a probability ρ based on which a relationship might exist or not between any pair of nodes. Based on a uniform probability, this randomized relationship generation procedure cannot satisfy a specific degree distribution.

Stochastic Adjacency Matrix Deterministic kronecker graphs [152] are based on successively applying the Kronecker product of an (N_1,N_1) initiator adjacency matrix G_1 with itself until reaching the desired scale. After k repetitions of the same procedure,

the resulting matrix of graph G_k will contain N_1^K cells. It recursively creates self-similar communities where each community in step k is expanded into G_1 in step $k+1$. Since deterministic Kronecker graphs present a staircase effect in the resulting degree distribution, a stochastic version was proposed producing Stochastic Kronecker Graphs (SKG). This model consists of filling the adjacency matrix with values between 0 and 1, thus generating a stochastic adjacency matrix that must be sampled to create the desired number of relationships. Variants of this model were also proposed, such as Fast Kronecker [150] and R-MaT [50].

Chung-Lu The Chung-Lu model [238, 56, 55, 54] proposes a random graph generator producing any given degree distribution. Indeed, it tunes the probability of relationship generation to a normalized value proportional to the product of the two relationship endpoints degrees. More concretely, the probability p of a relationship creation between a node u with degree d_u and another node v with degree d_v is equal to

$$p = \frac{d_u d_v}{\sum_0^N d_i}$$

This generation procedure approximates the required degree distribution so that the obtained degree of each node follows a Poisson distribution with a mean equal to its desired degree for a large number of samples.

Stochastic block model The stochastic block model is a generative model that controls the community structure of the resulting graphs [120]. This model takes a pair (P, M), as an input parameter, where P and M are defined as follows:

- P: A partitioning function that assigns each node of the graph to a partition from the set of partitions $\{b_0, \ldots, b_{k-1}\}$.
- M: A $(k \times k)$ block probability matrix whose element m_{ij} corresponds to the linkage probability between partitions b_i and b_j.

A common use of the SBM consists of setting the same within-community, and between-community linkage probabilities for the communities of the generated graphs [222, 101, 28, 66]. Some implementations of the SBM model distribute uniformly the nodes within communities [39, 87, 222]. Whereas other implementations distribute nodes following a skewed distribution which mimics better the communities in real-world graphs [222].

Block Two Erdős-Rényi The Block-Two Erdős-Rényi (BTER) is a community-aware graph generative model that satisfies both the clustering coefficient and degree distribution [140]. The generated graph is a collection of Erdős Rényi blocks (or communities) where each block contains $d+1$ nodes of degree d. After the insertion of within-community relationships, the next phase consists of generating between-community relationships following the Chung-Lu model. Thus, the BTER model adds between-community relationships for each node to satisfy its desired degree, where the probability of a relationship's existence is proportional to the product of the degree excess of its endpoints. Darwini [72] extends BTER by controlling the clustering coefficient distribution at a finer granularity. BTER and Darwini produce graphs with a small diameter due to their generation process.

Labeled and property graph generators

We describe in the following some of the generators that consider adding labels and properties to the resulting graphs.

gMark [20] is a graph and query generative framework. It creates directed relationship-labeled graphs that satisfy a given degree distribution where each node and relationship has a user-specified type. The user can also define, for each relationship type, an in and out-degree distribution and the permitted source and target types.

DataSynth [192] is a graph generator producing property graphs such that nodes and relationships have types and properties. DataSynth connects nodes based on the similarities between their property values. However, DataSynth defines a community as a set of nodes sharing the exact value of a specific property. This condition is not realistic since the values of the properties in real-world graph communities can be heterogeneous. Besides, it does not mirror a given degree distribution which is a critical property of graphs.

Comparison The similarity between the Erdős-Rényi, stochastic Kronecker, and Chung-Lu models resides in assigning a linkage probability for each possible relationship. However, the Erdős-Rényi generates random graphs with a uniform degree distribution. The stochastic Kronecker model preserves a recursive community structure that

poorly reflects real-world graphs. Besides, it fails to control the degree distribution of the graph. The significant advantage of the Chung-Lu model is that it can provide graphs with any desired degree distribution, a feature that stochastic Kronecker and Erdős-Rényi models provably do not offer. This motivated us to extend Chung-Lu's model to generate temporal graphs while controlling the evolution of the degree distribution.

The stochastic Kronecker, BTER, and SBM are models that generate graphs with a community structure. The stochastic Kronecker model generates graphs with a recursive community structure, a property that cannot be found in most real-world graphs. Now the BTER generates graphs with a community structure by controlling the clustering coefficient of the nodes. As previously mentioned, each community groups $d-1$ nodes where d is the degree of the nodes in that community. This partitioning strategy results in a large number of small communities and a relatively small number of large communities. Besides, all nodes belonging to the same community share the same degree. The general model of SBM is a simple yet powerful model that can be used for generating graphs while approximating any ground-truth community structure that is defined with a block probability matrix. Based on this comparison, we choose to use the SBM model to control the community structure of the temporal graphs.

Regarding the advantages and limitations of the aforementioned generative models, we chose to build upon the Chung-Lu model and SBM to generate temporal graphs. Indeed, these two models are simple and scale well for large-scale graphs. Besides, the Chung-Lu model and SBM allow the generation of graphs that obey a given degree distribution and ground truth community matrix, which cannot be satisfied by other generative models. Our second goal is to extend this model with time such that the generated graphs represent a sequence of temporal updates while preserving the evolution of the degree distribution and community structure. Towards this, we also studied the related work in temporal graph generation, which we will present in the next section.

2.4 Temporal graph generators

The previously described graph generators produce non-temporal graphs. In this dissertation, we are particularly interested in the generation of temporal graphs, including changes in topology and the properties of nodes and relationships. In the following, we provide a detailed description of such generators.

DANCer This generative model [28] produces community-aware temporal property graphs. In this model, each community contains several "representative" nodes with which a new-coming node must be compared (property comparison) before inserting it in the community with the most similar representative. DANCer merges communities based on the similarity between their representatives and splits the communities with the largest diameter. It also defines criteria to create or remove nodes and relationships.

Generator	Community aware	Seed	Batch	Growth only	Property evolution	DD	PA	CT	S	WBC
CGGS	x			x			x	x		x
APA		x	x	x			x	x	x	
EGG		x	x		x	x				
DANCer	x	x	x				x	x	x	x

Table 2.3 – Comparison between different temporal graph generation tools. **DD** is an abbreviation of the degree distribution. **PA** is an abbreviation of the preferential attachment. **CT** is an abbreviation for the closing triangles. **S** is an abbreviation of similarity. **WBC** is an abbreviation of Within and Between community.

ComAwareNetGrowth This generator [107] produces community-aware growth-only graphs. The generation model consists of adding a node with several relationships attached to it or creating a link between two existing nodes. Inserting a node involves choosing a community based on a probability vector containing, for each community, its probability of attracting a node. Next, the node will connect inside or outside its community based on a fixed probability provided as an input parameter. Then, another parameter defines the probability of choosing the other end of the relationship randomly or based on preferential attachment. Inserting a relationship between two existing nodes is similar, except that one end is always chosen randomly, and the other end can also close a triangle or quadrangle based on a user-defined probability.

APA (Attribute-Aware Preferential Attachment) This model [6] is a graph generator capable of creating growth-only property graphs. The key feature of this model is closing a triangle based on the similarity (i.e., the similarity between their property values) between the candidate relationship's endpoints. In each snapshot, newly inserted nodes are batched, and new connections are created.

EGG (Evolving Graph Generator) This generative tool [10] extends gMark [20] with time-evolving properties.The evolution of these properties is controlled by a set of user-specified constraints, such as possible values of each property, values that cannot appear in two successive snapshots, the correlation between the properties of a single node, etc. Authors of EGG disregard the topological changes to the network and narrow the temporal evolution of the graph to property updates.

DSNG-M (Dynamic Social Network Generator based on the Modularity) This graph generator [71] produces temporal graphs by mirroring a given community structure. The generation principle consists of flipping the relationships of a given static graph to satisfy a randomly chosen modularity value. The procedure consists of running a community-detection algorithm on the provided initial graph. Then, for each snapshot, a modularity value is selected. A randomly chosen relationship is flipped to obtain a new graph snapshot with the desired modularity value, and the new modularity value is computed. The only temporal operation allowed is relationship flipping which limits the generality of the generated temporal graphs.

Comparison We identify a list of properties to characterize a temporal graph generator. Table 2.3 selects, for each generator, the properties that it satisfies. The first property indicates whether the generator is community-aware or not. The seed property suggests that the graph starts from an initial graph snapshot that represents the state of the graph at an initial time instant. The batch property indicates that the generator groups graph updates at a single time instant to produce a new graph snapshot instead of generating a sequence of time-respective graph updates. The growth-only property indicates that the graph only includes addition graph updates, such as adding a node or relationship. Property evolution indicates that the values of properties change over time. The degree distribution column (DD) indicates that the generator creates relationships to satisfy a given degree distribution. The preferential attachment (PA) implies that the 'Rich getting richer' principle is applied. The closing triangle (CT) property indicates that the creation of a relationship is favored when the inserted relationship participates in a triangle. The similarity criteria (S) indicate if the generator adds a relationship between two nodes if they share similar properties. Finally, the within and between community (WBC) indicates that the relationship creation is strictly related to pre-configured between and within community linkage probabilities. It can be

derived from the table that no graph generation tool controls the evolution of the degree distribution and community structure and includes deletions in the set of possible graph updates. These limitations motivated us to build our graph generator RTGEN (presented in Chapter 7), capable of fulfilling those three requirements that we firmly find essential for mirroring the evolution of real-world graphs.

2.5 Conclusion

We started this chapter by discussing the prior work on developing non-temporal graph management systems. This study shows the advantages and limitations of current methods in handling the temporal dimension. We presented graph models, query languages, and system design characteristics for graph management systems and their temporal counterparts. Besides the design of temporal graph management systems, we presented the prior work on graph generation needed for benchmarking these systems. The available approaches fail to produce temporal graphs while controlling: the evolution of the degree distribution and community structure.

To address the challenges presented in this chapter, we introduce Clock-G, a temporal graph management system we have designed to efficiently store and query temporal graphs. The main contributions are defining a query language, new storage, and query processing techniques optimized for managing the additional temporal dimension. We also present a temporal graph generator configured with degree distribution parameters describing its temporal evolution while controlling the community structure. These contributions are presented in the following chapters.

Chapter 3

Clock-G: A temporal graph management system

From the analysis of the related work (Chapter 2), we found that existing solutions present severe limitations or do not fit our requirements. Besides, current graph databases are not designed to support the temporal dimension. Hence, we designed Clock-G, a temporal graph management system we aim to integrate into the Thing'in platform. Our main goal is to couple Clock-G with expressive temporal graph query language while also focusing on the design considerations such as storage and query processing. This chapter provides an overview of the Ph.D. work, which mainly focuses on the design of Clock-G. In the following chapters, each of the critical components of the design of Clock-G is further discussed.

3.1 Overview

Clock-G is a temporal graph management system that we have developed to answer the need of the Thing'in platform. We developed this system using the programming languages Java and Golang. In the following, we describe each of the components presented in Figure 3.1. It should be noted that the following description is general and brief, but we will develop each aspect in the next chapters.

Request Handler The request handler is responsible for managing the read and write requests. As presented in Figure 3.1, the request handler comprises two functional components: Reader and Writer.

The reader is responsible for processing temporal graph queries written in our proposed language, T-Cypher. This component is further composed of a query extractor, planner, and evaluator. The role of the query extractor is to convert a T-Cypher query

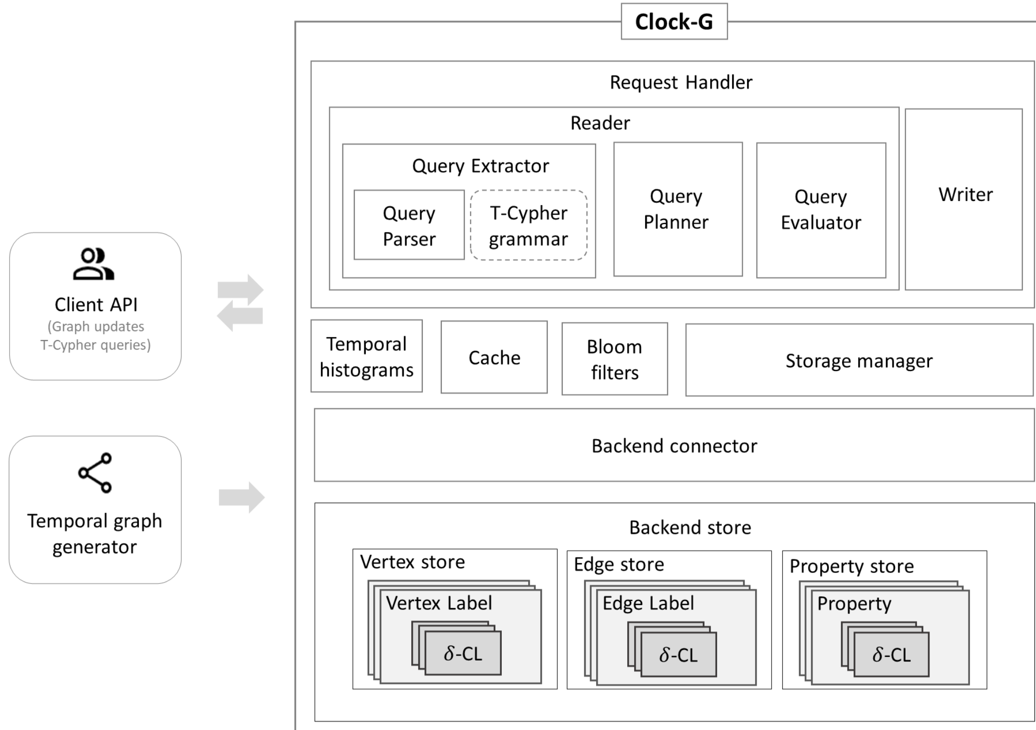


Figure 3.1 – Overview of the system architecture of Clock-G

into a query object recognized by the system. This conversion builds upon grammatical rules (T-Cypher grammar) that we have proposed to define the syntax of T-Cypher queries. The query planner is responsible for converting a query object into an execution plan based on our proposed plan selection algorithm. This algorithm greedily selects an evaluation plan that minimizes the estimated cardinality of sub-results. The cardinality estimation is based on a cost model we defined to assign each operator an estimated cost. Besides, we maintain temporal histograms in special data structures that facilitate the extraction of the cardinalities during a time range. Finally, the query executor is responsible for evaluating query operators. Hence, it maintains a pool of atomic executors where each executor is responsible for executing a single query operator. Now, the intermediate results are shared between the atomic executors.

On the other hand, the writer is responsible for inserting graph updates. It first informs the storage manager about the insertion and asks for storage elements to which the new operation should be added. Then, it sends an insertion request to the backend connector, translating this request into atomic write operations sent to the backend

store.

Backend store The backend store is responsible for storing the temporal graphs following our proposed storage technique called the δ-*Copy+Log*. We rely on the column-oriented database Apache Cassandra [147] for robustness, engineering maturity, and scalability. Besides, Cassandra sorts blocks of data according to a given column or combination of columns. We utilize this feature to sort graph updates according to their chronological order, accelerating their sequential read.

For instance, the storage is separated based on the graph entity type, resulting in node, relationship, and dynamic property stores. For each node/relationship label or dynamic property, we partition the storage based on a Hash partitioning strategy. Each of these partitions corresponds to a storage unit and is stored following the δ-*Copy+Log* method (denoted δ-CL in Figure 3.1 for simplicity). We note that the δ-*Copy+Log* is a storage technique we have proposed to mitigate the apparent trade-off between space and query execution time presented by anterior storage techniques. This technique will be further discussed in Chapter 5.

Storage manager The storage manager is responsible for applying the rules of the δ-*Copy+Log* method to the storage. Besides, it maintains metadata that helps direct read or write operations to the corresponding storage entities.

Backend connector The backend connector connects to and executes requests against the backend store. Hence, it receives read or write requests from the request handler and converts them into Cassandra queries before executing them against the backend store.

Auxiliary data structures To reduce the prohibitive cost of accessing the secondary storage, we use auxiliary data structures maintained in memory and queried when needed. These auxiliary data structures include Temporal histograms, Cache, and Bloom filters. The temporal histograms represent the evolution of the cardinality of graph elements through time. The cache is used to preserve data in the main memory to avoid costly disk accesses. The Bloom filters are probabilistic data structures that we use to check the existence of an element in a set.

Client API Clock-G offers a client API enabling a client to connect, ingest graph updates or query the stored graphs. For instance, users can define the graph space's schema that includes labels of nodes and relationships and a set of properties for each node/relationship label.

Users can insert graph operations individually or in batches into the system. In both cases, graph operations are attached to a transactional time based on the system's internal clock. Besides, users can query the temporal graph using the temporal graph query language T-Cypher that we have proposed to facilitate the expression of temporal graph pattern matching and temporal navigational queries.

Temporal graph generator To assess the performance of Clock-G, we needed access to a large temporal graph dataset. However, real-world datasets are either subject to confidentiality agreements or do not fit the scale requirement. Hence, we referred to synthetic datasets apart from using real-world temporal graphs. We proposed and implemented a temporal graph generator capable of generating large-scale temporal graphs whose characteristics mirror real-world graphs. This graph generator, called RTGEN, will be presented in Chapter 7. Besides, we referred to an existing generative tool (LDBC datagen [76]) that we have chosen because it covers a significant fraction of aspects such as degree distribution, all types of graph updates that include the addition and deletion of a node or relationship, or the update of a property. This feature is not available in any other publicly available dataset or other available generative tools.

3.2 Conclusion

In this chapter, we mainly focused on providing a general overview of our temporal graph management system Clock-G. In the following chapters, we will present each key component of Clock-G while highlighting our contributions.

CHAPTER 4

TEMPORAL GRAPH QUERY LANGUAGE

In this chapter, we introduce T-Cypher, a query language for temporal graphs that has been integrated into Clock-G. This chapter is structured as follows: we begin by discussing the motivations and contributions of our work. Next, we provide an overview of the preliminaries that form the foundation of our proposal. We also introduce the temporal constructs that we utilize to augment Cypher with temporal capabilities. Afterward, we present the syntax of T-Cypher and the translation rules that allow T-Cypher queries to be expressed in Cypher. Finally, we conclude the chapter.

4.1 Motivations and contributions

Although a temporal graph query language can be highly useful, the majority of current temporal graph management systems [112, 165, 138, 223] do not offer such a feature. Instead, these systems typically prioritize optimizing storage techniques and overlook the importance of temporal query expressiveness or providing a language for expressing such queries. Since querying temporal graphs cannot be done with conventional graph query languages such as Cypher [86], PGQL [200], or G-Core [14], the addition of time-version support is necessary to accommodate both structural and temporal constraints.

To address this limitation, we propose T-Cypher: a time-extended version of Cypher. Knowing that the standard graph query language GQL [1] has not been released by the time of writing this thesis (October 2022), we choose to extend Cypher instead of any other available query language. The rationale behind this choice is that the syntax of Cypher is graph-like (i.e., graph patterns are expressed using “ASCII art”) and user-friendly, making it a popular choice amongst graph query languages. Many features extracted from Cypher will be echoed in the standardization of GQL. Besides, Cypher

1. https://gql.today/

is expressive, declarative, normalized, and open source.

Our proposed extension of Cypher is conservative, as it incorporates temporal constructs without modifying existing grammar rules. This approach makes it easy for practitioners who are already familiar with Cypher to transition to T-Cypher. The primary difference between T-Cypher and the original language is that the former allows for the expression of temporal variables that refer to the time validity intervals of graph variables, in addition to the expression of graph variables such as nodes, relationships, or properties. This enables the expression of temporal constraints that can be applied to the query's temporal variables. Another essential feature of T-Cypher is the trim statement, which can be used at the beginning of a query to limit the search space to one or more time intervals, ensuring that all variables defined in the query are valid during at least one of these intervals. Additionally, T-Cypher allows for the expression of temporal values (such as time instants, intervals, and duration), temporal functions, and operators that can be used to define constraints or predicates on the query's temporal variables. Another critical feature of T-Cypher is that it enables the expression of temporal paths. In this context, we propose three types of temporal relationship patterns: continuous, sequential, and pairwise-continuous. Using T-Cypher, we can fulfill our required querying functionalities presented in the Introduction chapter: temporal slicing (Functionality $\boldsymbol{R_1}$), graph pattern matching (Functionality $\boldsymbol{R_2}$), navigation (Functionality $\boldsymbol{R_3}$) and aggregation (Functionality $\boldsymbol{R_4}$).

The main contributions presented in this chapter reduce to the following:

- Proposing a temporal graph query language that allows the expression of a wide range of queries with a user-friendly syntax.
- Presenting the translation rules of a T-Cypher query into a Cypher query.

4.2 Preliminaries

For clarity, some of the key definitions used throughout this section are given in Table 4.1.

4.2.1 Time domain

We define in this section the time domain and temporal elements already presented in the literature of temporal databases [129]. Specifying the time domain is needed in

Symbol	Set notation
Property keys	k
Node identifiers	ID
Relationship identifiers	ID^R
Node labels	L
Relationship types	T
Names	A
Values	V
Time domain	Ω^T

Table 4.1 – Symbols used in the preliminary definitions

data management systems when data items should be assigned with temporal ontologies such as temporal validity information [168]. We consider a discrete temporal flow such that time is quantified by time granules. A time granule, also referred to as a **chronon**, is the smallest non-decomposable unit of time defined by a certain temporal granularity (e.g., a second or a millisecond). Hence, we consider $\Omega^T = \mathbb{N}_0 \cup \{\infty\}$ where $k < \infty, \forall k \in \mathbb{N}_0$ to be the temporal domain defined as a totally ordered set of instants such that the duration between consecutive instants is equal to the chronon. The symbol ∞ is used as the right boundary of a time interval to indicate that an entity is not yet deleted. A time instant t is a point on the time axis or an element of the time domain such that $t \in \Omega^T$. The time instant $t_0 = 0$ is usually used to refer to the origin, or standard point of a temporal domain [53]. Time intervals are often used to obtain a compact representation of convex sets of time instants. Assuming that a fact is valid over a time interval, it is considered valid at each time instant of that interval. A time interval $i = [i^s, i^e) \in \Omega^T \times \Omega^T$ where $i^s \leq t < i^e$.

4.2.2 Temporal property graph model

We extend the original property graph model proposed by Francis et al. in [86] by assigning each graph element that can be a node, relationship, or property value with a set of time validity intervals during which the graph entity was valid. Having this, we define a temporal property graph as a tuple $G =< N, R, src, tgt, \lambda, \tau, \phi, \iota^T >$ where ϕ and ι^T are functions added to the original model to incorporate the temporal dimension. We define each element of the tuple as follows:

— N is a finite subset of ID where ID represents the set of node identifiers.

— R is a finite subset of ID^R where ID^R represents the set of relationship identifiers.
— src: $R \rightarrow N$ is a function that maps each relationship identifier to its source node identifier.
— tgt: $R \rightarrow N$ is a function that maps each relationship identifier to its target node identifier.
— $\lambda : N \rightarrow 2^L$ is a function that maps a node identifier to a set of labels where L is a set of node labels.
— $\tau : R \rightarrow 2^T$ is a function that maps a relationship identifier to a set of types where T is a set of relationship types.
— $\phi : N \cup R \rightarrow 2^{\Omega^T \times \Omega^T}$ is a finite partial function that maps a node or relationship identifier to a set of validity intervals during which the corresponding graph entity was valid.
— $\iota^T : (N \cup R) \times k \rightarrow 2^{\Omega^T \times \Omega^T \times V}$ is a finite partial function that maps a dynamic property key associated with a node or relationship to a set of value and validity interval pairs. Every element in the mapped set is a pair (v, i) where v is a value and i is a time interval, indicating that the value v is valid during i.

As mentioned in Chapter 2 (Section 2.2.1), this model is interval-based since it assigns each entity with a time interval. This time representation is semantically equivalent to assigning each element with a set of time instants during which the entity was valid. However, we use intervals to compact the representation of convex sets of time instants. Note that this model is similar to the model proposed by Moffitt et al in [167] and presented in Chapter 2. The main difference between our model and the original one is that our model assigns edges with a set of types to adhere with the data model of our particular use case Thing'in.

4.3 Temporal graph relation

The output of a T-Cypher query is a temporal graph relation. Each graph relation is a bag of tuples where a tuple u is a partial function that maps names to values. A tuple with named fields $u = (a_1 : v_1, \ldots, a_n : v_n)$ where $(a_1, \ldots, a_n)$ are distinct names and each element in $(v_1, \ldots, v_n)$ is either a value, node or relationship state, set of node or relationship states, or paths. These states represent a node or relationship in a time interval during which their property values did not change. Recall that V denotes the

set of values and k denotes the set of property keys.

A **node state** in n is a tuple (id_n, l, k, τ) such that:

— $id_n \in ID$ is the node identifier.

— $l \in 2^L$ is the set of node labels.

— $k = \{k_1 : v_1, \dots, k_m : v_m\}$ is a map of property names and values such that $k_i \in k, 1 \leq i \leq m$ and $v_i \in V, 1 \leq i \leq m$.

— $\tau \in \Omega^T \times \Omega^T$ is the validity time interval during which the node state was valid.

A **relationship state** in r is a tuple $(id_{n_s}, id_{n_t}, t, k, \tau)$ such that:

— $id_{n_s} \in ID$ is the source node identifier.

— $id_{n_t} \in ID$ is the target node identifier.

— $t \in 2^T$ is the set of relationship types.

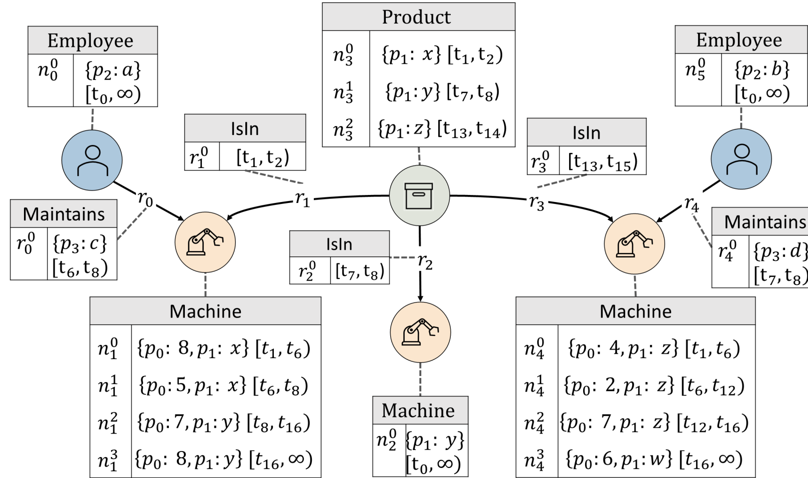
— $k = \{k_1 : v_1, \dots, k_m : v_m\}$ is a map of property names and values such that $k_i \in k, 1 \leq i \leq m$ and $v_i \in V, 1 \leq i \leq m$.

— $\tau \in \Omega^T \times \Omega^T$ is the validity time interval during which the relationship state was valid.

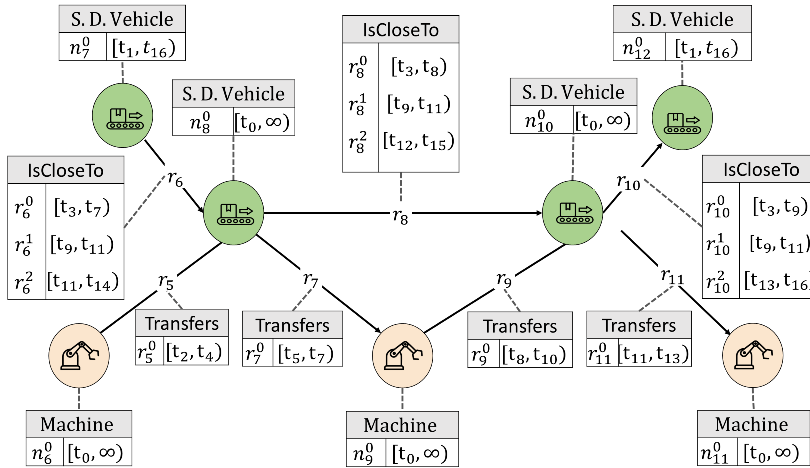
The initial state of a graph entity, either a node or a relationship, results from its creation. Subsequent modifications to the entity, including property creation, update, deletion, or the entity's deletion, result in new states. The initial state of an entity includes all the property values set at the time of its creation, and each property is associated with a time interval $[t, \infty)$, where t is the creation time instant. Upon modification of a property, a new state is created, which includes all the previous property values that remain unaffected and the new value of the modified property. The time interval of the modified property is replaced with $[t_{prev}, t)$, where t_{prev} is the time of the last update of the entity and t is the time instant of the modification. The modified property in the new state is associated with the time interval $[t, \infty)$. Property addition also results in a new state, where the new property is associated with the time interval $[t, \infty)$, where t is the time instant of the addition. In contrast, deleting an entity does not create a new state but instead updates its last state by setting the right bound of its time interval to the time instant of deletion.

Example 4.3.1. To clarify the definition of node and relationship states, we present a concrete example of a temporal property graph inspired by the use case of smart factories. Figures 4.1(a) and 4.1(b) show two toy graphs (A and B) inspired by this use-case. In these graphs, nodes $\{n_1, ..., n_{12}\}$ model products, machines, self-driving vehicles (S.D. vehicles), and employees. Whereas, relationships $\{r_1, ..., r_{11}\}$ represent the connections

between nodes. Properties $\{p_0, \ldots, p_3\}$ are attached to nodes and relationships to describe these graph entities.



(a) Toy graph A



(b) Toy graph B

Figure 4.1 – Toy graphs illustrating the traversal of products through machines, the maintenance of these machines (Toy graph A), the transfer of products between machines, and the closeness between self driving vehicles (Toy graph B)

In the manufacturing process, a product may traverse through various machines, which is denoted by the ***isIn*** relationship between them. This relationship captures the progression of the product through the different stages of the manufacturing process. The machines are regularly maintained by employees through the ***maintains*** relationship. Additionally, products are transported from one machine to another through an S.D. ve-

Nodes	States
n_0	$n_0^0 = (id_{n_0}, \text{Employee}, \{p_2 : a\}, [t_0, \infty))$
n_1	$n_1^0 = (id_{n_1}, \text{Machine}, \{p_0 : 8, p_1 : x\}, [t_1, t_6))$ $n_1^1 = (id_{n_1}, \text{Machine}, \{p_0 : 5, p_1 : x\}, [t_6, t_8))$ $n_1^2 = (id_{n_1}, \text{Machine}, \{p_0 : 7, p_1 : y\}, [t_8, t_{16}))$ $n_1^3 = (id_{n_1}, \text{Machine}, \{p_0 : 8, p_1 : y\}, [t_{16}, \infty))$
n_2	$n_2^0 = (id_{n_2}, \text{Machine}, \{p_1 : y\}, [t_0, \infty))$
n_3	$n_3^0 = (id_{n_3}, \text{Product}, \{p_1 : x\}, [t_0, t_2))$ $n_3^1 = (id_{n_3}, \text{Product}, \{p_1 : y\}, [t_7, t_8))$ $n_3^2 = (id_{n_3}, \text{Product}, \{p_1 : z\}, [t_{13}, t_{14}))$
n_4	$n_4^0 = (id_{n_4}, \text{Machine}, \{p_0 : 4, p_1 : z\}, [t_1, t_6))$ $n_4^1 = (id_{n_4}, \text{Machine}, \{p_0 : 2, p_1 : z\}, [t_6, t_{12}))$ $n_4^2 = (id_{n_4}, \text{Machine}, \{p_0 : 7, p_1 : z\}, [t_{12}, t_{16}))$ $n_4^3 = (id_{n_4}, \text{Machine}, \{p_0 : 6, p_1 : w\}, [t_{16}, \infty))$
n_5	$n_5^0 = (id_{n_2}, \text{Employee}, \{p_2 : b\}, [t_0, \infty))$
n_6	$n_6^0 = (id_{n_6}, \text{Machine}, \{\}, [t_0, \infty))$
n_7	$n_7^0 = (id_{n_7}, \text{S.D. Vehicle}, \{\}, [t_1, t_{16}))$
n_8	$n_8^0 = (id_{n_8}, \text{S.D. Vehicle}, \{\}, [t_0, \infty))$
n_9	$n_9^0 = (id_{n_9}, \text{Machine}, \{\}, [t_0, \infty))$
n_{10}	$n_{10}^0 = (id_{n_{10}}, \text{S.D. Vehicle}, \{\}, [t_0, \infty))$
n_{11}	$n_{11}^0 = (id_{n_{11}}, \text{Machine}, \{\}, [t_0, \infty))$
n_{12}	$n_{12}^0 = (id_{n_{12}}, \text{S.D. Vehicle}, \{\}, [t_1, t_{16}))$

Table 4.2 – Nodes and node states of the graph in Figure 4.1(a) and 4.1(b)

Relationships	States
r_0	$r_0^0 = (id_{n_0}, id_{n_1}, \text{Maintains}, \{p_3 : c\}, [t_6, t_8))$
r_1	$r_1^0 = (id_{n_3}, id_{n_1}, \text{IsIn}, \{\}, [t_1, t_2))$
r_2	$r_2^0 = (id_{n_3}, id_{n_2}, \text{IsIn}, \{\}, [t_7, t_8))$
r_3	$r_3^0 = (id_{n_3}, id_{n_4}, \text{IsIn}, \{\}, [t_{13}, t_{15}))$
r_4	$r_4^0 = (id_{n_5}, id_{n_4}, \text{Maintains}, \{p_3 : d\}, [t_7, t_8))$
r_5	$r_5^0 = (id_{n_6}, id_{n_8}, \text{Transfers}, \{\}, [t_2, t_4))$
r_6	$r_6^0 = (id_{n_7}, id_{n_8}, \text{IsCloseTo}, \{\}, [t_3, t_7))$ $r_6^1 = (id_{n_7}, id_{n_8}, \text{IsCloseTo}, \{\}, [t_9, t_{11}))$ $r_6^2 = (id_{n_7}, id_{n_8}, \text{IsCloseTo}, \{\}, [t_{11}, t_{14}))$
r_7	$r_7^0 = (id_{n_8}, id_{n_9}, \text{Transfers}, \{\}, [t_5, t_7))$
r_8	$r_8^0 = (id_{n_8}, id_{n_{10}}, \text{IsCloseTo}, \{\}, [t_3, t_8))$ $r_8^1 = (id_{n_8}, id_{n_{10}}, \text{IsCloseTo}, \{\}, [t_9, t_{11}))$ $r_8^2 = (id_{n_8}, id_{n_{10}}, \text{IsCloseTo}, \{\}, [t_{12}, t_{15}))$
r_9	$r_9^0 = (id_{n_9}, id_{n_{10}}, \text{Transfers}, \{\}, [t_8, t_{10}))$
r_{10}	$r_{10}^0 = (id_{n_8}, id_{n_{12}}, \text{IsCloseTo}, \{\}, [t_3, t_9))$ $r_{10}^1 = (id_{n_{10}}, id_{n_{12}}, \text{IsCloseTo}, \{\}, [t_9, t_{11}))$ $r_{10}^2 = (id_{n_{10}}, id_{n_{12}}, \text{IsCloseTo}, \{\}, [t_{13}, t_{16}))$
r_{11}	$r_{11}^0 = (id_{n_{10}}, id_{n_{11}}, \text{Transfers}, \{\}, [t_{11}, t_{13}))$

Table 4.3 – Relationships and relationship states of the graph in Figures 4.1(a) and 4.1(b)

hicle using the ***transfers*** relationship. To indicate the proximity between S.D. vehicles, a temporary relationship ***isCloseTo*** is established if the distance between them is lower than a predetermined threshold. The properties p_0 and p_1 of a machine can indicate its temperature or position whereas the property p_2 of an employee can its skills. The tools used during maintenance can be represented by p_3 of the maintains relationship. Each node and relationship in the temporal graph contains several states that map property names to values during specific time intervals. Querying this temporal graph allows for analyzing the causes of system failures by tracking the trajectory of products and monitoring the evolution of machine states. We present in Tables 4.2 and 4.3 the node and relationship states of $N = \{n_0, \ldots, n_{12}\}$ and $R = \{r_0, \ldots, r_{11}\}$ of the temporal property graphs (A and B) presented in Figure 5.1.

We present in Tables 4.2 and 4.3 the node and relationship states of $N = \{n_0, \ldots, n_{12}\}$ and $R = \{r_0, \ldots, r_{11}\}$ of the temporal property graphs (A and B) presented in Figure 5.1.

Let us now discuss the creation of node states $\{n_1^0, n_1^1, n_1^2, n_1^3\}$ in the Toy graph A (Figure 4.1(a)). For instance, the first node state n_1^0 is bound with values $(8, x)$ for property keys (p_0, p_1). This state is valid during $[t_1, t_6)$ since an update of the property p_1 occurred at time instant t_6 which results in a new node state n_1^1. Both node states have the same value for the unmodified property (p_1) and different values for the updated property (p_0). Similarly, the node state n_1^2 is created after the update of the properties p_0 and p_1 at time

instants t_8. Finally, the last modification of the node is an update of the property p_0 at time instant t_{16} which results in a new node state n_1^3 valid in $[t_{16}, \infty)$.

4.4 Temporal query constructs of T-Cypher

This section presents the temporal constructs proposed to extend the Cypher language with temporal features. These query constructs consist of temporal values, a time-slicing clause, temporal functions and operators, and temporal paths. Using these query constructs, we fulfill the required functionalities of temporal slicing (Functionality $\boldsymbol{R_1}$), graph pattern matching (Functionality $\boldsymbol{R_2}$), navigation (Functionality $\boldsymbol{R_3}$), and aggregation (Functionality $\boldsymbol{R_4}$), listed in the Introduction chapter.

4.4.1 Temporal slicing clause

To fulfill the functionality $\boldsymbol{R_1}$, we propose a temporal slicing clause to prune the search space of a query to a single time instant or time interval. Hence, the temporal selection will be applied to all the variables of a temporal query such that the returned states of graph entities should be valid at the requested time instant or during the requested time interval. We use different time slicing techniques using the tokens `SNAPSHOT`, `RANGE_SLICE`, `LEFT_SLICE` and `RIGHT_SLICE`.

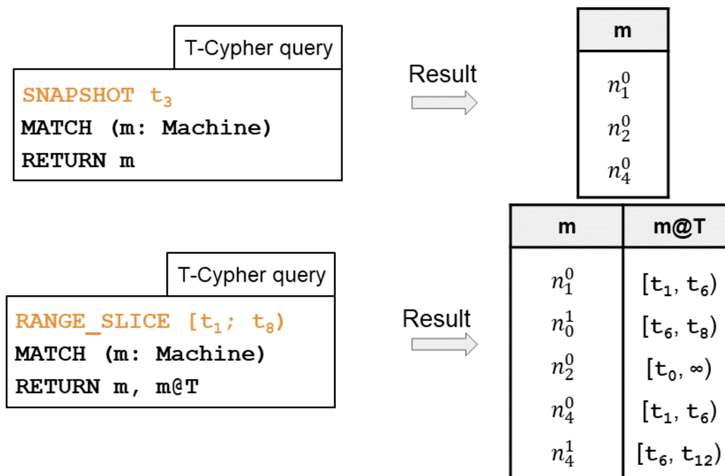


Figure 4.2 – Example of temporal slicing

A query starting with the `SNAPSHOT` token searches for graph entities that are valid at

a single requested time instant. On the other hand, a query starting with a time-slicing token, such as `RANGESLICE`, `LEFTSLICE`, or `RIGHTSLICE`, searches for graph entities whose time intervals intersect with the requested time interval, starts before, or ends after the requested time instant, respectively. If a query does not start with a time-slicing token, it is applied to the latest version of the graph.

Figure 4.2 shows two queries applied to the Toy graph A (Figure 4.1(a)), with their results. The first query returns the machine states valid at t_3, while the second query returns machine states with time intervals intersects with $[t_1, t_8)$.

4.4.2 Temporal functions and operators

To fulfill the functionality $\boldsymbol{R_2}$, we define a set of temporal functions and operators that can be applied to the temporal variables of a pattern to define temporal predicates.

We list the temporal functions in Table 4.4. The semantics of these functions is detailed in the Appendix A. We use Allen's operators to define temporal relations between the temporal variables of a pattern. These operators are illustrated in Figure 4.3. We also provide a definition for each operator in Appendix A.

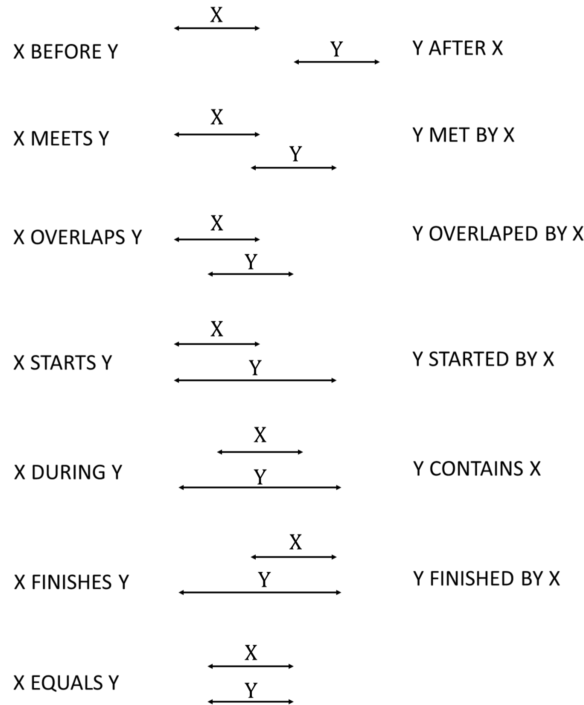


Figure 4.3 – Allen's temporal relations [11]

Function (Syntax construct)	Description	Return type
START(i)	returns the starting time instant of the time interval i	Instant
END(i)	returns the ending time instant of the time interval i	Instant
ADD(i, d)	returns an interval starting with i and ending d time units after i	Interval
SUB(i, d)	returns an interval ending with i and starting d time units before i	Interval
ELAPSED_TIME(i, i')	returns the elapsed time between intervals i and i'	Duration
DURATION(i)	returns the duration of interval i	Duration
INTERSECTION($i_0, \ldots, i_n$)	returns the intersection between intervals $\{i_0, \ldots, i_n\}$	interval
RANGE($i_0, \ldots, i_n$)	returns the time range of intervals $\{i_0, \ldots, i_n\}$	interval

Table 4.4 – Description of temporal functions used in T-Cypher

Figure 4.4 provides an example of a T-Cypher query using temporal functions and operators and its result when applied to the Toy graph A (Figure 4.1(a)). This query returns the elapsed time [2] between the maintenance of a machine and its failure. The failure of a machine can be detected if the value of property p_0 (e.g., temperature) is higher than a threshold. The expression (`n@T AFTER e@T`) indicates that the system failure must have occurred after the maintenance. We notice that the machine state n_1^2 is returned since it has a value of p_0 higher than the threshold and it occurred after the maintenance of the machine.

