Clock-G: Temporal Graph Management System

Maria Massri, Zoltan Miklos, Philippe Raipin, Pierre Meye

Clock-G: Temporal graph management system

Maria Massri[1,2], Zoltan Miklos[1], Philippe Raipin[2], and Pierre Meye[2]

[1] Univ Rennes CNRS IRISA, Rennes, France {maria.massri, zoltan.miklos}@irisa.fr
[2] Orange Labs, Cesson Sévigné, France {maria.massri, philippe.raipin}@orange.com, meye_pierre@yahoo.fr

Abstract. Graphs are a ubiquitous data model for capturing entities and their relationships. Since most graphs that model real-world networks evolve over time, efficiently managing temporal graphs is an important problem from both a theoretical and practical perspective. Querying the history of temporal graphs can lead to new applications such as object tracking, anomaly detection, and predicting future behavior. However, existing commercial graph databases lack native temporal support, hindering their usefulness in these use cases.
This paper introduces Clock-G, a temporal graph management system designed to handle the history temporal graphs. What differentiates Clock-G from other temporal graph management systems is its comprehensive approach, covering query language, query processing, and physical storage. We define T-Cypher, a temporal extension of Cypher query language, enabling user-friendly and concise querying of the graph's history. Additionally, we propose a query processor that utilizes temporal statistics collected from underlying temporal graphs to offer a good evaluation plan for T-Cypher queries. We also propose a novel storage technique that balances space usage and query evaluation time.

Keywords: Temporal Graph management · Storage · Query language · Query processing.

1 Introduction

Graphs are frequently used to model real-world interactions as a collection of vertices and relationships providing generally a fertile ground to analyze relationship-centered domains. Despite the wealth of studies on managing static graphs, a time version support is seldom provided.

This work is motivated by the industrial use case of Thing'in[3], an Orange-initiated platform that manages a graph of connected (machines, traffic lights, cameras, etc.) and non-connected (doors, roads, shelves, etc.) objects with structural and semantic environment descriptions. Clients include companies and public administrations developing smart city services and private object owners building analytical IoT applications. The graph is maintained by a commercial database lacking temporal support. However, there has been an extensive

[3] https://www.thinginthefuture.com/

and recent demand by the clients of Thign'in for preserving the past states and connections of the graph for the interest of tracking objects, anomaly detection and forecasting the future behaviour. Concrete use cases include Mo.Di.Flu[4], a project tracking product positions in a manufacturing pipeline to detect delays or losses. To address these requirements, we designed the temporal graph management system Clock-G. Although initially designed for the particular use case of Thing'in, Clock-G is a general purpose system that can be used in other application domains requiring temporal graph management.

Storing and querying temporal graphs are possible by exploiting a commercial graph database with temporal metadata [7,5]. However, these systems do not natively offer time-version support which might lead to unpredictable performances. Hence, we argue that time should be considered as a first-class citizen rather than a simple add-on.

Existing temporal graph management systems often lack comprehensive coverage of the different layers that should be addressed to account for the temporal dimension, as they may not provide a native temporal query language or an efficient query processor for temporal queries. Many existing systems [23,31,44] prioritize storage techniques and only offer simple, general-purpose temporal graph queries that cannot meet the requirements of specific applications such as the Thing'in use case. To address this issue, our paper takes a comprehensive approach to managing temporal graphs by addressing the different layers of query language, query processing, and physical storage.

This paper is an extension of our previous work [30]. A major improvement of this version compared to our previous work is the inclusion of a query language that supports complex temporal queries into Clock-G, including graph pattern matching and navigational queries with temporal predicates. Additionally, we have developed a query processor capable of evaluating temporal graph queries. Unlike the previous version, which focused primarily on storage techniques, this version addresses the challenges of query languages and processing, making our system more comprehensive.

Various temporal graph querying solutions have been proposed in the literature, extending OLAP and OLTP queries with time. OLAP queries include finding most durable connected components [41], temporal shortest paths [20], and temporal centrality [34], while OLTP queries include temporal graph pattern matching [32,39] and temporal navigational queries [36,2]. In this paper, we focus on extending OLTP queries with the temporal dimension. Hence, we propose T-Cypher, a temporal extension of the well-known graph query language Cypher [10] designed to enhance graph pattern matching and navigational queries with the temporal dimension.

Example 1.1 Figure 1 illustrates a graph pattern and its corresponding Cypher query, as well as a temporal graph pattern and its corresponding T-Cypher query. In this example, the non-temporal pattern retrieves machines ($m1$ and $m2$) that indicate the same alert (a) and are situated in the same room (r). To enhance

[4] https://www.pole-emc2.fr/projet/mo-di-flu/

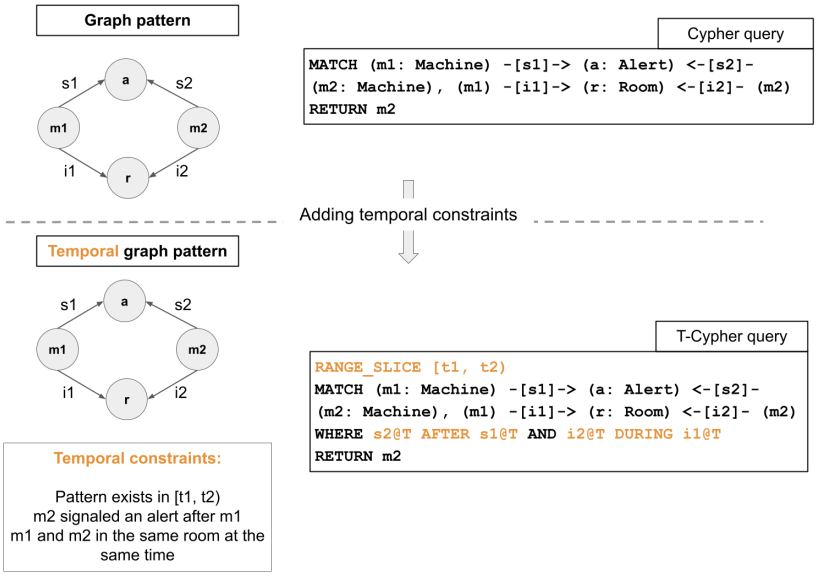


Fig. 1: Example showing a T-Cypher query with temporal constructs compared to a non-temporal Cypher query

machine maintenance efficacy, one might want to identify machines that are affected by malfunctions in other machines. In such a scenario, the order in which alerts were triggered becomes significant since it allows for the retrieval of machines that indicated an alert after another machine signaling the same alert, thus implying machine-to-machine influence. This is translated in the T-Cypher query by the inclusion of the temporal constraints (`s2@T AFTER s1@T AND i2@T DURING i1@T`). Besides we trim the search space to a time interval [t1, t2) of interest instead of searching the full history which is translated in the T-Cypher query by `RANGE_SLICE [`t_1`,` t_2`)` clause at the beginning of the query.

We propose a query processor to evaluate T-Cypher queries. Our processing pipeline involves an algebra, cost model, and plan selection algorithm. We introduce a temporal graph algebra that extends the graph algebra proposed by Hölsch et al. [21] for Cypher queries with temporal operators. The evaluation plan composed of algebraic operators is chosen based on a cost model that relies on changing cardinalities provided by the backend store. Unlike traditional query processors, we consider the optimal plan to vary within the requested time interval due to changes in cardinalities, and thus preserve the history of cardinalities in temporal histograms. We implemented this query processor in Clock-G and evaluated it by executing various queries on synthetic datasets, comparing its performance with an alternative solution that is based on extend-

ing a non-temporal graph database (Neo4j[5]) with the temporal dimension. The results demonstrate the efficiency of our cost model and query processor.

Besides proposing a query language and a query processing pipeline, we propose and implement into Clock-G a storage technique for temporal graphs. Various storage approaches have been proposed in literature for managing temporal graphs, including the *Log* and the *Copy+Log* methods. The former involves preserving all graph updates as timestamped logs, while the latter stores the updates in time windows, along with snapshots (i.e. states of the graph) at the end of each window. However, the space usage of the *Copy+Log* method, especially for growth-mostly graphs, can be space-consuming due to redundant graph entities shared between snapshots. On the other hand, the impact of the *Log* approach on query evaluation time can be detrimental. To address these limitations, we propose the *δ-Copy+Log* method, which stores only the difference between successive snapshots, called deltas. Snapshots are stored every M time windows and used as starting points for query evaluation. Specifically, half of the time windows and their corresponding deltas are stored in a forward fashion, while the other half are stored in a backward fashion. During query evaluation, the choice between forward or backward construction of the result is determined based on the requested time instant. This approach results in a significant reduction in the maximum execution time of queries by up to 50%. We also conducted experiments to evaluate the performance of Clock-G. A comparison between traditional methods and the *δ-Copy+Log* validates that our technique offers a good compromise between the performances of the *Log* and *Copy+Log* methods. Besides, we showcase how the parameters of Clock-G can be calibrated in order to tune the overall performance with an adequate configuration that adheres most with the acceptable threshold of query latency and available storage resources.

The main contributions of this work reduce to the following:

- Proposing a user-friendly extension of the Cypher query language that enables to express a large fraction of temporal queries.
- Proposing a temporal graph algebra and query processor for T-Cypher queries.
- Proposing *δ-Copy+Log* as a space-efficient variant of the traditional *Copy+Log* method.
- Taking a holistic approach into managing temporal graphs by addressing the different layers of storage, query language, and processing.