4.4.3 Temporal paths

The relationships in a temporal graph are valid during certain time intervals. Hence, the connectivity between two nodes can be subject to temporal conditions defined over the relationships of a path which results in diverse types of temporal paths. To fulfill requirement $\boldsymbol{R_3}$, we refer to three temporal types that can cover a large subset

2. The elapsed time between two time intervals i and i' is equal to the difference between the starting time instant of i' and the ending time instant of i.

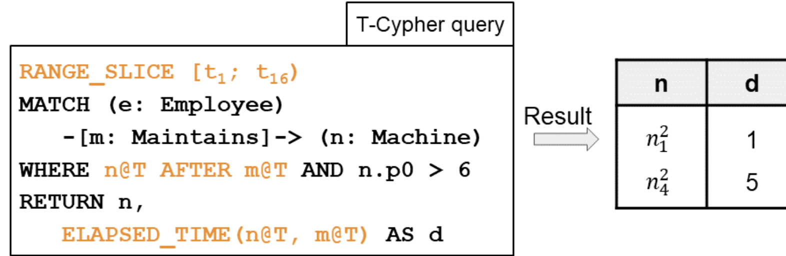


Figure 4.4 – Example of temporal functions and operators

of queries: Continuous, Sequential, and Pairwise-continuous (Figure 4.5). A **Continuous path** is defined as a path having a non-empty intersection between the time validity intervals of all its relationships [201]. A **Sequential path**, also known as the time-respecting path, is defined as a path where the outgoing relationship of each path node occurs after the incoming relationship to the same node [244, 136, 185, 198]. A **Pairwise-continuous path** is a path where the time interval of the outgoing relationship of each path node intersects with the time interval of the incoming relationship to the same node [65]. Since continuous paths imply a strict condition on the relationships, pairwise-continuous paths represent a simplified variant of continuous paths. In the following, we define temporal paths, then introduce continuous, sequential, and pairwise-continuous paths.

Temporal path

A temporal path is defined as a tuple $(n_1^s, r_1^s, \ldots, r_k^s, n_{k+1}^s, \tau_p)$ containing a sequence of k relationship states $(r_i^s, \forall\, 1 < i < k)$ and $k+1$ node states $(n_i^s, \forall\, 1 < i < k+1)$ and a time interval during which the path is valid. Each relationship state $(r_i^s, \forall\, 1 < i < k)$ is a tuple $(id_{n_i}, id_{n_{i+1}}, t_{r_i^s}, k_{r_i^s}, \tau_{r_i^s})$ connecting two node states of the path $n_i^s = (id_{n_i}, l_{n_i^s}, k_{n_i^s}, \tau_{n_i^s})$ and $n_{i+1}^s = (id_{n_{i+1}}, l_{n_{i+1}^s}, k_{n_{i+1}^s}, \tau_{n_{i+1}^s})$. The time interval of the path τ_p is derived from the time intervals of the path relationships and depends on the type of the temporal path.

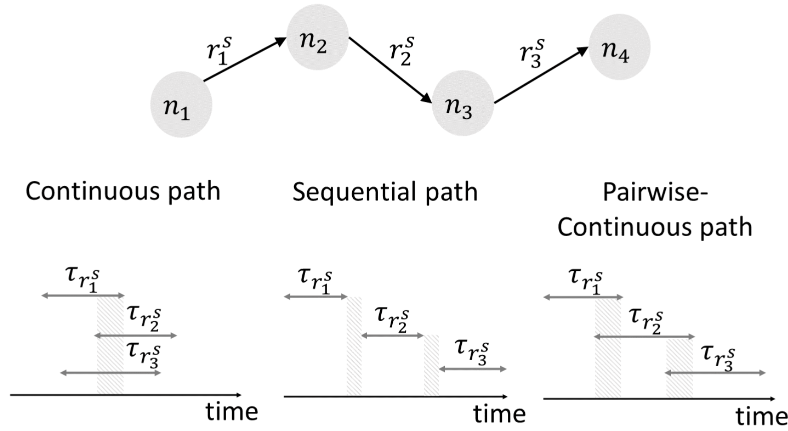


Figure 4.5 – Different types of temporal relationship patterns: Continuous, Pairwise Continuous and Sequential ($\tau_{r_1^s}$, $\tau_{r_2^s}$ and $\tau_{r_3^s}$ refer to time validity intervals of relationships r_1^s, r_2 and r_3^s)

Continuous path

A continuous path is a temporal path where the intersection between the time intervals ($\tau_{r_i^k}, \forall\, 1 < i < k$) of the relationship states (r_i^k) of the path is an interval of non zero duration and τ_p is equal to the intersection between time intervals $\{\tau_{r_1^s}, \ldots, \tau_{r_k^k}\}$.

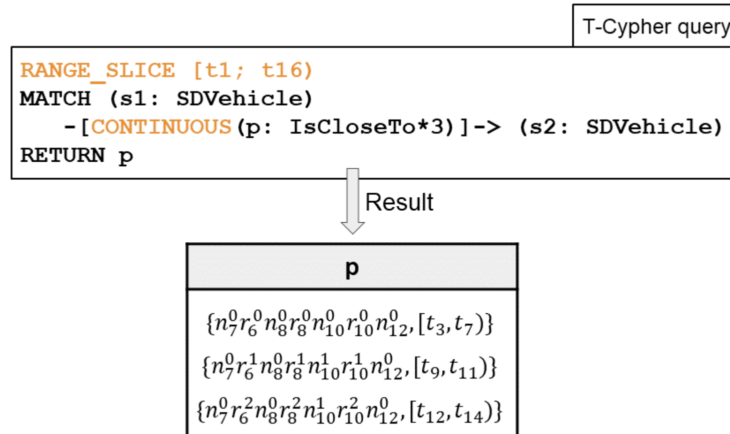


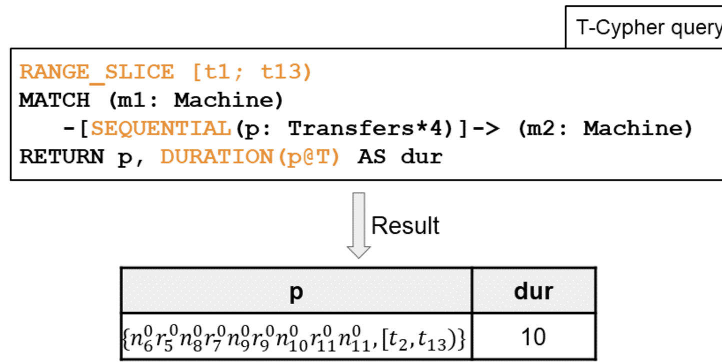
Figure 4.6 – Example of a continuous path

Figure 4.6 presents a T-Cypher query with a continuous path and its result when applied to Toy graph B (Figure 4.1(b)). This query returns the path between self-driving vehicles that were 3-Hop close to each other during the time interval $[t_1, t_{16})$. Hence, the self-driving vehicles of the path were close during the intersection of the time intervals

of the path relationships. Notice that three continuous paths of length 3 exist between the self-driving vehicles n_7 and n_{12}. The time interval $[t_3, t_7)$ of the first path is equal to the intersection between the time intervals of its relationship states ($[t_3, t_7)$, $[t_3, t_8)$ and $[t_3, t_9)$).

Sequential path

A sequential path is a temporal path where each relationship state r^s_{i+1} should occur after the relationship state r^s_i ($\forall\, 1 \leq i < k$). Hence, the ending time instant of $\tau_{r^s_i}$ should be lower than the starting time instant of $\tau_{r^s_{i+1}}$. The time interval of the path is the range of time covered by the time intervals of the path.



p	dur
$\{n^0_6 r^0_5 n^0_8 r^0_7 n^0_9 r^0_9 n^0_{10} r^0_{11} n^0_{11}, [t_2, t_{13})\}$	10

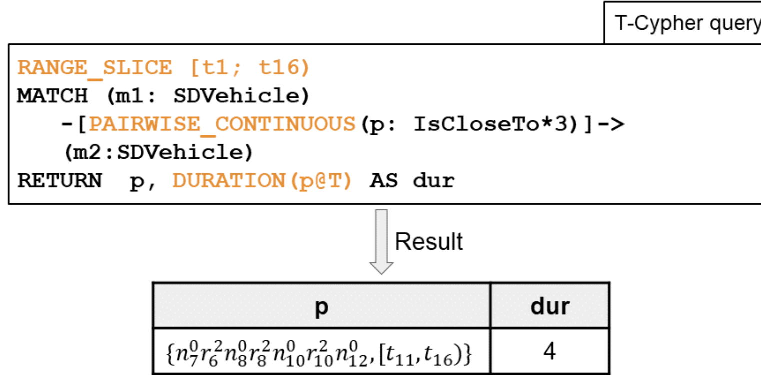
Figure 4.7 – Example of a sequential path

Figure 4.7 presents a T-Cypher query with a sequential path and its result when applied to the Toy graph B (Figure 4.1(b)). It returns a product's transfer path of length 4 between two machines, implying that a self-driving vehicle or a machine transfers a product after receiving it. Note that a sequential path of length 4 exists between the node states n_3 and n_9. This path is valid during the time interval $[t_2, t_{13})$ that represents the range of the time intervals of its relationship states ($[t_2, t_4)$, $[t_5, t_7)$, $[t_8, t_{10})$ and $[t_{11}, t_{13})$).

Pairwise-continuous path

A pairwise-continuous path is a temporal path where the time interval of each relationship state r^s_i should overlap with that of the outgoing relationship state r^s_{i+1}

($\forall\, 1 \leq i < k$). Therefore, $\tau_{r_i^s}$ starts within the time boundaries of $\tau_{r_{i-1}^s}$ and ends within the time boundaries of $\tau_{r_{i+1}^s}$.



p	dur
$\{n_7^0 r_6^2 n_8^0 r_8^2 n_{10}^0 r_{10}^2 n_{12}^0, [t_{11}, t_{16})\}$	4

Figure 4.8 – Example of pairwise-continuous path

Let us now consider that a vehicle a transfers a product to a close vehicle b. Now, b also looks for a close vehicle, c, and transfers the product to it. Similarly, the vehicle c transfers a product to a close vehicle d. The path between the vehicles is pairwise continuous since the time intervals of each pair of consecutive relationships are overlapping. To illustrate, Figure 4.8 presents a T-Cypher query with a pairwise-continuous path and its result when applied to the Toy graph B (Figure 4.1(b)). Notice that a single row is returned, corresponding to the pairwise-continuous path between node states n_7 and n_{12}. The time interval of the path $[t_{11}, t_{16})$ is equal to the range of the time intervals of its relationship states ($[t_{11}, t_{14})$, $[t_{12}, t_{15})$ and $[t_{13}, t_{16})$).

4.4.4 Temporal aggregation

A temporal aggregation query computes for a single node an aggregated value, in a range of time, based on the properties of the node itself, its relationships, or neighbors. We list in the following some query examples that include temporal aggregation.

The following query returns the minimum duration when the property p_0 of a machine state was less than a threshold 6 after t_0. The grouping key (m) is defined in the returned statement to group the values going into the aggregate functions that share the same value for the node state m.

```
RIGHT_SLICE t0
```

```
MATCH (m: Machine)
WHERE m.p_0 < 6
RETURN m, MIN(DURATION(m@T)) AS min
```

This query returns the following relation when evaluated on Toy graph A (Figure 4.1(a)). We notice that the value of m is a set of node states. These states are returned by the match clause for instance n_1^1 is the only state of n_1 satisfying the condition $m.p_0 < 6$ and $\{n_4^0, n_4^1\}$ are the states of n_4 satisfying this condition. The grouping key m indicates that these states are grouped based on a common node identifier. For example, node states n_4^0 and n_4^1 are grouped since they refer to the same node n_4. The aggregated value is the minimum duration of the grouped states.

m	min
$\{n_1^1\}$	1
$\{n_4^0, n_4^1\}$	4

The following query returns the maximum manufacturing time of products by machines during $[t_7, t_{15})$. Similarly to the previous query, the grouping key is the node state variable m.

```
RANGE_SLICE [t7;t15)
MATCH (p: Product) -[i:IsIn]->
(m: Machine)
RETURN m, MAX(DURATION(i@T)) AS max
```

This query returns the following relation when evaluated on Toy graph A (Figure 4.1(a)). The node states satisfying the matching pattern in $[t_7; t_{15})$ are $\{n_3^1, n_3^2\}$. These states are grouped since they refer to the same node n_3. The aggregated value is the maximum duration of the relationship i, indicating the time during which a product was inside a machine. The returned value 1 corresponds to the duration of the relationship r_3^0.

m	max
$\{n_3^1, n_3^2\}$	1

4.5 Syntax of T-Cypher

We present in this section the syntax of T-Cypher. We start by describing the syntax of temporal values, including time instants, intervals, and duration. We then describe

the grammar rules of expressions mainly used in the *where* clause to set the filtering constraints on the variables of a query. Also, we present the grammar rules of graph patterns. Finally, we describe the syntax of queries, including the *time slicing* and *return* clauses.

4.5.1 Temporal values

In T-Cypher, a temporal value can be either an instant, interval, or duration. In the following, we show the grammatical rules applied to these values. Note that these rules are listed in Figure 4.9.

v^T ::= instant | interval | duration	
instant ::= cd | epoch	cd is a calendar date
interval ::= [instant; instant)	
epoch ::= **EPOCH** t	$t \in \Omega^T$
duration ::= n g	$n \in \mathbb{N}$ and g is a temporal granularity

Figure 4.9 – Syntax of temporal values used in T-Cypher

A **time instant** is expressed either as a calendar date or as the total number of chronons since Epoch. We use ISO-8601 format to describe calendar dates (e.g., 2022-07-18T12:33:00). Assuming that the system defines a time domain with a granularity of milliseconds and a default time zone GMT, then `2021-03-08T08:00:00` is parsed to a time instant whose value is equal to 1615190400000.

A **time interval** is composed of a starting and ending time instant (e.g., `[2021-03-08T00:00:00; 2021-03-09T00:00:00)`).

A **duration** is expressed as the number of chronons followed by a granularity that can be of days, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds (e.g., `1 day`, `15 sec`). For instance, a duration can be used to filter temporal variables such as filtering the states of a node variable (`a`) that lasted for more than 2 hours (`DURATION(a@T) > 2 hour`). We consider that each system stores temporal data based on a single granularity (e.g., milliseconds) where all the duration values used in a T-Cypher query are internally converted to the system granularity during the parsing phase (e.g., `1 hour` is parsed to 3600000 milliseconds). If the duration value is

expressed with a granularity lower than the system's, then a semantic error should be raised when parsing the query.

4.5.2 Expressions

Expressions are primarily used in the *where* clause to add predicates on the variables of a T-Cypher query. We present the grammar rules applied to expressions in the following. These rules are listed in Figure 4.10.

```
ex ::=   v | a | f(ex_list)                                          v ∈ V, a ∈ A, f ∈ F    values/variables
         | ex. k | {prop_list} | {}                                                          maps
         | [ex_list] |ex IN ex | [ ]                                                         lists
         | ex[ex] | ex[ex. . ] | ex[. . ex] | ex[ex. . ex]
         | ex STARTS_WITH ex| ex ENDS_WITH ex|ex CONTAINS ex                                 strings
         | ex OR ex |ex AND ex| ex XOR ex | NOT ex | ex IS_NULL | ex IS_NOT_NULL              logic
         | ex < ex | ex <= ex | ex >= ex | ex > ex | ex = ex| ex <> ex                         comparison
         | ex BEFORE ex | ex MEETS ex | ex OVERLAPS ex | ex STARTS ex                         temporal comparison
         | ex DURING ex | ex FINISHES ex | ex AFTER ex | ex MET_BY ex
         | ex OVERLAPPED_BY ex | ex STARTED_BY ex | ex CONTAINS_T ex
         | ex FINISHED_BY ex | ex EQUALS ex

ex_list ::= ex | ex, ex_list
```

Figure 4.10 – Syntax of expressions used in T-Cypher (V is the set of values (including temporal values), A is the set of names, and F is the set of functions (including temporal functions))

An expression can represent a value $v \in V$ where V is the set of values, including temporal values whose syntactic rules are given in Section 4.5.1 and can represent an instant, interval, or duration. Besides, an expression can represent a variable ($a \in A$) where A is the set of names. Similarly, an expression can represent a temporal variable `a@T` applied on a non-temporal variable `a`. If `a` represents a node state variable, then `a@T` is interpreted as a temporal variable representing the validity interval of `a`. An expression can also represent a function $f \in F$ where F represent the set of functions including the temporal functions listed in Appendix A (e.g., duration and intersection

functions). We also define the rules of expressing lists and maps. For instance, one can express a condition on the property value of a node using the mapping rule `ex.k` (e.g. `a.Measurement` where `a` is a node state variable and `Measurement` is a property name). Also, we use string operators (e.g. (`STARTS WITH`), logic (e.g. (`OR`)) and comparison operators (e.g. <) that are inherited from the Cypher language. We include temporal comparison operators representing Allen's temporal relations (e.g., `BEFORE`). For instance, the expression `a@T STARTS b@T` indicates that the node state (`a`) should have ended when the node state (`b`) started. Whereas the expression `a@T OVERLAPS [2021-03-08T00:00:00; 2021-03-09T00:00:00]` indicates that the validity interval of the node state (`a`) should overlap with the given time interval.

4.5.3 Patterns

A graph pattern is expressed in the *Match* clause that can be followed by a *Where* sub-clause. The grammar rules of patterns are listed in Figure 4.11. The `pattern_tuple` is used to define a sequence of patterns composed each of a `node_pattern` or a concatenation of a `node_pattern`, relationship pattern (`rel_pattern`) and a pattern.

A node pattern comprises an optional name that refers to the node state, an optional list of labels, and an optional map. The optional map filters the node states based on the values of their properties. For instance, applying the *match* clause: `MATCH (a: Machine id:` n_1`,` p_0`: 5,` p_1`:` x`)` on the toy graph presented in Figure 4.1(a) returns the node state that corresponds to the state of the machine n_1 having values $(5, x)$ for properties (p_0, p_1), hence, the state n_1^1 that was valid during the time interval $[t_6, t_8)$.

A relationship pattern can either be left-to-right, right-to-left, or undirected. The temporal relationship details (`temp_rel_details`) are used to define the temporal type of the path, the variable name, the list of types, the length, and the property values. The temporal path parameters (`temp_path_param`), based on the **type** of the temporal path, can be used as follows:

— **Continuous path**: Minimum or maximum duration of the intersection between the time intervals of the relationships of the path.

— **Pairwise-continuous path**: Minimum or maximum duration of the intersection between a pair of structurally consecutive relationships of the path.

— **Sequential path**: Minimum or maximum elapsed time between a pair of structurally consecutive relationships of the path.

```
clause ::= MATCH pattern_tuple [WHERE expr]                                   relational clauses
pattern_tuple ::=pattern | pattern, pattern_tuple                             tuples of patterns
pattern ::= node_pattern | (node_pattern rel_pattern pattern)
node_pattern ::= (a? label_List? map? )
rel_pattern ::= – [temp_rel_details] → | ← [temp_rel_details] –
               |–[temp_rel_details] –
temp_rel_details ::= temp_rel_type (rel_details? ) temp_path_param ? |
                     rel_details ?
temp_path_param ::= (MIN_DELTA | MAX_DELTA) ? duration
rel_details ::= a? type_list? len? map?                                       a ∈ A
label_list ::= : l | : l label_list                                           l ∈ L
type_list ::= :t | :t type_list                                               t ∈ T
map ::= {prop_list}
prop_list ::= k: expr | k: expr, prop_list                                    k ∈ K
Len ::= ∗ | ∗ d | ∗ d1.. | ∗..d2 | ∗ d1..d2                                   d1, d2 ∈ ℕ
temp_rel_type ::= CONTINUOUS, PAIRWISE_CONTINUOUS, SEQUENTIAL
```

Figure 4.11 – Syntax of patterns used in T-Cypher

4.5.4 Queries

A T-Cypher query starts with an optional slicing statement used to specify a time instant or interval that is, by default, applied to all the variables of a query. This statement filters the valid variables at the requested time instant or during the requested time interval. The `SNAPSHOT` token is followed by a time instant to indicate that all query variables must be valid at this time instant. The `RANGE_SLICE` token is used to set the bounds of a time interval such that the validity intervals of these variables must intersect with the requested time interval. The `LEFT_SLICE` or `RIGHT_SLICE` tokens are used to set the right or left bounds of the time interval indicating that the variables of the query should exist before or after that time instant, respectively. We recall that the slicing statement is optional, and when omitted, it applies to the most recent snapshot of the graph.

Similar to a Cypher query, a T-Cypher query ends with a *return* statement. This

```
queryT ::= slice ? query                                                  slicing
query ::= RETURN ret | clause query                            clause sequences
ret ::= * | expr [AS a] | ret, expr [AS a]                         return lists
slice ::=( slice | SNAPSHOT instant_list )
slice ::=RANGESLICE interval_list |( RIGTHSLICE | LEFTSLICE ) t        t ∈ Ω^T
instant_list ::= instant | instant, instant_list
interval_list ::= interval | interval, interval_list
```

Figure 4.12 – Syntax of T-Cypher queries

statement is used to indicate which query variables should be returned. Each returned expression can be optionally named using the token `AS` followed by an alias. For instance, `RETURN INTERSECTION(a@T, b@T) AS i` returns the intersection between the time intervals of node states (`a` and `b`) and binds the expression with the alias i which will be referred to in the returned result instead of the full expression.

4.6 Industrial integration of T-Cypher

In this section, we present the implementation details of T-Cypher in Thing'in [3]. As previously mentioned, Thing'in is an Orange-initiated platform managing a graph of IoT objects. This platform proposes to its clients a customized query language TiQL and uses ArangoDB [172] as a backend store such that TiQL queries are translated into AQL (official language used in ArangoDB) queries.

Regarding the importance of querying the history of the graphs managed by Thign'in, we added a time version support, allowing storing and querying the history of these graphs. Although our ultimate goal is the integration of our system Clock-G into the platform Thing'in, building a robust data management system requires a team of full-time engineers and years of progressive development. Regarding the urgent demand for temporal support by the clients of Thing'in, we decided to rely on an existing non-

3. https://wiki.thinginthefuture.com/

temporal graph database to accelerate the implementation. The details of this implementation will be described in the following.

We present in Figure 4.13 the platform's new architecture, including the added temporal layer. Note that we developed the temporal layer using the programming language Java. As shown in the figure, the clients of Thing'in can send CRUD requests (Create, Read, Update, and Delete) such that the read requests can be written in T-Cypher or TiQL languages. We prefer to keep both languages since an abrupt elimination of the original language (TiQL) is not convenient for users who are already familiar with the language and building applications using it. The main difference between both languages is that TiQL queries can only investigate the current graph state (i.e., the last updated version), whereas T-Cypher enables querying past and current graph states. To handle the history of the graph, we choose to integrate a Neo4j server [175]. Although running two separate database systems create consistency problems, using Neo4j simplifies the task of translating T-Cypher queries and evaluating them. Hence, we decided to keep an ArangoDB server to manage the most recent version of the graph and a Neo4j server to manage its history.

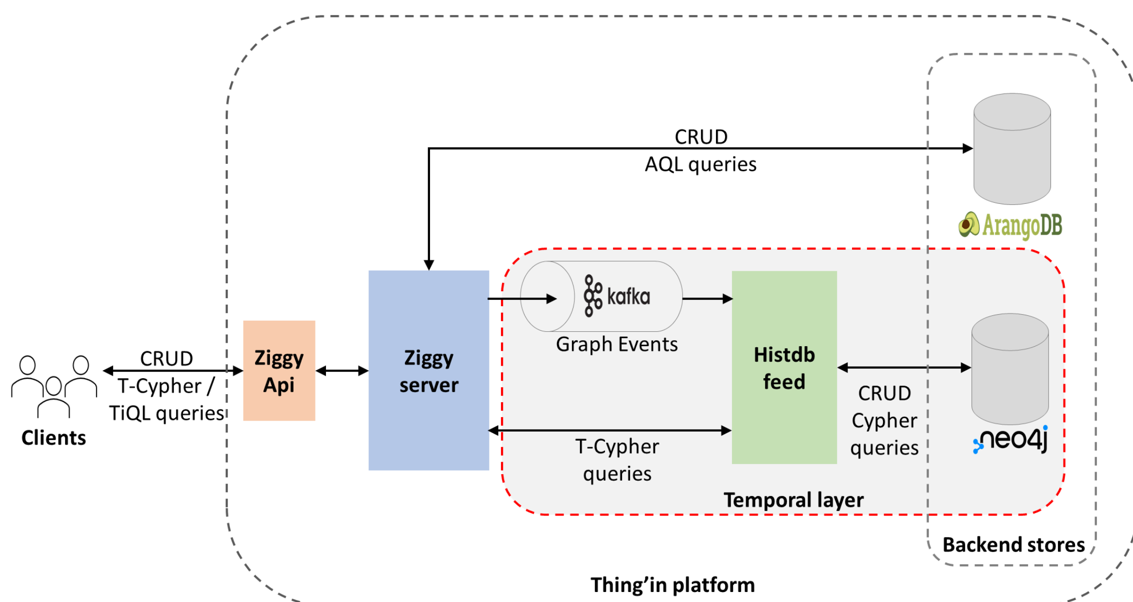


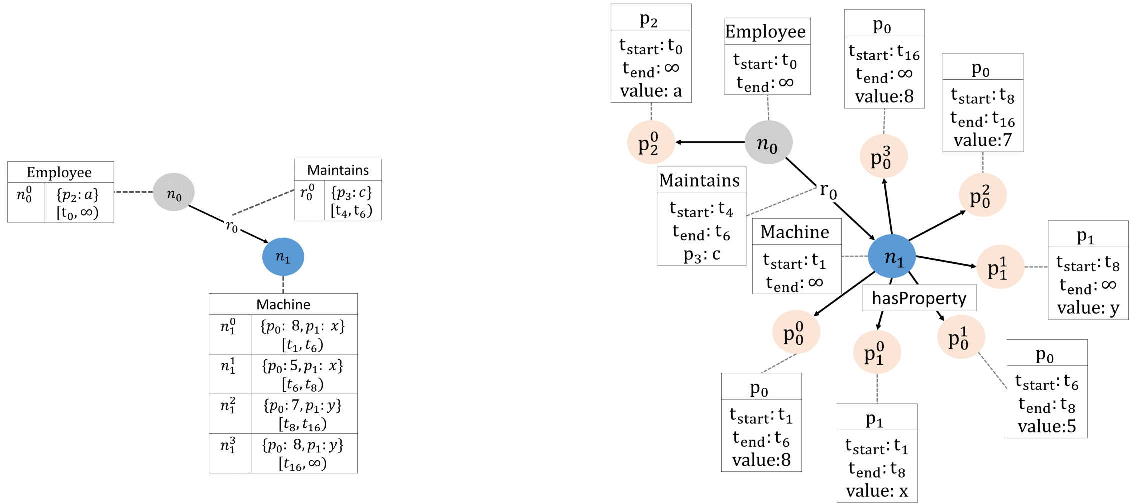
Figure 4.13 – Architecture of the Thing'in platform with the added temporal layer enabling the execution of T-Cypher queries

As presented in Figure 4.13, the **CRUD** requests are received by **Ziggy API**, which sends these requests to the Ziggy server, which is a critical component in the architecture of Thing'in. The Ziggy server is primarily responsible for interpreting the API requests and converting them into queries that can be evaluated against the backend stores.

When receiving a create, delete or update query, the **Ziggy server** will first send the equivalent operation to ArangoDB to update the last version of the graph. Then, it will assign the operation the current timestamp and send it as a graph event to the **Kafka channel**. These graph events will be collected by **Histdbfeed**, responsible for managing the temporal dimension of the graph using Neo4j. These requests can be sent and collected individually or in batches. We fixed two criteria for reading from the Kafka channel. The criterion with the highest priority is setting the number of graph events in a batch. In other words, when this threshold is reached, Histdbfeed will read all the graph events from the channel. This has the advantage of controlling the number of inserted graph events into Neo4j. Now, the batched write queries will be performed in a single transaction to limit the overhead of creating and running many transactions. The second criterion consists of fixing the duration of the graph events in the Kafka channel, which forces the collection of graph events after a period of time even if the threshold number is not reached. In some situations where the rate of the received graph events is relatively low, the graph events can become *stale* by the time of collecting them by Histdbfeed and persisting them in Neo4j. To ensure the persistence of these graph events regardless of their rates, we fixed the duration after which they will be collected and persisted. The HistdbFeed reads the graph events from the Kafka channel to execute their equivalent Cypher write queries against Neo4j.

After receiving a T-Cypher query, the Ziggy server will send it to Histdbfeed, translating it into a Cypher query that can be evaluated against Neo4j. The translation of T-Cypher queries into Cypher queries is based on syntax and model translation rules we present in the next section. The result of an evaluated query is then converted from Neo4j's default format into customized formats that we have proposed to serve the needs of different endpoint applications. For instance, the result can be converted into **Full** or **Log-based**. The **Full-based** format returns all the node and relationship states satisfying a given pattern, including their starting and ending time instants. The **Log-based** format, however, returns the result as a series of timestamped logs (i.e., graph events) that corresponds to the creation or deletion of a node or relationship or

an update of their properties. This format is preceded by a complete snapshot of the returned subgraph, allowing the reconstruction of the result valid at the beginning of the requested time interval.



(a) Representation of a subgraph with the temporal property graph model

(b) Translated representation of a subgraph with the non-temporal property graph model

Figure 4.14 – Example of a temporal graph represented with the logical model and its translation into the non-temporal graph model

4.6.1 Model translation rules

The model used to represent the graph in Figure 5.1 is the logical model practitioners should follow when reasoning about T-Cypher queries. This model is translated into the non-temporal graph model supported by Neo4j by representing time intervals using properties. It is important to point out that this translation is completely transparent for end users. For each node or relationship occurrence, a new node or relationship instance is created with properties representing the starting and ending time instants of the occurrence (t_{start}) and (t_{end}). For each occurrence of a new property value of a node, a new property node is created that is connected to the actual node with a relationship having the type (*hasProperty*). This node is attached with three properties: (*value*), (t_{start}) and (t_{end}) to set the value and validity interval of the corresponding dynamic property.

Figure 4.14 illustrates the translation of a subgraph extracted from the toy graph in Figure 5.1. As shown in the figure, each node has two properties (t_{start}) and (t_{end}) indicating its time interval and is attached to several property nodes with a *hasProperty* relationship. Each property node corresponds to a single property value valid during a time interval defined by the properties t_{start} and t_{end}. For example, the node n_1 is attached to 6 property nodes. The property node p_0^0 connected to n_0 includes the property p_0, the value 8 and the time interval $[t_1, t_6)$ by the setting the properties t_{start} and t_{end}.

4.6.2 Query translation rules

To translate a T-Cypher query into a Cypher query, we use the parser generator ANTLR [188] and the query translation rules listed in this section. Figure 4.15 shows the query translation pipeline. First, a **lexer** breaks up a T-Cypher query into vocabulary symbols (i.e., tokens) and sends the symbol stream to the **parser**. The parser will then convert this stream into an **A**bstract **S**yntax **T**ree (AST) representing its content. The **query builder visitor** traverses the AST to analyze its content. In our translation procedure, a visitor traverses the tree to generate a **query object**. This query object is recognized by the **Cypher query constructor**, whose role is to interpret the query object and convert it into an equivalent Cypher query based on our **query translation rules**. To clarify these rules, we give in Figure 4.16 the translation of a T-Cypher query into a Cypher query. T-Cypher and the fragment of Cypher that we extended have the same expressive power, which allows the translation of any T-Cypher query into a Cypher query.

Temporal slicing clause

A temporal slice implies that the node and relationship states should exist in a requested time interval. To translate temporal slicing, we add a conjunction of temporal predicates on the properties (t_{start} and t_{end}) referring to the starting and ending time instants of the nodes and relationships. For example, to convert the right slice in the T-Cypher query of Figure 4.16, we add for each node and relationship variable, defined in the query, a temporal condition in the Where sub-clause such as `u.tEnd > t` to indicate that the node state u should begin end after the requested time instant t. To get the property nodes, we call a user-defined procedure that yields a list of all the

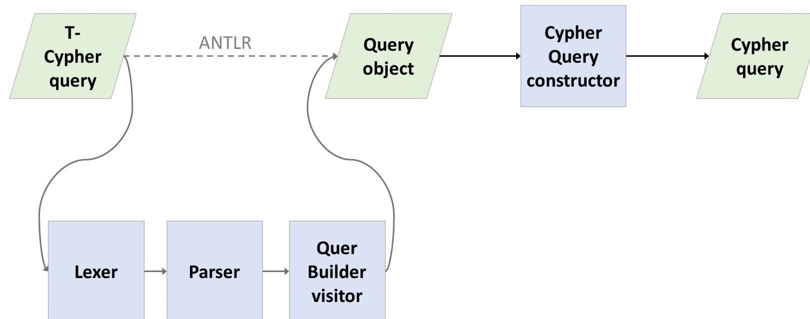


Figure 4.15 – Pipeline of translating a T-Cypher query into Cypher query

property nodes connected to each node variable defined in the query. We also specify in the called procedure the type of the temporal slice to exclude the property nodes not valid in the requested time interval or at the requested time instant. As presented in the example (Figure 4.16), we add `CALL proc.getPropertyNodes` on nodes u and v to get their property nodes. We also include a filtering condition that excludes the property nodes and the relationships connecting them to the original nodes. In our example, the label of node variables u and v is not specified. Hence, we add the condition `NOT(u:‘property‘) AND NOT(v:‘property‘)`.

Temporal relationship pattern

Rigid temporal relationship patterns with the same minimum and maximum depth (n) are translated to n intermediate one-hop relationship patterns. Then, the temporal constraints applied to each type of temporal relationship pattern (continuous, sequential, or pairwise-continuous) are added to the Where sub-clause. Whereas queries with temporal relationship patterns of variable length between n and m are first converted into the union of sub-queries with a rigid pattern. The length of these rigid patterns varies between n and m. Each sub-query is then translated based on the translation rules of a rigid pattern. Our query example (Figure 4.16) includes a temporal path of variable length. Hence, we create a Cypher query corresponding to each possible depth (between 2 and 3), then combine their results using the union operator. To include the temporal condition of the path, we add temporal constraints to the Where

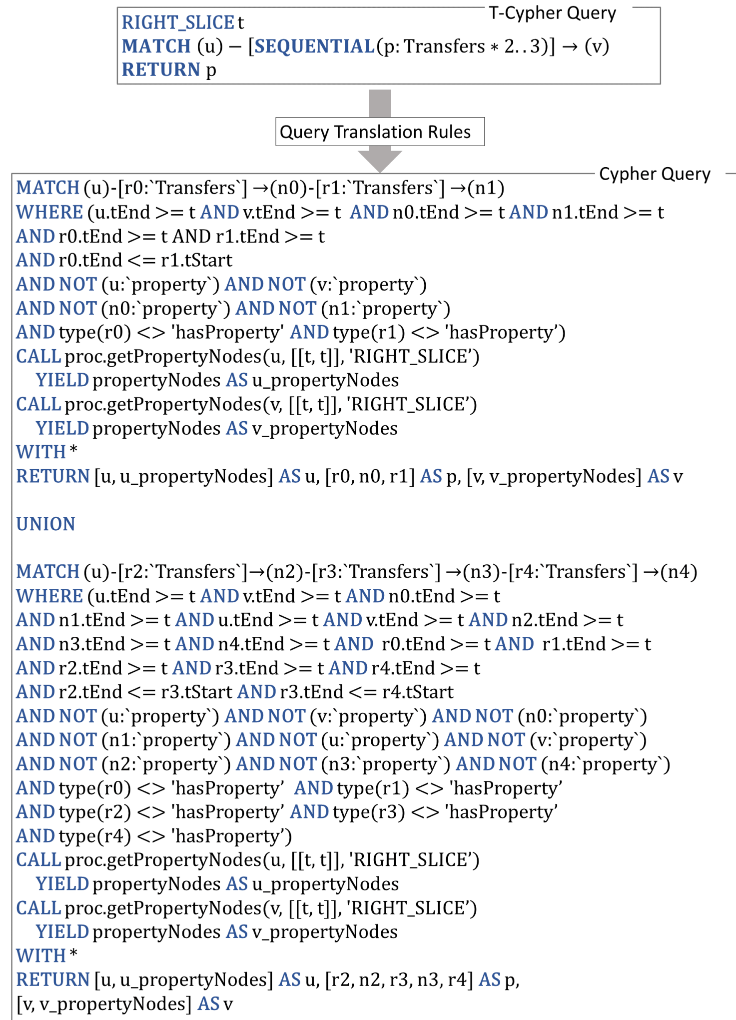


Figure 4.16 – A T-Cypher query and its Cypher translation

sub-clause. For example, we add the constraint (`r0.tEnd <= r1.tStart`) to indicate the sequentiality of the consecutive relationship variables $r0$ and $r1$.

Temporal functions and operators

An expression with temporal operators or functions is translated to a Cypher-based expression that is semantically equivalent using the built-in comparison operators and functions. For example, expression (`a@T BEFORE b@T`) is translated to a valid Cypher expression (`a.tEnd < b.tStart`). However, functions and operators that cannot be ex-

pressed with the syntax of Cypher are implemented as Neo4j's user-defined functions [4]. These functions added as plugins can be used in Cypher queries as built-in functions.

We can conclude from the example given in Figure 4.16 that the translated Cypher query is much more verbose than the T-Cypher query, which is a key advantage of using a native temporal query language.

4.7 Conclusion

We present in this chapter our proposed query language, T-Cypher. We first defined preliminary concepts to introduce the language, such as the temporal domain and temporal graph data model. Then, we presented the added temporal constructs with their syntax. Finally, we described the integration details of T-Cypher into the Thing'in platform. This process consists of coupling a non-temporal graph database with a temporal layer. This layer is responsible for converting the temporal graph model into the non-temporal counterpart and T-Cypher queries into Cypher ones. By presenting the query translation rules, we showed that Cypher queries are more verbose than their equivalent T-Cypher queries, which motivates using a query language with special temporal constructs.

4. https://neo4j.com/docs/cypher-manual/current/functions/user-defined/

Chapter 5

Temporal graph storage technique

This chapter presents our proposed storage technique that targets the reduction of the data volume while maintaining query latencies. The main goal is to integrate this storage technique into our system Clock-G.

As presented in Chapter 2, several approaches have been proposed to store temporal graphs, such as the *Log* and *Copy+Log* methods. The *Log* approach [94, 93] consists of storing graph updates as a series of timestamped logs which induces a severe performance bottleneck since recovering the state of the graph at a single time instant requires reloading all graph updates whose timestamps are lower than the requested one. To reduce query latency, the *Copy+Log* approach [138] consists of storing graph updates in temporally disjoint time windows along with snapshots representing each graph's state valid between two successive time windows. Evaluating a temporal query at a given time instant implies reading from the closest snapshot and constructing the initial state of the result valid at the time instant of the snapshot. Then, subsequent graph updates are loaded from the time window following that snapshot and applied incrementally on the returned result until reaching the requested time instant. Hence, this storage technique prunes the search space of the query evaluation process to, at most, a single snapshot and its subsequent time window. These methods present an apparent trade-off between space and query execution time, such that space usage is favored over query execution time or vice-versa. Besides, the underlying mechanism of the *Copy+Log* method materializes the state of the entire graph regularly, which is space-consuming, significantly when relatively small portions of the graph are changing while the rest remains static. This characteristic can be found in the majority of real-world temporal graphs. Notably, a large part of the graph of Thing'in is primarily static, implying that successive graph snapshots share many similarities. In the use case of smart factories, the relationships representing the connections between the machines and sensors, machines and locations (rooms, buildings, etc.) are almost static, and only the relations between the products and machines are dynamic. The problem with

the *Copy+Log* method is that unchangeable graph entities will be copied repeatedly in each snapshot which implies a considerable amount of redundant graph entities across snapshots.

In this chapter, we propose the δ-*Copy+Log* storage approach to mitigate the cost of storing full graph snapshots. Our solution mainly differentiates from the *Copy+Log* by replacing graph snapshots with deltas representing the difference between a pair of consecutive snapshots. Hence, a delta contains a set of graph operations between two snapshots. In this set, we omit the graph operations occurring between consecutive snapshots and canceling each other. For example, the addition and deletion operators of the same node that occur sequentially between two graph snapshots are canceling operations. Besides, we propose an optimization technique to mitigate the query execution overhead caused by storing deltas instead of snapshots. This technique is applied on N-Hop traversal queries and assigns each delta with a Bloom filter that will be checked whenever data is requested from that delta.

We also conducted experiments on real and synthetic datasets. The results align with previous findings on the apparent trade-off between space and query execution time of the *Copy+Log* and *Log*. Furthermore, a comparison between these traditional methods and the δ-*Copy+Log* validates the superior performance of our proposed method as it offers a compromise between the space usage and query execution time.

This chapter is organized as follows. Section 5.1 defines a formal model of temporal property graphs. Section 5.2 describes our proposed storage technique δ-*Copy+Log*. Section 5.3 shows the implementation details of the δ-*Copy+Log* method and its integration into Clock-G. Section 5.4 presents the results of evaluating the performance of Clock-G with large-scale real-world and synthetic temporal graphs and validates the gain of using δ-*Copy+Log* technique compared to traditional methods.

5.1 Preliminaries

5.1.1 Temporal property graph model

In this section, we introduce the Operation-based Property Graph Model (OPGM) used throughout the Chapter to define key concepts of our proposal.

Let V and E denote finite sets of node and relationship identifiers (ids), respectively. Nodes and relationships can have a single label and a set of dynamic property keys

denoted as P. Let R denote an infinite set of atomic values that can have any type from a finite set of data types D (e.g., string), L denote a finite set of strings, id denote an identifier, 2^X denote the set of all finite subsets of the domain X. Let Ω^T denote the time domain defined in Chapter 4 (Section 4.2.1). Finally let a, d and u denote an addition, deletion and update respectively. We define a graph operation ϵ^i as an action applied on a graph entity that translates to an addition/deletion of a node/relationship or the update of a dynamic property. We define finite sequence of temporally ordered graph operations $\Upsilon = \{\epsilon^i, i \in \mathbb{N}\}$. Following the OPGM, a temporal property graph model is a tuple:

$$G^O = \left\{\Upsilon, V, E, P, \rho, \alpha, f^G, f^T, f^E\right\}$$

We explain the functions below:

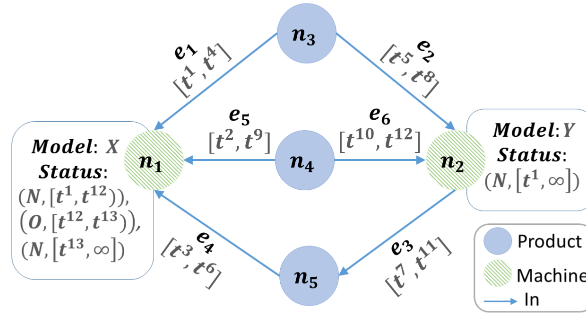
— $\rho : (V \cup E) \rightarrow 2^L$ maps each node and relationship to a finite set of labels from L.
— $\alpha : E \rightarrow (V \times V)$ maps each relationship to its source and target ID.
— $f^G : \Upsilon \rightarrow (V \cup E \cup ((V \cup E) \times R \times P))$ maps a graph operation to its corresponding graph entity.
— $f^T : \Upsilon \rightarrow D^T$ maps a graph operation to its corresponding time instant.
— $f^E : \Upsilon \rightarrow \{a, d, u\}$ maps a graph operation to an addition (a), deletion (d) or update (u).