Outline Section 2 provides an overview of related work. Section 3 introduces key definitions for our proposed approaches. Section 4 introduces our proposed temporal graph query language. Section 5 describes the query processor used to evaluate our temporal graph queries. Section 6 presents our proposed storage approach. Section 7 presents the architecture and overall design features of Clock-G. Section 8 presents the results of our experiments conducted on real and synthetic datasets. Finally, Section 9 concludes the paper and gives a future perspective.

[5] https://neo4j.com

2 Related work

This paper proposes a comprehensive approach to managing temporal graphs with versioning support, which includes addressing challenges related to storage techniques, query languages, and query evaluation. We discuss the related work on these challenges in subsequent sections.

2.1 Query language

In the field of graph querying, subgraph pattern matching or navigational queries are the core concepts. Many proposals to extend these queries with the temporal dimension have been posited.

Some extensions focus only on extending navigational queries with the temporal dimension. Temporal reachability queries where extended with the temporal dimension [40,43]. Granite [36] is a query engine that implements temporal navigational queries by adding temporal predicates and temporal ordering constraints, as well as temporal aggregations. A temporal extension of regular path queries (TRPQ) was proposed in [2] by introducing structural and temporal navigational operators. The T-GQL [7] query language is a temporal extension of the standard query language for graph databases GQL [8]. The proposed extension allows the expression of different types of temporal paths. However, these solutions focus on navigational queries rather than graph pattern matching queries.

Other proposals present temporal extensions of graph pattern matching queries. For instance, non-decreasing time flow pattern are defined as each path between two nodes follows a non-decreasing time flow [34,37,45]. It is useful for studying the spread of a disease or the flow of rumors in a social network. Most Durable Graph Pattern (MDGP) returns the most durable matches of a given non-temporal pattern, which is useful for analyzing the tightness of connectivity between nodes [41]. Despite the usefulness of these proposals in some applications, they cannot express more general temporal predicates between the elements of a pattern.

The Temporal Graph Algebra (TGA) [32] is a temporal generalization based on temporal relational algebra for some graph operators. These operators can filter the search to a time instant or interval or returns subgraphs that are isomorphic to a given pattern during a given time instant or interval. GRALA [39] is a temporal analytical language that offers temporal operators to determine graph snapshots, the difference between two snapshots, and the subgraphs satisfying a given time-dependent graph pattern. Despite the novelty of these extensions, they do not support navigational queries.

T-SPARQL [**?**] is a temporal graph query language for RDF that embeds the features of TSQL2 [**?**]. To express temporal predicates over timestamp variables, the authors propose using a subset of Allen's temporal operators [1]. 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 and temporal slicing. In this paper, temporal predicates and slicing are used on the property graph model, extended to include temporal navigation functionalities.

This paper presents a novel method for querying temporal graphs that builds upon graph pattern matching and navigational queries by incorporating the temporal dimension. Our primary objective is to propose a concise and user-friendly syntax that is easy to learn and facilitates intuitive reasoning and query construction for temporal graphs. Motivated by this goal, we proposed T-Cypher (Section 4), a temporal graph query language that extends the popular Cypher query language [10]. 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 upcoming standrad graph query language GQL [8]. Besides, Cypher is expressive, declarative, normalized, and open source.

2.2 Query processing

A query processor uses an algebra to convert a query into a set of algebraic operators. In the context of temporal graph management, a Temporal Graph Algebra (TGA) was proposed in [32], which includes graph operators that are extended with the temporal dimension. In this work, we define a temporal graph algebra that extends the graph algebra defined by Hölsch et al. for Cypher queries in [21]. Our choice of extending this algebra, rather than other alternative graph algebras (such as GraphQL [18] and GRAD [12]), is based on its compatibility with our proposed query language, which extends Cypher.

Evaluating a query implies choosing a good evaluation plan that ideally minimizes the cardinality of sub-results, reducing thus the overall execution time. The plan selection technique is usually coupled with a cost model that defines a cost function for each algebraic operator which allows to approximate the resulting cardinality of an operator before evaluation. This evaluation pipeline was followed in [13] for processing Cypher queries in a graph database. However, our goal in this paper is to extend this pipeline for the temporal graph model.

A query processor for temporal navigational queries can be found in Granite [36]. This plan selection approach splits the query path into sub-path segments to reduce cardinality, and uses a cost model based on temporal histograms to estimate plan cost. However, this approach is limited to path queries and cannot handle temporal graph pattern matching, which requires a more complex plan selection approach. To address this problem, we present a query processor that evaluates temporal graph pattern matching queries (Section 5).