Example 5.1.1. To illustrate, we provide an example of a toy graph inspired by the use case of a smart factory. Figures 5.1(a) and 5.1(b) illustrate a temporal property graph and its representation based on the OPGM model. The temporal graph models the dynamic connections between the machines and products and the different values recorded by the sensors monitoring the state of the machines. We show how, for a sample of nodes and relationships, the history of this graph can be represented using the previous definitions:

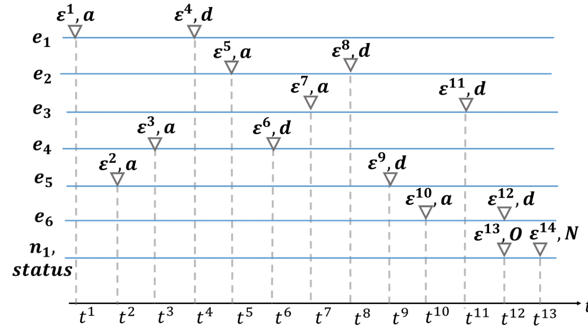
— $V = \{n_1, ..., n_5\}$, $E = \{e_1, ..., e_6\}$, $P = \{\text{Status}\}$;
— $\rho(r) = \begin{cases} \{\text{Machine}\}, & \text{if } r \in \{n_1, n_2\} \\ \{\text{In}\}, & \text{if } r \in \{e_1, ..., e_6\} \end{cases}$;
— $\alpha = \{e_1 \rightarrow (n_3, n_2), e_2 \rightarrow (n_3, n_2)\}$;
— $\Upsilon = \{\epsilon^1, ..., \epsilon^{14}\}$;
— $f^G = \{\epsilon^1 \rightarrow e_1, \epsilon^4 \rightarrow e_1, \epsilon^{13} \rightarrow (n_1, O, \text{Status})\}$;
— $f^T = \{\epsilon^1 \rightarrow t^1, \epsilon^4 \rightarrow t^4, \epsilon^{13} \rightarrow t^{12}\}$;

— $f^E = \{\epsilon^1 \rightarrow a, \epsilon^4 \rightarrow d, \epsilon^{13} \rightarrow u\}$;

As presented in Figure 5.1(a), the validity intervals of nodes indicate the time during which the corresponding machine or product was recorded in the database. The validity intervals of the 'In' relationships indicate the time during which a product was in a machine. The property 'Status' indicates the value and the time interval of the status of a machine.



(a) Temporal property graph



(b) OPGM representation

Figure 5.1 – Example illustrating a temporal property graph and its representation based on the OPGM model. We used (a), (d), (u), (N), (O) to denote addition, deletion, update of a dynamic property, normal and out-of-service, respectively

5.2 δ-Copy+Log

The δ-*Copy+Log* is a variant of the Copy+Log storage approach, which we propose to mitigate the space cost induced by storing full snapshots. Recall that the Copy+Log consists of storing valid snapshots between the boundaries of a time window such

that each time window contains a fixed number of graph operations. The δ-*Copy+Log* follows a similar mechanism with the main difference but stores deltas instead of snapshots. A critical point is that a delta differs from a time window. A time window contains every graph operation between two snapshots, whereas a delta contains only the minimum number of graph operations that transform a snapshot into another one. Indeed, adding an element is canceled by deleting the same element. Hence, both operations are stored in a time window but omitted from the delta.
We store a snapshot after several time windows to serve as a starting point for query evaluation. Having this, we store graph operations in consecutive time buckets containing a number M of time windows such that the first $M - 1$ time windows end with a delta, whereas the final time window ends with a snapshot. A critical optimization is the forward and backward data storage and retrieval. Half of the deltas and time windows in a bucket are constructed in a forward fashion, whereas the other half is built in a backward fashion. The rationale behind this choice is the acceleration of the query execution time. That is, we choose the closest snapshot from which to start the retrieval and then compute the result in a forward or backward fashion, whether the time instant of that snapshot is lower or greater than the requested one.

To illustrate, we present the internals of the δ-*Copy+Log* technique in Figure 5.2. This example is used throughout this section to clarify key concepts of our proposal.

All the symbols used in this section can be found in table 5.1. In the following, we describe the critical components of the δ-*Copy+Log* approach.

Time buckets: We keep graph updates in temporally disjoint time buckets, each containing M time windows and their corresponding checkpoints that can be a delta or a snapshot. We present in Figure 5.2 the internals of a time bucket parameterized with $M = 6$. That is, we store a snapshot (S^6) valid at the highest time instant of the last time window of each bucket ($\omega^6_{\Leftarrow}$) and a delta ($\delta^1_{\Rightarrow}, \delta^2_{\Rightarrow}, \delta^4_{\Leftarrow}, \delta^5_{\Leftarrow}$) at the boundary of each of the remaining time windows ($\omega^1_{\Rightarrow}, \omega^2_{\Rightarrow}, \omega^4_{\Leftarrow}, \omega^5_{\Leftarrow}$). Having this, M time windows exist between the snapshots of two consecutive buckets.

The first $M/2$ time windows ($\omega^1_{\Rightarrow}, \omega^2_{\Rightarrow}, \omega^3_{\Rightarrow}$) are constructed forward. The rest of the time windows are constructed backward ($\omega^4_{\Leftarrow}, \omega^5_{\Leftarrow}, \omega^6_{\Leftarrow}$).

To formalize, we consider the sequence of time buckets $B = \{b^i | i \in \mathbb{N}\}$ s.t. a time bucket is defined as the tuple $b^i = \{\Omega^i, \Gamma^i\}$:

— $\Omega^i = \{\omega^j | j \in [iM + 1, (i + 1)M]\}$ is the sequence of time windows that can be a

Symbol	Description
B	Sequence of time buckets
b^i	Time bucket
Ω^i	Sequence of time windows
Γ^i	Sequence of checkpoints
$\omega^i_{\Rightarrow}$, $\omega^i_{\Leftarrow}$	Forward and Backward time window
s^i	Snapshot
$\delta^i_{\Rightarrow}$, $\delta^i_{\Leftarrow}$	Forward and Backward delta
M	Number of time windows in a bucket
N	Number of graph operations in a time window

Table 5.1 – Symbols and their descriptions used in the formalization of the δ-*Copy+Log* approach

forward or backward time window s.t.:

$$\omega^j = \begin{cases} \omega^j_{\Rightarrow} \text{ if} j \in [iM+1, (i+1/2)M] \\ \omega^j_{\Leftarrow} \text{ if} j \in [(i+1/2)M+1, (i+1)M] \end{cases} \tag{5.1}$$

— $\Gamma^i = \{\gamma^j | j \in j \in [iM+1, (i+1)M] - (i+1/2)M\}$ is the sequence of checkpoints that can be a snapshot, forward or backward delta s.t.:

$$\gamma^j = \begin{cases} s^j \text{ if j = (i+1)M} \\ \delta^j_{\Rightarrow} \text{ if } j \in [iM+1, (i+1/2)M-1] \\ \delta^j_{\Leftarrow} \text{ if } j \in [(i+1/2)M+1, (i+1)M-1] \end{cases} \tag{5.2}$$

Time windows: A time window is a physical container of N graph operations. The time windows presented in Figure 5.2 are configured with $N = 3$. A forward time window ($\omega^1_{\Rightarrow}, \omega^2_{\Rightarrow}, \omega^3_{\Rightarrow}$) contains graph operations sorted in ascending chronological order. A backward time window ($\omega^4_{\Leftarrow}, \omega^5_{\Leftarrow}, \omega^6_{\Leftarrow}$) contains graph operations that are first reversed, meaning that an addition operation is stored as a deletion and vice versa, then sorted following a decreasing order of their timestamps.

Snapshots: A snapshot is persisted at the ending time instant of the last time window of a bucket and represents a valid state at that time instant. As presented in Figure 5.2, snapshot (S^6) is stored at the ending time instant of the time window $\omega^6_{\Leftarrow}$. If the bucket

corresponds to a node or relationship label, then a snapshot contains all nodes and relationships at the time of the snapshot. Whereas, if the bucket corresponds to a dynamic property, then a snapshot includes all nodes and relationships that have that property with the last updated value before or at the time instant of a snapshot.

Deltas: A delta between two snapshots S and S' contains the minimum number of graph updates that permit the transformation of S into S'. If an addition operation is followed by a deletion of the same graph entity, these graph operations cancel each other and will not be added to the corresponding delta. For example, the delta $\delta^1_{\Rightarrow}$ in Figure 5.2 corresponds to the difference between snapshots S^0 and S^1 and contains the deletion operation of relationship e_2 denoted as $-e_2$. The two additional graph operations ($-e_5$ and $+e_5$) are not included in the delta because they represent a canceling pair of operations. We formally define a forward delta $\delta^i_{\Rightarrow}$, using constructs from the OPGM model, as follows:

$$\begin{aligned}\delta^i_{\Rightarrow} = \{\ \epsilon^k | \forall \epsilon^k \in \omega^i_{\Rightarrow} (\forall \epsilon^l \in \omega^i_{\Rightarrow} - \{\epsilon^k\} (f^G(\epsilon^k) = f^G(\epsilon^l) \\ \implies f^T(\epsilon^k) > f^T(\epsilon^l)))\ \}\end{aligned} \tag{5.3}$$

Equation (5.3) states that a graph operation is contained in a forward delta if the former is not followed by any other graph operation in the same time window that maps to the same graph entity. A formal definition of a backward delta can be derived from Equation (5.3) by replacing $\delta^i_{\Rightarrow}$ and $\omega^i_{\Rightarrow}$ by $\delta^i_{\Leftarrow}$ and ω_i and $(f^T(\epsilon^k) > f^T(\epsilon^l))$ by $(f^T(\epsilon^k) < f^T(\epsilon^l))$.

Bloom filter Bloom filters are assigned to each delta to mitigate the execution time overhead of queries induced by the storage of deltas instead of snapshots. For each graph operation in a delta, we add the identifier of the corresponding node to the Bloom filter. Having this, queries are accelerated by skipping the retrieval of graph operations related to the requested node if the identifier of the latter is not found in the Bloom filter.

The example in the Figure 5.2 shows how a temporal graph is stored using the δ-*Copy+Log* method. More particularly, this example shows the internals of a time bucket configured with parameters $N = 3$ and $M = 6$. The history of the graph is composed of 18 graph operations distributed on 6 time windows. These graph operations represent

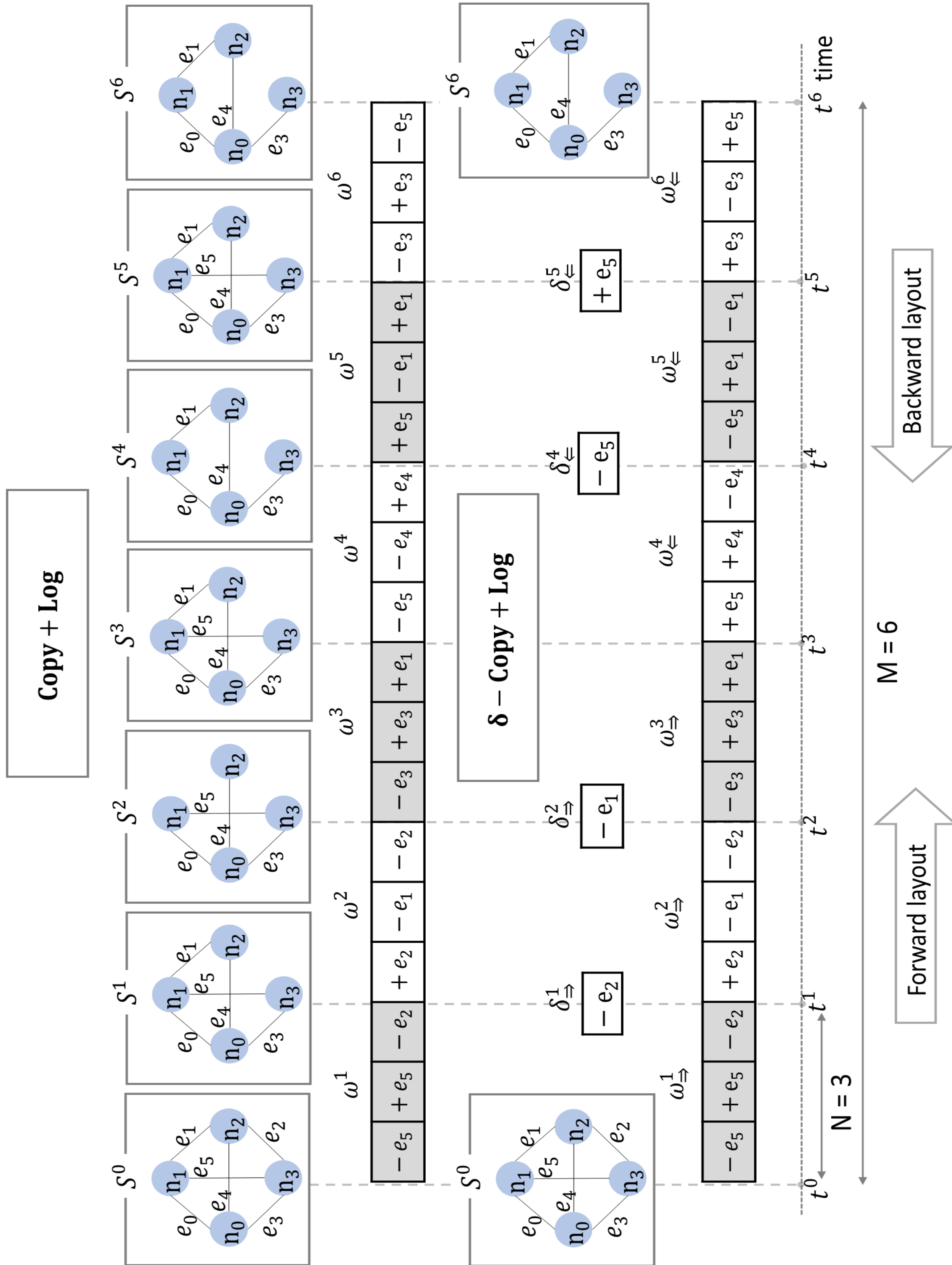


Figure 5.2 – An example of the internals of the δ-Copy+Log showing a time bucket (b^1) with $M = 6$, forward and backwards time windows ($\omega^i_{\Rightarrow}$, $\omega^i_{\Leftarrow}$) , deltas ($\delta^i_{\Rightarrow}$, $\delta^i_{\Leftarrow}$) and snapshot S^i.

either additions or deletions of relationships $\{e_0, \ldots, e_5\}$ connecting the nodes of the graph $\{n_0, \ldots, n_3\}$. We consider that the snapshot S^0 represents the state of the graph at time instant t^0, and no graph operations are recorded into the system with a timestamp lower than t^0. However, the snapshot S^6 corresponds to the state of the graph at time instant t^6. Each forward time window ($\omega^1_{\Rightarrow}$ and $\omega^2_{\Rightarrow}$) contains three graph operations and ends with a forward delta ($\delta^1_{\Rightarrow}$ and $\delta^2_{\Rightarrow}$). The delta $\delta^1_{\Rightarrow}$ corresponds to the difference between snapshots S^0 and S^1 and contains the deletion operation of relationship e_2 denoted as $-e_2$. The two additional graph operations ($-e_5$ and $+e_5$) are not included in the delta because they represent a canceling pair of operations. The backward time windows ($\omega^6_{\Leftarrow}$ and $\omega^5_{\Leftarrow}$) each contains 3 graph operations and ends with a delta ($\delta^6_{\Leftarrow}$ and $\delta^5_{\Leftarrow}$). The delta $\delta^6_{\Leftarrow}$ corresponds to the difference between snapshots S_5 and S_6 and only contains the addition of the relationship e_5. The two additional graph operations ($-e_3$ and $+e_3$) represent a pair of canceling operations. Hence, they can be removed from the delta. Notice that the forward and backward time windows $\omega^3_{\Rightarrow}$ and $\omega^4_{\Leftarrow}$ do not end with a delta since it is not needed for query evaluation.

By referring to this example, we explain the process of computing the neighborhood of a single node at a given time instant based on the forward and backward layouts.

Forward computation The computation of the direct neighbors of the node n_1 at a time instant t falling between time instants t_2 and t_3 implies a forward evaluation since the requested time instant between the boundaries of the time windows in forward layout. First, relationships e_0, e_5, and e_1 are loaded from snapshot S^0. Second, a Bloom filter associated with $\delta^1_{\Rightarrow}$ will be queried to check the existence of any graph operations representing the addition or removal of an outgoing relationship of n_1. The corresponding Bloom filter will return a negative response indicating that no graph operation related to n_1 exists in $\delta^1_{\Rightarrow}$; hence, it can be skipped. Third, a Bloom filter associated with $\delta^2_{\Rightarrow}$ will return a positive response indicating that the delta might contain graph operations related to n_1 and should be fetched. The graph operation ($-e_1$) will be returned from $\delta^2_{\Rightarrow}$ and applied to the neighborhood of n_1 that has been computed so far. This implies that the relationship e_1 will be removed from the computed result. Finally, the time window $\omega^3_{\Rightarrow}$ will be fetched to complete the result. Suppose that the requested time instant is equal to the timestamp of the graph operation ($-e_3$), then no graph operations will be applied to the result since graph operations $(+e_3)$ and $(+e_3)$ are related to the node n_0. Hence, the neighborhood of n_1 at time instant t contains relationships e_0 and

e_5.

Backward computation The computation of the neighborhood of n_1 at a time instant t falling between time instants t_3 and t_4 implies computing the result in a backward fashion since the requested time instant between the boundaries of the time windows in forward layout. First, the outgoing relationships e_0 and e_1 will be loaded from snapshot S_6. Second, the delta $\delta^5_{\Leftarrow}$ will be fetched since the corresponding Bloom filter will return a positive response. The loaded graph operation ($+e_5$) will be applied on the neighborhood of n_0 loaded from snapshot S^6. Hence, the relationship e_5 will be added to the result. Similarly, the delta $\delta^4_{\Leftarrow}$ will be fetched since the corresponding Bloom filter will return a positive response. Then, the graph operation ($-e_5$) will be applied to the returned result, implying the removal of relationship e_5. Finally, the time window $\omega^4_{\Leftarrow}$ will be searched to complete the result. Suppose that the time instant t is equal to the timestamp of the graph operation ($+e_4$) of the time window $\omega^4_{\Leftarrow}$, then only graph operations ($-e_4$) and ($+e_4$) will be returned. However, the returned graph operations are not related to node n_1. Hence, they will be skipped, and the computation will end. Having this, the computed neighborhood of n_0 at time instant t will contain the relationships e_0 and e_1.

5.2.1 Space and time complexities

This section presents the space and time complexities of δ-*Copy+Log*, Log, and Copy+Log methods. We consider the system parameters: N, M, c_1, c_2, r_1 and r_2 and the graph parameters: γ and p_d. Parameters γ, N, and M are previously defined and correspond to the set of all the graph operations, the number of graph operations in a time window, and the number of time windows in a bucket, respectively. Now, parameters c_1 and c_2 are constants corresponding to the space occupied by a single graph operation or graph element. Whereas r_1 and r_2 are constants that correspond to the time taken to read a graph operation or graph element. Parameter p_d corresponds to the probability of deleting a graph element. For simplicity, we assume that all deleted elements are created in the same time window.

The space usage of the δ-*Copy+Log* method is the sum of the space occupied by graph operations (χ_o), deltas (χ_d) and snapshots (χ_s). We compute (χ_o), (χ_d) and (χ_s) separately, as follows.

Space occupied by graph operations is equal to the total number of graph operations ($|\gamma|$) times the space occupied by each graph operation (c_1) as indicated in the following equation:

$$\chi_o = |\gamma| c_1$$

Space occupied by deltas is equal to the total number of deltas ($\frac{(M-2)|\gamma|}{NM}$) times the space occupied by each delta. The space occupied by each delta is equal to the total number of graph operations that are not canceled by a deletion ($N - 2p_d N$) times the space occupied by each graph operation (c_1). Having this, the space occupied by deltas can be computed as follows:

$$\chi_d = (1 - 2p_d)\frac{M-2}{M} c_1 |\gamma|$$

Space occupied by snapshots The total number of graph elements in the ith snapshot is equal to the total number of graph operations that were not canceled by any deletion ($iNM\,(1 - 2p_d)$) whereas the total number of snapshots is equal to $\frac{|\gamma|}{NM}$. Given that, the number of elements in snapshots represent an arithmetic sequence: $\{NM(1 - 2p_d), 2NM(1 - 2p_d), \ldots, \frac{|\gamma|}{NM} NM(1 - 2p_d)\}$. The space usage of snapshots is equal to the sum of the number of elements in all snapshots times the space usage of a graph element c_2, leading to the following equation:

$$\begin{aligned} \chi_s &= \left(\frac{|\gamma|}{NM} + 1\right) \frac{(1 - 2p_d)\,|\gamma|}{2} c_2, \frac{|\gamma|}{NM} \gg 1 \\ &= \frac{(1 - 2p_d)}{2NM} c_2 |\gamma|^2 \end{aligned}$$

Having this, the total space usage of the δ-*Copy+Log* method ($\chi_{\delta-CL}$) can be formulated as follows:

$$\chi_{\delta-CL} = \left(1 + (1 - 2p_d)\,\frac{(M-2)}{M}\right) c_1 |\gamma| + \frac{(1 - 2p_d)}{2NM} c_2 |\gamma|^2$$

The space usage of the *Log* approach (χ_{Log}) is equal to the space occupied by all graph operations (χ_o), implying the following:

$$\chi_{Log} = c_1 |\gamma|$$

The space usage of the *Copy+Log* method (χ_{CL}) is equal to the space occupied by

graph operations and snapshots ($\chi_o + \chi_s$) where $M = 1$. Having this, we derive the following:

$$\chi_{CL} = c_1|\gamma| + \frac{(1-2p_d)}{2N}c_2|\gamma|^2$$

From the obtained equations for χ_{Log}, $\chi_{\delta-CL}$ and χ_{CL}, we can derive the following:

$$\chi_{Log} \leq \chi_{\delta-CL} \leq \chi_{CL}$$

We analyze the time complexity of a unary query evaluation operator for point-based traversal queries. The expand operator ($\uparrow_\tau (v)$) retrieves all the relationships of a node v whose validity intervals contain the time instant τ. Note that the static version of the expand operator was first introduced in the graph algebra proposed by Hölsch et al. in [126]. In this complexity analysis, we use a basic temporal variant of this operator.

Execution time of the expand operator: We consider the worst-case execution time of the operator. Expanding a node at a given time instant implies reading at most from the snapshot whose timestamp is the closest to τ. Then, it induces reading all the operations in the deltas of the selected time bucket whose time interval is before τ, which implies reading $((\frac{M}{2}-1)N)$ graph operations. Finally, it induces reading all the graph operations in the time window that follows the last selected delta. Having this, we obtain the following:

$$T_{\delta-CL}\left(\uparrow_\tau (v)\right) = \left(r_2 + \left(\frac{M}{2}-1\right)Nr_1 + Nr_1\right)$$

Expanding a node using the Log method might incur loading all graph operations in γ. Having this, we derive the following:

$$T_{Log}\left(\uparrow_\tau (v)\right) = |\gamma|r_1$$

Finally, the expansion of a node using the Copy+Log method incurs a single snapshot read which implies the following:

$$T_{Copy+Log}\left(\uparrow_\tau (v)\right) = r_2$$

Consider $|\gamma| \gg (\frac{NM}{2})$ and $|\gamma| \gg \frac{r_2}{r_1}$, then we can derive the following:

$$T_{Copy+Log}(\uparrow_\tau (v)) \leq T_{\delta-CL}(\uparrow_\tau)(v) \leq T_{Log}(\uparrow_\tau)(v)$$

This analysis validates that δ-*Copy+Log* presents a compromise between the *Log* and *Copy+Log* methods.

5.3 Implementation

This section provides the implementation details of the delta Copy+Log technique and its integration into Clock-G. We will refer throughout this section to the components integrated into the architecture of Clock-G as presented in Chapter 3 (Section 3.1, Figure 3.1).

5.3.1 System components

We present in the following the functionality of each of the components: Request handler, Storage manager, Backend connector, Backend store, and Bloom filter briefly described in Chapter 3 and presented in Figure 3.1.

Request handler

The request handler maintains a pool of workers such that each worker is assigned a client request. A client request corresponds to either the insertion of a graph operation or a temporal query. When handling a write request, the request handler worker will first require the metadata information from the storage manager, which will send instructions to store the graph operation. Once received, the request handler sends a message containing the instructions from the storage manager to the backend connector to insert the graph operation. When handling a read request, the request handler will convert it into a set of atomic operators. It will execute each query operator by sending atomic read requests to the backend connector. Finally, the worker sends the obtained result back to the query client. The extended version of the request handler, including a query planner for T-Cypher queries, is provided in Chapter 6.

Storage manager

The storage manager maintains a metadata structure, presented in Figure 5.3, that directs the insertion of a graph operation to the corresponding storage entities. Besides insertion, the storage manager uses this data structure to direct the read operations

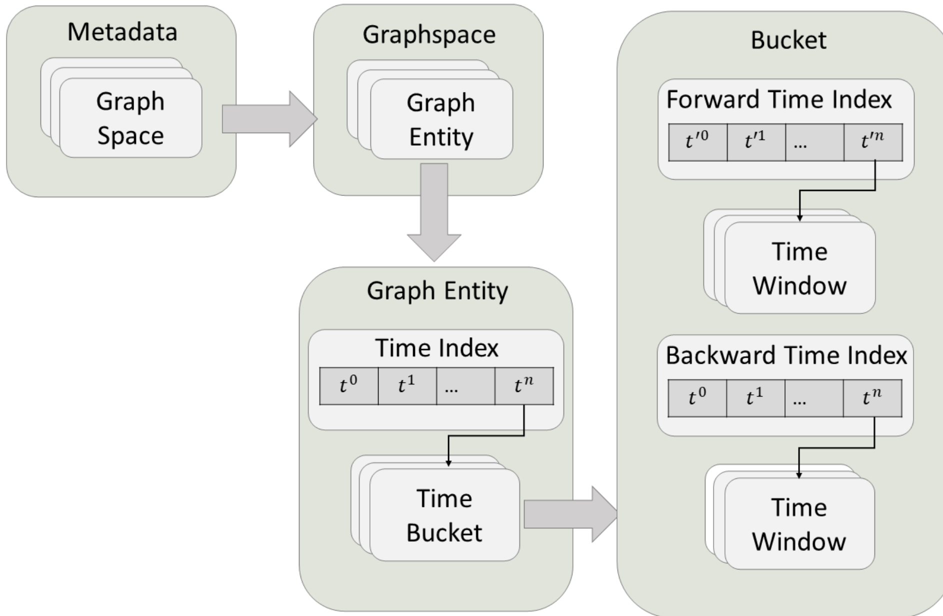


Figure 5.3 – Internals of the metadata managed by the storage manager

to corresponding storage entities. Besides, the storage manager orders the creation of a delta or snapshot when needed. When the number of graph operations in a time window reaches the threshold N, the storage manager sends a creation order to the backend connector, which will create the delta. When the number of created time windows in a bucket reaches the threshold M, the storage manager orders the creation of a full graph operation by sending a message to the backend connector. The creation of the delta and snapshot will be further discussed in Section 5.3.1. The metadata is composed of several graph spaces, each corresponding to a predefined collection of nodes and relationships that belongs to a user application. Each graph space contains the metadata of several graph entities, each corresponding to a given node label, relationship type, or property name. For instance, the graph space corresponding to the toy graph presented in Example 5.1.1 (Figure 2.1) will be associated with entities { `Machine`, `Product`, `In`, `Status` }. The storage manager will maintain the metadata of each of these graph entities. A graph entity holds the metadata related to time buckets, including a time index. The time index is composed of a sorted list of timestamps where each timestamp corresponds to the maximum timestamp recorded in a time bucket. An auxiliary map is used to assign each timestamp in the time index to the metadata of a

time bucket. Now, the metadata of a time bucket contains the time indexes and metadata of forward and backward time windows. The forward and backward time indexes are composed of a sorted array of timestamps corresponding to the maximum timestamp recorded in a time window.

Backend connector

The backend connector is the direct connector with the backend store and is responsible for executing the atomic read and write operations of the request handler. The backend store relies on the column-oriented data store Cassandra [147]. Hence, the backend connector connects and sends atomic requests to a Cassandra instance. Besides, the backend connector is composed of several workers. Each worker is responsible for a number of graph partitions and executes atomic read and write operations on its local graph partitions. The worker is responsible for inserting atomic graph operations in the time window tables and creating snapshots and deltas.

Insertion of a graph operation In order to insert an atomic graph operation, the worker receives a write request from the request handler worker. This write request contains the metadata that directs the graph operation to the corresponding Cassandra table, along with other information about the operation. For instance, a write request for a graph operation corresponding to a node creation contains the identifier of the node, timestamp, type of the event, and the identifier of the time window in which the operation will be inserted, along with a flag that indicates if the operation should be written in forward or backward layouts. If the graph operation corresponds to a node creation or deletion, then the label of the node and its identifier and the type of the event will be sent. Note that the static properties of the node should also be sent in the write request if the operation corresponds to an insertion. The write request of a graph operation corresponding to a relationship contains the identifier of the source and target nodes, the timestamp, and the type of the event. Note that the graph operation corresponds to addition and also includes the static properties of the relationship. The write request of a graph operation corresponding to the update of a dynamic property contains the identifier of the corresponding node or the identifiers of the source and target nodes, the value of the property, and the timestamp of the graph operation. After receiving the write request and the corresponding metadata, the worker generates

a CQL (Query language of Cassandra) insertion query and executes it against the backend store.

Delta creation The creation of a delta corresponding to a time window implies reading all of its graph operations. A critical point is that these graph operations are stored sequentially on disk based on their chronological order. Hence, when creating a delta, the graph updates are loaded sequentially with the correct chronological order, which avoids a costly sorting operation. This implies that the space and time complexities of creating a delta are $O(N)$. Now in order to create a delta that corresponds to the node store, we create an empty map that associates a node identifier with a graph operation. Then, we iterate through the loaded graph operations and insert the graph operation in the map if the latter does not contain a graph operation for that node identifier. Otherwise, the node identifier and graph operation, previously added to the map, are removed. After iterating over the graph operations, the worker transforms the resulting graph operations contained in the map into several batches of CQL insertion queries. Finally, the worker sends the batch requests to the backend store in order to create the delta.

Snapshot creation The creation of the snapshot of a bucket implies reading the closest snapshot (i.e., the snapshot of the previous time bucket), the forward and backward deltas in the time bucket, and the final time window. In order to create a node snapshot, we create a map that assigns a node identifier to a graph element representing a node and containing its static properties and creation timestamp. Then, we iterate through the graph operations of the forward deltas. If the graph operation is an addition, we add the graph element to the map; otherwise, we remove it. This procedure is followed by iterating through the graph operations of the backward deltas. Each graph operation is first inverted, then applied to the map (e.g., a deletion operation is converted into an addition). Then, we add or remove graph elements from the map following the same principle applied to forward deltas. The final task consists of reading from the last time window in a bucket which is stored in a backward layout. The operations loaded from the final time window will be inverted and then applied to the map. The final map will be converted to several batches of CQL insertion requests. Finally, these requests will be sent to the backend store to create the snapshot.

Besides writing individual graph operations and creating deltas and snapshots, the

backend connector worker also evaluates individual reads. When evaluating a temporal query, the request handler sends read operations to the backend connector.

Backend store

The backend store of Clock-G is the column-oriented data store Apache Cassandra [147]. We designed a graph translation layer in order to transition from the flat data model (tables) offered by Cassandra into a temporal graph model.

Store	Clustering key	Regular columns
node	node Id, Timestamp	Event, Static properties
relationship (Out)	Source Id, Timestamp, Target Id	Event, Static properties
relationship (In)	Target Id, Timestamp, Source Id	Event, Static properties
node property	node Id, Timestamp	Value
relationship property (Out)	Source Id, Target Id, Timestamp	Value
relationship property (In)	Target Id, Source Id, Timestamp	Value

Table 5.2 – Description of the internals of Cassandra tables corresponding to the time windows of node, relationship, and node/relationship dynamic property stores

For instance, we separate the storage based on the graph entity type. That is, we store node operations in the node store, relationship operations in the relationship store, and dynamic property operations in the property store. The rationale behind this choice is that for a large subset of queries, attribute information is not used, and only the topology (i.e., graph structure) is important. Thus, we propose storing properties in separate collections (Cassandra tables). In each store, the storage is separated based on the type of graph element. That is, node and relationship stores separate the storage of different node and relationship labels, respectively. Whereas property store

separates the storage of different property names. Note that the property store contains the history of dynamic properties, whereas static properties are stored in either node or relationship store whether the property corresponds to a node or relationship, respectively. The fact that graph elements are more likely to be queried independently motivated the separation of their separation on storage level. However, if several graph elements are requested in a single query, one can achieve an acceleration due to this separation by asynchronously fetching graph updates from different storage units.

For each node/relationship label or dynamic property name, we partition the storage based on a Hash partitioning strategy. Each of these partitions corresponds to a storage unit and is stored following the δ-*Copy+Log* method (denoted δ-CL in Figure 3.1). Hence, we distribute node or relationship operations across storage units based on the hashed value of the node or source identifier respectively. The dynamic property operations are distributed based on the hashed value of the corresponding node or source identifier if the dynamic property is assigned to a node or relationship property, respectively. Indeed, this partitioning can reduce the cost of checkpoint materialization, where a checkpoint is created after the insertion of N graph operations related to a partition of graph entities rather than the entire graph.

We differentiate between static and dynamic properties to reduce disk space usage so that static properties will be stored along with the corresponding node or relationship, whereas dynamic properties are stored separately. The storage internals of the Cassandra tables that constitute node, relationship, and dynamic property stores are depicted in Table 5.2. In this table, we show for each store and each δ-*Copy+Log* component that can be a time window, delta, or snapshot, the composition of the columns of the corresponding Cassandra table. Note that a Cassandra table has a Partition Key that can be a composition of a set of column names such that for each distinct set of values of these columns, a new partition of that table is created. We omit the Partition Key from Table 5.2 as it.

The node store contains the history of the nodes, which includes their identifier (node Id) and the values of the static properties. The **partition key** of each store is the composition of the partition and component identifiers. Having this, each partition in the node store contains the history of a group of nodes. The **clustering key** of the time window table is a composition of the node identifier and the timestamp of the operations. Following the order of the columns in the clustering key, the rows (graph operations) in a single partition will be sorted on disk first by the node identifier and

then by the timestamp. The **regular columns** include the event of the graph operation (i.e., addition or deletion). The clustering key of the checkpoint table is the node identifier, whereas the regular columns are the timestamp, event, and static properties. The snapshots and deltas are stored within the same table. However, the rows belonging to a snapshot do not set the regular column named (Event) since the rows stored in a snapshot correspond to graph elements rather than graph operations.

As shown in Table (5.2), we store two distinct tables for the outgoing and incoming relationships. The relationship store contains the history of relationships which includes the identifiers of the source and target nodes and a set of static properties. Now, each partition in the relationship table contains the history of a group of relationships. The relationships in the outgoing relationship store are partitioned based on the identifier of the source node. The relationships in the incoming relationship store are partitioned based on the identifier of the target node. The clustering key of the time window table in the outgoing relationship store corresponds to the composition of the source identifier, timestamp, and target identifier. Hence, graph operations will be first sorted by the source identifier. Then, for each source node, the graph operations are sorted based on their timestamps. The same principle holds for the time window of the outgoing relationship store with the main difference of sorting graph operations based on the identifier of the target node. Sorting graph operations on disk simplify the task of reading consecutive graph operations related to a source or target node. Now, the clustering key of the checkpoint table is the composition of source and target identifiers. The regular columns of the time window table contain the event of the graph operations and the static properties of the relationship. However, the regular columns of the checkpoint table contain the timestamp, event, and static properties. We create a separate Cassandra table for the dynamic properties of nodes and incoming and outgoing relationships. The partitions of these dynamic stores contain the history of a group of nodes and the relationships of a group of source or target nodes.

Bloom Cache

In order to mitigate the execution time overhead of graph traversals induced by the storage of deltas instead of snapshots, we assign each relationship delta with a Bloom filter. Now, a relationship delta is a delta that corresponds to a storage unit of a relationship label. Indeed, for each graph operation in a delta, we add the identifier of the source node to the Bloom filter. Having this, we can accelerate graph traversals

by skipping the retrieval of the neighborhood of a node if the identifier of the latter is not found in the Bloom filter. It should be noted that we keep Bloom filters in the main memory such that we fix the threshold of memory usage as a design parameter to limit the creation of Bloom filters. That is, whenever the space occupied by Bloom filters reaches the specified threshold, we follow a FIFO (**F**irst **I**n **F**irst **O**ut) policy to evict the oldest Bloom Filters from the cache.

5.3.2 Querying

In this Section, we provide the implementation details of basic temporal graph queries: Local/Global Point/Range queries described in Chapter 2.2.1. Note that the evaluation of more complex queries (T-Cypher queries) is provided in Chapter 6.

Point-based local queries

A point-based local query retrieves the N-Hop neighborhood of a node given several predicates. These predicates are used to express constraints on the *id*, label, or values of properties of the starting node, length of the traversal, type of the relationships, and the time instant at which all the nodes and relationships must be valid.
For the special use case of a smart factory presented in this thesis, a local query (N-Hop queries, i.e.) retrieves, for example, the surroundings of a malfunctioning machine, including sensors, other machines, and products up to a fixed length, given a set of property predicates and a given time instant. The returned result is in the form of a Bag of records such that each record u is a tuple mapping a field name k to a value v such that $u(k) = v$. The notation $dom(u)$ is used to denote the domain of u. Now, each tuple represents a valid path that maps a depth d', such that $0 \leq d' \leq d$ where d is the depth of the traversal, to the *id* of the node that belongs to the path at d'. To introduce the algorithm of the traversal, we define the following unary operators:

— **Select:** $\sigma_c(r)$ This operator selects the tuples of a bag r for which the propositional formula c holds and returns a new bag with the selected tuples.

— **Distinct:** $\vartheta(r)$ This operator returns a bag containing all distinct tuples of a bag r such that it de-duplicates the tuples sharing the same set of fields and mapped to the same values.

— **Projection:** $\pi_{f_0,f_1,...,f_n}(r)$ This operator keeps a specific set of the names of the fields $(f_0, f_1, ..., f_n)$ of a bag r.

— **Expand:** $\xrightarrow[i,(x,y)]{(r)}$ This operator expands a tuple u, if u is contained in a bag r and maps the field i to the value x such that $u(i) = x$, by adding the field $i + 1$ to $dom(u)$ such that $u(i + 1) = y$ and keeps u unchanged otherwise. Then, the operator adds the tuple u to the returned bag.

— **Shrink** $\xleftarrow[i,(x,y)]{(r)}$ This operator shrinks a tuple u, if u is contained in a bag r and maps fields i and $i + 1$ to values x and y such that $u(i) = x$ and $u(i + 1) = y$, by removing the field $i + 1$ from dom(u) and keeping u unchanged otherwise. Then, the operator adds the tuple u to the returned bag.

Algorithm 1 depicts how the N-Hop traversal is computed. The Algorithm takes an *id* of the source node s, a number of hops d and a time instant t and returns a traversal T that contains all paths of depth d starting from s and existing at t. First, $InitBag$ initializes T as a bag with a single tuple u mapping hop 0 to the *id* of the source node such that $u(0) = s$. Then, $GetCheckpointIDs$ returns an array of the identifiers of the δ-*Copy+Log* components fetched in each hop of the traversal. Indeed, those identifiers are sorted incrementally or decreasingly whether the traversal has to be computed in a forward or backward fashion, respectively.

The result is then computed in a BFS (**B**readth **F**irst **S**earch) fashion. Indeed, for each hop n, we get the set $nextnodes$ using the projection and distinct operators that extracts the distinct nodes found at depth n from the traversal T. Then, the nodes of depth $(n + 1)$ are computed by finding the neighbors of each node found in $nextnodes$. That is, the algorithm finds the neighbors by visiting every component whose *id* is contained in ids.

In case the visited component corresponds to a snapshot, the algorithm expands the traversal T at depth $(n+1)$ with each node returned from the $GetNeighbors$ function using the expand operator.

Now, in case the component corresponds to a delta, then before fetching the neighborhood of a node, the function $CheckBloomFilter$ checks for the existence of that node in that delta by testing the corresponding Bloom filter. Now, in case of a positive response, the $GetOperations$ returns all relationship operations of a source node.

In case the component corresponds to a forward or backward time window, then the function $GetOperationsT$ returns the relationship operations belonging to a source node and having a timestamp that is either lower than t, or greater than or equal to t, respectively.

Next, for every retrieved relationship operation ϵ, the algorithm gets the event and

the *id* of the target node (lines 18-20). Note that functions f^E, f^G and α are defined in section 5.1 and $Target(x)$ returns the *id* of the target node of a relationship x. In case of an addition (a), T is expanded at the depth $(n+1)$ with the target node (Line 22). Otherwise, that target node is removed from T using the shrink operator (Line 24). Finally, the select operator is used (Line 25) to filter every tuple u from T such that the condition $d \in dom(u)$ does not hold. Indeed, this condition states that all returned tuples should represent a path of length d.

Algorithm 1: N-Hop traversal

```
   Input: id of the source node s; Number of hops d; Time instant t
   Output: Traversal T
1  T ← InitBag(0, s) ;                                   ▷Initialize traversal
2  ids ← GetCheckpointIDs(t) ;                           ▷Find checkpoints
3  for n ← 0, 1, 2, ..., (d − 1) do
4      nextnodes ← ϑ(π_n(T));
5      for id ∈ ids do
6          for i ∈ nextnodes do
7              if id corresponds to a snapshot then
8                  neighbors ← GetNeighbors(i, id) ;
9                  for j ∈ neighbors do
10                     T ← --(T)--> n:(i,j)) ;           ▷Expand traversal
11             else if id corresponds to a delta then
12                 if CheckBloomFilters(i, id) then
13                     ops ← GetOperations(i, id);
14             else
15                 ops ← GetOperationsT(i, id, t);
16             for ε ∈ ops do
17                 e ← f^E(ε) ;                          ▷Get event from operation
18                 j ← Target(α(f^G(ε))) ;               ▷Get target node from operation
19                 if e == a then
20                     T ← --(T)--> n:(i,j)) ;           ▷Expand traversal
21                 else
22                     T ← <--(T)-- n:(i,j) ;            ▷Shrink traversal
23 T ← σ_{d∈dom(u)}(T) ;                                 ▷Select d hops paths in Traversal
```

Range-based local queries

Range local queries apply a time interval to the computed path such that the time intervals of the returned nodes and relationships should overlap with the requested time interval. We implemented a special type of range-based local queries that we refer to as **sequential paths**. As described in Chapter 4 (Section 4.4.3), every outgoing relationship of a node should have occurred after the incoming relationships to the same node. For the use case of Thing'in, this type of temporal path finds applicability in logistic chains. For example, a product starting from a given station can reach another station if there exists a sequential path representing product transfer between the stations.

Global queries

Point-based global queries retrieve the state of a sub-graph given the labels of the nodes, types of relationships, and a time instant at which all the returned nodes and relationships must be valid. Similarly, range-based global queries retrieve a sub-graph that was valid during a time range, meaning that the validity interval of the returned nodes and relationships should intersect with the requested time interval. Note that one can choose to return a full snapshot of the graph containing all the nodes and relationships without any constraints on the nodes and relationships. In the use case of Thing'in, a global query can recover the status of all the machines and their connections with other devices at a given time instant or follow their evolution during a time interval.

5.4 Experimental evaluation

In this section, we present the evaluation of the overall performance of Clock-G. Our main goal is to validate that the δ-*Copy+Log* produces a compromise between the performances of traditional methods *Copy+Log* and *Log*. Another goal is to show that one can tune the performance of Clock-G by choosing the adequate configuration of the system parameters. Based on this evaluation, one should be able to configure the parameters of Clock-G in order to account for the threshold of accepted query latency and available storage resources. Furthermore, we want to compare Clock-G with a non-temporal graph database to verify that our system outperforms an implementation based on a non-temporal graph system.

5.4.1 Experimental setup

Machine configuration The experiments were conducted on a single machine equipped with 32 Intel(R) Xeon(R) E5-2630L v3 1.80GHz CPUs, 264 GB memory, 1 TB SSD, running 64-bit Ubuntu 18.04.4 LTS with 5.0.0-23-generic Linux kernel. We use OpenJDK 11.0.9, Go 1.14.4, DSE 6.8.4, CQL spec 3.4.5 and Neo4j 4.4.

Datasets In order to validate the performance of the proposed methods, we conducted experiments on synthetic and real temporal graphs. Since the space reduction obtained from the δ-*Copy+Log* is strictly related to the elimination of redundant graph elements that exist across snapshots, we generated synthetic datasets by varying the probability of addition p_a. Indeed, higher values of p_a imply that graph elements will have a longer validity duration. Hence they will be more frequently copied across snapshots. That is, we generated three temporal graph datasets referred to as DS_{p_a} by choosing a value of p_a in $\{0.9, 0.75, 0.6\}$.

Although we work in this thesis on a specific use case (Thing'in), Clock-G can also be deployed for other categories of temporal graphs. Given this, we use in these experiments different real-world datasets such as DBLP dataset (DS_{DBLP} [143], Stack overflow dataset (DS_{stack}) and Wiki talk dataset (DS_{wiki}) [234]. We assume that these graphs are growth-only graphs in the sense that once a relationship is added, it will not be deleted. To evaluate the time-increasing paths, we used the CitiBike dataset [1] (DS_{citi}), which includes bike trips between stations in New York City. We transformed 3 months of data into a series of timestamped graph updates.

We present the characteristics of the generated datasets in Table 5.3 where $|V|$ refers to the total number of nodes, $|\gamma|$ refers to the total number of graph operations.

Dataset	$\|V\|$	$\|\gamma\|$	p_a	Time span (Days)	Space usage (GB)
DS_{citi}	1K	2.5M	_	90	0.066
DS_{wiki}	1.1 M	7.8 M	_	2320	0.173
DS_{p_a}	500K	10 M	0.9, 0.75, 0.6	116	0.315
DS_{DBLP}	1.8 M	29.5 M	_	29930	0.831
DS_{stack}	2.6 M	63.4 M	_	2774	1.7

Table 5.3 – Characteristics of the generated graphs

1. https://ride.citibikenyc.com/system-data

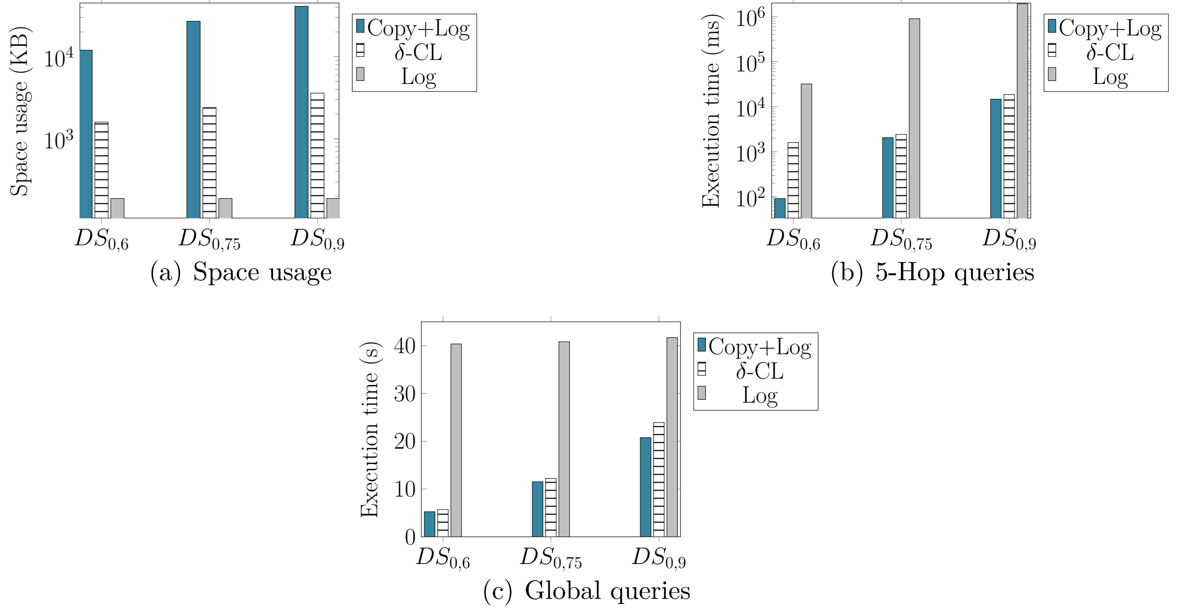


Figure 5.4 – Comparison with state-of-the-art techniques

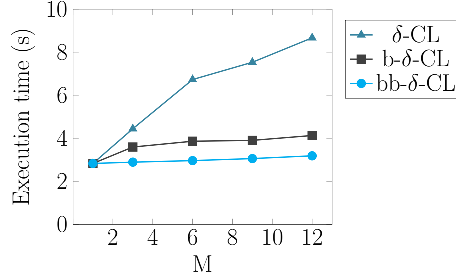


Figure 5.5 – Evaluation of 8 Hop queries with f-δ-CL, b-δ-CL and δ-CL methods on dataset $DS_{0,6}$ with N set to $10K$

5.4.2 Space usage and query execution time

We evaluate the disk space usage and query execution time with different configurations by tuning the system parameters. We compute local queries, detailed in Section 5.3.2, by randomly choosing $1k$ nodes to be the starting nodes of the traversals. Global queries, detailed in Section 5.3.2, retrieve a snapshot of the graph that was valid at a requested time instant. It should be highlighted that time instants used in queries are uniformly chosen within the time span of the datasets in order to avoid a biased distribution that favors only time instants that are closer to checkpoints.

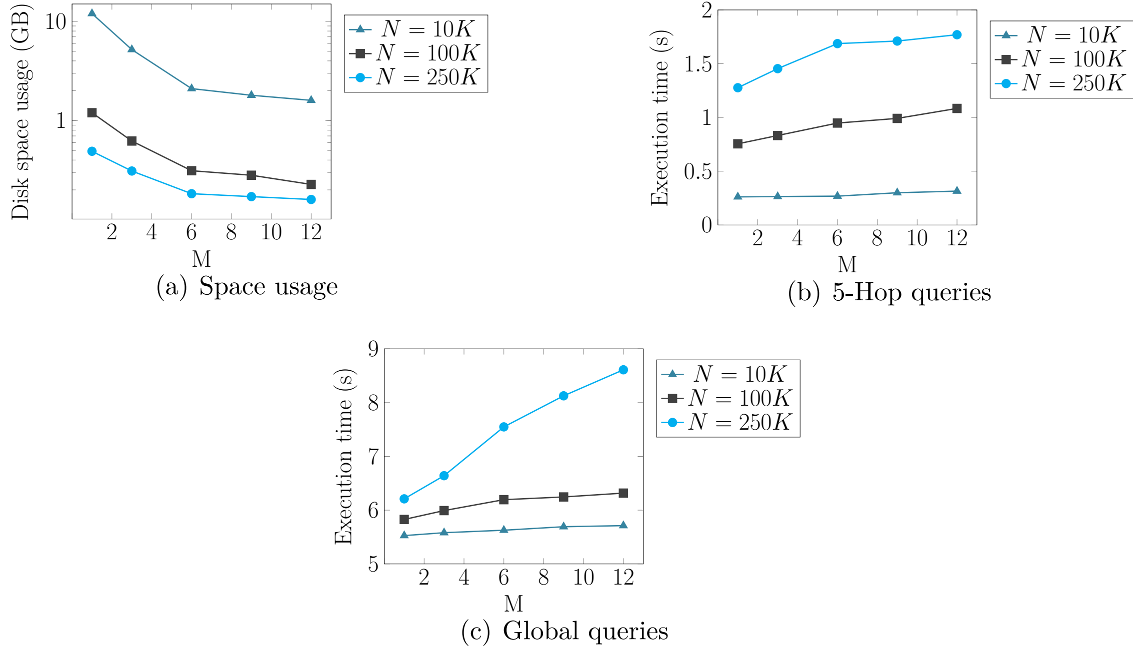
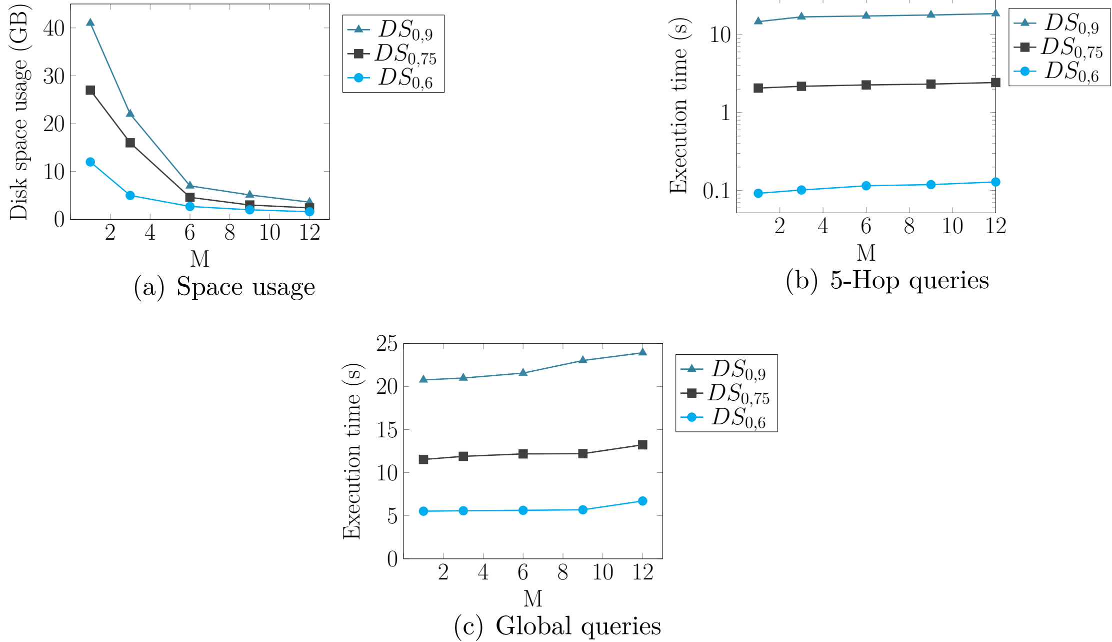


Figure 5.6 – Evaluation of the disk space usage and execution time of queries while varying the system's configuration parameter N. The evaluation is conducted on the synthetic dataset $DS_{0,6}$

Comparison with state-of-the-art methods We compare the results of the proposed method δ-*Copy+Log* with those of the traditional methods *Copy+Log* and *Log*. The implementation of *Copy+Log* in Clock-G is fairly straightforward since it consists of setting parameter M to 1. However, the implementation of the *Log* method consists of creating time windows of size N that are not bounded by any checkpoint. That is, the evaluation of a query with a requested time instant t implies reading from time windows whose time intervals fall before or contains t. Figures 5.4(a), 5.4(b) and 5.4(c) display the space usage, the execution time of 5-Hops and global queries on datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$. Note that we set the system parameters N and M to $10k$ and 12, respectively.

The results are directly in line with previous findings on the apparent trade-off between the space usage and execution time of the *Copy+Log* and *Log* methods. The *Copy+Log* results in a space usage that is 144 times higher than that resulting from the *Log* approach, whereas it results in a query execution time of 5-Hop queries that is 433 times faster than that resulting from the *Log* approach. It is clear from the results that the proposed δ-*Copy+Log* method offers a compromise between the *Log*



(a) Space usage

(b) 5-Hop queries

(c) Global queries

Figure 5.7 – Evaluation of the disk space usage and execution time of queries with $N = 10K$. The evaluation is conducted on synthetic datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$ having each a different value of parameter p_a

and *Copy+Log* as it reduces the storage obtained by the *Copy+Log* by a factor of 12 whereas it reduces the query execution time offered by the *Log* approach by a factor of 340.

Validating the use of Bloom filters As previously discussed, storing deltas instead of snapshots induces a query execution time overhead. Hence, we developed optimization techniques to reduce the induced query latency. We evaluated the execution time of queries with 3 methods, namely: f-δ-CL, b-δ-CL, and δ-CL. The f-δ-CL method, standing for forward-δ-*Copy+Log*, follows the same approach as the δ-*Copy+Log* with the difference of storing only forward time windows and deltas and omitting the use of Bloom filters. The b-δ-CL, standing for bloomed-δ-*Copy+Log*, consists of adding Bloom filters to the f-δ-CL. Finally, the δ-CL refers to the δ-*Copy+Log* method. Hence, it consists of adding forward and backward time windows and deltas to the b-δ-CL. Indeed, comparing the aforementioned methods emphasizes the gain of adding Bloom filters and of storing backward time windows and deltas separately. Figure 5.5 shows the average execution time of traversal queries with a fixed depth equals to 8 on the dataset

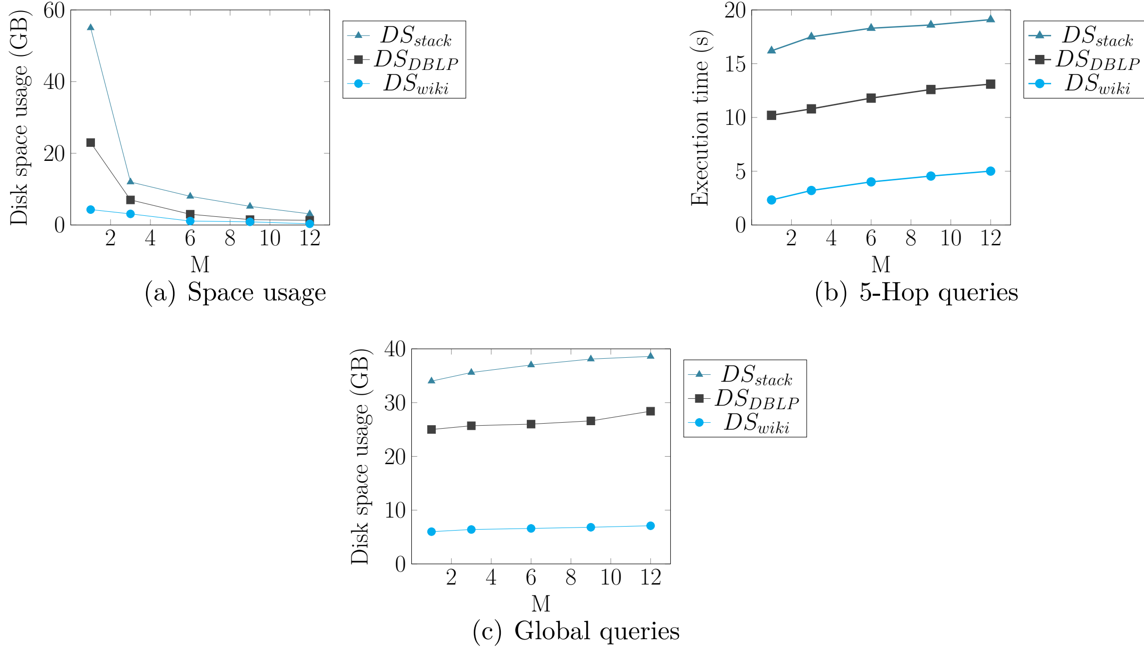


Figure 5.8 – Evaluation of the disk space usage and execution time of queries with $N = 250K$. The evaluation is conducted on real datasets DS_{stackO}, DS_{DBLP} and DS_{wiki}

$DS_{0,6}$ while increasing the system parameter M from 1 to 12. Note that the system parameter N is set to $10k$. It is clear that using the f-δ-CL significantly increases the execution time with the increase of M. Now, adding Bloom filters to the b-δ-CL method reduces the execution time as compared to the f-δ-CL s.t. the speedup can reach 52% for $M = 12$. Furthermore, adding forward and backward time windows and deltas to the δ-CL speeds up the traversals by a factor of 23% as compared to the b-δ-CL. The execution time overhead of the f-δ-CL is equal to 206% when the value of M is increased from 1 to 12. Indeed, this overhead is reduced to $12,5\%$ when using the δ-CL. The rest of the evaluations depicted in this section are executed using the δ-CL method s.t. the f-δ-CL and b-δ-CL methods are used as proof-of-concept implementations and will not be further discussed.

Variation of N and M We evaluate the disk space usage in the function of system parameters N and M. Figure 5.6(a) shows the disk space usage of checkpoints for different configurations s.t. each configuration is set with a value of N in $\{10k, 100k, 250k\}$ and a value of M in $\{1, \ldots, 12\}$. We ingest the dataset $DS_{0,6}$ in Clock-G with every combination of the values of parameters N and M. It is notable that the smaller the

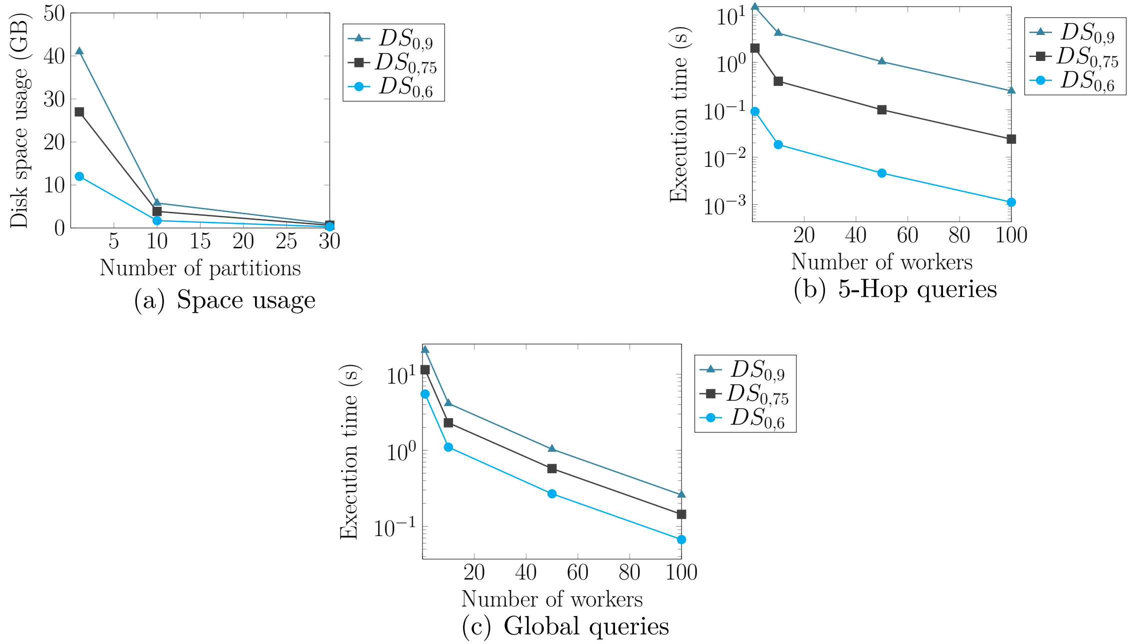


Figure 5.9 – Evaluation of the effect of varying the number of partitions and backend connector workers on the disk space usage and execution time of queries with $N = 10K$. The evaluation is conducted on synthetic datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$

value of N, the higher the space usage of checkpoints. Now, increasing M while fixing the value of N significantly reduces the space usage as compared to the *Copy+Log* method (corresponding to $M = 1$). Besides, the disk space gain obtained by increasing the value of M is more significant for smaller values of N, which is intuitively justified by the fact that lower values of N cause the creation of more time windows. Consequently, a larger number of snapshots will be substituted by deltas which leads to a more significant overall space reduction as compared to that obtained with higher values of N. We also evaluate the variation of N and M on the execution time of 5-Hop and global queries with the same system configuration whose results are given in Figures 5.6(b) and 5.6(c). For these results, it is notable that the higher the value of N, the higher the execution time. That is, fewer checkpoints are created in configurations tuned with larger time windows (higher values of N) which increases, in general, the duration between requested time instants and the time instant of the closest snapshots that are used as starting points for query evaluation.

Variation of p_a and M We study the effect of varying the system parameter M and the dataset characteristic p_a on the disk space. Figure 5.7(a) displays the space usage occupied by the checkpoints of datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$ with different system configurations corresponding each to a value of M in $\{1, \ldots 12\}$. The obtained results demonstrate that space usage strictly decreases with the increase of the value of M. Besides, superior space gains are obtained for graphs with a higher probability of additions. This is due to the fact that snapshots of such graphs are more space consuming which implies that replacing them with deltas emphasizes more significantly the space gain.
We also evaluate the effect of varying p_a and the system parameter M on the execution time of 5-Hop queries and Global queries. Figures 5.7(b) and 5.7(c) show that the execution time of queries increases with the increase of parameter p_a. That is, the average degree of a node increases with the value of p_a, which results in a higher number of computations to evaluate the result of a query.

Evaluation on real datasets We evaluate the space gain obtained from ingesting real-world datasets following the δ-*Copy+Log* method. Figure 5.8(a) displays the disk space usage of checkpoints created by the ingestion of datasets DS_{stack}, DS_{DBLP} and DS_{wiki} into Clock-G while increasing the value of the system parameter M from 1 to 12. It can be noticed that the δ-*Copy+Log* approach markedly reduces the space usage occupied by the dataset when increasing the value of M from 1 to 12. We also evaluate 5-Hop traversal and global queries on these real-world datasets. The obtained results, given in Figures 5.8(b) and 5.8(c), validate that our solution gives clearly good results as compared to the *Copy+Log* method such as it significantly reduces the space usage while adding a slight query execution time overhead.

Variation of the number of partitions and backend workers We evaluate the space usage by changing the total number of partitions. As depicted in Section 3.1, we separate each graph element store into a number of partitions. It is clear from Figure 5.9(a) that the space gain decreases with the increase of the total number of partitions. Figure 5.9(b) and 5.9(c) validate that the query execution time is reduced with the increase in the number of backend connector workers. Indeed, this is due to the fact that fetching tasks are executed by a pool of backend workers in parallel.

Comparison with a non-temporal graph database We compare the performance of *Clock-G* with that of a commercial graph database Neo4j. That is, we developed a temporal layer on top of Neo4j to enable the storage and evaluation of temporal graphs. To add the validity intervals to the graph elements, we created for each node and relationship occurrence two properties: *tStart* and *tEnd* to indicate the starting and ending time instants of the temporal validity interval of the occurrence. Furthermore, we added an index on the identifiers of nodes, properties *tStart* and *tEnd* of the nodes and relationships to accelerate the traversal. We refer to the implementation without indexes as **Neo4j** and the one with the use of indexes as **Neo4j**$_i$. We ingested the dataset DS_{citi} in Clock-G, Neo4j, Neo4j$_i$ with a fixed batch size of 500 graph operations per batch. Then, we evaluated a time increasing path query for each node (station) of the graph and for each depth $1 \rightarrow 8$ and time range 1 hour $\rightarrow 8$ hours.

Figures 5.10(a) and 5.10(b) show the ingestion throughput and space usage of Clock-G, Neo4j, Neo4j$_i$. It can be derived from the plots that Clock-G significantly outperforms Neo4j and Neo4j$_i$. This difference in the ingestion throughput is due to the fact that performing deletes incur the update of the property *tEnd*, which induces a read operation to match the graph element before setting the new property value. Another key factor is the parallelism of Clock-G. That is, a number of backend workers are delegated to batch each, in parallel, a partition of the received graph operations. However, inserting graph updates in parallel into Neo4j is not possible since the chronological order of the updates should be guaranteed.

Figures 5.10(c) and 5.10(d) show the execution time of time increasing path queries while varying the depth and time range of the queries. Note that for each depth and time range, we run the query on all the nodes of the graph and plot the average of the obtained results. It can be noticed from the plots that Clock-G outperforms Neo4j and Neo4j$_i$ such that the difference is more significant with the increase of the depth and time range of the query. Indeed, when evaluating a time range query in Clock-G, the search space is trimmed to a number of selected time windows whose time interval intersects with the time range of the query. Furthermore, backend connector workers compute the result of the query in parallel such that each worker computes a sub-result that contains nodes and relationships belonging to its local partitions. This, however, is not possible with Neo4j and Neo4j$_i$ even when indexes are used to accelerate the traversals.

The results obtained from this experiment highlight the need to develop a graph

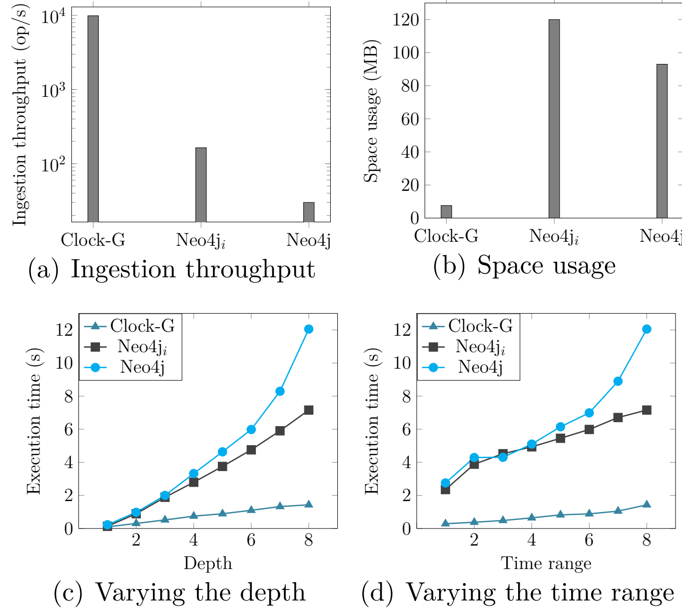


Figure 5.10 – Evaluation of the ingestion throughput and space usage of Clock-G, Neo4j and Neo4j$_i$

management system with native temporal support instead of using an existing non-temporal commercial system.

5.5 Conclusion

In this chapter, we introduced a space-efficient storage technique referred to as δ-*Copy+Log*. It mainly differentiates from the *Copy+Log* by storing deltas instead of snapshots. Besides, we referred to forward and backward data storage and retrieval to accelerate the query evaluation time. We also provided a detailed description of the architecture of Clock-G and the special implementation of Bloom filters that permits the acceleration of traversal queries. Compared to traditional methods, we conducted an evaluation test on synthetic and real-world graphs that validate the efficiency of the δ-*Copy+Log* method. The results demonstrate that our solution significantly reduces the space usage of the *Copy+Log* method and the execution time of the *Log* method, hence it mitigates the space-execution time tradeoff presented by these methods. The queries evaluated in this chapter are basic queries. The evaluation of more complex queries is given in Chapter 6.

Chapter 6

Temporal graph query processing

Chapter 4 introduces our temporal graph query language, T-Cypher. Chapter 5 introduces our temporal graph storage technique and describes its implementation in our system Clock-G. Chapter 5 also presents the evaluation technique of basic temporal graph queries (i.e., Local/Global and Point/Range query presented in Section 2.2.1). However, our primary goal is to integrate more complex temporal graph queries into Clock-G, such as T-Cypher queries. To capture this, we steered our attention toward the processing and optimization of temporal graph queries that we present in this chapter.

Defining a temporal graph algebra for T-Cypher queries is essential to convert a T-Cypher query into a set of algebraic operators. Hence, we define in this chapter a **temporal graph algebra** that extends the operators defined by Hölsch et al. in [126] with the temporal dimension. Then, we define a **cost model** to compute an estimated cost for each operator. This cost model is based on cardinality estimation extracted from histograms storing information about the cardinality of each graph entity. For instance, histograms are adopted in many database systems to estimate the frequencies or cardinalities of selected data values. However, in temporal databases, the frequencies of these data values are likely to evolve over time. Hence, the traditional two-dimensional histograms should be extended with the temporal dimension to capture the cardinality evolution. In this chapter, we use **temporal histograms** to represent the evolution of the cardinalities of graph entities (i.e., node labels, relationship type, or property values). These temporal histograms allow the estimation of the cost of an operator in a requested time interval. To optimize query evaluation, we propose a **greedy algorithm** that selects an evaluation plan for a T-Cypher query. Our query processing approach might lead to different query plans for different time intervals since the plan costs might change over time. To store these temporal histograms, we propose a customized implementation of **segment trees** [29], a data structure commonly used for querying interval-based data. These segment trees hold information about the tem-

poral evolution of histograms used to estimate the cardinalities of T-Cypher queries. To test the performance of our query processor, we evaluated T-Cypher queries using Clock-G. The obtained results demonstrate the effectiveness of our query selection algorithm. Besides, we added time-version support to Neo4j [175] to enable the evaluation of T-Cypher queries and compared the performance of Clock-G with that of Neo4j in evaluating temporal queries. The results show that Clock-G outperforms Neo4j as it resulted in better query evaluation time.

We propose a temporal graph query processor to achieve these goals. In the following, we provide the outline of the chapter:

— We first provide in Section 6.1 a general overview of the query evaluation pipeline implemented in Clock-G.
— We propose in Section 6.2 a temporal graph algebra for T-Cypher queries where we extend the most common graph operators with time.
— In Section 6.3, we present a cost model used to compute an estimated cost of T-Cypher queries.
— We propose in Section 6.4 a greedy algorithm that yields an optimal logical plan for a T-Cypher query.
— We present in Section 6.5 the data structure we implement to store temporal histograms.
— We present in Section 6.6 the results of evaluating our query processor integrated in Clock-G and compare the performance of Clock-G with that of Neo4j.

The work presented in this chapter is ongoing work that we plan to extend shortly. Hence, we offer the first version of our query processor that lays the foundation for further extensions that are part of our ongoing and future work.

6.1 General overview

In this section, we describe the query processing pipeline that we integrate into our system Clock-G. Recall that the version of Clock-G presented in Chapter 5 allows the evaluation of basic temporal graph queries. In this chapter, we extend Clock-G with T-Cypher queries by implementing the pipeline shown in Figure 6.1. The components presented in this pipeline are given in the overall architecture of Clock-G (Figure 3.1).

The **query parser** translates a T-Cypher query into an **A**bstract **S**yntax **T**ree (AST) after verifying its syntactic correctness based on our defined grammar rules(defined

in Chapter 4 (Section 4.5)). Then, the parser traverses the AST to generate a parsed query object recognizable by the **query planner**. After receiving the parsed query, the query planner generates an algebraic plan by applying a greedy algorithm that relies on a cost-based model to limit the cardinalities of subqueries. Estimating these cardinalities is based on **temporal histograms** extracted from the underlying storage engine. These temporal histograms keep track of the evolution of the cardinalities of different graph entities, hence can return an estimated cardinality value that is relevant for a requested time interval or instant. The **query evaluator** executes each atomic query operator by communicating with the storage engine and sending atomic read requests. As presented in Chapter 5, we apply our technique the δ-*Copy+Log* to the storage engine.

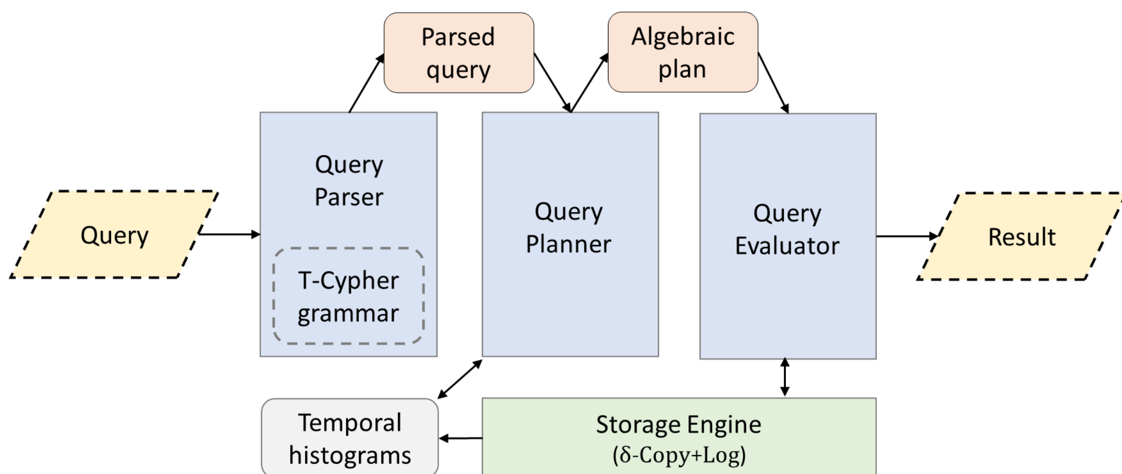


Figure 6.1 – The query processing pipeline implemented in Clock-G

6.2 Temporal graph algebra

In this section, we extend the graph algebra proposed by Hölsch et al. in [126] with the notion of time. Our goal is to have a set of algebraic operators that allows translating a substantial portion of T-Cypher queries into their algebraic representations. This algebra covers a fragment of T-Cypher queries described in later sections. Note that we

leave advanced operators that correspond to aggregations, optional pattern matching, and sub-queries for future work and instead focus on core operators and their temporal extensions.

As described in Chapter 2 (Section 2.1.2), the graph algebra proposed by Hölsch et al. includes the graph operators, **GetNodes** and **Expand**. Besides, they also include basic relational operators, **select** and **join**. The GetNodes operator yields all the database nodes that satisfy a given label. The Expand operator yields the relationships incident to the nodes of the previous result, given the relationship's type and target node's labels. We extend these operators by considering an additional temporal constraint implying that the returned graph entities should be valid during a given time interval or at a given time instant. We limit the description of the non-temporal versions of these operators here, as a more detailed description is given in Section 2.1.2.

A critical concept in our extension is that our temporal graph algebra operators are temporal graph relations, as defined in Section 4.3. A temporal graph relation is a bag of tuples mapping names to values, node or relationship states, set of node or relationship states, or temporal paths. Recall that a graph entity state represents a node or relationship in a time interval during which their property values did not change.

6.2.1 Operators

Let E denote an algebraic expression, $\mu(E)$ denote the set of variables defined in the expression. For example, if E corresponds to matching a relationship between two node variables (a and c) such as $(a-[b]->c)$, then $\mu(E)$ is the set of variables $\{a, b, c\}$.

We illustrate the utilization of our operators to convert a T-Cypher query into an algebraic expression. Specifically, we demonstrate the process using a sample query Q that is applied to toy graph A. The objective of this query is to retrieve the state of a machine and a product that was present in the machine before it underwent maintenance by an employee during a specified time period. We present in Figure 6.2 a possible evaluation plan with the results of the different algebraic expressions ($\{E_0, \ldots, E_5\}$) composing the plan. Note that these expressions are given in the following description of the operators.

```
Query Q

RANGE_SLICE [t1; t8)
```

```
MATCH (m: Machine) <-[r: Maintains]- (e: Employee),
(m) <- [i: IsIn] - (p:Product)
WHERE m.p_0 > 2 AND m@T BEFORE r@T
AND i@T BEFORE r@T AND p@T DURING i@T
RETURN m, r, e, i, p;
```

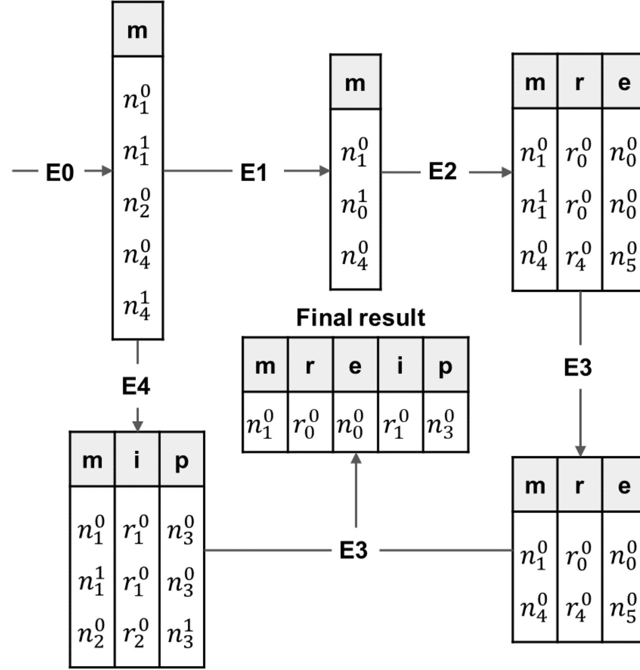


Figure 6.2 – Example showing the result of evaluating the algebraic operators $(E_0, \ldots, E_5)$ on the toy graph in Figure 4.1(a)

GetNodes operator The GetNodes operator returns a temporal graph relation containing node states from the underlying graph G. We use $\bigcirc_{\tau,a,\rho_a}$ to denote this operator, where

- τ: is a time interval such that the returned node states should have time intervals that overlap with it.
- a: is the name of the node variable.
- ρ_a: is the label of the node variable.

Every node state in the underlying graph having a label ρ_a and its time interval overlaps with τ will be returned by this operator. To illustrate this operator, we consider the

following expression returning the machine states valid in $[t_1, t_8)$.

$$E_0 = \bigcirc_{[t_1,t_8),m,Machine}$$

The result of this operator is given in Figure 6.2.

Select operator The select operator, denoted as $\sigma_{\tau,\theta}(E)$, filters input tables based on property values of node or relationship states. It uses a Boolean expression θ defined over validity intervals and property values of variables from $\mu(E)$. This operator filters tuples from input graph relations that satisfy θ during the time interval τ. To illustrate this operator, consider the following expression:

$$E_1 = \sigma_{[t_1,t_8),m.p_0>2}(E_0)$$

This operator filters the input relation resulting from applying E_1 such that the value of the property p_0 of m is lower than 2. The result of this operator is illustrated in Figure 6.2.

Expand operator The expand operator creates a new relation by expanding input relation tuples with direct relationships and target nodes. It is denoted as $\uparrow_{\tau,a,b,ab,\rho_b,\rho_{ab}}$ (E), and ensures that added relationship states are valid within a specified time interval, where

— τ: is a time interval such that the returned relationship states should have time intervals that overlap with it.
— a: is the name of the node in the input relation.
— b: is the name of the added target node.
— ab: is the name of the added relationship.
— ρ_b: is the label of the added target node b.
— ρ_{ab}: is the type of the added relationship ab.

To denote an expansion with an incoming direction, we write $\downarrow_{\tau,a,b,ab,\rho_b,\rho_{ab}}$ (E).

The expand operator can express joins between the input relation and underlying graph when a node in the input relation reaches another node in the graph through a relationship. However, expressing paths through a recursion of join operators leads to a limited relational model. The expansion operator is more general and convenient, as it does not restrict the data model. To illustrate this operator, consider the following

operator:

$$E_2 = \downarrow_{[t_1,t_8),m,e,r,\text{Employee},\text{Maintains}} (E_1)$$

Consider that this operator's input is the previous expression E_1 resulting from the select operator. Notice that the node states (n_1^0 and n_1^1) are each expanded with (r_0^0 and n_0^0) and the node state n_4^0 is expanded with (r_4^0 and n_5^0). Let us filter the returned result to keep the machine states valid before the maintenance, as follows:

$$E_3 = \sigma_{[t_1,t_8),\text{m@T BEFORE r@T}}(E_2)$$

The result of this operator is given in Figure 6.2.

Join operator The Join operator joins two expressions based on a Boolean expression. We use $E \bowtie_\theta E'$ to denote this operator where θ is a Boolean expression. To illustrate this operator, consider joining the previously described expression E_3 with the expression E_4 given below. This expression returns the product states valid when the product was in a machine in $[t_1, t_8)$.

$$E_4 = \sigma_{[t_1,t_8),\text{p@T DURING i@T}}(\downarrow_{[t_1,t_8),m,i,p} (\bigcirc_{[t_1,t_8),m}))$$

The following operator joins E_3 and E_4 with a temporal condition. We refer to a junction with a temporal condition as a temporal join. The result of this operator is illustrated in Figure 6.2.

$$E_5 = E_3 \bowtie_{\text{i@T BEFORE r@T}} E_4$$

It should be mentioned that more complex operators can be defined including the aggregation operator that we keep for later work.

6.3 Cost model

In this section, we define the cost model used by the query planner to estimate the cost of an evaluation plan. This cost model is based on cardinality estimation such that the cost of each operator is equal to the estimated cardinality of its output temporal graph relation. This is reminiscent of classical query optimization. However, our cost model differs because we consider the cost of a query to change over the history

of the underlying graph. In other terms, an optimal plan of a query for a given time interval might not be the same for another since the cardinalities change over time. Our query planner is aware of this evolution and computes the cardinality of each algebraic operator according to the requested time interval. In the following, we use $card(E)$ to denote the estimated cardinality of an algebraic expression E.

The temporal histograms (described in Section 6.5) are used to estimate the cardinalities of algebraic operators for a given requested time. We create a temporal histogram for the **evolution** of each of the following:

— Number of node states with a given label.
— Number of relationship states with a given type and labels for the source and target nodes.
— Number of node states with a given label and a value for a property name.
— Number of relationship states with a given type and a value for a property name.

GetNodes operator The cost of the getNodes operator, given in the equation below, is equal to the estimated cardinality of the node states with a given label ρ_a valid during a given time interval τ.

$$card(\bigcirc_{\tau,a,\rho_a})) = C_{(\tau,\rho_a)}$$

Expand operator The cost of the expand operator, given in the equation below, is equal to the average cardinality of the relationship states given the label of the source and target node states (ρ_a, ρ_b), type of the relationship state (ρ_{ab}), requested time interval (τ) multiplied by the cost of the previous expression E (card(E)).

$$card(\uparrow_{\tau,a,b,ab,\rho_b,\rho_{ab}} (E)) = \frac{C_{(\tau,\rho_a,\rho_b,\rho_{ab})}}{C_{(\tau,\rho_a)}} * card(E)$$

The average cardinality of the relationships of a node is equal to the cardinality of all the existing relationships divided by the cardinality of the nodes given the input parameters $(\tau, \rho_a, \rho_b, \rho_{ab})$.