2.3 Storage

Available temporal graph storage techniques can be categorized as follows: ***Log***, ***Copy***, ***Copy-On-Write*** and ***Copy+Log***. These methods are mainly motivated by concepts of logging and *checkpointing* which reflects on lessons learned from

classical techniques of database state recovery. The ***Log*** storage approach used in [15,11] stores graph updates as timestamped logs, allowing recovery of any graph state by loading logs with a timestamp lower than or equal to the requested one. In contrast, the ***Copy*** approach materializes and persists graph snapshots. These methods represent two extremes in storing temporal graphs, favoring either space optimization or query computation time optimization. ***Copy-On-Write*** [19,27,29,4] involves copying a single graph entity whenever it gets updated, while ***Copy+Log*** [17,31,25,23,24,16,44] stores graph updates in temporally disjoint partitions (called time windows), along with snapshots representing valid states of the graph. The advantage of the Copy+Log approach is that the state of the graph at a given time instant can be recovered by reading a single snapshot and all graph updates recorded in a time window.

In this work, we address a critical limitation of the *Copy+Log* storage approach which relates to the high space consumption of full graph snapshots. To mitigate this issue, we propose the ***δ-Copy+Log*** (Section 6) approach which considers the difference between snapshots instead of materializing full snapshots. We preserve a number of snapshots to serve as starting points for query evaluation and after a fixed number of delta, a full snapshot is materialized. This approach differs from traditional methods such as RMAN in that it replaces full backups with deltas that contain only the difference between two snapshots.

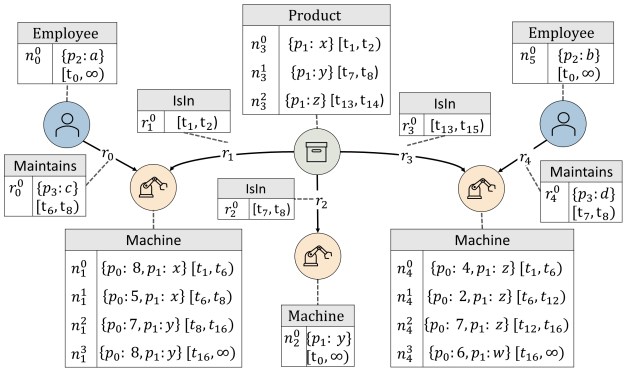
3 Formal definitions

3.1 Time domain

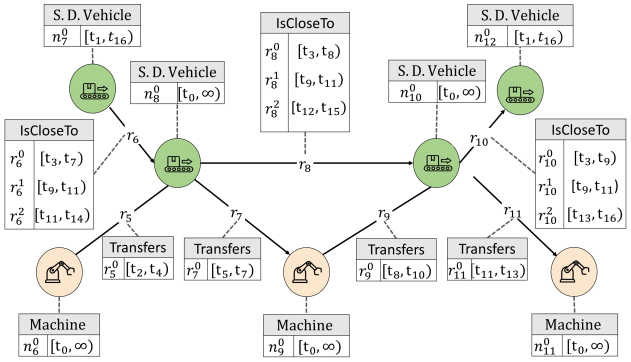
In this section, we present a definition of the time domain, which is essential in the development of data management systems that incorporate temporal ontologies. The time domain definition is particularly important in assigning temporal validity information to data items [33]. Our approach to modeling time involves selecting a discrete temporal flow, which is achieved by quantifying a time axis with time granules [6]. Time granules, also known as chronons, are the smallest indivisible units of time defined by a specific temporal granularity (such as a second or a millisecond). We define the time domain, denoted as Ω^T, as a totally ordered set of instants that includes a sequence of discrete time granules: $\Omega^T = \{t^i | i \in \mathbb{N}\} \bigcup \{Now, \infty\}$. The duration between consecutive instants in the sequence is equal to a chronon. In addition, we assume that the system assigns a transactional time to each graph update.

3.2 Temporal graph relation

A T-Cypher query produces a temporal graph relation as output. Each relation is represented as a bag of tuples, where a tuple u is a partial function that maps names to values. The named fields of a tuple are defined as $u = (a_1 : v_1, \ldots, a_n : v_n)$, where $(a_1, \ldots, a_n)$ are distinct names, and each element in $(v_1, \ldots, v_n)$ can be a value, node or relationship state, set of node or relationship states, or paths.



(a) Toy graph A



(b) Toy graph B

Fig. 2: 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)

These states correspond to nodes or relationships within a specific time interval during which their property values remained constant. We consider V, k, ID, L, and T to denote the set of values, property keys, node identifiers, node labels, and relationship types, respectively.