Select operator The cardinality of the select operator (given in the equation below) applied on an expression E is equal to the selectivity of the graph entity states $sel_\theta(E)$ satisfying the given condition θ multiplied by the cardinality of E. In this section, we define the cost of the selection operator in which we only consider filtering on the

values of the node and relationship properties defined in the input expression. Hence, we only consider boolean expressions of the form $a.p = v$ where $a \in \mu(E)$, p is a property name and v is a value to which the property p is compared.

$$card(\sigma_{\tau,a.p=v}(E)) = sel(\tau, \rho_a, p, v) * card(E)$$

The selectivity of graph entities is computed as follows:

$$sel(\tau, \rho_a, p, v) = (\frac{C_{(\tau,\rho_a,p,v)}}{C_{(\tau,\rho_a)}})$$

The selectivity of a condition θ is equal to the cardinality of all the graph entities satisfying it $a.p = v$ divided by the cardinality of all graph entities with a label or type ρ_a that existed during the time interval τ.

Join operator The cost of the join operator applied on expressions E, and E' (given in the equation below) is equal to the product of the cardinalities of these expressions.

$$card(E \bowtie E') = card(E) * card(E')$$

6.4 Greedy plan selection algorithm

This section describes an algorithm that greedily generates an evaluation plan for a T-Cypher query (Algorithm 2). The main idea is to iteratively compute the optimal plan such that an optimal decision is chosen at each iteration by selecting the less costly algebraic operator and adding it to the final plan.

The input is a query object Q extracted by the query parser as described in Section 6.1, whereas the output is the algebraic plan p_{final}. The first step is to compute all the GetNodes operators representing the leaves of the logical plan tree and add them to the set of sub-plans P. In each iteration, a candidate set P_{cand} is initialized, which will then contain the possible operators that can be applied to the set of sub-plans P to cover all the nodes defined in the query. Then, the possibility of joining two sub-plans is first checked, and every possible join operator is added to the candidate plans P_{cand}.

The method **joinExists**(p, p') returns the following:

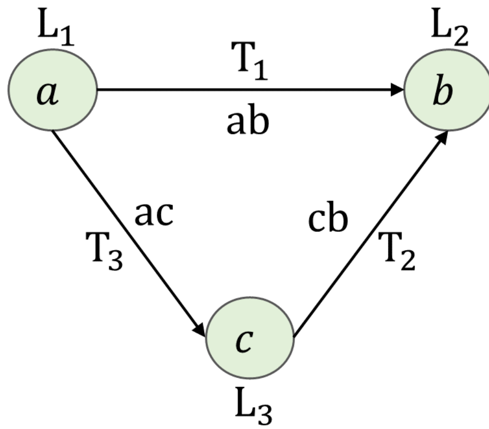
$$joinExists(p, p') = \begin{cases} True, & \text{if } \mu(E_p) \cap \mu(E_{p'}) \neq \{\} \\ False, & \text{Otherwise} \end{cases}$$

Consider $\mu(E_x)$ to denote the set of variables of the expression of the plan x, then the method returns true if the variables of the plan p intersect with the set of variables of the plan p' and false otherwise. After including the possible joins in P_{cand}, each candidate plan is extended with an Expand operator such that the added node variable does not exist in the original plan. Every extended plan will be added to P_{cand}. Now, if no candidate operators are available, the final plan, which encloses all the node variables of the query Q is found, and the iterations stop. Otherwise, the most optimal plan p_{opt} is chosen between the set of candidate plans P_{cand} such that the cost of each plan corresponds to the requested time interval τ. Note that the computations of the costs of each operator are described in Section 6.3. After adding p_{opt} to P, the other plans contained in P and enclosed by p_{opt} are removed from P. The method **enclose** returns the following:

$$p.enclose(p') = \begin{cases} True, & \text{if } \mu(E_p) \supseteq \mu(E_{p'}) \neq \{\} \\ False, & \text{Otherwise} \end{cases}$$

This implies that a plan p is considered to enclose another plan p' if the set of variables of p contains all the variables of p'. Finally, the iterations stop when no candidate subplans are added to the P_{cand} and the final plan p_{final} contained in P is returned.

We show how an optimal plan for the graph pattern presented in Figure 6.3(a) is computed by applying Algorithm 2. In this example, we present three node variables $\{a$, b, $c\}$ labelled with $\{L_1$, L_2, $L_3\}$ and the relationship variables $\{ab, cb, ac\}$ having types $\{T_1, T_2, T_3\}$. We show in Figure 6.3(b) two of the many possible execution plans for this graph pattern. We assume that the cardinalities of the graph entities change over time which conduces to a change of the (greedily) optimal plan. Hence, we consider that the plans presented in Figure 6.3(b) corresponds to time intervals $[t, t')$ and $[t', t'')$, respectively. Figures 6.4 and 6.5 present the selection of operators in each iteration of Algorithm 2, yielding to plans p and p' presented in Figure 6.3(b). Note that we omit some parameters from the notation of operators when they can be derived from the context.



(a) Graph pattern

Plan p valid in $[t, t')$

$\sigma_{a=a'}$

$\downarrow_{b,a',T_1}$

$\uparrow_{c,b,T_2}$

$\uparrow_{a,c,T_3}$

$\bigcirc_{a,L_1}$

Plan p′valid in $[t', t'')$

$\sigma_{b=b'}$

$\uparrow_{a,b',T_1}$

$\bowtie$

$\downarrow_{b,c,T_2}$ $\uparrow_{a,c,T_3}$

$\bigcirc_{b,L_2}$ $\bigcirc_{a,L_1}$

(b) Two equivalent algebraic plans p and p' corresponding to time intervals $[t, t')$ and $[t', t'')$

Figure 6.3 – Example illustrating a graph pattern and two possible logical plans, each corresponding to a time interval

6.5 Temporal Histograms

The cost of an execution plan is based on the cardinalities corresponding to a requested time interval which is related to the fact that the cardinalities of the graph entities change over time. We describe in this section the data structure (segment tree) that we use to store temporal histograms. Then, we provide a compression strategy to reduce the space usage of these histograms.

We illustrate in Figure 6.6(a), the evolution of the cardinalities of three node labels (x, y and z) where c_i $(0 \leq i \leq 4)$ represent the cardinalities of a given label (i.e., the total number of node states with a given label). For example, this figure shows that during time interval $[t_4, t_6)$, c_2, c_2 and c_1 node states with labels x, y and z, respectively were valid.

6.5.1 Segment trees

Extracting cardinalities from temporal histograms implies fetching the total number of graph entity states with given constraints (node label or relationship type) that were valid during the requested time interval (i.e., their time intervals overlap with the requested time interval). A possible way of handling this is to keep all the cardinalities in

Algorithm 2: Greedy selection of a logical plan for a T-Cypher query

```
   Input: Query object Q, τ
   Output: Logical plan p_final
1  P ← InitPlans() N ← ExtractNodes(Q) ;
2  τ ← ExtractTimeInterval(Q) ;
3  for n ← N do
4  |   p ← getNodes(n) ;
5  |   P.insert(p) ;
6  do
7  |   P_cand ← initPlans() ;
8  |   for p ∈ P do
9  |   |   for p' ∈ P do
10 |   |   |   if joinExists(p, p') then
11 |   |   |   |   p'' ← join(p, p') ;
12 |   |   |   |   P_cand.insert(p'') ;
13 |   for p ∈ P do
14 |   |   p' ← expand(p) ;
15 |   |   P_cand.insert(p') ;
16 |   if P_cand.size ≥ 1 then
17 |   |   p_opt ← chooseOptimal(P_cand, τ) ;
18 |   |   P.insert(p_opt) ;
19 |   |   for p ∈ P do
20 |   |   |   if p_opt.enclose(p) then
21 |   |   |   |   P.remove(p) ;
22 while P_cand.size ≥ 1;
23 p_final ← P.get(0)
```

an array such that querying it for a given time interval implies reading all the records until reaching the end time instant of the requested time interval. Despite its compact space usage, an array data structure implies an $O(N)$ time complexity where N is the total number of recorded cardinalities in the array. To mitigate this complexity, we propose the use of segment trees [29].

A segment tree is a data structure that keeps information related to intervals as a full binary tree to allow an efficient response to range queries. For example, querying a segment tree allows finding an aggregated value (e.g., sum, maximum, average) of consecutive array elements in a range. For our query planner, we use segment trees to

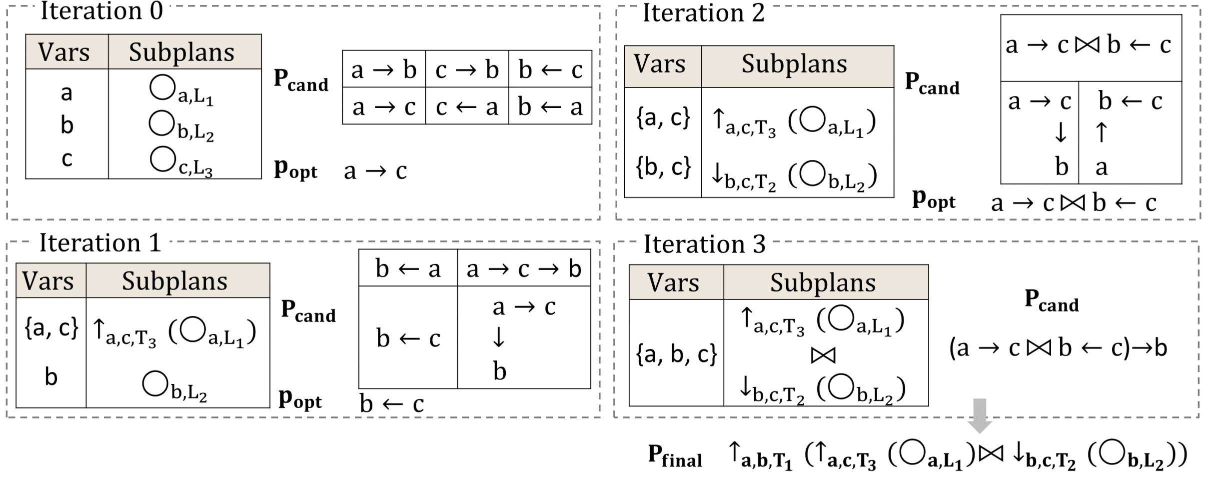


Figure 6.4 – Greedy selection of a logical plan for a T-Cypher query in time interval $[t, t')$

compute the maximum cardinality recorded in a time range to estimate the overall cost of a query plan. We choose the maximum cardinality since it can result in worst-case cost estimation.

We construct a segment tree with the following properties:

— Consider N to be the total number of added cardinalities corresponding each to a sub-time interval during the entire time interval I, then the height H of the tree is $log(N) + 1$. The total number of non-leaf nodes is $N - 1$; hence the total number of nodes is $2N - 1$.

— Each node at position (n, k) corresponds to a time interval $[l_{n,k}, r_{n,k})$ such that $n \in [0, H - 1]$ and $k \in [0, 2^n - 1]$ and contains the cardinality $c_{n,k}$ of a graph entity during the corresponding time interval. The time interval of the root node covers the entire time interval I.

— The intervals of the nodes on the same level $n \in [0, H - 1]$ are non-overlapping such that $r_{n,k} = l_{n,k+1}$ ($k \in [0, 2^n - 2]$).

— The integrity of the segment tree is satisfied in each level since the intervals of the nodes on the same level $n \in [0, H - 1]$ cover the full interval such that $\bigcup_{k=0}^{k=2^n-1} [r_{n,k}, l_{n,k}) = I$.

— Each non-leaf node at position (n, k) has two child nodes such that $[l_{n+1,2k}, r_{n+1,2k})$ and $[l_{n+1,2k+1}, r_{n+1,2k+1})$ are the time intervals of the left and right child nodes at positions $(n + 1, 2k)$ and $(n + 1, 2k + 1)$, respectively. The time interval of a non-leaf node covers the time intervals of its child nodes such that $[l_{n,k}, r_{n,k}) = [l_{n+1,2k}, r_{n+1,2k+1})$. A non-leaf node contains the maximum of the cardinalities of

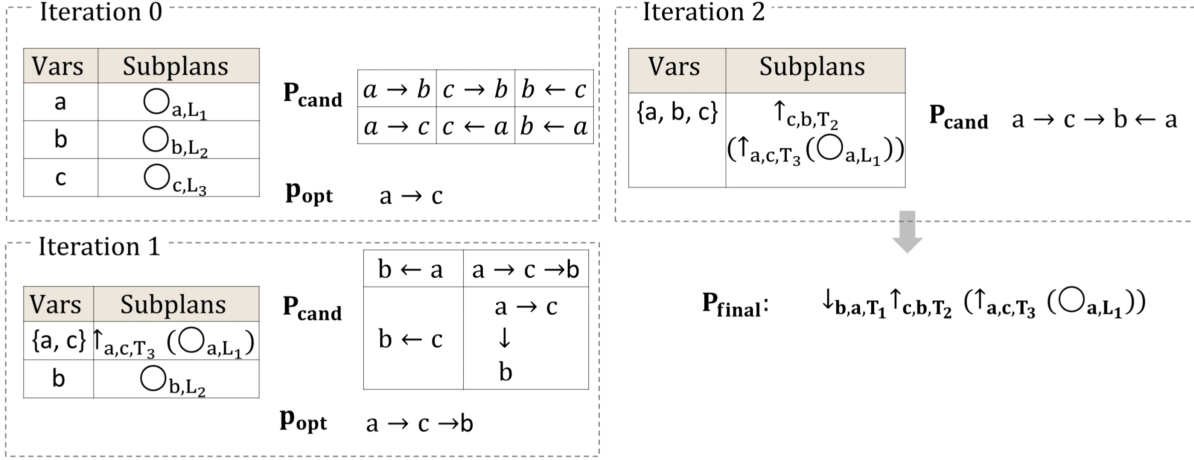


Figure 6.5 – Greedy selection of a logical plan for a T-Cypher query in time interval $[t', t'')$

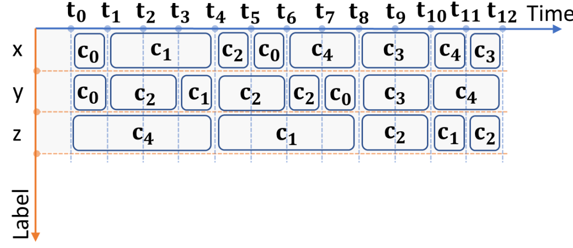
its child nodes such that $c_{n,k} = \text{MAX}(c_{n+1,2k}, c_{n+1,2k+1})$. This implies that the root node contains the maximum cardinality recorded in the entire interval.

— Each leaf node contains the cardinality during its assigned time interval I.

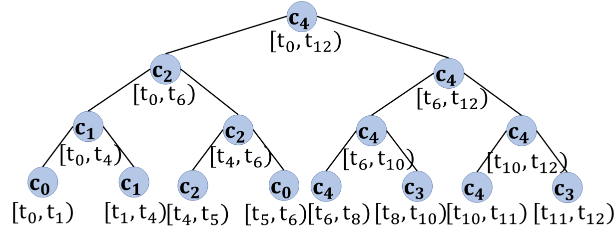
We provide in Figure 6.6(b) an example of a segment tree representing the evolution of the cardinalities of a node label x given in Figure 6.6(a). We assume that the cardinalities satisfy the condition $c_i < c_j$ if $i < j$. For instance, each leaf node represents a time interval during which the cardinality did not change. For example, the leftmost node corresponds to time interval $[t_0, t_1)$ and cardinality value c_0 and the next neighboring node corresponds to the time interval $[t_1, t_4)$ and cardinality value c_1. Now, each non-leaf node contains the maximum of its child nodes. For example, the parent node of the two leftmost leaf nodes corresponds to the maximum value of the cardinalities of its child nodes (c_1) and the union of their time intervals ($[t_0, t_4)$). We can easily verify the above characteristics on the rest of the tree.

6.5.2 Compression of temporal histograms

Preserving all the cardinalities recorded after each change, such as the addition, deletion, or update of a node or relationship state, is space-consuming, especially for highly volatile data. Since query planning does not require very accurate estimates, we can trade the accuracy of cost estimation for efficiency. Hence, we propose a compression technique, neglecting the cardinality fluctuations with a low impact on the cost estimation. This method consists of setting a threshold α such that a new cardinality c_j



(a) Evolution of the cardinalities of labels x, y and z



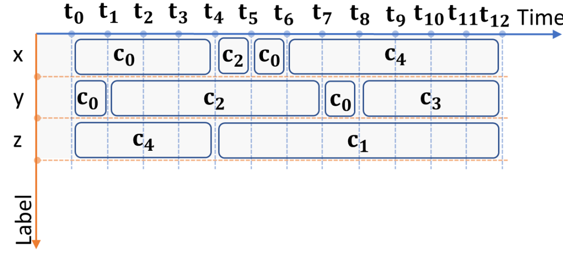
(b) Segment tree corresponding to the temporal histogram of node label x

Figure 6.6 – Example illustrating the evolution of the cardinality of node labels and the segment tree that corresponds to a node label

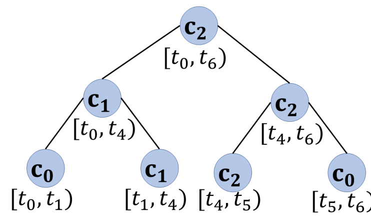
at time instant t_j that follows a cardinality c_i at time instant t_i is only preserved if the absolute normalized difference $\frac{|c_i - c_j|}{c_i}$ between c_i and c_j is higher than α and no cardinality between t_i and t_j is recorded. We present in Figure 6.7, an example of a compressed version of the temporal histogram in Figure 6.6. We consider α as the **compression threshold** and assume the following for $0 \leq i \leq 3$ and $1 \leq j \leq 4$:

$$\frac{|c_i - c_j|}{c_i} = \begin{cases} a \leq \alpha, \text{ if } j = i + 1 \\ b > \alpha, \text{ otherwise} \end{cases}$$

To illustrate, Figure 6.7(a) shows the cardinalities of the tree node labels x, y, and z after applying the compression threshold α. We notice that the small fluctuations are neglected, and the new segment tree presented in Figure 6.7(b) is more compact than the original segment tree presented in Figure 6.6(b), as it reduces the total number of nodes.



(a) Evolution of the cardinalities of labels x, y and z after compression



(b) Compressed Segment tree corresponding to the temporal histogram of node labels x

Figure 6.7 – Example illustrating a compressed version of the temporal histogram and segment tree presented in Figure 6.6

6.6 Evaluation

In this section, we evaluate the performance of our query processor that we integrated into our system Clock-G by implementing the pipeline presented in Section 6.1. We present the execution time of T-Cypher queries when evaluated with a best, random, and worst plan selection. The best execution plan is greedily chosen by applying Algorithm 2. The worst execution plan is chosen following a modified version of this algorithm. Instead of choosing the *cheapest* algebraic operator (i.e., the query operator resulting in the minimum estimated cardinality) using the function *chooseOptimal* (Line 17, Algorithm 2) at each iteration, we choose the most costly one. Similarly, we apply the same algorithm to compute a random plan but randomly choose the algebraic operator at each iteration.

6.6.1 Experimental setup

Machine configuration The experiments were conducted on a single machine equipped with 32 Intel(R) Xeon(R) E5-2630L v3 1.80GHz CPUs, 264 GB memory, 1 TB SSD, running 64-bit Ubuntu 18.04.4 LTS with 5.0.0-23-generic Linux kernel. We use OpenJDK 11.0.15, Go 1.17.7, DSE 6.8.4, CQL spec 3.4.5 and Neo4j 4.4.6.

Datasets We used the LDBC dataset [76] to perform the evaluations. This dataset represents a temporal social graph where people know each other or like each other's posts and comments. We modified the original schema of the LDBC dataset to accommodate the test requirements. The original LDBC schema does not account for dynamic properties. Hence, we transformed some outgoing relationship types and target nodes into dynamic properties attached to the incident nodes. We present our modified version of the schema in Figure 6.8. As shown in this figure, a node in the graph can be either a person, post, or comment, whereas a relationship can have the type likes or knows. The LDBC generator, Datagen, includes a creation date and deletion date on the nodes and edges, if the *raw* output format is used. We refer to these dates as t_{start} and t_{end} in Figure 6.8. Each node and relationship are attached with a starting and ending time instant that defines the boundaries of the validity time interval of each graph entity. Each node has a set of dynamic and static properties that characterizes it. We consider that the dynamic properties might change over time, and their evolution is needed for analyzing the dataset, whereas the static properties are those that do not alter, or the analysis of their history brings no further information to the analysis of the dataset. For instance, the property university of a person node was, in the original LDBC dataset, a relationship connected to that node and to another university node that we have converted into a dynamic property. Similarly, we converted the relationships connecting a person to a company and a post or comment to a tag into dynamic properties.

We used the LDBC datagen tool [1] to generate datasets with a different **scale factor** [2] (SF) as shown in Table 6.1.

1. https://github.com/ldbc/ldbc_snb_datagen_spark
2. The scale factor is the amount of GB of uncompressed data in comma-separated value (CSV) representation.

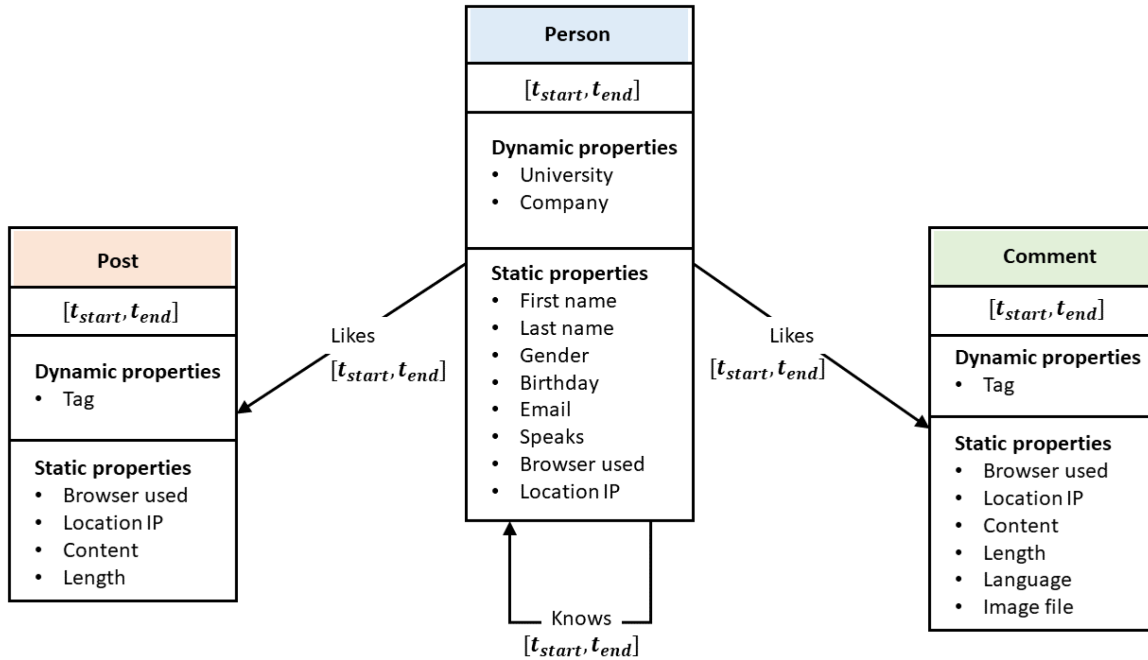


Figure 6.8 – The modified schema of the LDBC dataset used in the evaluation tests

Dataset	# graph updates	# nodes	SF	Space usage (MB)
LDBC$_0$	12K	4.2K	0.003	3
LDBC$_1$	1.9M	406.3K	0.1	100
LDBC$_2$	3.9M	1.1M	0.3	300

Table 6.1 – Characteristics of the generated LDBC graphs

6.6.2 Queries

We ran the test with several T-Cypher queries listed below. Note that all these queries apply to a time interval $[2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]$ which covers the full history of the LDBC datasets.
Query Q_0 returns the pairs of persons who knew each other in the time interval.

Query Q_0

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k1:knows]-> (p2:person)
RETURN p1, p2
```

Query Q_1 returns the person's 2-hop friendship paths.

Query Q_1

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k:knows*2]-> (p2:person)
RETURN p1, k, p2
```

Query Q_2 returns the person's 3-hop friendship paths.

Query Q_2

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k:knows*3]-> (p2:person)
RETURN p1, k, p2
```

Query Q_3 returns the friends of friends of a person who went to the university x such that the friendship started when the person studied in that university.

Query Q_3

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k1:knows]-> (p2:person)
-[k2:knows]-> (p3:person)
WHERE p1.university=x AND p1@T STARTS k1@T AND p1@T STARTS k2@T
RETURN p1, p2, p3
```

Query Q_4 returns all friends of friends of each person such that the intermediate person likes a post.

Query Q_4

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k1:knows]-> (p2:person) -[k2:knows]->
(p3:person), (p2:person) -[l:likes]-> (p:post)
RETURN p1, p2, p3, p
```

Query Q_5 returns the friends who like the same post.

Query Q_5

```
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k:knows]-> (p2:person) -[l1:likes]->
(p:post), (p1:person) -[l2:likes]-> (p:post)
RETURN p1, p2, p
```

6.6.3 Plan selection

We present in this section the best, random, and worst execution plan of Queries Q_3, Q_4, and Q_5. Multiple random queries are possible, which can also be the best or worst execution plan. However, we limit the following description to a single random plan. It is essential to mention that, in the LDBC dataset, the cardinality of the node label *person* is lower than that of the label *post*, and the cardinality of the relationship *like* is higher than that of the type *know*.

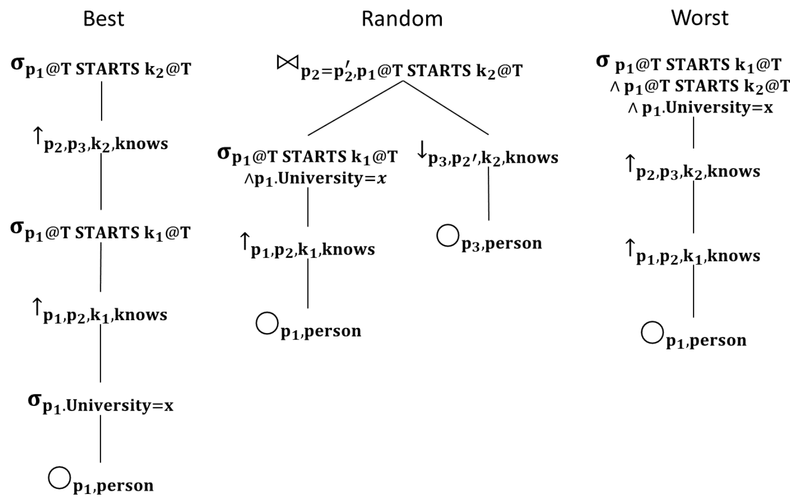


Figure 6.9 – Best, random, and worst evaluation plan of Query Q_3

Finally, we discuss the evaluation plans of Query Q_3 presented in Figure 6.9. As shown in the figure, the best execution plan starts by getting nodes p_1 and then selecting those who went to the given *university*. The next operator consists of expanding nodes p_1 with relationship k_1 then selecting those starting when the *person* node went to the given *university* ($p_1@T$). This is followed by the second expansion with relationship k_2 and the selection to keep those starting by p_1. The worst execution plan is straightforward since the selections are pushed to the last. The random plan presented in the figure consists of computing two sub-parts of the query and then joining the results. The first sub-part computes the nodes p_1 and their direct neighbors p_2 connected through the relationship k_1 and selecting the nodes who studied in the given *university* such that k_1 started by p_1 going to that *university*. The second part computes the nodes p_3 with their direct neighbors p'_2 connected through relationship k_2. Finally, both sub-parts are joined based on the condition that p_2 should be equal to p'_2 and the

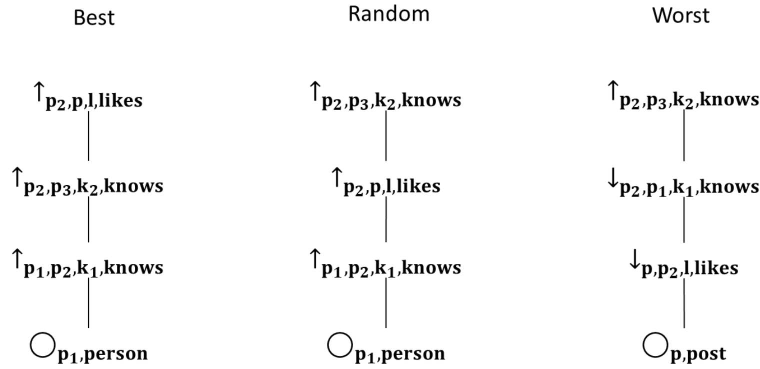


Figure 6.10 – Best, random, and worst evaluation plan of Query Q_4

temporal condition stating that k_2 should be started by p_1.

The evaluation plans of Query Q_4 are presented in Figure 6.10. The best execution plan is to get one of the nodes labeled *person* (e.g., p_1 as presented in the figure), then expand these nodes two compute the 2-Hop friendship path ($p_1 k_1 p_2 k_2 p_3$). The last operator expands the nodes p_2 with the relationship of type *likes* l and finds the target nodes labeled *post* p. Indeed, keeping the likes to the last step helps reduce the cardinalities of p_2 since computing the relationships k_2 before l reduces the cardinality of the nodes p_2 that do not have incoming and outgoing knows relationships. In the random query plan, the expansion of nodes n_2 with relationships l comes before the expansion with relationship k_2. The worst execution plan starts with getting the nodes labeled *post* p, then expanding them with relationship l. This maximizes the cardinality of sub-queries since the nodes labeled *post* and *like* present higher cardinalities than the nodes labeled *person* and *know*.

The evaluation plans of Query Q_5 are given in Figure 6.11. Knowing that the cardinality of nodes labeled *post* is higher than that of the nodes labeled *person*, the query planner chooses to start the best execution plan with one of the *person* nodes (e.g., p_1 as presented in the figure). The second operator expands the returned nodes with the relationship *knows* instead of the relationship *likes* since the former has the lower cardinality. The next operator expands the nodes p_2 with the relationship *likes* to get the nodes labeled *posts* p. Then, the query planner chooses to expand nodes p_1 with the relationship *likes* to get the nodes labeled *posts* p'. The final operator consists of

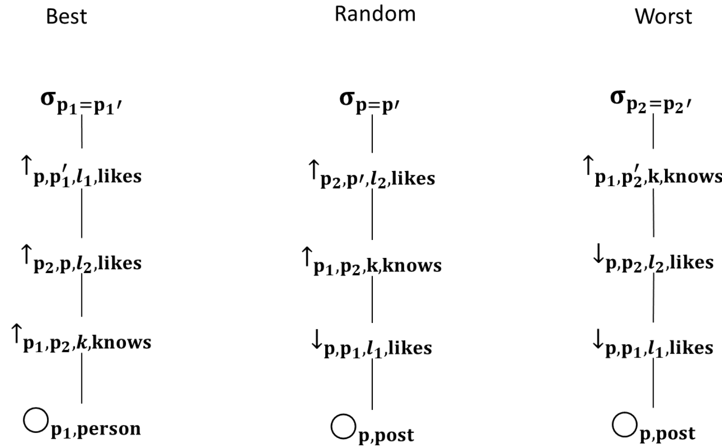


Figure 6.11 – Best, random, and worst evaluation plan of Query Q_5

filtering based on the condition p_1 and p'_1. The worst plan starts with getting the nodes labeled *posts* p. The next operator consists of expanding p to the nodes p_1, then, expanding p to the nodes p_2. Similar to the best execution plan, the final operator filters based on the equality between nodes p_2 and p'_2. The difference between the worst and random execution plans is that the expand operator k is pushed after the expansions l_1 and l_2, increasing the sub-queries cardinality.

We present in Figure 6.12 the result of computing the queries $\{Q_0, \ldots, Q_5\}$ with best, random, and worst plan selection strategies on datasets LDBC$_0$, LDBC$_1$, and LDBC$_2$. Each query is repeated 10 times, and the average execution time is presented in the plots.

The execution time of queries Q_0, Q_1, and Q_2 increases with the number of traversed hops for all the evaluated datasets. However, the execution time of the best, random and worst evaluation plans present a negligible difference. There is no difference between the best, random, and worst evaluation plans for these three queries. We can clearly notice, from these plots, the difference between the execution time following the best and worst plan selection for queries Q_3, Q_4, and Q_5. These results are expected since all the node and relationship variables in these queries share the same label (*person*) and type (*knows*); hence the same cardinality. As previously discussed, the worst plan selection pushes the selection to the end and starts the computation with the nodes with the highest cardinalities, which affects the overall cost of evaluating the queries. The execution time of these queries with a random plan selection presents a

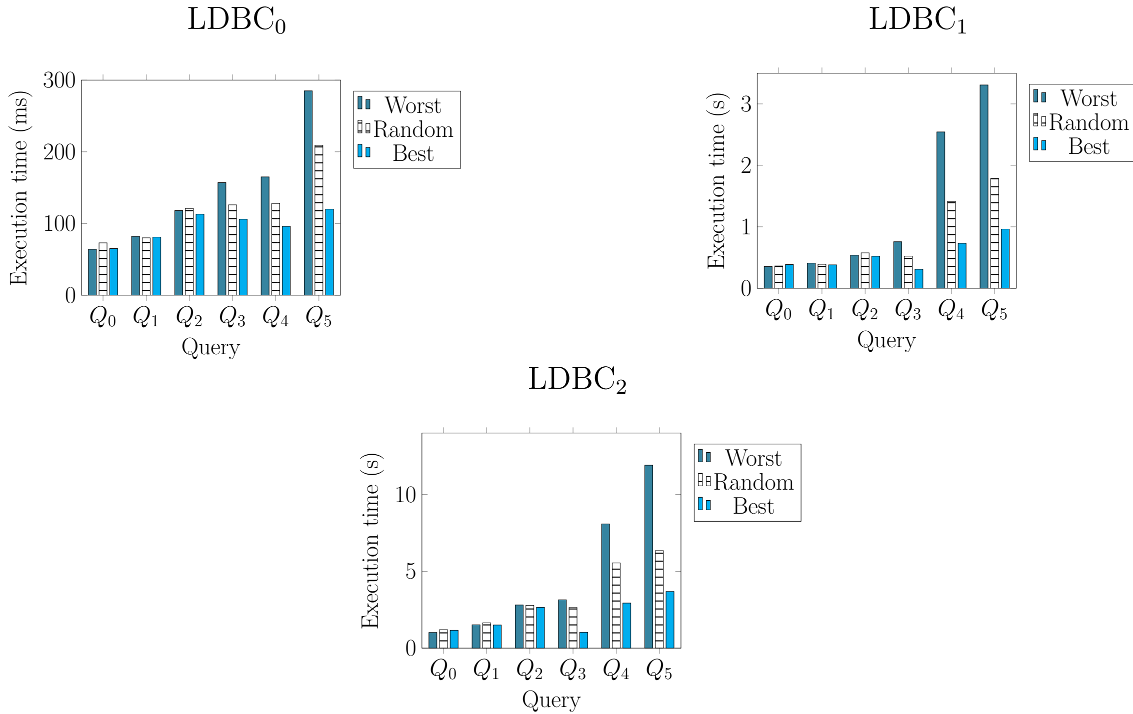


Figure 6.12 – Comparison between the execution time of T-Cypher queries with worst, random, and best execution plans

compromise between the best and worst plan selection. This can be explained by the fact that the best and worst execution plans can be selected along with other possible query plans that present an execution time between the best and worst. Hence, these results prove the efficiency of our cost model and plan selection algorithm.

6.6.4 Comparison with Neo4j

To compare Clock-G with a non-temporal graph system, we implemented a temporal layer on top of Neo4j (See Section 4.6 for a detailed description of this implementation). This layer manages the temporal dimension by persisting the time instants of each graph update and translating T-Cypher queries into Cypher queries that can be evaluated using a backend store that supports the Cypher language, such as Neo4j [175].

We evaluated queries $\{Q_0, \ldots, Q_5\}$ with Neo4j and Clock-G. The obtained results are given in Figure 6.13. Note that we evaluate these queries using the Algorithm 2 to find the best execution plan. As shown in the plots, clock-G outperforms Neo4j such

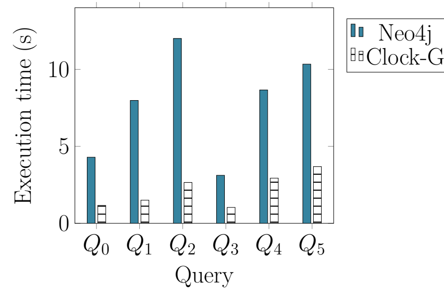


Figure 6.13 – Comparison between the execution time of T-Cypher queries with Neo4j and Clock-G on $LDBC_2$

that it can speed up the evaluation time by 80% for some queries. This difference can be explained by the fact that the query processor can prune the search space according to the requested time interval. That is, Clock-G chunks the history of the graph into multiple time windows such that evaluating a temporal query implies searching directly from the selected time windows and corresponding snapshots and deltas as presented in Chapter 5.

6.7 Conclusion

This chapter describes the temporal graph processor we implemented in Clock-G. We defined temporal graph algebra that extends the algebra presented by Hölsch et al. in [126] with the temporal dimension by introducing algebraic operators that composes the algebraic plans of a T-Cypher query. We also presented a cost model based on cardinality estimation that allows the computation of the cost of a query plan. We defined a greedy algorithm based on this cost model to select a good evaluation plan for a T-Cypher query. Our cost estimator differs from traditional ones because it considers the cardinalities to evolve over time. Hence, we use temporal histograms to store the evolution of the different graph entities. Finally, we presented the performance of our query planner we integrated into Clock-G. The obtained results prove the efficiency of our cost model and plan selection algorithm. We also compared the performance of Clock-G with that of Neo4j. The obtained results demonstrated that Clock-G outperforms Neo4j in query execution time. This motivates our choice to implement a native temporal graph system instead of relying on a non-temporal graph database. In future work, we will extend the current evaluation tests with larger datasets from our graph

generator RTGEN (Chapter 7).

Chapter 7

Temporal graph generation

This chapter presents our proposed temporal graph generator to benchmark temporal graph systems.

Although real-world temporal graphs already exist, these datasets do not often fit the scale requirements or cover all graph updates, such as the addition, deletion, and update of property values. Therefore, practitioners must rely on a temporal graph generator that can produce large-scale graphs whose evolution correlates with real-world temporal graphs.

Current graph generation techniques fill the gap between real and synthetic graphs by mirroring the characteristics of real graphs as presented in this survey [37], such as controlling the degree distribution [9, 49, 150, 140, 72, 19] or community structure [120, 134, 132]. Despite the importance of controlling these two graph features, they are not considered in available temporal generative models [28, 107, 6, 10, 71].

To tackle this challenge, we proposed RTGEN: a relative temporal graph generator that produces temporal property graphs by controlling several key features that characterize the evolution of real-world graphs. That is, our generation procedure controls the evolution of the degree distribution by extending a very common generation technique [56] referred to as the Chung-Lu model (CL) with temporal support. We also extend the CL model to partition the graph into ground-truth communities that coexist with the aforementioned time-dependent degree distribution. We characterize our generator as relative since it generates a temporal graph snapshot valid at time instant t by applying a number of graph updates on its ancestor snapshot valid at time instant $t-1$.

The outline of this chapter is as follows. We first provide an overview description of the generation pipeline. We then describe the baseline generation technique that produces a static graph with a degree distribution approximating a given one. We also describe our proposed method to extend this static generator with a community structure. We then describe a relative graph generation technique that can produce temporal graphs while approximating the given evolution of degree distributions and community

structure. This technique relies on an optimal transport solver that we use to minimize the effort of transforming a graph snapshot into another one. Hence, we also describe how the optimal transport solver is included in our generator. We define metrics to measure the distance between the generated graphs and the desired parameters. Finally, Our contributions are validated through experimental results showing the evolution of the degree distribution and community structure approximating the ground-truth input parameters.

7.1 Overview

This section describes the overall generation procedure (illustrated in Figure 7.1). Given the characteristics of graph snapshots, our generation procedure produces the series of synthetic graph snapshots $\{G_1, \ldots, G_n\}$ whose features approximate the given ones. These graph snapshots are relatively computed by applying graph updates on each snapshot to produce its successor snapshot. To clarify, we apply a number of graph updates on a graph snapshot G_{i-1} to create another graph snapshot G_i whose characteristics approximate the given parameters assigned for the ith graph snapshot.

We define a graph snapshot G_i valid at a time instant t_i as $\{V_{G_i}, E_{G_i}, \phi_{G_i}, M_{G_i}\}$ where V_{G_i} is the set of vertices, E_{G_i} is the set of relationships, ϕ_{G_i} is a degree distribution and M_{G_i} is the density community matrix. For instance, we consider ϕ_{G_i} of the form $\{(x_1^{G_i}, \omega_1^{G_i}), \ldots, (x_n^{G_i}, \omega_n^{G_i})\}$ as a discrete distribution over $\mathbb{N}$ where $x_j^{G_i}$ refers to the degree of a node and $\omega_j^{G_i}$ refers to the total number of vertices in the graph whose total number of relationships is equal to $x_j^{G_i}$. A density community matrix M_{G_i} defines the community structure of the generated graphs, each element m_{uv} of which is equal to the density of relationships between the source and target communities c_u and c_v.

Given the number of vertices in each graph snapshot $k_i \in \{k_1, \ldots, k_n\}$, a stochastic community matrix M and a sequence of degree distributions $\{\phi_1, \ldots, \phi_n\}$, we generate a sequence of graph snapshots $\{G_1, \ldots, G_n\}$ such that each snapshot G_i is relatively generated by applying a number of graph updates to G_{i-1}. This transformation is based on morphing the $\phi_{G_{i-1}} = \{(x_1^{G_{i-1}}, \omega_1^{G_{i-1}}), \ldots, (x_n^{G_{i-1}}, \omega_n^{G_{i-1}})\}$ into $\phi_i = \{(x_1^i, \omega_1^i), \ldots, (x_k^i, \omega_k^i)\}$ and preserving the community structure that is represented by the stochastic community matrix M such that $M_{G_{i-1}} = M_{G_i} = M$. Note that each element m_{uv} of M is equal to the probability of relationship creation between the source and target communities c_u and c_v. Figure 7.1 illustrates the relative graph generation

procedure. The transformation between G_{i-1} and G_i is based on computing a transportation matrix T that minimizes the cost of morphing $\phi_{G_{i-1}}$ into ϕ_i. The computation of the transportation matrix reduces to an optimal transport problem. Based on the computed transportation matrix, each node belonging to the graph G_{i-1} is assigned a linkage or breakage probability to indicate the probability of adding or removing a relationship. This phase is followed by creating or removing relationships to or from the graph G_{i-1} to produce the graph G_i. These graph updates follow the linkage or breakage probabilities assigned for each node. Finally, the graph G_i is computed by applying the generated updates on G_{i-1}. Note that the generation procedure depicted in this figure shows a simplified scenario where the number of vertices does not change. However, if that number changes, a phase consisting of the addition or deletion of vertices should precede the computation of the transportation matrix to ensure the following constraint:

$$\sum_{s=1}^{k} \omega_s^{G_i} = \sum_{t=1}^{m} \omega_t^i, \quad \forall\, 1 \leq i \leq n$$

This constraint implies that the sum of weights of distributions ϕ_{G_i} and ϕ_i should be equal.

7.2 Generation with degree distribution

This section describes the generation procedure of random static graphs with a given degree distribution that sets the base of our temporal and community-aware extensions discussed in further sections.

Random graphs were introduced by Erdős and Rényi (ER) [75]. The ER model generates a number of nodes and connects them by a relationship after picking each endpoint with a fixed probability p. However, this model produces graphs whose degree distribution follows a binomial distribution with a mean degree equal to $(N-1)p$ where N is the total number of nodes. Hence, it fails to mimic real-world graphs, usually following a power-law degree distribution. The Chung-Lu model (CL) generates a random graph that approximately matches a given degree distribution [8], relying on a simple generation procedure that can be considered a variant of the ER model.

Consider the degree distribution ϕ as the input parameter to the CL model and the undirected, unweighted, and unlabeled graph $G = \{V, E, \phi_G\}$ as the output where ϕ_G denotes the degree distribution of G, V, and E represent the set of nodes and rela-

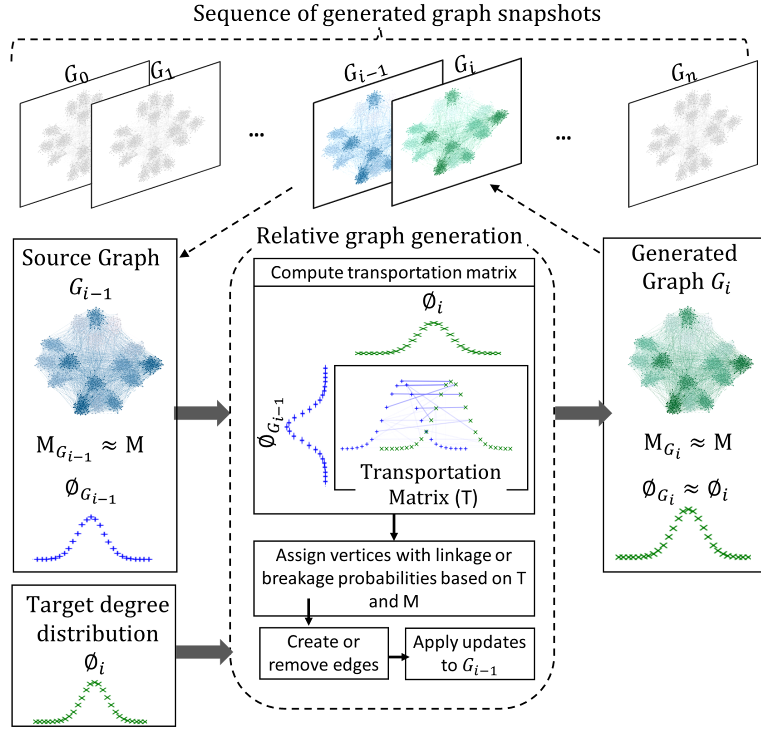


Figure 7.1 – Relative graph generation procedure.

tionships, respectively. The CL model produces a graph G such that ϕ_G approximates ϕ.

The main idea is to pick each relationship endpoint given a linkage probability such that the total number of incident relationships to each node resulting from the generation procedure is close to its assigned degree. Hence, the starting phase consists of assigning each node $v_i \in V$ with a degree d_{v_i} and a linkage probability $p_{v_i} \propto d_{v_i}$. Considering that D is the sum of the degrees extracted from ϕ, we define the CL linkage probability p_{v_i} in the following Equation:

$$p_{v_i} = \frac{d_{v_i}}{D} \tag{7.1}$$

Subsequently, a linkage phase consists of picking $|E| = \frac{D}{2}$ pairs of nodes to connect such that for a sufficiently large $|E|$ the random variable denoting the degree of node v_i is Poisson distributed with a mean equals to d_{v_i}. Iterating the linkage phase $|E|$ times where a relationship is equally likely to be chosen in both directions for undirected graphs, the insertion probability of a relationship connecting node v_i and node

v_j is $p_{v_i vj} = 2p_{v_i}p_{v_j}\frac{D}{2}$. The relationship insertion probability can be rewritten in a more convenient form:

$$p_{v_i v_j} = \frac{d_{v_i} d_{v_j}}{D}$$

To optimize the generation procedure, we gather all nodes sharing the same degree together in a pool $\gamma_d = \{v_i | v_i \in V \wedge d_{v_i} = d\}$. Each node in a pool is equally likely to be chosen, assuring that the aforementioned linkage probability p_{v_i} is not affected for a sufficiently large number of nodes. After the degree assignment phase, nodes are distributed across the pools having each the following linkage probability:

$$p_{\gamma_d} = \frac{d|\gamma_d|}{D}$$

Following this, a pool is first picked then the node is randomly chosen in the selected pool. It should be highlighted that self-loops or multi-edges can be created since each endpoint of a relationship is picked independently. However, the number of these relationships is independent of the number of nodes and thus can be neglected for large-scale graphs.

7.3 Community-aware generation with degree distribution

Although the Chung-Lu (CL) model produces graphs with a given degree distribution, it is unaware of the community structure in most real-world graphs. Hence, we propose a community-aware extension of the CL model based on the stochastic block model (SBM) [120]. In the following, we introduce the concept of a community (Section 7.3.1), describe the SBM model (Section 7.3.2), present our proposed extension that generates graphs with respect to a given degree distribution and community structure (Section 7.3.3), and propose an auto-filling technique to generate the community parameters with no exogenous effort.

7.3.1 Graph community

A graph community is not quantitatively well defined, leading to many definitions posited in the literature. Intuitively, a community can be considered as a subgraph

whose nodes are more densely connected with each other than they are with the rest of the graph.

Given the set of communities $C = \{c_k, 0 \leq k \leq N-1\}$, we consider that a node should belong to a single community and relationships should be differentiated into **within** and **between relationships** as follows:

— Given a community c_i, a relationship e is called a **within relationship** if the source node $\in c_i$ and the target node $\in c_i$.

— Given two non-overlapping communities c_i and c_j, a relationship e is called a **between relationship** if the source node $\in c_i$ and target node $\in c_j$ or vice versa.

Having this, we define $p_{c_i}^{in}$ as the probability of creating a within relationship that belongs to the community c_i and $p_{c_i}^{out}$ as the probability of creating a between relationship whose one endpoint node belongs to c_i whereas the other endpoint belongs to another community $c_j \in C - c_i$. To insure that nodes belonging to a community are more densely connected between each other than they are with the rest of the graph, the within and between relationship creation probabilities $p_{c_i}^{in}$ and $p_{c_i}^{out}$ of each community c_i must satisfy the condition $p_{c_i}^{in} > p_{c_i}^{out}, \quad \forall c_i \in C$.

7.3.2 Stochastic block model

This section describes the SBM model [120] (also known as the planted partition model), which is commonly used for generating random graphs with a given community structure. This generation procedure only considers controlling the community structure of the graph and overlooks the resulting degree distribution.

The input of the generation procedure is a stochastic community matrix M, each element m_{ij} of which defines the probability of relationship creation between the source and target communities c_i and c_j. The output is a graph $G = \{V, E, M_G\}$ where M_G is the obtained density community matrix, each element m_{ij}^G of which defines the relative relationship density between the source and target communities c_i and c_j.

The generation procedure starts with the distribution of nodes among the planted (non-overlapping) communities such that each node belongs to a single community. Now, the linkage probability between a node belonging to the community c_i and another node belonging to the community c_j is equal to m_{ij}. However, the extracted community density matrix M_G from the resulting graph G approximates M. That is, each element m_{ij}^G is binomially distributed with a mean equal to m_{ij} and Poisson distributed with the

same mean for a sufficiently large number of created relationships.

7.3.3 Stochastic block model with degree distribution

This section describes our generative model that extends the CL model with the control of the community structure based on the previously described SBM model. Given a degree distribution ϕ and a stochastic community matrix M, our proposed model generates a graph G in which degree distribution ϕ_G is an approximation of ϕ and density community matrix M_g is an approximation of M.

Since the generated graph G is undirected, the matrix M is symmetric such that $m_{ij} = m_{ji}$. Having this, we consider $\omega_{ij} = \omega_{ji} = 2m_{ij}$ and $\omega_{ii} = m_{ii}$. Furthermore, we assign each community c_i with a within relationship creation probability $p_{c_i}^{in}$, a between relationship creation equal to $p_{c_i}^{out}$ and a probability of relationship creation p_{c_i} such that:

$$p_{c_i} = p_{c_i}^{in} + p_{c_i}^{out} = \omega_{ii} + 0.5 \sum_{j=1, j \neq i}^{|C|} \omega_{ij} \tag{7.2}$$

Given that D_{c_m} is the sum of the degrees of nodes belonging to community c_m and p_{c_m} is the probability of choosing c_m, we define the linkage probability p_{v_i} of choosing a node v_i belonging to c_m as follows:

$$p_{v_i} = \frac{d_{v_i}}{D_{c_m}} p_{c_m}, v_i \in c_m \tag{7.3}$$

The linkage probability of a node is the product of the probability of choosing the community to which the node belongs (p_{c_m}) and the probability of choosing the node v_i in that community $(\frac{d_{v_i}}{D_{c_m}})$. Hence, Equation 7.3 assures the approximation of the community matrix. However, p_{v_i} should be equal to $\frac{d_{v_i}}{D}$ (Equation 7.1) to assure the approximation of the degree distribution. Therefore, we define the following condition in order to reduce Equation (7.3) to Equation (7.1):

$$D_{c_m} = D p_{c_m} \tag{7.4}$$

By replacing D by $(\frac{D_{c_m}}{D p_{c_m}})$ in the original CL linkage probability (Equation 7.1) which assures the control of the degree distribution, we obtain Equation 7.3 which assures the control of the community structure. Having this, the duality of the linkage probability

given in Equations (7.1) and (7.3) ensures that both requirements are satisfied by our generation procedure. Hence, this simple yet powerful assumption allows our generative model to produce graphs with respect to a given degree distribution and community structure.

Towards improving performance, we consider the selection of pools rather than nodes such that a pool is local to one community. That is, nodes having the same desired degree and belonging to the same community c_m are grouped in a pool $\gamma_d^{c_m} = \{v_i | v_i \in c_m \wedge d_{v_i} = d\}$ such that the probability of a pool selection for relationship insertion $p_{\gamma_d^{c_m}}$ is defined as follows:

$$p_{\gamma_d^{c_m}} = \frac{d|\gamma_d^{c_m}|}{D_{c_m}} \tag{7.5}$$

The generation procedure starts with assigning nodes with a degree drawn from the desired degree distribution. Then, the nodes are distributed among communities such that the Equality 7.4 is satisfied. Subsequently, pools are created per community to group the nodes belonging to the same community and sharing the same desired degree. Once all the communities and corresponding pools are filled with nodes, the relationship creation phase starts. To link two nodes, we first choose the community to which each node belongs. The selection of a community is affected by the probability presented in Equation 7.2. Once the communities of the two endpoints are chosen, a pool grouping nodes sharing the same expected degree is selected based on the linkage probability presented in Equation 7.5. Finally, a node is uniformly chosen inside each of the selected pools forming the endpoints of the created relationship.

7.3.4 Hierarchical community structure

Exhaustively filling the stochastic matrix can be cumbersome when creating many communities. To fill the stochastic matrix with no exogenous effort, we propose an auto-generative procedure. Given only two parameters, this procedure auto-fills the matrix to produce a hierarchical community structure in many real-world graphs.

In a hierarchical community matrix, communities recursively embed subsequent communities in a self-similar fashion such that the community structure can be represented by a hierarchical tree, each node of which corresponds to a community. Each non-leaf node (community) is expanded into b other nodes (communities) until reaching

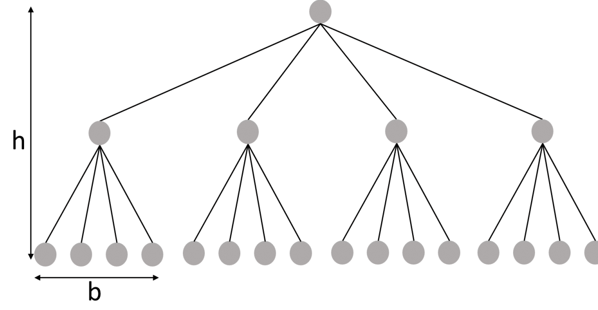


Figure 7.2 – Hierarchical community tree with height $h = 2$ and branching factor $b = 4$.

a desired tree height h (Figure 7.2). The ending recursion results in $n_c = b^h$ leaf-nodes corresponding to the finest scale communities.

The within and between linkage probabilities are defined based on the distance between the communities of the resulting trees. That is, the between linkage probability ω_{ij} of communities c_i and c_j is proportional to the distance between c_i and c_j. The distance between two communities $d(c_i, c_j)$ is equal to the number of branches in the hierarchical tree that should be traversed to reach the least common ancestor (node) of these communities. In order to satisfy the condition stating that within relationship linkage probability must be higher than between linkage probability ($p_{c_i}^{in} > p_{c_i}^{out}$), we define ω_{ii} as follows:

$$\omega_{ii} = 0.5 \sum_{j=0, j \neq i}^{n_c - 1} \omega_{ij} + k$$

Where k is a tunable parameter whose calibration steers the difference between the within and between relationship densities. The effect of varying k is further highlighted in Section 7.6.

7.4 Relative graph generation

This section proposes a generative model that can produce a sequence of graph snapshots while controlling the evolution of the degree distribution and community structure. This procedure consists of transforming a graph snapshot into its successor by applying a number of graph updates.

7.4.1 Earth mover's distance

To mirror the fact that real-world graphs gradually evolve, we minimize the effort of transforming a graph snapshot into its successor. Hence, we reduce the number of graph updates between snapshots by relying on an optimal transport solver proposed by Flamary et al. in [83] to compute the minimal distance (Earth mover's distance) between the degree distributions of each pair of successive graph snapshots. The Earth mover's distance is a measure of distance over a domain D between two distributions of the form $\{(x_1, \omega_1), ..., (x_n, \omega_n)\}$ where $x_i \in D$ and ω_i is the density of x_i. Having this, the problem reduces to the computation of an optimal flow (transportation matrix) $T = [t_{ij}]$ between two distributions $P = \{(x_1, p_1), ..., (x_n, p_n)\}$ and $Q = \{(y_1, q_1), ..., (y_n, q_n)\}$ such that t_{ij} is the mass transported between p_i and q_j which minimizes the overall cost:

$$\min_T \sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij} d_{ij}$$

where $d_{ij} = d(x_i, y_j)$ is a measure of distance between x_i and y_j. The following constraints must hold for the optimal flow T:

$$t_{ij} \geq 0,\ 1 \geq i \geq n,\ 1 \geq j \geq m$$

$$\sum_{j=1}^{m} t_{ij} \leq p_i,\ 1 \geq i \geq n, \quad \sum_{i=1}^{n} t_{ij} \leq q_j,\ 1 \geq j \geq m$$

Once the optimal flow T is found, the EMD between P and Q is computed as follows:

$$EMD(P, Q) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij} d_{ij}}{\sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij}}$$

The EMD is fundamental in our generation procedure since it is used to compute the distance between degree distributions, as described in the following section.

7.4.2 Baseline relative graph generation

In this section, we provide the baseline procedure of transforming a graph G with degree distribution ϕ into G' with degree distribution ϕ', which we refer to as the Baseline relative graph generation. Note that we use this technique for generating temporal graphs such that G and G' corresponds to successive graph snapshots. Towards the

generalization of the mechanism, we remove the notion of time in this section. This transformation is enabled by atomic graph operations, including adding and deleting a node or a relationship. Following the assumption that temporal graphs gradually evolve, this number of graph operations between successive snapshots should be minimized, which is assured in our model by applying an optimal transport method.

Consider the input graph $G = \{V, E, \phi\}$ and degree distribution ϕ', the generated output graph $G' = \{V', E', \phi_{G'}\}$ such that $\phi_{G'}$ is an approximation of ϕ'. We define the distance between two degree distributions ϕ and ϕ' as the earth mover's distance $EMD(\phi, \phi')$.

Consider $\delta n = |V'| - |V|$ as the total number of nodes to be added to or removed from the graph based on whether δn is a positive or negative number, respectively. When adding a new node, this node is assigned a degree equal to 0, and deleting a node consists of removing the node with its corresponding incident relationships. This transformation phase assures that G and G' share the same number of nodes, hence, enabling the transformation of ϕ into ϕ'. To morph ϕ into ϕ', a transportation matrix T is computed, where each row corresponds to a degree d in the set of degrees in the source distribution ϕ and each column corresponds to a degree d' in the set of degrees in the target distribution ϕ'. Now, each cell consists of the portion of nodes having a degree d for which links are to be inserted or removed to be assigned a total number of relationships equals to degree d'. That is, a node v_i, with a degree $d_{v_i} = d$, will be assigned a degree variation of $\delta d_{v_i} = d' - d$ resulting in a total number of relationship insertions and deletions defined as D^+ and D^-, respectively.
We assign, for each node v_i, a linkage probability $p^+_{v_i}$ or a breakage probability $p^-_{v_i}$ defined as extensions of the CL linkage probability (7.1):

$$p^+_{v_i} = \frac{\delta d_{v_i}}{D^+}, \delta d_{v_i} > 0 \tag{7.6}$$

$$p^-_{v_i} = \frac{-\delta d_{v_i}}{D^-}, \delta d_{v_i} < 0 \tag{7.7}$$

We collect nodes sharing the same degree variation $\delta d = d' - d$ into a linkage pool if $\delta d > 0$ and in a breakage pool if $\delta d < 0$. Consider $\gamma_{d \to d'} = \{v_i | v_i \in V \wedge \delta_{v_i} = d' - d\}$ to be the pool containing nodes having a degree d that should be transformed into d'. We compute the probability of picking a linkage or breakage pool $p^+_{\gamma_{d \to d'}}$ and $p^-_{\gamma_{d \to d'}}$ as

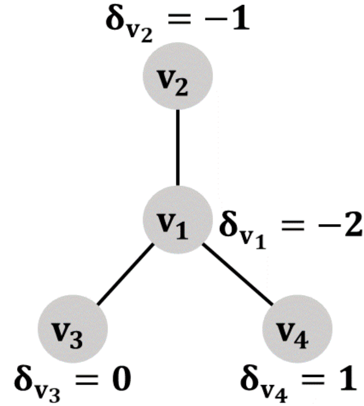


Figure 7.3 – Graph representing the case of a non-possible relationship breakage.

follows:

$$p^{+}_{\gamma_{d\to d'}} = \frac{\delta d|\gamma_{d\to d'}|}{D^{+}}, \delta d > 0$$

$$p^{-}_{\gamma_{d\to d'}} = \frac{-\delta d|\gamma_{d\to d'}|}{D^{-}}, \delta d < 0$$

However, breaking a relationship might be impossible in situations where the source degree variation δd is negative and the sum of the negative degree variations of its neighbors is higher than δd. For the sake of illustration, we present in Figure 7.3 a graph in which the number of relationships to remove from a node is higher than the sum of the number of relationships to remove from its neighboring nodes. That is, the transformation of this graph implies removing 2 relationships from node v_1 since $\delta v_1 = -2$. However, the number of the relationships that have to be removed from the neighboring nodes of v_1 is equal to $\delta_{v_2} = -1$ since $\delta_{v_3} = 0$ and $\delta_{v_4} = 1$. To overcome this, we repeat the morphing procedure until EMD(ϕ, ϕ') reaches the desired threshold. Our simulations have proved that the value of EMD(ϕ, ϕ') converges rapidly towards the minimum threshold after a tolerable number of iterations. This statement will be further highlighted in Section 7.6.

7.4.3 Relative community-aware graph generation

A more complex version of the previously described relative graph generation consists of preserving the graph community structure in the transformation procedure. That is, the input of our community-aware relative graph generator is the graph $G = \{V, E, \phi_G, M_G\}$, the desired degree distribution ϕ and the stochastic block matrix M. However, the output consists of a graph $G' = \{V', E', \phi_{G'}, M_{G'}\}$ where $\phi_{G'}$ is an approximation of ϕ and $M_{G'}$ is an approximation of M.

Recall that the generation procedure depicted in section 7.3.3 produces a graph with a given expected degree distribution and stochastic community matrix based on the proposed linkage probability duality presented in Equations (7.1) and (7.3). Indeed, a relative community-aware graph generation is based on extending the duality mentioned above by considering the node's degree variation instead of its desired degree. That is, the following linkage and breakage probabilities ($p^+_{v_i}$, $p^-_{v_i}$) of a node v_i belonging to a community c_m present a straightforward extension of Equations (7.6) and (7.7):

$$p^+_{v_i} = \frac{\delta d_{v_i}}{D^+_{c_m}} p_{c_m}, v_i \in c_m$$

$$p^-_{v_i} = \frac{\delta d_{v_i}}{D^-_{c_m}} p_{c_m}, v_i \in c_m$$

We consider $D^+_{c_m}$ and $D^-_{c_m}$ to be the total number of relationship insertions and deletions in community c_m, respectively. We also consider p_{cm} to be the linkage probability of the community c_m and δ_{vi} to be the degree variation of the node v_i.

As described in Section 7.4.2, we use a transportation matrix to compute the fraction of nodes (n_{ij}) that should be affected with a given degree variation ($\delta_v = d_j - d_i$). In the community-aware procedure, each node belongs to a community with an expected between and within relationship densities. Hence, the fraction of nodes having a given degree variation is put under several conditions to account for the community structure. That is, finding the portion n^{cm}_{ij} of nodes in the community c_m should satisfy the three conditions that are detailed below. Each condition i results in a system of linear equations of the form $A_i X = B_i$ where X is a vector composed of n^{cm}_{ij} such that $X = \{n^{c_k}_{ij} \,|\, \forall 1 \leq i \leq |\phi_G| \wedge \forall 1 \leq j \leq |\phi| \wedge 0 \leq k \leq |C|\}$.

Condition 1**:** For each community $c_m \in C$, conditions stating that $D^+_{c_m} = D^+ p_{c_m}$ and $D^-_{c_m} = D^- p_{c_m}$ must hold, where D^+ and D^- are the total number of relationship insertions and deletions in all communities of C, respectively. Incorporating n^{cm}_{ij} in the

previous condition translates to the following equality:

$$\sum_{i=0}^{|\phi_G|}\sum_{j=0}^{|\phi'|}(d_j - d_i)n_{ij}^{c_m} = (\sum_{i=0}^{|\phi_G|}\sum_{j=0}^{|\phi'|}(d_j - d_i)n_{ij})p_{c_m}$$

Where ϕ_G and ϕ' are the source and target degree distributions.

Algorithm 3: CRGG