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.
- $k = \{k_1 : v_1, \ldots, k_m : v_m\}$ is a map of property names and values such that $k_i \in k$ and $v_i \in V, \forall 1 \leq i \leq m$.
- $\tau \in \Omega^T \times \Omega^T$ 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.
- $k = \{k_1 : v_1, \ldots, k_m : v_m\}$ is a map of property names and values such that $k_i \in k$ and $v_i \in V, \forall 1 \leq i \leq m$.
- $\tau \in \Omega^T \times \Omega^T$ is the validity time interval during which the relationship state was valid.

Example 3.1 To clarify the previous definitions, we present a concrete example of a temporal property graph inspired by the use case of smart factories. Figures 2(a) and 2(b) show two toy graphs (A and B) inspired by this use-case. In these graphs, nodes $\{n_0, ..., n_{12}\}$ model products, machines, self-driving vehicles (S.D. vehicles), and employees. Whereas, relationships $\{r_0, ..., r_{11}\}$ represent the connections between nodes. Properties $\{p_0, \ldots, p_3\}$ are attached to nodes and relationships to describe these graph entities.

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. vehicle 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 1(a) and 1(b) 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 2.

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 2(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 $[t_{16}, \infty)$.

Table 1: A fraction of the relationships and their states of the graph in Figures 2(a) and 2(b)

(a) Node states

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	$n_2^0 = (id_{n_2}, \text{Machine}, \{p_1 : y\}, [t_0, \infty))$

(b) Relationship states

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	$r_2^0 = (id_{n_3}, id_{n_2}, \text{IsIn}, \{\}, [t_7, t_8))$

4 Temporal graph query language

This section presents our temporal graph query language T-Cypher that extends Cypher with temporal constructs. Throughout this section, we provide clear query examples and their results to clarify the semantics of our temporal constructs. A more detailed description of the syntax and semantics of T-Cypher is given the online documentation [6]. Our proposed extension, T-Cypher, is designed to incorporate temporal constructs without requiring modifications to the existing grammar rules. This approach ensures a straightforward transition for practitioners who are already familiar with Cypher, while reducing query verbosity.

With T-Cypher, graph variables such as nodes, relationships, and properties, as well as temporal variables referring to time validity intervals, can be expressed. This enables the application of temporal constraints to the temporal variables of the query. Furthermore, T-Cypher introduces the trim statement, which can be used at the beginning of a query to prune the search space to single or multiple time intervals. This guarantees that all variables defined in the query are valid during at least one of these intervals. Temporal functions and operators, can also be used in T-Cypher to define constraints and predicates on temporal variables of the query. Another key feature of T-Cypher is the ability to express different types of temporal paths. We define these key temporal constructs in the following.

Temporal slicing clause We propose a temporal slicing clause to prune the search space of a query to a single time instant or time interval. Hence, the tem-

[6] https://project.inria.fr/tcypher/

poral 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`.

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 3 shows two queries applied to the Toy graph A (Figure 2(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 $[t_1, t_8)$.

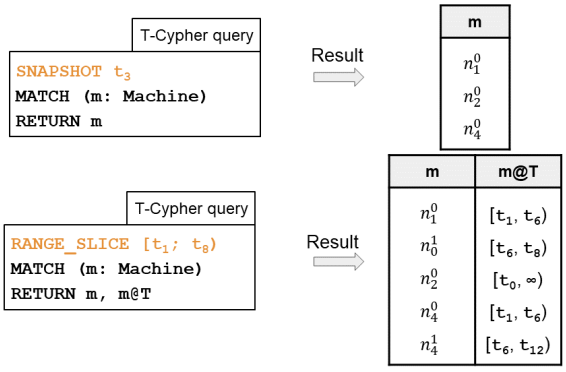


Fig. 3: Example of temporal slicing

Temporal functions and operators We define a set of temporal functions that can be applied to the temporal variables of a pattern to define temporal predicates. For space limitations, we present some of these functions in Table 2, whereas a more comprehensive description is given in the online documentation of T-Cypher. Besides, we use Allen's operators [1] (e.g., before, after, during) to define temporal relations between the temporal variables of a pattern.

Figure 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 2(a)). This query returns the elapsed time[7] 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`)

[7] 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.

Table 2: Description of some temporal functions used in T-Cypher

Function	Description	Return type
ELAPSED_TIME(i, i')	Returns the elapsed time between i and i'	Duration
DURATION(i)	Returns the duration of i	Duration
INTERSECTION($i_0, \ldots, i_n$)	Returns the intersection between $\{i_0, \ldots, i_n\}$	interval

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.

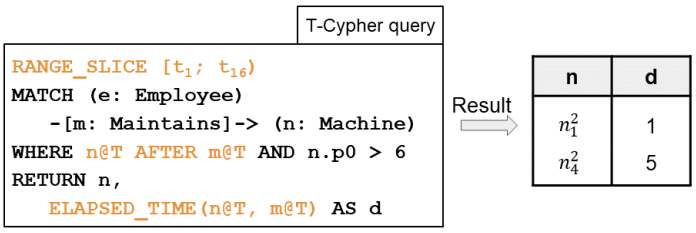


Fig. 4: Example of temporal functions and operators

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. In T-Cypher, we include three temporal types that can cover a large subset of queries: Continuous, Sequential, and Pairwise-continuous (Figure 5), which we describe in the following.

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.

Continuous path [38] 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

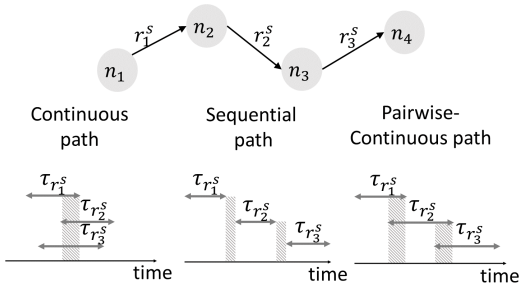


Fig. 5: Different types of temporal relationship patterns: Continuous, Pairwise Continuous and Sequential ($\tau_{r_1^s}$, $\tau_{r_2^s}$ and $\tau_{r_3^s}$ refer to time validity intervals of relationships r_1^s, r_2 and r_3^s)

(r_i^k) of the path is not null and τ_p is equal to the intersection between time intervals $\{\tau_{r_1^s}, \ldots, \tau_{r_k^k}\}$.

Figure 6 presents a T-Cypher query with a continuous path and its result when applied to Toy graph B (Figure 2(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 $[t_3, t_9)$).

T-Cypher query

```
RANGE_SLICE [t1; t16)
MATCH (s1: SDVehicle)
   -[CONTINUOUS(p: IsCloseTo*3)]-> (s2: SDVehicle)
RETURN p
```

Result

p
$\{n_7^0 r_6^0 n_8^0 r_8^0 n_{10}^0 r_{10}^0 n_{12}^0, [t_3, t_7)\}$
$\{n_7^0 r_6^1 n_8^0 r_8^1 n_{10}^1 r_{10}^1 n_{12}^0, [t_9, t_{11})\}$
$\{n_7^0 r_6^2 n_8^0 r_8^2 n_{10}^1 r_{10}^2 n_{12}^0, [t_{12}, t_{14})\}$

Fig. 6: Example of a continuous path

Sequential path [45,22,37] A sequential path is a temporal path where each relationship state r_{i+1}^s should occur after the relationship state r_i^s ($\forall\, 1 \leq i < k$). Hence, the ending time instant of $\tau_{r_i^s}$ 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.

To illustrate, Figure 7 presents a T-Cypher query with a sequential path and its result when applied to the Toy graph B (Figure 2(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 $[t_{11}, t_{13})$).

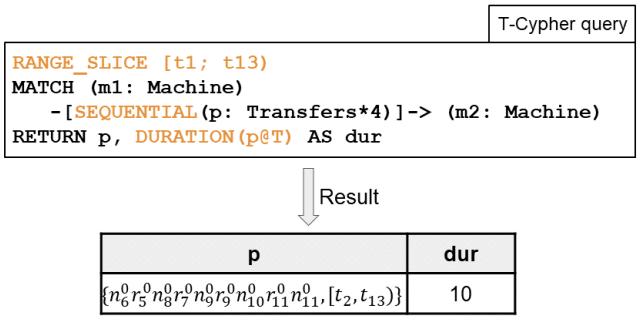


Fig. 7: Example of a sequential path

Pairwise-continuous path [7] 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, $\tau_{r^s_i}$ starts within the time boundaries of $\tau_{r^s_{i-1}}$ and ends within the time boundaries of $\tau_{r^s_{i+1}}$.

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 8 presents a T-Cypher query with a pairwise-continuous path and its result when applied to the Toy graph B (Figure 2(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 $[t_{13}, t_{16})$).

5 Temporal graph query processor

In this section, we give an overview of the query processing pipeline presented in Figure 9.

T-Cypher query

```
RANGE_SLICE [t1; t16)
MATCH (m1: SDVehicle)
   -[PAIRWISE_CONTINUOUS(p: IsCloseTo*3)]->
   (m2:SDVehicle)
RETURN  p, DURATION(p@T) AS dur
```

Result

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

Fig. 8: Example of pairwise-continuous path

The query parser checks the syntax of a T-Cypher query according to defined grammar rules and generates an Abstract Syntax Tree (AST). The parser then uses the AST to create a parsed query object that is understandable by the query planner. Using a cost-based model, the query planner generates an algebraic plan with cardinalities of subqueries based on temporal histograms. The query evaluator executes each query operator by communicating with the storage engine using *δ-Copy+Log* technique presented in Section 6. In the following, we will introduce our temporal graph algebra, cost model, and plan selection algorithm.