```
    Input: G = {V, E, φ_G, M_G}, φ, M, ε, max_iter, cur_iter
    Output: G' = {V', E', φ_G', M_G'}
 1  loop
 2      T ←getTransportMatrix(φ_G, φ);
 3      X ←getVector(φ_G, φ, T, M);
 4      (D+, D−) ←getNumberOfEdges(T);
 5      cdfCom ←getCDFComs(M);
 6      (cdfPools+, cdfPools−) ←initCDFPools();
 7      L ←initLogs();
 8      for c_m ∈ C do
 9          (cdfPools+_cm, cdfPools−_cm) ←getCDFPools(X, c_m);
10          cdfPools+[m] ← cdfPools+_cm;
11          cdfPools−[m] ← cdfPools−_cm;
12      for i ← 0 to D+ do
13          (c_n, c_m) ←chooseComs(cdfCom);
14          (n_i, n_j) ←chooseVertices(cdfPools+[n], cdfPools+[m]);
15          L.addEdge(n_i, n_j);
16      for i ← 0 to D− do
17          (c_n, c_m) ←chooseComs(cdfCom);
18          (n_i, n_j) ←chooseVertices(cdfPools−[n], cdfPools−[m]);
19          L.removeEdge(n_i, n_j);
20      G' ←applyLogs(G, L);
21      ε' ←getEMD(φ, φ'_G);
22      if ε' ≥ ε ∧ cur_iter < max_iter then
23          cur_iter ← cur_iter + 1 ;
24          G' ← CRGG(G', φ, M, cur_iter, max_iter);
25      else
26          return G';
```

Condition 2**:** This condition states that the sum of all portions of nodes with degree variation $d_j - d_i \; \forall d_j \in \phi_t$ in c_m should be equal to the portion $n_i^{c_m}$ of nodes in c_m having a degree d_i resulting in the following equality:

$$\sum_{j=0}^{m} n_{ij}^{c_m} = n_i^{c_m}$$

Condition 3**:** This condition states that the portion n_{ij} of nodes with degree variation $d_j - d_i$ in the graph must be equal to the sum of all portions $n_{ij}^{c_m} \; \forall c_m \in C$.

$$\sum_{c_m=0}^{n_c} n_{ij}^{c_m} = n_{ij}$$

By solving the concatenated system of equations obtained from the previous conditions $\texttt{concat}(A_1, A_2, A_3)X = \texttt{concat}(B_1, B_2, B_3)$, we find the vector X, hence the portions of nodes with a given degree variation and belonging to a given community ($n_{ij}^{c_m}$).

To accelerate the generation procedure, we gather nodes in linkage or breakage pools and assign each pool with the linkage or breakage probability. These pools are created on a local basis (i.e., in each community) such that the nodes with the same degree variation $\delta d = d' - d$ and belonging to the same community c_m are collected in a single pool $\gamma_{d \to d'}^{c_m}$. We compute the probability of picking a linkage or breakage pool $p^+_{\gamma_{d \to d'}, c_m}$ and $p^-_{\gamma_{d \to d'}, c_m}$ as follows:

$$p^+_{\gamma_{d \to d'}, c_m} = \frac{\delta d |\gamma_{d \to d'}^{c_m}|}{D^+}, \delta d > 0$$

$$p^-_{\gamma_{d \to d'}, c_m} = \frac{-\delta d |\gamma_{d \to d'}^{c_m}|}{D^-}, \delta d < 0$$

Let us now describe our community-aware relative graph generation (CRGG) algorithm (Algorithm 3). The input parameters are the graph snapshot G, desired degree distribution ϕ, density community matrix M, the threshold of the EMD distance between ϕ_G and ϕ, the maximum number of repetitions max_iter, and the current number of repetitions cur_iter. Whereas the output is a new graph snapshot G'. Note that the value of cur_iter is equal to 0 in the first iteration. The transportation matrix T is computed using the function `getTransportMatrix` by taking the degree distributions ϕ_G and ϕ as input. The function `getVector`, computes A and B based on the Conditions 1, 2 and 3 and solves the system of equations defined by $AX = B$ to find the vector X. The

total number of edges to add (D^+) and delete (D^-) are then computed based on the transportation matrix T. The function `getCDFComs` computes the cumulative distribution function $cdfCom$ based on the density community matrix M. Then, vectors $cdfPools^+$ and $cdfPools^-$ represent the cumulative density functions of the linkage and breakage pools, and a list of logs (graph updates) L are initialized. The function `getCDFPools` is used to compute the cumulative distribution functions $cdfPools^+$ and $cdfPools^-$ based on the probabilities $p^+_{\gamma_{d\to d'},c_m}$ and $p^-_{\gamma_{d\to d'},c_m}$. Adding and removing edges are repeated D^+ and D^- times, respectively. In each iteration, communities c_n and c_m are picked based on $cdfComs$ and vertices n_i and n_j are picked using $cdfPools^+[n]$ and $cdfPools^-[m]$. Now, an addition or deletion graph update between the chosen vertices is added to the list of logs using functions `addEdge` and `removeEdge` whether the vertices were chosen from the linkage or breakage pools. However, breaking an edge might be impossible in some situations, as shown in Figure 7.3. In such a use case, no graph update is added to the list of logs L. Finally, the EMD distance ϵ is computed between the obtained degree distribution ϕ'_G and the desired one ϕ. If ϵ' is higher than ϵ and the number of repetitions cur_iter has not yet reached max_iter, the same algorithm is repeated on the newly computed graph snapshot G'. The computation stops when ϵ' is lower than or equal to ϵ, or the number of repetitions has already been reached.

7.4.4 Accuracy of the generation procedure

To measure how far the characteristics of the generated graphs are from the ground truth parameters, we define two distance metrics ε_d and ε_c.

The first metric ε_d measures the inaccuracy of approximating the degree distributions of the generated graphs with the given sequence of degree distributions. That is, it measures the root mean square of the EMD distances between each degree distribution ϕ_i in the given sequence $\{\phi_1, \ldots, \phi_n\}$ and its corresponding degree distribution ϕ_{G_i} in the sequence $\{\phi_{G_1}, \ldots \phi_{G_n}\}$ extracted from the generated graphs. We compute ε_d as follows:

$$\varepsilon_d = \frac{\sqrt{\sum_{i=1}^{n}(EMD(\phi_i, \phi_{G_i}))^2}}{n}$$

The second metric, ε_c, measures the inaccuracy of approximating the community density matrix of the generated graphs with a given stochastic matrix. More precisely, it measures the root mean square of the difference between the Frobenius norms of the

given stochastic matrix M and the stochastic matrix M_{G_i} extracted from every generated graph snapshot. The formula we use to compute ε_c is given below:

$$\varepsilon_c = \frac{\sqrt{\sum_{i=1}^{n}(F(M) - F(M_{G_i}))^2}}{n}$$

where $F(M)$ is the Frobenius norm of the stochastic community matrix M. We recall that the Frobenius norm of a matrix A of dimensions (n,m) is defined as follows:

$$F(A) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}|a_{ij}|^2}$$

These distance metrics will be further outlined in the experimental evaluation section (Section 7.6).

7.5 Generating evolving properties

Real-world graphs are usually enriched with various node and relationship properties. Nodes and relationships may be of different *types* and have several *properties* with values that may change over time. Hence, we included a decorator [1] to RTGEN to add time-evolving properties to the nodes and relationships of the generated temporal graphs.

7.5.1 Model

Our decorator works in a post-processing phase such that it operates on an existing (already generated) dynamic graph with no properties on the nodes and relationships. We assume that in some use cases, the values of a node's properties highly depend on those of its neighbors. In a social network, the interests of a person are more likely to be identical to those of his friends (i.e., neighbors in the graph). Similarly, the neighboring nodes in the Thing'in graph share similarities (i.e., common properties). Moreover, the value evolution of a node property at a time step might also depend on the previous values. We capture this by allowing the users to tune the degree of dependency

1. Amaury Bouchra Pilet (A post-doctoral fellow in Orange Labs) and Thomas Hassan (A research engineer in Orange Labs) have helped us include time-evolving properties by assisting with the design of the decorator.

between neighboring nodes and the different consecutive states of each node. That is, we consider the values of the properties to be Markovian in time (Markov chain) and space (Markov field). A Markov chain is a stochastic model describing a sequence of possible actions in which the probability of each action depends only on the state obtained in the previous action.

Algorithm 4: Decorator for dynamic graphs

Input: $G = \{N, R\}$, Nt, $Ncom$, $Nspe$, $Ninf$, $Nevo$, Rt, $Rcom$, $Rspe$, Rvo, T

Output: $G = \{N, R\}$ (decorated)

`/* Initialisation */`

1. $Ntypes \leftarrow$ RANDOMSTRINGS(Nt); ▷Generate Nt types
2. $Rtypes \leftarrow$ RANDOMSTRINGS(Rt); ▷Generate Rt types
3. $NcomPro \leftarrow$ RANDOMSTRINGS($Ncom$); ▷...common properties (node)
4. $RcomPro \leftarrow$ RANDOMSTRINGS($Ncom$); ▷...common properties (edge)
5. $NspePro \leftarrow$ MAP($Ncom$);
6. **for** $type \in Ntypes$ **do**
7. $NspePro[type] \leftarrow$ RANDOMSTRINGS($Nspe$); ▷...specific properties (node)
8. **for** $type \in Rtypes$ **do**
9. $RspePro[type] \leftarrow$ RANDOMSTRINGS($Rspe$); ▷...specific properties (edge)

`/* Decoration */`

10. **for** $t \in T$ **do**
11.   **for** $n \in N(t)$ **do** ▷Processing nodes
12.   **if** $\exists n(t-1)$ **then** ▷Existing node
13. $n.type \leftarrow n(t-1).type$;
14.   **for** $p \in NcomPro \cup NspePro[n.type]$ **do**
15. $s \leftarrow 0$ **for** $n \in n.k$ **do**
16.   **if** $\exists n.props[p]$ **then** $s+= \mathbb{1}_{n.props[p]}$ **else** $s+=$ RAND($[0,1]$);
17. $n.props[p] \leftarrow ([Ninf \times s + (1 - Ninf) \times$ RAND($[0,1]$)$] > 0.5)$;
18.   **else** ▷New node
19. $n.type \leftarrow$ RAND($Ntypes$);
20.   **for** $p \in NcomPro \cup NspePro[n.type]$ **do**
21. $n.props[p] \leftarrow ([(1 - Nevo)\mathbb{1}_{n(t-1).props[p]} + Nevo \times$ RAND($[0,1]$)$] > 0.5)$;
22.   **for** $r \in R(t)$ **do** ▷Processing relationships
23.   **if** $\exists r(t-1)$ **then** ▷Existing relationship
24. $r.type \leftarrow r(t-1).type$;
25.   **for** $p \in RcomPro \cup RspePro[r.type]$ **do**
26. $r.props[p] \leftarrow$ RAND($\top, \bot$);
27.   **else** ▷New relationship
28. $r.type \leftarrow$ RAND($Rtypes$);
29.   **for** $p \in RcomPro \cup RspePro[r.type]$ **do**
30. $r.props[p] \leftarrow ([(1 - Revo)\mathbb{1}_{r(t-1).props[p]} + Revo \times$ RAND($[0,1]$)$] > 0.5)$;

We list the rules we apply to the labels, types, and properties of nodes and relationships as follows:

— The labels and types are static, whereas properties are dynamic.

— The label and property values of a node should depend on the types of its incident relationships and the labels and properties of its neighbors.

— The value of the property of a node at a given snapshot depends on its value in the previous snapshot.

This rather generic framework would allow using measures taken from real graphs to parameterize the decorator. It cannot cover more complex cases (such as dependency on neighbors' neighbors), but such dependencies would be harder to extract anyway without relying on more advanced methods, e.g., machine learning-based methods, which are out of the scope of this dissertation.

However, we still need a more straightforward, necessarily more limited, implementation of this model, enabling us to decorate graphs based only on a few numeric parameters.

7.5.2 Implementation

To propose an easy-to-configure implementation of our model, we focus on generating boolean properties as an initial step but plan to extend it later with other types. Some of these properties are common to all node labels, and others are specific to each node label. In this implementation, the labels and types of nodes and relationships are random. Node properties initially depend on neighbors' properties but then evolve following the Markov chain model. Relationship properties are initiated randomly and then change as a Markov chain.

Our decorator takes the following parameters:

— $Nt \in \mathbb{N}$ and $Rt \in \mathbb{N}$ number of node and relationship types (random strings).

— $Ncom \in \mathbb{N}$ and $Rcom \in \mathbb{N}$ number of properties shared by all nodes and relationships.

— $Nspe \in \mathbb{N}$ and $Rspe \in \mathbb{N}$ number of properties specific to each node and relationship type.

— $Nevo \in [0, 1], Revo \in [0, 1]$ the evolution factor of nodes and relationships (0: static, 1: independent from previous values).

— $Ninf \in [0, 1]$ the influence factor of neighbors on a node's properties (0: inde-

pendent, 1: deterministic).

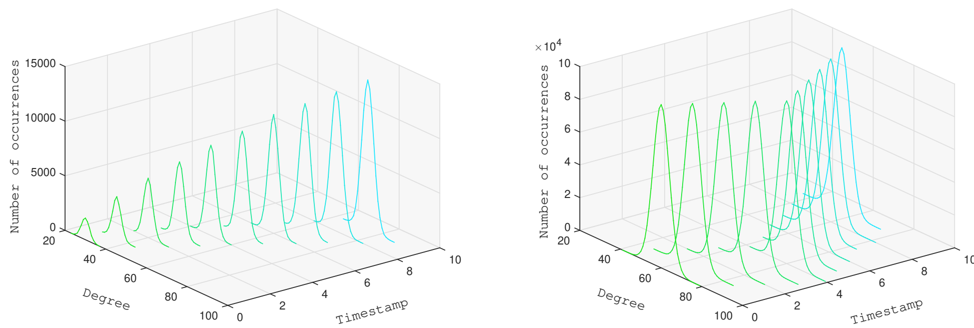
That is, the evolution factor controls the dependency of a property value at a time step with a previous one. The influence factor controls the value dependency of a node property with its neighbors. Since the generation process decorates nodes sequentially, it is likely that its neighbors are not already decorated when initializing a node. In addition, for type-specific properties, values do not exist for neighbors of a different type. In this case, the value is replaced by a random one. The decoration process is presented in Algorithm 4.

We denote by $G(t)$, $N(t)$ and $R(t)$ a graph, its set of nodes and its set of relationships at time t. For a given node n/edge r (time t implicit), we denote by $n(t-1)$/$r(t-1)$ the previous state of the node/edges, if any. That is, $\exists n(t-1)$/$\exists r(t-1)$ is $\top$ (true) if the node/edge exists at time $t-1$, $\bot$ (false) otherwise. T denotes a sequence of time steps (corresponding to graph snapshots). $n.k$ denotes the set of the neighbors of a node. We use ($\exists n.props[p]$) to indicate whether property p is set for node n at the time step corresponding to the current iteration.

7.6 Experimental evaluation

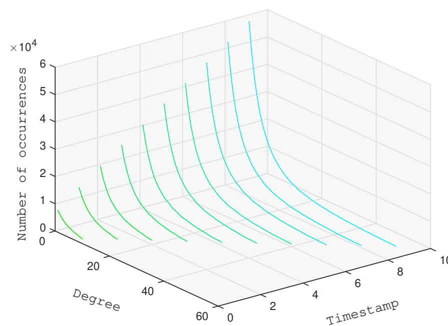
In this section, we present the results of the experiments we have conducted to evaluate the performance of our generator RTGEN. We also provide insight into how changing the input parameters can steer the characteristics of the generated temporal graphs.

RTGEN tool the users of our RTGEN tool can pass the input parameters to describe the desired sequence of degree distributions or stochastic community matrix and the format of the generated output files to RTGEN using a terminal command. RTGEN proposes two output types: **snapshot-based** and **event-based**. The snapshot-based format consists of a sequence of graph snapshots in a separate file. The event-based format consists of generating the sequence of graph updates (events) that we applied between successive snapshots to transform one snapshot into the next.



(a) Gaussian degree distribution of a growth-only graph

(b) Gaussian degree distribution of a graph with relationship deletions



(c) Zipfian degree distribution of a growth-only graph

Figure 7.4 – Evolution of the degree distribution of the obtained graph snapshots

Experimental setup

The experiments were conducted on a single machine equipped with Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz 1.90 GHz, 16 GB memory and 500 GB SSD. We used Go 1.17.5 and Python 3.8.0. Besides, we referred to the optimal transport solver proposed by Flamary et al. in [83]. The graphs shown in this section are visualized using the tool Gephi [26], which offers network visualization facilities and community detection algorithms [31].

Preliminaries

In the following experiments, we refer to two types of common degree distributions, Gaussian f_G and Zipfian f_Z, defined as follows:

$$f_G(x) = \frac{1}{\sigma\sqrt{\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

$$f_Z(x) = \frac{1}{(x+v)^s} \quad x \in [0, d_{max}]$$

We consider that the value of a parameter k in iteration i depends on its value in the previous iteration $i-1$ such that $k_i = k_{i-1} + \delta k$. This is applied to the parameters of the degree distributions μ, σ, d^{max}, s, v, and the total number of nodes denoted n. That is, δn denotes the number of nodes to be added or removed from the graph in the relative generation process. RTGEN also generates the first snapshot, implying that the parameters of the degree distribution of the first snapshot should be given.

7.6.1 Controlling the evolution of the degree distribution

In this experiment, we show the evolution of the degree distribution of a sequence of graph snapshots generated with the relative generation procedure given a set of input parameters. Hence, we consider Gaussian and Zipfian degree distributions with different parameters and plot the obtained degree distributions in Figures 7.4(a), 7.4(b) and 7.4(c). Figure 7.4(a) shows the evolution of the degree distribution of a generated sequence of 10 graph snapshots given the following parameters: $\{n^0 = 10K, \quad \mu^0 = 30, \quad \sigma^0 = 2, \quad \delta n = 10k, \quad \delta\mu = 5, \quad \delta\sigma = 0.1\}$. By setting $\delta\mu$ to 5, we increase the average degree by 5 between each pair of snapshots. This can model a growth-only graph where the average relationship degree regularly increases.

However, some real-world graphs are not growth-only because they are subject to relationship deletions. This is indeed the case of the Thing'in graph, where a number of short-term connections between IoT devices are only valid during peak hours. To model this characteristic, RTGEN also supports relationship deletions. The evolution of the degree distribution with relationship deletions is presented in Figure 7.4(b). Let the following parameters define the evolution of degree distribution for $i \in [0, 4]$: $\{n^0 = 1M, \quad \mu^0 = 60, \quad \sigma^0 = 4, \quad \delta n = 0, \quad \delta\mu = 5, \quad \delta\sigma = 0\}$. Whereas the following parameters define its evolution for $i \in [5, 9]$: $\{n^0 = 10K, \quad \mu^0 = 80, \quad \sigma^0 = 2, \quad \delta n =$

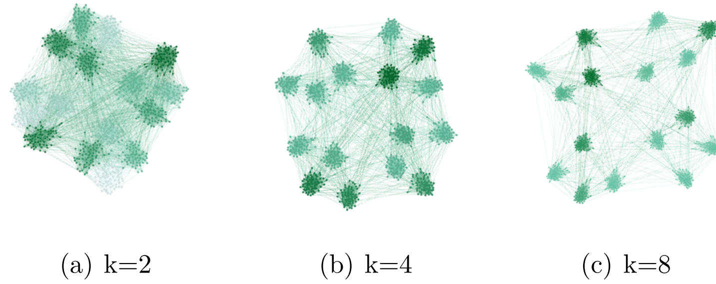
$0, \quad \delta\mu = -5, \quad \delta\sigma = 0\}$. Indeed, setting $\delta\mu$ to -5 indicates that the average degree decreases by a value of 5 between each pair of successive graph snapshots.

Since real-world temporal graphs usually exhibit a power law degree distribution, we also generated graphs with an evolutionary Zipfian degree distribution composed of 10 graph snapshots as shown in Figure 7.4(c). For this generated temporal graph, we set the following parameters $\{n^0 = 50k, \quad s^0 = 2.5, \quad v^0 = 10, \quad d^0_{max} = 10, \quad \delta n = 50k, \quad \delta s = 0, \quad \delta v = 0, \quad \delta d_{max} = 5\}$. By setting parameter δd_{max} to 5, we consider that the maximum degree of nodes increases by a value of 5 between each pair of successive snapshots. The value of δn indicates that $50k$ new nodes join the graph between successive snapshots. These parameters reflect the growth of many real-world temporal graphs where new nodes join the graph, and new connections are created as time elapses.

7.6.2 Controlling the community structure

In this experiment, we present the generated community structure with different parameters of the stochastic community matrix and the effect of varying parameter k of the hierarchical tree. As described in Section 7.3.4, RTGEN can auto-generate the stochastic community matrix representing a hierarchical community structure. Consider a stochastic community matrix generated by setting $b = 4$ and $h = 2$. As depicted in Equation 7.2, one can tune the parameter k to control the within and between relationship densities (see Section 7.3.4 for a description of parameter k). Hence, we select three different values of k in $\{2, 4, 8\}$. Furthermore, consider $n = 1000$ to be the total number of nodes and $\mu = 30$ and $\sigma = 2$ to be the parameters of a Gaussian distribution. In this experiment, we generate a single graph snapshot relying on the generation procedure proposed in Section 7.3.3. The generated graphs are shown in Figures 7.5(a), 7.5(b) and 7.5(c) using the Gephi tool. It can be noticed that the difference between the within and between relationship densities is proportional to k since $k \propto p^{in}_{c_i} - p^{out}_{ci}$ where $p^{in}_{c_i}$ and p^{out}_{ci} are the within and between linkage probabilities of a community c_i.

Figure 7.6 presents the modularity in function of parameter k which we vary from 0 to 32. Modularity is a measure to quantify the goodness of community structure. It compares, for all the communities, the fraction of relationships falling within the given community with the expected fraction if relationships were distributed at random. It is clear from the results that the modularity increases with the increase of the value of k.



(a) k=2 (b) k=4 (c) k=8

Figure 7.5 – A visualization of the generated graphs with a hierarchical community structure with parameters: $b = 4$, $h = 2$, $c = 4$ and a varying k.

This is justified by the fact that k is proportional to the difference between within and between relationship linkage probabilities $p_{c_i}^{in} - p_{ci}^{out}$.

7.6.3 Generating graphs with deletions

As mentioned in Section 7.4, our relative graph generation procedure can produce several relationship deletions. This can be cumbersome when the number of relationships to delete for a given node is higher than the total sum of relationships to delete from its neighboring nodes. We solve this problem by repeating the generation process until reaching an acceptable error threshold defined by the EMD between the obtained and desired degree distributions.

Figure 7.10 shows the variation of the number of iterations and the execution time of the generation process in function of the threshold error defined by the EMD. The results show that our generation procedure converges rapidly towards a tolerable threshold. That is, a threshold equal to 0.001 can be reached with only 7 iterations. By comparing the execution time of a single iteration with that of repetition (e.g., 7 iterations), we can derive that the execution time of repeating the generation is lower than that of the first iteration. This is justified by the fact that the majority of modifications are added in the first iteration, and only the remaining nodes whose linkage probability does not satisfy the sum of the linkage probabilities of their neighboring nodes are considered in the next iteration. Note that these results are obtained from the generation of two successive snapshots with the following input parameters of a Gaussian degree distribution: $\{n_0 = 500k, \quad \mu_0 = 60, \quad \sigma_0 = 2, \quad \delta n = 0, \quad \delta mu = -30, \quad \delta\sigma = 0\}$.

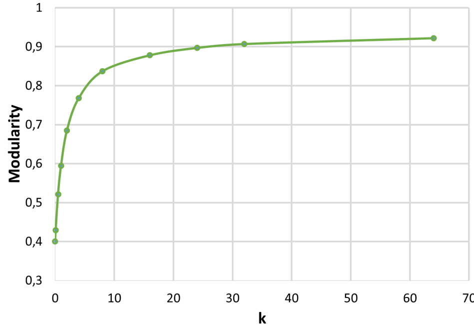


Figure 7.6 – Modularity value in function of parameter k ranging from 0 to 32.

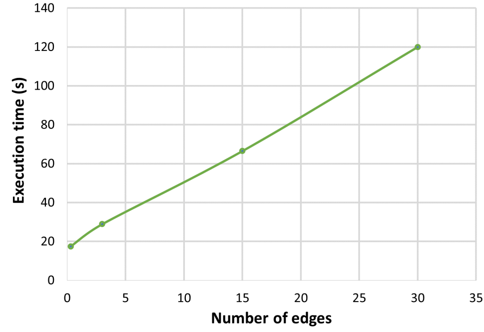


Figure 7.7 – Execution time in function of the number of relationships.

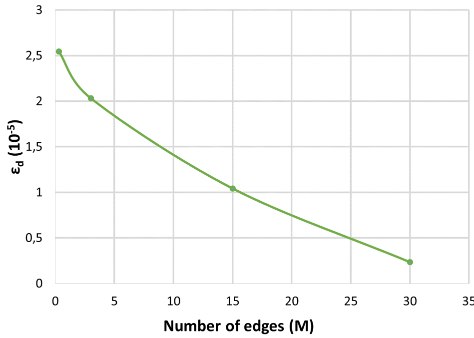


Figure 7.8 – The variation of ε_d in function of the number of relationships.

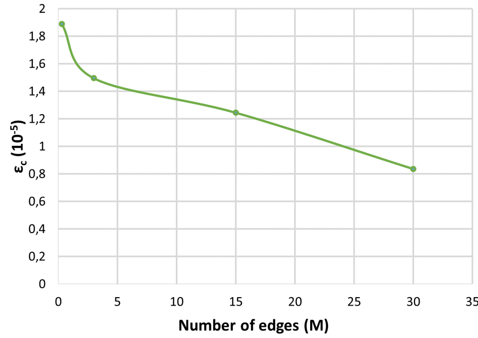


Figure 7.9 – The variation of ε_c in the function of the number of relationships.

7.6.4 Accuracy of the generation procedure

We quantify the accuracy of the generated graphs with the given parameters by computing the distance metrics ε_d and ε_c defined in Section 7.4.4. We generated a sequence of $n = 5$ snapshots with the following parameters of Gaussian degree distribution: $\{n_0 \in \{10k, \quad 100k, \quad 500k, \quad 1M\}, \quad \mu_0 = 30, \quad \sigma_0 = 2, \quad \delta_n = 0, \quad \delta\mu = 10, \quad \delta\sigma = 0\}$. Besides, we controlled the community structure by fixing the following parameters of a hierarchical tree: $h = 2, \quad b = 2, \quad c = 4, \quad k = 0$.

Figures 7.7, 7.8 and 7.9 plot the execution time, value of ε_d and ε_c in function of the total number of created relationships from applying the Gaussian degree distribution whose parameters are given above. The execution time increases with the number of generated relationships. However, the distances ε_d and ε_c decrease with the increase of the number of created relationships. This implies our generator RTGEN approximates

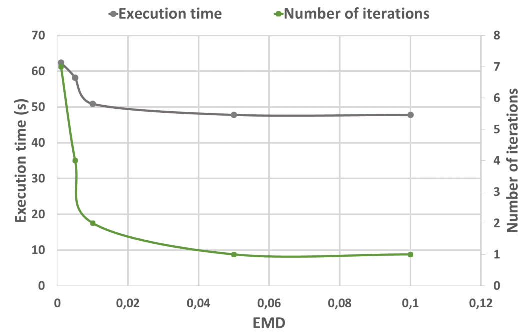


Figure 7.10 – The variation of the number of iterations and execution time in function of the EMD.

the given sequence of degree distribution and community structure more accurately as the number of relationships grows. Hence, our generator performs better for large-scale graphs.

7.7 Conclusion

In this chapter, we addressed the generation of temporal graphs that are critical for benchmarking temporal graph management systems. We proposed RTGEN, a novel temporal graph generator that produces a sequence of graph snapshots whose community structure and evolution of the degree distribution results from approximating user-defined parameters. This generation procedure consists of relatively generating a graph snapshot from a previous one by applying several atomic graph operations. Our generation technique relies on an Optimal transport solver to approximate a user-defined sequence of degree distributions while minimizing the number of operations needed to transform one snapshot into its successor. We conducted several experiments that validated our generation procedure's efficiency and accuracy. Although we tested the accuracy of the generated graphs' characteristics with the given user-defined parameters, we plan to evaluate their accuracy with parameters extracted from real-world temporal graphs.

The work described in this chapter paves the way for several possible extensions, such as adding a dynamic community structure. Indeed, the communities in real-world graphs are subject to splits, merges, shrinks, or expansions, which should also be modeled in synthetic graphs. However, such an extension falls outside the scope of

this thesis.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The goal of this thesis has been to address the challenges in the design of a temporal graph system. We were particularly interested in the use case of Thing'in [1], where many queries about the evolution of graphs are more insightful for end users than traditional non-temporal queries. Since existing graph databases do not account for the temporal dimension, we decided to design our temporal graph system Clock-G from the ground up. We believe that a non-temporal graph system might lead to performance deterioration when extended with the temporal dimension because it is not designed with native temporal support. We summarize in Section 8.1 the different aspects that we have addressed to design our system Clock-G. Besides, we outline the possible expansions and improvements which may be made to Clock-G in Section 8.2. Finally, Section 8.3 summarizes our contributions to the industrial use case of the platform Thing'in and provides our future plans.

8.1 Summary of contributions

The main contributions of this thesis are related to the design of Clock-G. These contributions are categorized under the four chapters which discuss them, namely: The temporal graph query language T-Cypher in Chapter 4, the temporal graph storage technique δ-*Copy+Log* in Chapter 5, the temporal graph processing engine in Chapter 6, the temporal graph generator RTGEN in Chapter 7.

Chapter 2 discussed the related work on the approaches used in the design of non-temporal graph systems and showed how temporal graph systems extend or differ from these approaches. We also outlined the main challenges addressed in the related work.

Chapter 3 presented the overall architecture of our system Clock-G briefly to give a

1. https://tech2.thinginthefuture.com/

broad perspective of the work accomplished during this thesis.

Chapter 4 presented the temporal graph query language T-Cypher. We designed T-Cypher to fulfill our required querying functionalities: Temporal slicing, temporal graph pattern matching, temporal paths, and temporal aggregation. We also described our industrial integration of T-Cypher into the Thing'in platform. This integration consisted of building a temporal layer to manage the temporal dimension on top of an existing non-temporal graph database. Evaluating T-Cypher queries consisted of finding the equivalent Cypher query based on translation rules and evaluating it against Neo4j. This translation demonstrated that using T-Cypher significantly reduced the verbosity of queries compared to Cypher, motivating the proposal of a native temporal query language.

Chapter 5 presented a new storage technique for temporal graphs. As we highlighted in the analysis of related work (Chapter 2), two of the most common methods in this context are the Log and Copy+Log. These techniques lead to an apparent trade-off between space/ query computation time complexities. The main limitation of the Copy+Log technique is the storage of full graph snapshots. Since the graph of Thing'in is growth mostly, the difference between consecutive snapshots contains mainly the added graph entities. In our proposed storage technique, δ-*Copy+Log*, we reduced the space usage of snapshots by storing the differences between consecutive snapshots instead of keeping them entirely. We also presented the implementation details of the δ-*Copy+Log* method in Clock-G. To evaluate the space usage and the execution time of our proposed method, we conducted several experiments using real and synthetic datasets. The results demonstrated that our approach can significantly reduce space usage compared to the Copy+Log technique while only adding a slight overhead to the execution time of queries compared with the Log technique. We also compared the performance of Clock-G with that of a non-temporal graph database Neo4j [175]. The obtained results demonstrated that Clock-G significantly outperforms Neo4j in terms of space usage and query execution time. Indeed, these results motivated our choice of building a graph system with native temporal support instead of relying on a non-temporal graph database.

Chapter 6 presented the processing of T-Cypher queries and the implementation details of our processing pipeline that we have integrated into Clock-G. We extended the graph algebra presented by Hölsch et al. in [126] with temporal dimension to translate T-Cypher queries into a set of algebraic operators. Also, we defined a cost model

that allows the estimation of the cardinality of an operator during a time interval. This cost model is used to compute the evaluation plan with the minimum estimated cost in a greedy manner. We then presented the data structure we use to keep track of the temporal histogram containing information about the evolution of the graph entities' cardinalities. We conducted several experiments using a synthetic dataset to evaluate the performance of our query processor. We compared the execution time of the best, random and worst evaluation plans. The results demonstrated that the best plans resulted in the minimum execution time compared to the random and worst. We also implemented T-Cypher queries using the non-temporal graph database Neo4j [175]. The results demonstrate that Clock-G outperforms Neo4j in terms of query execution time. This is in line with the results obtained in Chapter 5. However, the results in this chapter were obtained from evaluating more complex queries (i.e., T-Cypher queries) rather than basic queries.

Chapter 7 presented a generator for temporal graphs that can be used in benchmarking temporal graph systems. Available generators fail to produce temporal graphs given the evolution of their degree distribution which is a key characteristic of real-world graphs. Hence, we proposed RTGEN, a temporal graph generator based on a relative generation procedure that produces a snapshot from a previous one by applying a series of graph updates. A critical consideration in the design of RTGEN was the commonalities between the successive snapshots which imply that we apply the minimum number of graph updates that allow the transformation between two snapshots while respecting the parameters of the degree distribution. Besides the degree distribution, our generator controls the community structure of the generated graphs, which is another key characteristic of real-world graphs. We also coupled RTGEN with time-evolving properties. The evaluation of RTGEN demonstrated that it can generate temporal graphs given degree distributions and community structure parameters.

8.2 Future directions

From a future perspective, there are many expansions and possible improvements to the contributions elaborated in this thesis. Within this, we distinguish perspectives for the querying language, storage, query processing, and generation techniques. We list these improvements in the following.

8.2.1 Directions in querying temporal graphs

T-Cypher

As discussed in Chapter 4, the syntax of T-Cypher can cover a wide variety of insightful temporal queries. However, this syntax can be improved to cover full Cypher [86] queries instead of a fragment, including projections, and union of sub-queries. Besides, it would be interesting to study the expressive power versus efficiency of T-Cypher queries. A balance between expressiveness and efficiency (complexity of evaluation) means a balance between practice and theory. Indeed, some expressive T-Cypher queries can be challenging to evaluate in practice. For example, using Allen's algebra [11] to define temporal predicates between the variables of a temporal graph pattern is very useful but induces temporal joins whose implementation can be cumbersome as presented in Section 8.2.3.

In future work, we are also planning to include a comparison of the expressiveness of T-Cypher with alternative proposals.

Transformation of graph patterns

An interesting functionality in temporal graph querying would be the expression of different states of a pattern and returning the subgraphs that satisfy this transformation. Figure 8.1 illustrates an example of the evolution of a graph pattern composed of nodes $\{n_0, \ldots, n_3\}$ and relationships $\{r_0, \ldots, r_5\}$. As presented in the figure, the relationships appear and disappear between the time instants $\{t_0, t_1, t_2\}$. The returned sub-graphs should contain all the node and relationship states that satisfy this topological transformation at the requested time instants.

Spatio-temporal support

Besides the temporal dimension, it would be interesting to add the spatial dimension to graph queries. Indeed, Spatio-temporal databases have spurred interest in literature [108, 225, 189]. For instance, SPARQL-ST [189] extends SPARQL [113] with spatial and temporal constructs. The graph of Thign'in includes the geographic positions of each of the objects (i.e., nodes) such that the users of the platform can locate them by querying the graph. In this context, it would be interesting to study spatio-temporal queries that enable, for example, the expression of a geographic region (e.g.,

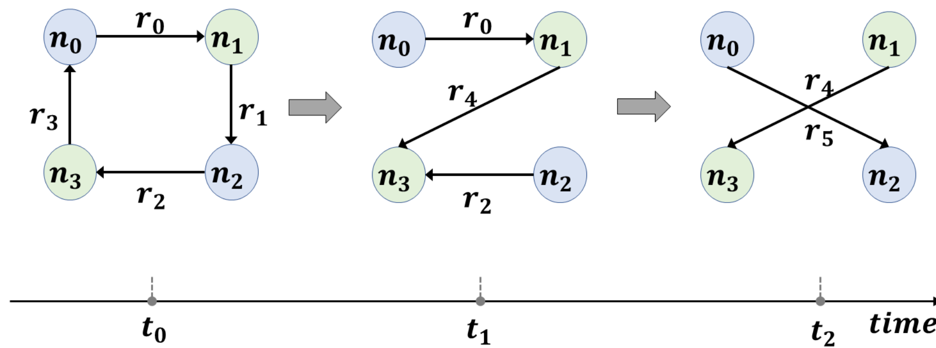


Figure 8.1 – Evolution of a graph pattern

polygon) in which the returned objects should be located during a given time interval. To capture this, we can include a spatial filtering clause (similar to the temporal slicing clause) into T-Cypher queries to specify the requested geographical region.

Temporal graph analytical support

An interesting research direction would be extending analytical queries with the temporal dimension. There has been substantial development in graph analytical systems. The available systems focus primarily on static graph processing namely Pregel [156], PowerGraph [98], and GraphLab [155]. However, some of the recent work focuses on building temporal graph analytical systems that extend current iterative models with the temporal dimension such as DynamoGraph [224] and Raphtory [223]. These systems enable the computation of the evolution of the Pagerank, Connected components, or centrality of the underlying temporal graphs. The users of Thing'in can benefit from analytical queries to track and analyze the evolution of key characteristics of the underlying graph. Besides, analytical queries could also allow the users of Thing'in to detect anomalies, such as the evolution of an IoT object in an unpredictable or uncontrollable manner.

8.2.2 Directions in storing temporal graphs

Extension of the δ-*Copy+Log* storage technique

As presented in Chapter 5, the δ-*Copy+Log* technique is based on the concept of storing M time windows in a single time bucket. This number is applied on all the time buckets such that the number of time windows between two successive snapshots is equal to M. Building upon the fact that recent data is more likely to be queried than old and stale data, we propose, as an extension for the δ-*Copy+Log* method, to increase the value of M of the buckets with the staleness of the data stored in each bucket. Having this, the older time buckets will contain more time windows, reducing the total number of graph snapshots that might be unnecessary or not frequently queried.

Data compression

Another amelioration related to storage is data compression. For our special use case (Thing'in), the data collected from IoT objects (e.g., sensor devices) are highly volatile and should be recorded with fine granularities (e.g., microseconds or milliseconds). These time-evolving values can be regarded as properties on the nodes of the graph of Thing'in. In this context, time series compression techniques are a sensible choice to reduce the space usage of time-evolving properties.

Time series compression has received attention in the literature, and several methods have been proposed. For example, Dictionary-based compression builds upon the redundancy peculiarity of time series. First, common segments are extracted through a training phase to create a dictionary of segments. Second, each segment in the time series is replaced with the key of the segment in the dictionary. However, if a time series segment does not match any dictionary segment, then the segment is left uncompressed. Examples of such methods can be found in [157, 137]. In functional-based compression methods, time series are represented as a function of time. Instead of finding a single function that represents the time series, these methods partition the time series into segments. For each of these segments, a function is assigned. An example of such methods can be found in [171].

Although these compression methods are interesting, they work better in offline systems since they necessitate a processing phase to extract the dictionary or functions. However, in the application domains of Thing'in, data should be inserted in real-time,

meaning that the insertion latency should be controllable, which implies using other online compression techniques.

8.2.3 Directions in processing temporal queries

Temporal joins

The query processor proposed in Chapter 6 allows the evaluation of T-Cypher queries. We proposed a time-extended temporal graph algebra that includes the operators getNodes and expand, retrieving the nodes in a graph or expanding an input relation with relationships. Besides these graph-oriented operators, we use relational operators: selection and join. The selection filters an input temporal graph relation, whereas the join operator is used to join two temporal graph relations based on a condition that applies to the variables of the relations. As outlined in the chapter, we refer to a join between two relations based on a temporal condition as a temporal join. In Figure 8.2, we present an example of a temporal graph pattern and its corresponding evaluation plan, including a temporal join. In this example the sub-patterns are evaluated separately, and then their results are joined. This join is based on the equality $n_0 = n'_0$ and $n_2 = n'_2$ which we omit from the operator for simplicity. The second join condition matches the tuples that satisfy Allen's relation stating that r_3 must have occurred before r_1. In this work, we proposed to perform temporal joins with a straightforward algorithm assuming no indices, implying scanning through both temporal graph relations, selecting the subsets matching the temporal condition, and then joining them. Assuming that the sizes of the relations are large with high joining selectivity, this joining technique will be costly. Having this, we propose to extend our query processor with a special technique that accelerates temporal joins. Such a technique can rely on an index structure optimized for range-interval queries (i.e., number of elements with a key range and a time interval) such as MVBT (Multi-Version B+Tree) [240] which can avoid scanning the entire relations.

Indexing

Data management systems often refer to indexes to prune the search space. In the context of graph management, auxiliary data structures are used to index paths [218], frequent subgraphs [239], trees [243] and neighborhood of a node [230]. Such indexes

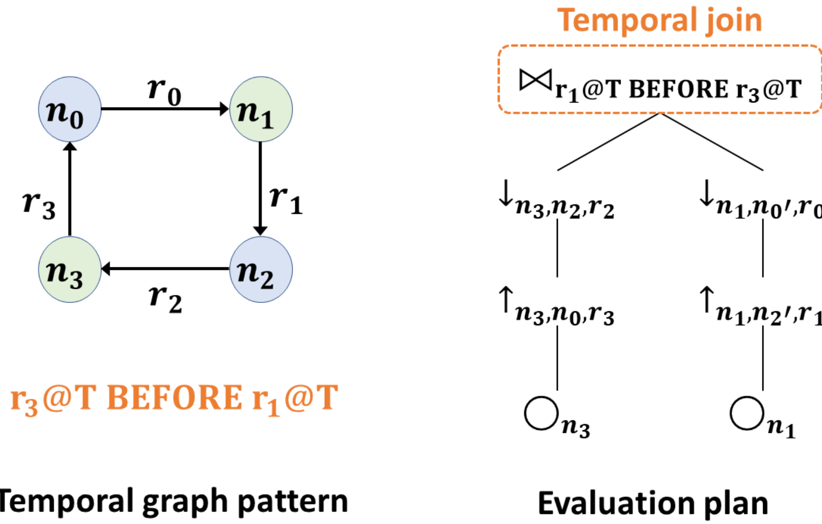


Figure 8.2 – Example of temporal graph pattern and its evaluation plan including a temporal join

filter out graphs that do not match the query. Regarding their usefulness in accelerating query processing, it would be interesting to include indexes into our system Clock-G to accelerate T-Cypher queries.

8.2.4 Directions in generating temporal graphs

We also propose improvements for our graph generator RTGEN. As outlined in Chapter 7, our generator produces temporal graphs while controlling the evolution of the degree distribution while maintaining a static community structure. Although nodes and relationships are joining and leaving the returned graphs, the between and within relationship densities of communities remain unchanged. This feature does not reflect real-world graphs since communities can be subject to a merge, split, shrink, or growth which affects the overall community structure [28]. To capture this, we propose to extend RTGEN with community-aware functionalities in which communities can change based on given evolutionary parameters that characterize these changes. However, the challenge we are aware of within this extension is controlling the evolution of the degree distribution since the desired changes in the community structure should not affect the expected evolution of the degree distribution.

8.3 Summary of contributions and directions in the industrial integration

8.3.1 Summary of contributions in the industrial integration

We recall that this thesis was motivated by the industrial use case of the Thing'in platform initiated by the telecommunication company Orange. Hence, some of the contributions of this dissertation, particularly T-Cypher, are already used in production.

This dissertation focused on the development of Clock-G. However, we provided a proof-of-concept implementation of our system, which is in an early development stage. Building a robust data management system takes years of progress and demands the efforts of a team of full-time engineers. Hence, we decided to rely on an existing graph database to accelerate the integration into the Thing'in platform and continue in parallel the development of Clock-G with the ultimate goal of eventually using it in production.

For instance, we added temporal support to the Thing'in platform [2] by relying on an existing graph database Neo4j [175] and translating T-Cypher queries into Cypher queries as presented in Chapter 4 (Section 4.6). The documentation of our public beta release of the temporal feature can be found on the official website of the platform [3].

We also added special techniques [4] to enable the users of Thing'in to visualize the evolution of the result. We also proposed two different formats to return the result of a T-Cypher query: Full and Log. The Full format returns node and relationship states. Whereas the Log format returns a sequence of graph updates. The latter format is particularly useful for visualizing the evolution of the returned result.

8.3.2 Directions in the industrial integration

As mentioned in the previous section, we have integrated the temporal dimension into the Thing'in platform. This feature is a public beta release we plan to develop progressively soon. For instance, the supported temporal queries cover a fragment of T-Cypher queries. Hence, we plan to include all the possible temporal features of T-

2. Thanks goes to David Crosson, a senior engineer at Orange Labs and member of the Thing'in team, who has helped me integrate my work into the Thing'in plateform

3. https://wiki.thinginthefuture.com/public/Historization

4. Thanks goes to Alain Dechorgnat, a senior developer at Orange Labs and member of the Thing'in team, for his efforts in implementing the visualization feature

Cypher, such as temporal paths and aggregation. To reduce query latency, a possible amelioration is the addition of indexes on special static properties of the nodes of the graph, which are commonly filtered by the platform users. Besides, we plan to benchmark the platform's performance to evaluate our implementation regarding ingestion throughput, space usage, and query response times. In this context, we plan to use RTGEN to generate the datasets of our benchmarks. Since we have recently rendered the temporal feature public, we are receiving feedback from end users, which will further help us improve the solution and propose new temporal functionalities.

Thing'in is currently managing digital twins, the digital representation of real-world physical devices that serve as their indistinguishable virtual counterparts. In this context, real-time management of the data collected by these devices is very critical. Note that real-time management in the use-case of digital twins implies controlling the latency of data collection, insertion, and querying, usually in milliseconds or even microseconds. This delicate demand should be further investigated in the future development of our temporal layer.

Bibliography

[1] Marko A. Rodriguez, *Gremlin's Time Machine*, 2016, URL: https://www.datastax.com/dev/blog/gremlins-time-machine.

[2] Emmanuel Abbe, « Community detection and stochastic block models: recent developments », *in*: *The Journal of Machine Learning Research* 18.*1* (2017), pp. 6446–6531.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of Databases*, Addison-Wesley, 1995, ISBN: 0-201-53771-0, URL: http://webdam.inria.fr/Alice/.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of Databases*, Addison-Wesley, 1995, ISBN: 0-201-53771-0.

[5] Amir Aghasadeghi, Vera Z. Moffitt, Sebastian Schelter, and Julia Stoyanovich, « Zooming Out on an Evolving Graph », *in*: *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT)* (2020), URL: https://par.nsf.gov/biblio/10137087.

[6] Amir Aghasadeghi and Julia Stoyanovich, « Generating evolving property graphs with attribute-aware preferential attachment », *in*: *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.

[7] Amir Pouya Aghasadeghi, « Querying Temporal Property Graphs », PhD thesis, New York University - Tandon School Of Engineering, 2022.

[8] William Aiello, Fan Chung, and Linyuan Lu, « A random graph model for power law graphs », *in*: *Experimental Mathematics* 10.*1* (2001), pp. 53–66.

[9] Leman Akoglu and Christos Faloutsos, « RTG: a recursive realistic graph generator using random typing », *in*: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2009, pp. 13–28.

[10] Karim Alami, Radu Ciucanu, and Engelbert Mephu Nguifo, « Synthetic Graph Generation from Finely-Tuned Temporal Constraints. », *in*: *TD-LSG@ PKDD/ECML*, 2017, pp. 44–47.

[11] James F Allen, « Maintaining knowledge about temporal intervals », *in*: *Communications of the ACM* 26.*11* (1983), pp. 832–843.

[12] *Amazon Neptune est un service de base de données orientée graphe entièrement géré | Amazon Neptune | Amazon Web Services*, fr-FR, URL: `https://aws.amazon.com/fr/neptune/` (visited on 05/07/2022).

[13] Renzo Angles, « A Comparison of Current Graph Database Models », *in*: *2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012, pp. 171–177, DOI: `10.1109/ICDEW.2012.31`.

[14] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al., « G-CORE: A core for future graph query languages », *in*: *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1421–1432.

[15] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč, « Foundations of Modern Query Languages for Graph Databases », *in*: *ACM Comput. Surv.* 50.*5* (2017), ISSN: 0360-0300, DOI: `10.1145/3104031`, URL: `https://doi.org/10.1145/3104031`.

[16] Marcelo Arenas, Pedro Bahamondes, Amir Aghasadeghi, and Julia Stoyanovich, « Temporal Regular Path Queries », *in*: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2412–2425, DOI: `10.1109/ICDE53745.2022.00226`.

[17] Eugene Asarin, Paul Caspi, and Oded Maler, « Timed regular expressions », *in*: *Journal of the ACM* 49.*2* (2002), pp. 172–206.

[18] *Azure Cosmos DB - Base de données NoSQL | Microsoft Azure*, fr, URL: `https://azure.microsoft.com/fr-fr/services/cosmos-db/` (visited on 06/02/2022).

[19] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat, « gMark: Schema-driven generation of graphs and queries », *in*: *IEEE Transactions on Knowledge and Data Engineering* 29.*4* (2016), pp. 856–869.

[20] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat, « gMark: Schema-driven generation of graphs and queries », *in*: *IEEE Transactions on Knowledge and Data Engineering* 29.*4* (2016), pp. 856–869.

[21] Albert-László Barabási and Réka Albert, « Emergence of scaling in random networks », *in*: *science* 286.*5439* (1999), pp. 509–512.

[22] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood, « Expressive Languages for Path Queries over Graph-Structured Data », *in*: *ACM Trans. Database Syst.* 37.*4* (2012), ISSN: 0362-5915, DOI: 10.1145/2389241.2389250, URL: https://doi.org/10.1145/2389241.2389250.

[23] Pablo Barceló, Leonid Libkin, and Juan L. Reutter, « Querying Graph Patterns », *in*: *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, Athens, Greece: Association for Computing Machinery, 2011, 199–210, ISBN: 9781450306607, DOI: 10.1145/1989284.1989307, URL: https://doi.org/10.1145/1989284.1989307.

[24] Pablo Barceló Baeza, « Querying Graph Databases », *in*: PODS '13, New York, New York, USA: Association for Computing Machinery, 2013, 175–188, ISBN: 9781450320665, DOI: 10.1145/2463664.2465216, URL: https://doi.org/10.1145/2463664.2465216.

[25] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi, « Semantic Acyclicity on Graph Databases », *in*: *SIAM Journal on Computing* 45.*4* (2016), pp. 1339–1376, DOI: 10.1137/15M1034714, eprint: https://doi.org/10.1137/15M1034714, URL: https://doi.org/10.1137/15M1034714.

[26] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy, « Gephi: an open source software for exploring and manipulating networks », *in*: *Third international AAAI conference on weblogs and social media*, 2009.

[27] Richard Bellman, « On a routing problem », *in*: *Quarterly of applied mathematics* 16.*1* (1958), pp. 87–90.

[28] Oualid Benyahia, Christine Largeron, Baptiste Jeudy, and Osmar R Zaïane, « Dancer: Dynamic attributed network with community structure generator », *in*: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2016, pp. 41–44.

[29] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, « Computational Geometry », *in*: *Computational Geometry: Algorithms and Applications*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–17, ISBN: 978-3-662-03427-9, DOI: 10.1007/978-3-662-03427-9_1, URL: https://doi.org/10.1007/978-3-662-03427-9_1.

[30] *Blazegraph Database*, URL: https://blazegraph.com/ (visited on 05/07/2022).

[31] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre, « Fast unfolding of communities in large networks », *in*: *Journal of statistical mechanics: theory and experiment* 2008.*10* (2008), P10008.

[32] Daniel Blum and Sara Cohen, « Grr: Generating Random RDF », *in*: *The Semanic Web: Research and Applications*, ed. by Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Pan, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 16–30.

[33] Stefano Boccaletti, Ginestra Bianconi, Regino Criado, Charo I Del Genio, Jesús Gómez-Gardenes, Miguel Romance, Irene Sendina-Nadal, Zhen Wang, and Massimiliano Zanin, « The structure and dynamics of multilayer networks », *in*: *Physics Reports* 544.*1* (2014), pp. 1–122.