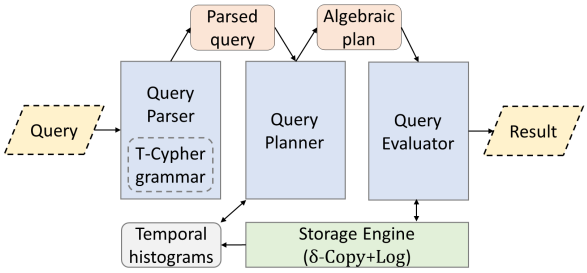


Fig. 9: The query processing pipeline implemented in Clock-G

5.1 Temporal graph algebra

We extend the graph algebra proposed by Hölsch et al. [21] by adding time-based operators to translate T-Cypher queries into algebraic representations. Our extension relies on temporal graph relations defined in Section 3.2. These

relations are bags of tuples that map names to various entities, including node or relationship states, sets of states, or temporal paths.

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 10 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;
```

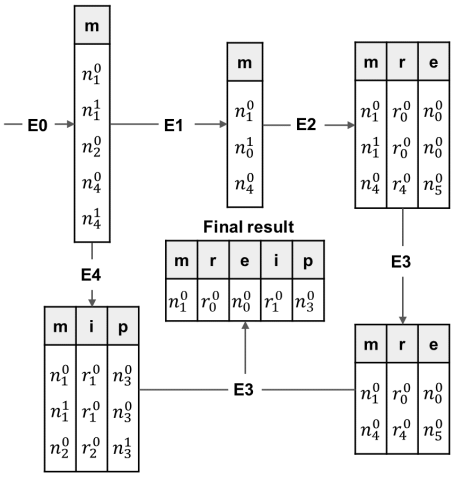


Fig. 10: Example showing the result of evaluating the algebraic operators $(E_0, \ldots, E_5)$ on the toy graph in Figure 2(a)

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 10.

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 10.

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:

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

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 10.

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 10.

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

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

5.2 Cost model

This section defines the cost model used by the query planner, which estimates the cost of an evaluation plan. The cost of each operator is equal to the estimated cardinality of its output relation. Our model differs from classical query processing models commonly used in relational databases because it considers the cost of a query to change over time, meaning an optimal plan for one time interval may not be optimal for another due to changing cardinalities. Our query planner accounts for this by computing the cardinality of each algebraic operator based on the requested time interval. We use $card(E)$ to denote the estimated cardinality of an algebraic expression E.

We use temporal histograms 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 is equal to the estimated cardinality of the node states with a given label ρ_a valid during a given time interval τ, as given in the equation below.

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

Expand operator The cost of the expand operator 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)), as given in the equation below.

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

Select operator The cardinality of the select operator 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, as given in the equation below.

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

The selectivity of graph entities is computed as follows:

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

Note that, 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. 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.

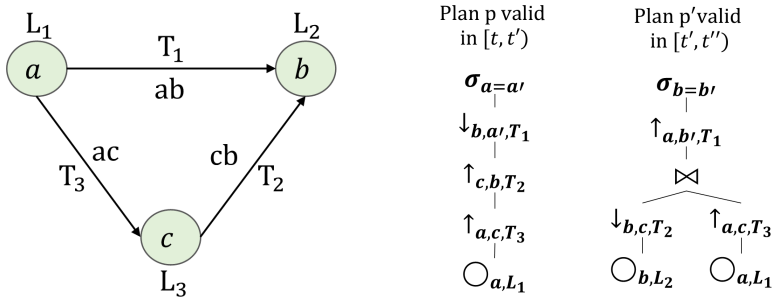
Join operator The cost of the join operator applied on expressions E, and E' is equal to the product of the cardinalities of these expressions, as given in the equation below.

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

5.3 Greedy plan selection algorithm

This section describes an algorithm that greedily generates an evaluation plan for a T-Cypher query (Algorithm 1). 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, 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



(a) Graph pattern

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

Fig. 11: Example illustrating a graph pattern and two possible logical plans, each corresponding to a time interval

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 5.2. 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

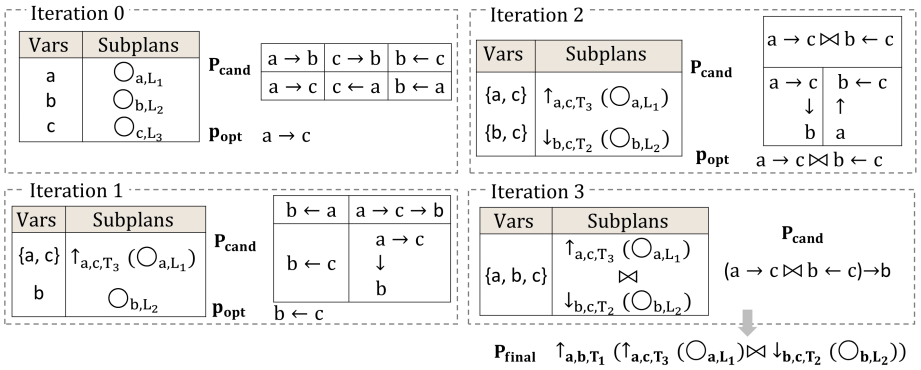
candidate sub-plans are added to the P_{cand} and the final plan p_{final} contained in P is returned.

Algorithm 1: Greedy selection of a logical plan for a T-Cypher query

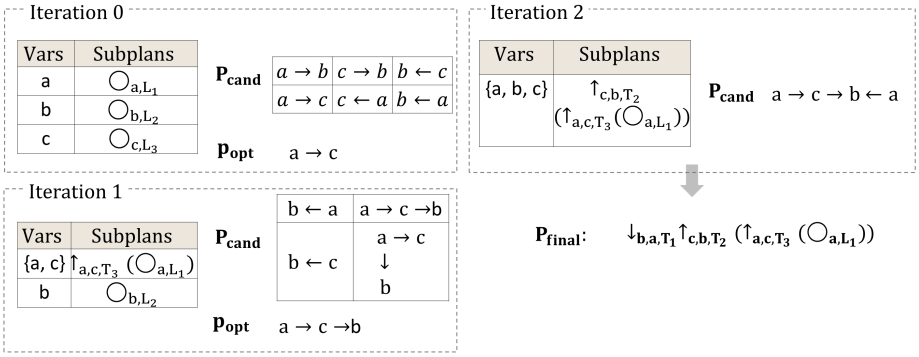
```
   Input: Query object Q, τ
   Output: Logical plan p_final
 1 P ← InitPlans() N ← ExtractNodes(Q) ;
 2 τ ← ExtractTimeInterval(Q) ;
 3 for n ← N do
 4 |   p ← getNodes(n) ;
 5 |   P.insert(p) ;
 6 P_cand.size ≥ 1 P_cand ← initPlans() ;
 7 for p ∈ P do
 8 |   for p' ∈ P do
 9 |   |   if joinExists(p,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