[34] M.H. Bohlen, R. Busatto, and C.S. Jensen, « Point-versus interval-based temporal data models », *in*: *Proceedings 14th International Conference on Data Engineering*, 1998, pp. 192–200, DOI: 10.1109/ICDE.1998.655777.

[35] Michael H. Böhlen, Christian S. Jensen, and Richard Thomas Snodgrass, « Temporal Statement Modifiers », *in*: *ACM Trans. Database Syst.* 25.*4* (2000), 407–456, ISSN: 0362-5915, DOI: 10.1145/377674.377665, URL: https://doi.org/10.1145/377674.377665.

[36] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets, « Querying graphs », *in*: *Synthesis Lectures on Data Management* 10.*3* (2018), pp. 1–184.

[37] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr, « Graph Generators: State of the Art and Open Challenges », *in*: *ACM Comput. Surv.* 53.*2* (2021), 36:1–36:30, DOI: 10.1145/3379445, URL: https://doi.org/10.1145/3379445.

[38] Karsten M. Borgwardt, Hans-peter Kriegel, and Peter Wackersreuther, « Pattern Mining in Frequent Dynamic Subgraphs », *in*: *Sixth International Conference on Data Mining (ICDM'06)*, 2006, pp. 818–822, DOI: `10.1109/ICDM.2006.124`.

[39] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner, « Experiments on graph clustering algorithms », *in*: *European Symposium on Algorithms*, Springer, 2003, pp. 568–579.

[40] Horst Bunke, « Graph matching: Theoretical foundations, algorithms, and applications », *in*: *Proc. Vision Interface*, vol. 2000, 2000, pp. 82–88.

[41] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader, « Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs », *in*: *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7, DOI: `10.1109/HPEC.2018.8547541`.

[42] Jaewook Byun, Sungpil Woo, and Daeyoung Kim, « ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time », *in*: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[43] Michael Böhlen, Christian Jensen, and Richard Snodgrass, « Evaluating the Completeness of TSQL2 », *in*: Jan. 1995, pp. 153–172, DOI: `10.1007/978-1-4471-3033-8_9`.

[44] Hongming Cai, Boyi Xu, Lihong Jiang, and Athanasios V. Vasilakos, « IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges », *in*: *IEEE Internet of Things Journal* 4.*1* (2017), pp. 75–87, DOI: `10.1109/JIOT.2016.2619369`.

[45] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi, « Rewriting of regular expressions and regular path queries », *in*: *Journal of Computer and System Sciences* 64.*3* (2002), pp. 443–465.

[46] Arnaud Castelltort and Anne Laurent, « Representing history in graph-oriented NoSQL databases: A versioning system », *in*: *Eighth International Conference on Digital Information Management (ICDIM 2013)*, 2013, pp. 228–234, DOI: `10.1109/ICDIM.2013.6694022`.

[47] Jaime Castro and Adrián Soto, « A Comparison between Cypher and Conjunctive Queries. », *in*: *AMW*, 2017.

[48] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch, « Time-Varying Social Networks in a Graph Database: A Neo4j Use Case », *in*: *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, New York: Association for Computing Machinery, 2013, ISBN: 9781450321884, DOI: 10.1145/2484425.2484442, URL: https://doi.org/10.1145/2484425.2484442.

[49] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos, « R-MAT: A recursive model for graph mining », *in*: *Proceedings of the 2004 SIAM International Conference on Data Mining*, SIAM, 2004, pp. 442–446.

[50] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos, « R-MAT: A recursive model for graph mining », *in*: *Proceedings of the 2004 SIAM International Conference on Data Mining*, SIAM, 2004, pp. 442–446.

[51] Hassan Nazeer Chaudhry, « FlowGraph: Distributed Temporal Pattern Detection over Dynamically Evolving Graphs », *in*: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, DEBS '19, Darmstadt, Germany: Association for Computing Machinery, 2019, 272–275, ISBN: 9781450367943, DOI: 10.1145/3328905.3332303, URL: https://doi.org/10.1145/3328905.3332303.

[52] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik, « Shortest paths algorithms: Theory and experimental evaluation », *in*: *Mathematical Programming* 73.*2* (May 1996), pp. 129–174, ISSN: 1436-4646, DOI: 10.1007/BF02592101, URL: https://doi.org/10.1007/BF02592101.

[53] Jan Chomicki, « Temporal query languages: A survey », *in*: *Temporal Logic*, ed. by Dov M. Gabbay and Hans Jürgen Ohlbach, Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 506–534, ISBN: 978-3-540-48585-8.

[54] Fan Chung, Fan RK Chung, Fan Chung Graham, Linyuan Lu, Kian Fan Chung, et al., *Complex graphs and networks*, 107, American Mathematical Soc., 2006.

[55] Fan Chung and Linyuan Lu, « Connected components in random graphs with given expected degree sequences », *in*: *Annals of combinatorics* 6.*2* (2002), pp. 125–145.

[56] Fan Chung and Linyuan Lu, « The average distances in random graphs with given expected degrees », *in*: *Proceedings of the National Academy of Sciences* 99.*25* (2002), pp. 15879–15882.

[57] Aaron Clauset, M. E. J. Newman, and Cristopher Moore, « Finding community structure in very large networks », *in*: *Phys. Rev. E* 70 (6 2004), p. 066111, DOI: 10.1103/PhysRevE.70.066111, URL: https://link.aps.org/doi/10.1103/PhysRevE.70.066111.

[58] James Clifford and Abdullah Uz Tansel, « On an Algebra for Historical Relational Databases: Two Views », *in*: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, Austin, Texas, USA: Association for Computing Machinery, 1985, 247–265, ISBN: 0897911601, DOI: 10.1145/318898.318922, URL: https://doi.org/10.1145/318898.318922.

[59] James Clifford and Abdullah Uz Tansel, « On an Algebra for Historical Relational Databases: Two Views », *in*: *SIGMOD Rec.* 14.*4* (1985), 247–265, ISSN: 0163-5808, DOI: 10.1145/971699.318922, URL: https://doi.org/10.1145/971699.318922.

[60] *CloudGraph*, URL: https://www.cloudgraph.dev/ (visited on 06/05/2022).

[61] James R Clough and Tim S Evans, « Time and citation networks », *in*: *arXiv preprint arXiv:1507.01388* (2015).

[62] Mariano P Consens and Alberto O Mendelzon, « GraphLog: a visual formalism for real life recursion », *in*: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1990, pp. 404–416.

[63] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood, « A graphical query language supporting recursion », *in*: *ACM SIGMOD Record* 16.*3* (1987), pp. 323–330.

[64] Soumyava Das, Ankur Goyal, and Sharma Chakravarthy, « Plan Before You Execute: A Cost-Based Query Optimizer for Attributed Graph Databases », *in*: *Big Data Analytics and Knowledge Discovery*, ed. by Sanjay Madria and Takahiro Hara, Cham: Springer International Publishing, 2016, pp. 314–328, ISBN: 978-3-319-43946-4.

[65] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman, « A model and query language for temporal graph databases », *in*: *The VLDB Journal* 30.*5* (Sept. 2021), pp. 825–858, ISSN: 0949-877X, DOI: 10.1007/s00778-021-00675-4, URL: https://doi.org/10.1007/s00778-021-00675-4.

[66] Daniel Delling, Marco Gaertler, and Dorothea Wagner, « Generating significant graph clusterings », *in*: *Proceedings of the European Conference of Complex Systems ECCS*, vol. 6, 2006.

[67] Eric V Denardo and Bennett L Fox, « Shortest-route methods: 1. reaching, pruning, and buckets », *in*: *Operations Research* 27.*1* (1979), pp. 161–186.

[68] *Dgraph | GraphQL Cloud Platform – GraphQL . Javascript . Distributed Graph Engine | Deploy a Production Ready GraphQL Backend in Minutes*, URL: `https://dgraph.io/` (visited on 06/05/2022).

[69] Anton Dignös, Michael H. Böhlen, and Johann Gamper, « Temporal Alignment », *in*: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, 433–444, ISBN: 9781450312479, DOI: `10.1145/2213836.2213886`, URL: `https://doi.org/10.1145/2213836.2213886`.

[70] Zhiming Ding and Ralf Hartmut Güting, « Modeling temporally variable transportation networks », *in*: *International Conference on Database Systems for Advanced Applications*, Springer, 2004, pp. 154–168.

[71] Binyao Duan, Wenjian Luo, Hao Jiang, and Li Ni, « Dynamic Social Networks Generator Based on Modularity: DSNG-M », *in*: *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, IEEE, 2019, pp. 167–173.

[72] Sergey Edunov, Dionysios Logothetis, Cheng Wang, Avery Ching, and Maja Kabiljo, « Darwini: Generating realistic large-scale social graphs », *in*: *arXiv preprint arXiv:1610.00664* (2016).

[73] *Enterprise Distributed Graph Database | DataStax*, URL: `https://www.datastax.com/products/datastax-graph` (visited on 05/04/2022).

[74] Paul Erdös and Alfréd Rényi, « On random graphs », *in*: *Publicationes mathematicae* 6.*26* (1959), pp. 290–297.

[75] Paul Erdős and Alfréd Rényi, « On the evolution of random graphs », *in*: *Publ. Math. Inst. Hung. Acad. Sci* 5.*1* (1960), pp. 17–60.

[76] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz, « The LDBC Social Network Benchmark: Interactive Workload », *in*: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 619–630, DOI: 10.1145/2723372.2742786, URL: https://doi.org/10.1145/2723372.2742786.

[77] Lingzhong Fan, Hai Li, Junjie Zhuo, Yu Zhang, Jiaojian Wang, Liangfu Chen, Zhengyi Yang, Congying Chu, Sangma Xie, Angela R. Laird, Peter T. Fox, Simon B. Eickhoff, Chunshui Yu, and Tianzi Jiang, « The Human Brainnetome Atlas: A New Brain Atlas Based on Connectional Architecture », *in*: *Cerebral Cortex* 26.*8* (July 2016), pp. 3508–3526, ISSN: 1047-3211, DOI: 10.1093/cercor/bhw157, eprint: https://academic.oup.com/cercor/article-pdf/26/8/3508/17333354/bhw157.pdf, URL: https://doi.org/10.1093/cercor/bhw157.

[78] Wenfei Fan, « Graph Pattern Matching Revised for Social Network Analysis », *in*: *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, Berlin, Germany: Association for Computing Machinery, 2012, 8–21, ISBN: 9781450307918, DOI: 10.1145/2274576.2274578, URL: https://doi.org/10.1145/2274576.2274578.

[79] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu, « Graph Homomorphism Revisited for Graph Matching », *in*: *Proc. VLDB Endow.* 3.*1–2* (2010), 1161–1172, ISSN: 2150-8097, DOI: 10.14778/1920841.1920986, URL: https://doi.org/10.14778/1920841.1920986.

[80] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A. Miller, « Towards efficient query processing on massive time-evolving graphs », *in*: *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2012, pp. 567–574, DOI: 10.4108/icst.collaboratecom.2012.250532.

[81] Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu, « Declarative specification of Web sites with Strudel », *in*: *The VLDB Journal* 9.*1* (2000), pp. 38–55, ISSN: 0949-877X, DOI: 10.1007/s007780050082, URL: https://doi.org/10.1007/s007780050082.

[82] Rémi Flamary and Nicolas Courty, *POT Python Optimal Transport library*, 2017, URL: https://github.com/rflamary/POT.

[83] Rémi Flamary, Nicolas Courty, Alexandre Gramfort, Mokhtar Z. Alaya, Aurélie Boisbunon, Stanislas Chambon, Laetitia Chapel, Adrien Corenflos, Kilian Fatras, Nemo Fournier, Léo Gautheron, Nathalie T.H. Gayraud, Hicham Janati, Alain Rakotomamonjy, Ievgen Redko, Antoine Rolet, Antony Schutz, Vivien Seguy, Danica J. Sutherland, Romain Tavenard, Alexander Tong, and Titouan Vayer, « POT: Python Optimal Transport », *in*: *Journal of Machine Learning Research 22.78* (2021), pp. 1–8, URL: `http://jmlr.org/papers/v22/20-451.html`.

[84] Santo Fortunato and Darko Hric, « Community detection in networks: A user guide », *in*: *Physics Reports* 659 (2016), Community detection in networks: A user guide, pp. 1–44, ISSN: 0370-1573, DOI: `https://doi.org/10.1016/j.physrep.2016.09.002`, URL: `https://www.sciencedirect.com/science/article/pii/S0370157316302964`.

[85] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor, « Formal Semantics of the Language Cypher », *in*: *CoRR* abs/1802.09984 (2018), arXiv: `1802.09984`, URL: `http://arxiv.org/abs/1802.09984`.

[86] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor, « Cypher: An evolving query language for property graphs », *in*: *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1433–1445.

[87] Marco Gaertler, Robert Görke, and Dorothea Wagner, « Significance-driven graph clustering », *in*: *International Conference on Algorithmic Applications in Management*, Springer, 2007, pp. 11–26.

[88] Libo Gao, Lukasz Golab, M. Tamer Özsu, and Güneş Aluç, « Stream WatDiv: A Streaming RDF Benchmark », *in*: *Proceedings of the International Workshop on Semantic Big Data*, SBD'18, Houston, TX, USA: Association for Computing Machinery, 2018, ISBN: 9781450357791, DOI: `10.1145/3208352.3208355`, URL: `https://doi.org/10.1145/3208352.3208355`.

[89] Shi Gao, Jiaqi Gu, and Carlo Zaniolo, « RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases », *in*: *EDBT*, 2016.

[90] Antonio García-Domínguez, Nelly Bencomo, Juan Marcelo Parra-Ullauri, and Luis Hernán García-Paucar, « Querying and annotating model histories with time-aware patterns », *in*: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2019, pp. 194–204.

[91] Michael R. Garey and David S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, USA: W. H. Freeman Co., 1990, ISBN: 0716710455.

[92] Betsy George, James M. Kang, and Shashi Shekhar, « Spatio-Temporal Sensor Graphs (STSG): A data model for the discovery of spatio-temporal patterns », *in*: *Intelligent Data Analysis* 13.*3* (2009), Publisher: IOS Press, pp. 457–475, ISSN: 1571-4128, DOI: 10.3233/IDA-2009-0376.

[93] Betsy George, James M Kang, and Shashi Shekhar, « Spatio-temporal sensor graphs (stsg): A data model for the discovery of spatio-temporal patterns », *in*: *Intelligent Data Analysis* 13.*3* (2009), pp. 457–475.

[94] Betsy George and Shashi Shekhar, « Time-aggregated graphs for modeling spatio-temporal networks », *in*: *Journal on Data Semantics XI*, Springer, 2008, pp. 191–212.

[95] Betsy George and Shashi Shekhar, « Time-Aggregated Graphs for Modeling Spatio-temporal Networks », *in*: *Journal on Data Semantics XI*, ed. by Stefano Spaccapietra, Jeff Z. Pan, Philippe Thiran, Terry Halpin, Steffen Staab, Vojtech Svatek, Pavel Shvaiko, and John Roddick, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 191–212, ISBN: 978-3-540-92148-6, DOI: 10.1007/978-3-540-92148-6_7, URL: https://doi.org/10.1007/978-3-540-92148-6_7.

[96] Amine Ghrab, Oscar Romero, Sabri Skhiri, Alejandro A. Vaisman, and Esteban Zimányi, « GRAD: On Graph Database Modeling », *in*: *CoRR* abs/1602.00503 (2016), arXiv: 1602.00503, URL: http://arxiv.org/abs/1602.00503.

[97] Michelle Girvan and Mark EJ Newman, « Community structure in social and biological networks », *in*: *Proceedings of the national academy of sciences* 99.*12* (2002), pp. 7821–7826.

[98] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, « PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs », *in*: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA: USENIX Association, Oct. 2012, pp. 17–30, ISBN: 978-

1-931971-96-6, URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez`.

[99] Fabio Grandi, « Multi-temporal RDF Ontology Versioning. », *in*: *IWOD@ ISWC*, 2009.

[100] Fabio Grandi, « T-SPARQL: A TSQL2-like Temporal Query Language for RDF. », *in*: *ADBIS (local proceedings)*, Citeseer, 2010, pp. 21–30.

[101] Clara Granell, Richard K Darst, Alex Arenas, Santo Fortunato, and Sergio Gómez, « Benchmark model to assess community structure in evolving networks », *in*: *Physical Review E* 92.*1* (2015), p. 012805.

[102] *Graph Analytics Platform | Graph Database | TigerGraph*, URL: `https://www.tigergraph.com/` (visited on 05/04/2022).

[103] *Graph Query Language GQL - Papers and Slides*, URL: `https://www.gqlstandards.org/resources/papers-and-slides` (visited on 09/10/2021).

[104] Andrey Gubichev, « Query Processing and Optimization in Graph Databases », PhD thesis, Technische Universität München, 2015.

[105] Carl A Gunter, *Semantics of programming languages: structures and techniques*, MIT press, 1992.

[106] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin, « LUBM: A benchmark for OWL knowledge base systems », *in*: *Journal of Web Semantics* 3.*2* (2005), Selcted Papers from the International Semantic Web Conference, 2004, pp. 158–182, ISSN: 1570-8268, DOI: `https://doi.org/10.1016/j.websem.2005.06.005`, URL: `https://www.sciencedirect.com/science/article/pii/S1570826805000132`.

[107] Furkan Gursoy and Bertan Badur, « A Community-aware Network Growth Model for Synthetic Social Network Generation », *in*: *arXiv preprint arXiv:1901.03629* (2019).

[108] Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras, « On-Line Discovery of Dense Areas in Spatio-temporal Databases », *in*: *Advances in Spatial and Temporal Databases*, ed. by Thanasis Hadzilacos, Yannis Manolopoulos, John Roddick, and Yannis Theodoridis, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 306–324, ISBN: 978-3-540-45072-6.

[109] Martin Haeusler, « Scalable Versioning for Key-Value Stores. », *in*: *DATA*, 2016, pp. 79–86.

[110] Martin Haeusler, Thomas Trojer, Johannes Kessler, Matthias Farwick, Emmanuel Nowakowski, and Ruth Breu, « ChronoGraph: A Versioned TinkerPop Graph Database », *in*: *International Conference on Data Management Technologies and Applications*, Springer, 2017, pp. 237–260.

[111] Wentao Han, Kaiwei Li, Shimin Chen, and Wenguang Chen, « Auxo: a temporal graph management system », *in*: *Big Data Mining and Analytics* 2.*1* (2019), pp. 58–71, DOI: `10.26599/BDMA.2018.9020030`.

[112] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen, « Chronos: A Graph Engine for Temporal Graph Analysis », *in*: *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands: Association for Computing Machinery, 2014, ISBN: 9781450327046, DOI: `10.1145/2592798.2592799`, URL: `https://doi.org/10.1145/2592798.2592799`.

[113] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux, « SPARQL 1.1 query language », *in*: *W3C recommendation* 21.*10* (2013), p. 778.

[114] Olaf Hartig, « RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF. », *in*: *ISWC (Posters, Demos & Industry Tracks)*, 2017.

[115] Olaf Hartig, « Reconciliation of RDF* and Property Graphs », *in*: *CoRR* abs/1409.3288 (2014), arXiv: `1409.3288`, URL: `http://arxiv.org/abs/1409.3288`.

[116] Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon, « Analyzing complex data in motion at scale with temporal graphs », *in*: (2020).

[117] Huahai He and Ambuj K. Singh, « Graphs-at-a-Time: Query Language and Access Methods for Graph Databases », *in*: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, 405–418, ISBN: 9781605581026, DOI: `10.1145/1376616.1376660`, URL: `https://doi.org/10.1145/1376616.1376660`.

[118] Huahai He and Ambuj K. Singh, « Graphs-at-a-Time: Query Language and Access Methods for Graph Databases », *in*: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, 405–418, ISBN: 9781605581026, DOI: `10.1145/1376616.1376660`, URL: `https://doi.org/10.1145/1376616.1376660`.

[119] Huahai He and Ambuj K. Singh, « Query Language and Access Methods for Graph Databases », *in*: *Managing and Mining Graph Data*, ed. by Charu C. Aggarwal and Haixun Wang, Boston, MA: Springer US, 2010, pp. 125–160, ISBN: 978-1-4419-6045-0, DOI: 10.1007/978-1-4419-6045-0_4, URL: https://doi.org/10.1007/978-1-4419-6045-0_4.

[120] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt, « Stochastic blockmodels: First steps », *in*: *Social networks* 5.*2* (1983), pp. 109–137.

[121] Petter Holme, « Modern temporal network theory: a colloquium », *in*: *The European Physical Journal B* 88.*9* (2015), p. 234.

[122] Petter Holme and Jari Saramäki, « Temporal networks », *in*: *Physics reports* 519.*3* (2012), pp. 97–125.

[123] *Home | OrientDB Community Edition*, en, URL: https://orientdb.org/ (visited on 05/04/2022).

[124] Haixing Huang, Jinghe Song, Xuelian Lin, Shuai Ma, and Jinpeng Huai, « TGraph: A Temporal Graph Data Management System », *in*: CIKM '16, Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, 2469–2472, ISBN: 9781450340731, DOI: 10.1145/2983323.2983335, URL: https://doi.org/10.1145/2983323.2983335.

[125] Wenyu Huo and Vassilis J. Tsotras, « Efficient Temporal Shortest Path Queries on Evolving Social Graphs », *in*: *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, Aalborg, Denmark: Association for Computing Machinery, 2014, ISBN: 9781450327220, DOI: 10.1145/2618243.2618282, URL: https://doi.org/10.1145/2618243.2618282.

[126] Jürgen Hölsch and Michael Grossniklaus, « An Algebra and Equivalences to Transform Graph Patterns in Neo4j », *in*: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016)*, ed. by Themis Palpanas and Kostas Stefanidis, CEUR Workshop Proceedings 1558, 2016, URL: http://ceur-ws.org/Vol-1558/paper24.pdf.

[127] *JanusGraph*, URL: https://janusgraph.org/ (visited on 11/17/2022).

[128] Tony Jebara, Jun Wang, and Shih-Fu Chang, « Graph Construction and b-Matching for Semi-Supervised Learning », *in*: ICML '09, Montreal, Quebec, Canada: Association for Computing Machinery, 2009, 441–448, ISBN: 9781605585161, DOI: `10.1145/1553374.1553432`, URL: `https://doi.org/10.1145/1553374.1553432`.

[129] Christian S. Jensen, James Clifford, Ramez Elmasri, Shashi K. Gadia, Pat Hayes, Sushil Jajodia, Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard Thomas Snodgrass, Mike D. Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold, « A Consensus Glossary of Temporal Database Concepts », English (US), *in*: *SIGMOD Record* 23.*1* (Jan. 1994), pp. 52–64, ISSN: 0163-5808, DOI: `10.1145/181550.181560`.

[130] Theodore Johnson, Vladislav Shkapenyuk, Pramod A Jamkhedkar, and Yaron Kanza, *Time-based querying of graph databases*, US Patent 10,685,063, 2020.

[131] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm, « Cypher-Based Graph Pattern Matching in Gradoop », *in*: GRADES'17, Chicago, IL, USA: Association for Computing Machinery, 2017, ISBN: 9781450350389, DOI: `10.1145/3078447.3078450`, URL: `https://doi.org/10.1145/3078447.3078450`.

[132] Bogumił Kamiński, Paweł Prałat, and François Théberge, « Artificial Benchmark for Community Detection (ABCD): Fast Random Graph Model with Community Structure », *in*: *arXiv preprint arXiv:2002.00843* (2020).

[133] Andrey Kan, Jeffrey Chan, James Bailey, and Christopher Leckie, « A query based approach for mining evolving graphs », *in*: *Proceedings of the Eighth Australasian Data Mining Conference-Volume 101*, Citeseer, 2009, pp. 139–150.

[134] Brian Karrer and Mark EJ Newman, « Stochastic blockmodels and community structure in networks », *in*: *Physical review E* 83.*1* (2011), p. 016107.

[135] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl, « RQL: A Declarative Query Language for RDF », *in*: *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, Honolulu, Hawaii, USA: Association for Computing Machinery, 2002, 592–603, ISBN: 1581134495, DOI: `10.1145/511446.511524`, URL: `https://doi.org/10.1145/511446.511524`.

[136] David Kempe, Jon Kleinberg, and Amit Kumar, « Connectivity and Inference Problems for Temporal Networks », *in*: *Journal of Computer and System Sciences* 64.*4* (2002), pp. 820–842, ISSN: 0022-0000, DOI: https://doi.org/10.1006/jcss.2002.1829, URL: https://www.sciencedirect.com/science/article/pii/S0022000002918295.

[137] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudré-Mauroux, « CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding », *in*: *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 2289–2298, DOI: 10.1109/BigData47090.2019.9005580.

[138] Udayan Khurana and Amol Deshpande, « Efficient snapshot retrieval over historical graph data », *in*: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 997–1008, DOI: 10.1109/ICDE.2013.6544892.

[139] Udayan Khurana and Amol Deshpande, « Storing and analyzing historical graph data at scale », *in*: *arXiv preprint arXiv:1509.08960* (2015).

[140] Tamara G Kolda, Ali Pinar, Todd Plantenga, and Comandur Seshadhri, « A scalable generative graph model with community structure », *in*: *SIAM Journal on Scientific Computing* 36.*5* (2014), pp. C424–C452.

[141] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura, « On Graph Deltas for Historical Queries », *in*: *CoRR* abs/1302.5549 (2013), arXiv: 1302.5549, URL: http://arxiv.org/abs/1302.5549.

[142] Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura, « HiNode: an asymptotically space-optimal storage model for historical queries on graphs », *in*: *Distributed and Parallel Databases* 35.*3* (Dec. 2017), pp. 249–285, ISSN: 1573-7578, DOI: 10.1007/s10619-017-7207-z, URL: https://doi.org/10.1007/s10619-017-7207-z.

[143] Jérôme Kunegis, « KONECT: The Koblenz Network Collection », *in*: WWW '13 Companion, Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, 1343–1350, ISBN: 9781450320382, DOI: 10.1145/2487788.2488173, URL: https://doi.org/10.1145/2487788.2488173.

[144] *Kyoto Cabinet: a straightforward implementation of DBM*, URL: https://dbmx.net//kyotocabinet/ (visited on 05/16/2021).

[145] Alan G Labouseur, Jeremy Birnbaum, Paul W Olsen, Sean R Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han, « The G* graph database: efficiently managing large distributed dynamic graphs », *in*: *Distributed and Parallel Databases* 33.*4* (2015), pp. 479–514.

[146] Mayank Lahiri and Tanya Y. Berger-Wolf, « Mining Periodic Behavior in Dynamic Social Networks », *in*: *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 373–382, DOI: 10.1109/ICDM.2008.104.

[147] Avinash Lakshman and Prashant Malik, « Cassandra: a decentralized structured storage system », *in*: *ACM SIGOPS Operating Systems Review* 44.*2* (2010), pp. 35–40.

[148] Avinash Lakshman and Prashant Malik, « Cassandra: A Decentralized Structured Storage System », *in*: 44.*2* (2010), 35–40, ISSN: 0163-5980, DOI: 10.1145/1773912.1773922, URL: https://doi.org/10.1145/1773912.1773922.

[149] Ora Lassila, Ralph R Swick, et al., « Resource description framework (RDF) model and syntax specification », *in*: (1998).

[150] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani, « Kronecker graphs: An approach to modeling networks », *in*: *Journal of Machine Learning Research* 11.*Feb* (2010), pp. 985–1042.

[151] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos, « Graphs over time: densification laws, shrinking diameters and possible explanations », *in*: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 177–187.

[152] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos, « Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication », *in*: *Knowledge Discovery in Databases: PKDD 2005*, ed. by Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 133–145, ISBN: 978-3-540-31665-7.

[153] Leonid Libkin, Wim Martens, and Domagoj Vrgoč, « Querying Graphs with Data », *in*: *J. ACM* 63.*2* (2016), ISSN: 0004-5411, DOI: 10.1145/2850413, URL: https://doi.org/10.1145/2850413.

[154] Wei Liu, Junfeng He, and Shih-Fu Chang, « Large Graph Construction for Scalable Semi-Supervised Learning », *in*: *ICML*, 2010, pp. 679–686, URL: https://icml.cc/Conferences/2010/papers/16.pdf.

[155] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein, « GraphLab: A New Framework For Parallel Machine Learning », *in*: *CoRR* abs/1408.2041 (2014), arXiv: 1408.2041, URL: http://arxiv.org/abs/1408.2041.

[156] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, « Pregel: A System for Large-Scale Graph Processing », *in*: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 135–146, ISBN: 9781450300322, DOI: 10.1145/1807167.1807184, URL: https://doi.org/10.1145/1807167.1807184.

[157] Alice Marascu, Pascal Pompey, Eric Bouillet, Michael Wurst, Olivier Verscheure, Martin Grund, and Philippe Cudre-Mauroux, « TRISTAN: Real-time analytics on massive time series using sparse dictionary compression », *in*: *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 291–300, DOI: 10.1109/BigData.2014.7004244.

[158] József Marton, Gábor Szárnyas, and Dániel Varró, « Formalising openCypher Graph Queries in Relational Algebra », *in*: *Advances in Databases and Information Systems*, ed. by Mārīte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos, Cham: Springer International Publishing, 2017, pp. 182–196, ISBN: 978-3-319-66917-5.

[159] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye, « Clock-G: A temporal graph management system with space-efficient storage technique », *in*: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2263–2276, DOI: 10.1109/ICDE53745.2022.00215.

[160] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye, « RTGEN: A Relative Temporal Graph GENerator », *in*: *DATAPLAT workshop at the EDBT/ICDT 2022 Joint Conference*, 2022.

[161] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye, *T-Cypher: A Temporal Graph Query Language.* https://project.inria.fr/tcypher/, 2021.

[162] Maria Massri, Zoltan Miklos, Philippe Raipin, Pierre Meye, Amaury Bouchra Pilet, and Thomas Hassan, « RTGEN++: A relative temporal graph generator (submitted) », *in*: *FGCS* (2022).

[163] Jim Melton and Alan R Simon, *Understanding the new SQL: a complete guide*, Morgan Kaufmann, 1993.

[164] Alberto O. Mendelzon and Peter T. Wood, « Finding Regular Simple Paths in Graph Databases », *in*: *SIAM Journal on Computing* 24.*6* (1995), pp. 1235–1258, DOI: 10.1137/S009753979122370X, eprint: https://doi.org/10.1137/S009753979122370X, URL: https://doi.org/10.1137/S009753979122370X.

[165] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen, « Immortalgraph: A system for storage and analysis of temporal graphs », *in*: *ACM Transactions on Storage (TOS)* 11.*3* (2015), pp. 1–34.

[166] Vera Zaychik Moffitt, « Framework for Querying and Analysis of Evolving Graphs », PhD thesis, Drexel University, 2017.

[167] Vera Zaychik Moffitt and Julia Stoyanovich, « Temporal Graph Algebra », *in*: DBPL '17, Munich, Germany: Association for Computing Machinery, 2017, ISBN: 9781450353540, DOI: 10.1145/3122831.3122838, URL: https://doi.org/10.1145/3122831.3122838.

[168] Angelo Montanari and Jan Chomicki, « Time Domain », *in*: *Encyclopedia of Database Systems*, ed. by LING LIU and M. TAMER ÖZSU, Boston, MA: Springer US, 2009, pp. 3103–3107, ISBN: 978-0-387-39940-9, DOI: 10.1007/978-0-387-39940-9_427, URL: https://doi.org/10.1007/978-0-387-39940-9$_$427.

[169] Peter J. Mucha, Thomas Richardson, Kevin Macon, Mason A. Porter, and Jukka-Pekka Onnela, « Community Structure in Time-Dependent, Multiscale, and Multiplex Networks », *in*: *Science* 328.*5980* (2010), pp. 876–878, DOI: 10.1126/science.1184819, eprint: https://www.science.org/doi/pdf/10.1126/science.1184819, URL: https://www.science.org/doi/abs/10.1126/science.1184819.

[170] Michael D Mueller, David Hasenfratz, Olga Saukh, Martin Fierz, and Christoph Hueglin, « Statistical modelling of particle number concentration in Zurich at high

spatio-temporal resolution utilizing data from a mobile sensor network », *in*: *Atmospheric Environment* 126 (2016), pp. 171–181.

[171] S.K. Mukhopadhyay, S. Mitra, and M. Mitra, « An ECG signal compression technique using ASCII character encoding », *in*: *Measurement* 45.*6* (2012), pp. 1651–1660, ISSN: 0263-2241, DOI: https://doi.org/10.1016/j.measurement.2012.01.017, URL: https://www.sciencedirect.com/science/article/pii/S0263224112000322.

[172] *Multi-model highly available NoSQL database - ArangoDB*, URL: https://www.arangodb.com/ (visited on 05/04/2022).

[173] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang, « Introducing the graph 500 », *in*: *Cray Users Group (CUG)* 19 (2010), pp. 45–74.

[174] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel, « On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity and Applications », *in*: *Database Theory — ICDT'99*, ed. by Catriel Beeri and Peter Buneman, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 236–256, ISBN: 978-3-540-49257-3.

[175] *Neo4j Graph Platform – The Leader in Graph Databases*, en, URL: https://neo4j.com/ (visited on 05/04/2022).

[176] M. E. J. Newman, « Finding community structure in networks using the eigenvectors of matrices », *in*: *Phys. Rev. E* 74 (3 2006), p. 036104, DOI: 10.1103/PhysRevE.74.036104, URL: https://link.aps.org/doi/10.1103/PhysRevE.74.036104.

[177] M. E. J. Newman and M. Girvan, « Finding and evaluating community structure in networks », *in*: *Phys. Rev. E* 69 (2 2004), p. 026113, DOI: 10.1103/PhysRevE.69.026113, URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[178] Mark Newman, *Networks*, Oxford university press, 2018.

[179] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz, « Random graph models of social networks », *in*: *Proceedings of the national academy of sciences* 99.*suppl 1* (2002), pp. 2566–2572.

[180] *Oracle | Cloud Applications and Cloud Platform*, URL: https://www.oracle.com/index.html (visited on 06/05/2022).

[181] *Oracle PGX 22.2.2 Documentation - Home*, URL: https://docs.oracle.com/cd/E56133_01/latest/index.html (visited on 05/07/2022).

[182] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66, Previous number = SIDL-WP-1999-0120, Stanford InfoLab, 1999, URL: http://ilpubs.stanford.edu:8090/422/.

[183] Raj Kumar Pan and Jari Saramäki, « Path lengths, correlations, and centrality in temporal networks », *in*: *Phys. Rev. E* 84 (1 2011), p. 016105, DOI: 10.1103/PhysRevE.84.016105, URL: https://link.aps.org/doi/10.1103/PhysRevE.84.016105.

[184] Raj Kumar Pan and Jari Saramäki, « Path lengths, correlations, and centrality in temporal networks », *in*: *Phys. Rev. E* 84 (1 2011), p. 016105, DOI: 10.1103/PhysRevE.84.016105, URL: https://link.aps.org/doi/10.1103/PhysRevE.84.016105.

[185] Raj Kumar Pan and Jari Saramäki, « Path lengths, correlations, and centrality in temporal networks », *in*: *Phys. Rev. E* 84 (1 2011), p. 016105, DOI: 10.1103/PhysRevE.84.016105, URL: https://link.aps.org/doi/10.1103/PhysRevE.84.016105.

[186] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec, « Motifs in Temporal Networks », *in*: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, Cambridge, United Kingdom: Association for Computing Machinery, 2017, 601–610, ISBN: 9781450346757, DOI: 10.1145/3018661.3018731, URL: https://doi.org/10.1145/3018661.3018731.

[187] Himchan Park and Min-Soo Kim, « TrillionG: A trillion-scale synthetic graph generator using a recursive vector model », *in*: *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 913–928.

[188] Terence Parr, *The definitive ANTLR 4 reference*, Pragmatic Bookshelf, 2013.

[189] Matthew Perry, Prateek Jain, and Amit P. Sheth, « SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries », *in*: *Geospatial Semantics and the Semantic Web: Foundations, Algorithms, and Applications*, ed. by Naveen Ashish and Amit P. Sheth, Boston, MA: Springer US, 2011, pp. 61–86, ISBN: 978-1-4419-

9446-2, DOI: 10.1007/978-1-4419-9446-2_3, URL: https://doi.org/10.1007/978-1-4419-9446-2_3.

[190] Ali Pinar, Comandur Seshadhri, and Tamara G Kolda, « The similarity between stochastic kronecker and chung-lu graph models », *in*: *Proceedings of the 2012 SIAM International Conference on Data Mining*, SIAM, 2012, pp. 1071–1082.

[191] Evaggelia Pitoura, « Historical graphs: models, storage, processing », *in*: *European Business Intelligence and Big Data Summer School*, Springer, 2017, pp. 84–111.

[192] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini, « Towards a property graph generator for benchmarking », *in*: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017, pp. 1–6.

[193] Riccardo Pucella, « On equivalences for a class of timed regular expressions », *in*: *Electronic Notes in Theoretical Computer Science* 106 (2004), pp. 315–333.

[194] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan, « A Distributed Path Query Engine for Temporal Property Graphs », *in*: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 499–508, DOI: 10.1109/CCGrid49817.2020.00-43.

[195] Sharvari Rautmare and D. M. Bhalerao, « MySQL and NoSQL database comparison for IoT application », *in*: *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*, 2016, pp. 235–238, DOI: 10.1109/ICACA.2016.7887957.

[196] *RedisGraph | Redis*, URL: https://redis.io/docs/stack/graph/ (visited on 06/05/2022).

[197] Ursula Redmond and Pádraig Cunningham, « Subgraph isomorphism in temporal networks », *in*: *arXiv preprint arXiv:1605.02174* (2016).

[198] Ursula Redmond and Pádraig Cunningham, « Subgraph Isomorphism in Temporal Networks », *in*: *CoRR* abs/1605.02174 (2016), arXiv: 1605.02174, URL: http://arxiv.org/abs/1605.02174.

[199] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng, « On querying historical evolving graph sequences », *in*: *Proceedings of the VLDB Endowment* 4.*11* (2011), pp. 726–737.

[200] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi, « PGQL: a property graph query language », *in*: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

[201] Flavio Rizzolo and Alejandro A Vaisman, « Temporal XML: modeling, indexing, and query processing », *in*: *The VLDB Journal—The International Journal on Very Large Data Bases* 17.*5* (2008), pp. 1179–1212.

[202] *RocksDB | A persistent key-value store*, URL: `http://rocksdb.org/` (visited on 06/05/2022).

[203] Marko A. Rodriguez, « The Gremlin Graph Traversal Machine and Language (Invited Talk) », *in*: *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, Pittsburgh, PA, USA: Association for Computing Machinery, 2015, 1–10, ISBN: 9781450339025, DOI: `10.1145/2815072.2815073`, URL: `https://doi.org/10.1145/2815072.2815073`.

[204] Marko A. Rodriguez and Peter Neubauer, « Constructions from Dots and Lines », *in*: *CoRR* abs/1006.2361 (2010), arXiv: `1006.2361`, URL: `http://arxiv.org/abs/1006.2361`.

[205] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm, « Distributed temporal graph analytics with GRADOOP », *in*: *The VLDB Journal* 31.*2* (2022), pp. 375–401, ISSN: 0949-877X, DOI: `10.1007/s00778-021-00667-4`, URL: `https://doi.org/10.1007/s00778-021-00667-4`.

[206] Martin Rosvall, Alcides V Esquivel, Andrea Lancichinetti, Jevin D West, and Renaud Lambiotte, « Memory in network flows and its effects on spreading dynamics and community detection », *in*: *Nature communications* 5.*1* (2014), pp. 1–13.

[207] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat, *Large-scale graph processing using Apache Giraph*, Springer, 2016.

[208] Betty Salzberg and Vassilis J. Tsotras, « Comparison of Access Methods for Time-Evolving Data », *in*: *ACM Comput. Surv.* 31.*2* (1999), 158–221, ISSN: 0360-0300, DOI: `10.1145/319806.319816`, URL: `https://doi.org/10.1145/319806.319816`.

[209] *SAP HANA Graph Reference - SAP Help Portal | SAP Help Portal*, URL: https://help.sap.com/docs/SAP_HANA_PLATFORM/f381aa9c4b99457fb3c6b53a2fd29c02/30d1d8cfd5d0470dbaac2ebe20cefb8f.html?version=2.0.02&locale=en-US (visited on 05/07/2022).

[210] Satu Elisa Schaeffer, « Graph clustering », *in*: *Computer Science Review* 1.*1* (2007), pp. 27–64, ISSN: 1574-0137, DOI: https://doi.org/10.1016/j.cosrev.2007.05.001, URL: https://www.sciencedirect.com/science/article/pii/S1574013707000020.

[211] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel, « $\mathrm{SP}^2 Bench : ASPARQLPerformanceBenchmark$ », *in*: *2009 IEEE 25th International Conference on Data Engineering*, 2009, pp. 222–233, DOI: 10.1109/ICDE.2009.28.

[212] Michael Schmidt, Michael Meier, and Georg Lausen, « Foundations of SPARQL Query Optimization », *in*: ICDT '10, Lausanne, Switzerland: Association for Computing Machinery, 2010, 4–33, ISBN: 9781605589473, DOI: 10.1145/1804669.1804675, URL: https://doi.org/10.1145/1804669.1804675.

[213] Konstantinos Semertzidis and Evaggelia Pitoura, « A Hybrid Approach to Temporal Pattern Matching », *in*: *arXiv preprint arXiv:2001.01661* (2020).

[214] Konstantinos Semertzidis and Evaggelia Pitoura, « Durable graph pattern queries on historical graphs », *in*: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 541–552, DOI: 10.1109/ICDE.2016.7498269.

[215] Konstantinos Semertzidis and Evaggelia Pitoura, « Durable Graph Pattern Queries on Temporal Graphs », *in*: *IEEE Transactions on Knowledge and Data Engineering* 31.*1* (2019), pp. 181–194, DOI: 10.1109/TKDE.2018.2823754.

[216] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis, « TimeReach: Historical Reachability Queries on Evolving Graphs », *in*: *EDBT*, 2015.

[217] Konstantinos Semertzidis, Evaggelia Pitoura, Evimaria Terzi, and Panayiotis Tsaparas, « Best Friends Forever (BFF): Finding Lasting Dense Subgraphs », *in*: *CoRR* abs/1612.05440 (2016), arXiv: 1612.05440, URL: http://arxiv.org/abs/1612.05440.

[218] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno, « Algorithmics and Applications of Tree and Graph Searching », *in*: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, Madison, Wisconsin: Association for Computing Machinery, 2002, 39–52, ISBN: 1581135076, DOI: 10.1145/543613.543620, URL: https://doi.org/10.1145/543613.543620.

[219] Shashi Shekhar and Dev Oliver, « Computational modeling of spatio-temporal social networks: A time-aggregated graph approach », *in*: *Specialist Meeting-Spatio-Temporal Constraints on Social Networks*, sn, 2010, pp. 6–10.

[220] Alexander Singh and Dimitrios Tsoumakos, « Towards an Algebraic Cost Model for Graph Operators », *in*: *International Workshop on Algorithmic Aspects of Cloud Computing*, Springer, 2017, pp. 89–105.

[221] Richard Thomas Snodgrass, Ilsoo Ahn, Gadi Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada, « TSQL2 Language Specification », *in*: *SIGMOD Rec.* 23.*1* (Mar. 1994), 65–86, ISSN: 0163-5808, DOI: 10.1145/181550.181562, URL: https://doi.org/10.1145/181550.181562.

[222] Christian Staudt and Robert Görke, « A generator of dynamic clustered random graphs », *in*: *ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH)(2009), http://i11www. iti. uni-karlsruhe. de/projects/spp1307/dyngen, informatik*, 2009.

[223] Benjamin Steer, Félix Cuadrado, and Richard Clegg, « Raphtory: Streaming analysis of distributed temporal graphs », *in*: *Future Generation Computer Systems* 102 (2020), pp. 453–464.

[224] Matthias Steinbauer and Gabriele Anderst-Kotsis, « DynamoGraph: A Distributed System for Large-Scale, Temporal Graph Processing, Its Implementation and First Observations », *in*: *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, 861–866, ISBN: 9781450341448, DOI: 10.1145/2872518.2889293, URL: https://doi.org/10.1145/2872518.2889293.

[225] J. Sun, Dimitris Papadias, Yufei Tao, and Bin Liu, « Querying about the past, the present, and the future in spatio-temporal databases », *in*: *Proceedings. 20th International Conference on Data Engineering*, 2004, pp. 202–213, DOI: 10.1109/ICDE.2004.1319997.

[226] Gábor Szárnyas, János Maginecz, and Dániel Varró, « Evaluation of optimization strategies for incremental graph queries », *in*: *Periodica Polytechnica Electrical Engineering and Computer Science* 61.*2* (2017), pp. 175–192.

[227] Jonas Tappolet and Abraham Bernstein, « Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL », *in*: *The Semantic Web: Research and Applications*, ed. by Lora Aroyo, Paolo Traverso, Fabio Ciravegna, Philipp Cimiano, Tom Heath, Eero Hyvönen, Riichiro Mizoguchi, Eyal Oren, Marta Sabou, and Elena Simperl, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–322, ISBN: 978-3-642-02121-3.

[228] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann, « Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs », *in*: *Proceedings of the VLDB Endowment* 10.*8* (2017), pp. 877–888.

[229] *Thing'in, the things' graph platform - Hello Future Orange*, en-US, Nov. 2018, URL: https://hellofuture.orange.com/en/thingin-the-things-graph-platform/ (visited on 07/29/2022).

[230] Y. Tian and J. M. Patel, « TALE: A Tool for Approximate Large Graph Matching », *in*: *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 963–972, DOI: 10.1109/ICDE.2008.4497505.

[231] J. R. Ullmann, « An Algorithm for Subgraph Isomorphism », *in*: 23.*1* (1976), ISSN: 0004-5411, DOI: 10.1145/321921.321925, URL: https://doi.org/10.1145/321921.321925.

[232] « User-friendly temporal queries on historical knowledge bases », *in*: *Information and Computation* 259 (2018), 22nd International Symposium on Temporal Representation and Reasoning, pp. 444–459, ISSN: 0890-5401, DOI: https://doi.org/10.1016/j.ic.2017.08.012, URL: https://www.sciencedirect.com/science/article/pii/S0890540117301554.

[233] Hannes Voigt, « Declarative Multidimensional Graph Queries », *in*: *Business Intelligence*, ed. by Patrick Marcel and Esteban Zimányi, Cham: Springer International Publishing, 2017, pp. 1–37, ISBN: 978-3-319-61164-8.

[234] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin, « Efficiently Answering Span-Reachability Queries in Large Temporal Graphs », *in*: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1153–1164, DOI: `10.1109/ICDE48307.2020.00104`.

[235] M Winlaw, H DeSterck, and G Sanders, *An in-depth analysis of the chung-lu model*, tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[236] Peter T. Wood, « Query Languages for Graph Databases », *in*: *SIGMOD Rec.* 41.*1* (2012), 50–60, ISSN: 0163-5808, DOI: `10.1145/2206869.2206879`, URL: `https://doi.org/10.1145/2206869.2206879`.

[237] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke, « Reachability and time-based path queries in temporal graphs », *in*: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 145–156, DOI: `10.1109/ICDE.2016.7498236`.

[238] Luo Xiangyu, Luo Yingxiao, Gui Xiaolin, and Yu Zhenhua, « An Efficient Snapshot Strategy for Dynamic Graph Storage Systems to Support Historical Queries », *in*: *IEEE Access* 8 (2020), pp. 90838–90846, DOI: `10.1109/ACCESS.2020.2994242`.

[239] Xifeng Yan, Philip S. Yu, and Jiawei Han, « Graph Indexing: A Frequent Structure-Based Approach », *in*: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, Paris, France: Association for Computing Machinery, 2004, 335–346, ISBN: 1581138598, DOI: `10.1145/1007568.1007607`, URL: `https://doi.org/10.1145/1007568.1007607`.

[240] Donghui Zhang, V.J. Tsotras, and B. Seeger, « Efficient temporal join processing using indices », *in*: *Proceedings 18th International Conference on Data Engineering*, 2002, pp. 103–113, DOI: `10.1109/ICDE.2002.994701`.

[241] Donhui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos, and Bernhard Seeger, « Efficient Computation of Temporal Aggregates with Range Predicates », *in*: *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, Santa Barbara, Califor-

nia, USA: Association for Computing Machinery, 2001, 237–245, ISBN: 1581133618, DOI: 10.1145/375551.375600, URL: https://doi.org/10.1145/375551.375600.

[242] Jinxiong Zhang, Cheng Zhong, Hai Xiang Lin, and Mian Wang, « Identifying protein complexes from dynamic temporal interval protein-protein interaction networks », *in*: *BioMed research international* 2019 (2019).

[243] S. Zhang, M. Hu, and J. Yang, « TreePi: A Novel Graph Indexing Method », *in*: *2007 IEEE 23rd International Conference on Data Engineering*, 2007, pp. 966–975, DOI: 10.1109/ICDE.2007.368955.

[244] Tianming Zhang, Yunjun Gao, Linshan Qiu, Lu Chen, Qingyuan Linghu, and Shiliang Pu, « Distributed time-respecting flow graph pattern matching on temporal graphs », *in*: *World Wide Web 23.1* (Jan. 2020), pp. 609–630, ISSN: 1573-1413, DOI: 10.1007/s11280-019-00674-0, URL: https://doi.org/10.1007/s11280-019-00674-0.

Appendix

A Description of temporal functions and operators

We present in Chapter 4 the temporal constructs added to our query language, T-Cypher. In this appendix, we describe the added temporal functions and operators used to enable the expression of temporal predicates.

The following temporal functions are applied on temporal values V^T that includes time instants, intervals, and duration values. That is, a temporal function, takes a set of temporal values and returns a temporal value. In the following, we describe each temporal function. We consider that i^s and i^e denote the starting and ending time instants of a time interval i.

- **Start**: takes a time interval and returns its starting time. Formally: Consider a time interval i, then $start(i) = i^s$.
- **End**: takes a time interval and returns its ending time. Formally, Consider a time interval i, then $end(i) = i^e$.
- **Add**: takes a time instant and a duration and returns the time interval starting at that time instant and ending at the time instant plus the input duration. Formally, consider a time interval i and a duration d, then add(t, d) = i s.t. $i^s = t$, $i^e = t + d$.
- **Sub**: takes a time instant and a duration and returns the time interval ending at that time instant and starting at the time instant minus the time input duration. Formally, consider a time instant and a duration d, then:

$$sub(t, d) = \begin{cases} \text{i s.t. } i^e = t \text{ and } i^s = t - d, & \text{if t>d} \\ \text{null}, & \text{otherwise} \end{cases}$$

- **ElapsedTime**: takes two time intervals and returns the duration between the ending time instant of the first time interval and the starting time of the second one. However, if the ending time instant is greater than the starting time instant,

the returned result is null. Formally, consider time intervals i and i', then:

$$elapsedTime(i, i') = \begin{cases} i^e - i'^s, & \text{if } i^e \leq i'^s \\ null, & \text{otherwise} \end{cases}$$

— **Duration**: takes a set of time intervals and returns the sum of a total number of chronons between each interval's starting and ending time instants. Formally, consider the set of time intervals $i = \{i_1, ..., i_n\}$, then:

$$duration(i) = \sum_{1 \leq k \leq n} (i_k^e - i_k^s)$$

.

— **Intersection**: takes a set of time intervals and returns a time interval containing all shared time instants between the input time intervals. Formally, consider $\{i_1, ..., i_n\}$ as a set of n time intervals, then:

$$intersection(i_1, ..., i_n) = \begin{cases} i, & \text{if } i^s = \text{MAX}(i_1^s, ..., i_n^s) \wedge i^e = \text{MIN}(i_1^e, ..., i_n^e) \wedge i^s \leq i^e \\ \text{null}, & \text{otherwise} \end{cases}$$

— **Range**: takes a set of time intervals and returns a time interval that covers the entire time range of input intervals. Formally, consider $\{i_1, ..., i_n\}$ as a set of n time intervals, then:

$$range(i_1, ..., i_n) = i \text{ if } i^s = \text{MIN}(i`_1^s, ..., i_n^s), i^e = \text{MAX}(i_1^e, ..., i_n^e)$$

.

Besides temporal functions, we also incorporate temporal operators that enable the comparison between time intervals. These temporal operators represent the thirteen operators of Allen temporal algebra presented in [11]. In the following, we describe the semantics of each operator. Consider (i, i') to denote time intervals, (i^s, i'^s) to be their starting time instants, and (i^e, i'^e) to denote their ending time instants, then:

— i **BEFORE** i' evaluates to true if i ends before i' starts.
— i **AFTER** i' evaluates to true if i starts after i'.
— i **OVERLAPS** i' evaluates to true if i' starts after the start of i and finishes after the end of i.

- i **STARTS** i' evaluates to true if i and i' starts at the same time instant and i ends before i' s.t. $i^s = i'^s$ and $i^e < i'^e$.
- i **DURING** i' evaluates to true if i after and ends before i' s.t. $i^s > i'^s$ and $i^e < i'^e$.
- i **FINISHES** i' evaluates to true if i and i' end at the same time instant and i starts after the start time instant of i' s.t. $i^e = i'^e$ and $i^s > i'^s$.
- i **EQUALS** i' evaluates to true if i and i' starts and ends at the same time instant s.t. $i^s = i'^s$ and $i^e = i'^e$.
- i **MEETS** i' evaluates to true if i starts after the finishing time of i' s.t. $i^s > i^e$.
- i **MET BY** i' evaluates to true if i starts after the finishing time of i'.
- i **OVERLAPPED BY** i' evaluates to true if i start after the starting time of i' and ends after the ending time of i' s.t. $i'^s < i^s$ and $i^s < i^e$.
- i **STARTED BY** i' evaluates to true if i and i' starts at the same time instant and i' ends before i s.t. $i^s = i's$ and $i^e > i'^e$
- i **CONTAINS** i' evaluates to true if i starts before the start of i' and finishes after the end of i' s.t. $i^s < i'^s$ and $i^e > i'^e$.
- i **FINISHED BY** i' evaluates to true if $i$and i' end at the same time instant and i' starts after i s.t. $i^e = i'^e$ and $i'^s > i^s$.