We show how an optimal plan for the graph pattern presented in Figure 11(a) is computed by applying Algorithm 1. In this example, we present three node variables $\{a, b, c\}$ labelled with $\{L_0, L_1, L_2\}$ and the relationship variables $\{ab, cb, ac\}$ having types $\{T_0, T_1, T_2\}$. We show in Figure 11(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 11(b) correspond to time intervals $[t, t')$ and $[t', t'')$, respectively. Figures 12(a) and 12(b) present the selection of operators in each iteration of Algorithm 1, yielding to plans p and p' presented in Figure 11(b). Note that we omit some parameters from the notations of operators when they can be derived from the context.

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



(a) Logical plan in $[t, t')$



(b) Logical plan in $[t', t'')$

Fig. 12: Greedy selection of logical plans in for different time intervals

overlap with the requested time interval). A possible way of handling this is to keep all the cardinalities in 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 at most searching all the elements of the array to retrieve the cardinality for a single time interval. To mitigate this complexity, we propose the use of segment trees [3].

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 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.

6 Temporal graph storage

The δ-*Copy+Log* is a variant of the Copy+Log storage approach that we propose to mitigate the space cost induced by storing full snapshots. Recall that the Copy+Log consists of storing snapshots that are valid between the boundaries of a time window s.t. each time window contains a fixed number of graph operations. Now, the δ-*Copy+Log* follows a similar mechanism with the main difference that consists of storing deltas instead of snapshots. A critical point is that a delta differs from a time window. That is, a time window contains every graph operation that exists between two snapshots whereas a delta contains the only the minimum number of graph operations that transform a snapshot into another one. Indeed, an addition of an element in cancelled by a deletion of the same element, hence, both operations are stored in time window but omitted from the delta. We store a snapshot after a number of time windows in order to serve as a starting point for query evaluation. Having this, we store graph operations in consecutive time buckets containing each 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. That is, half of the deltas and time windows in a bucket is constructed in a forward fashion whereas the other half is constructed 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 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.

Figure 13 illustrates the storage internals of the δ-*Copy+Log* and *Copy+Log* methods. It shows that the *Copy+Log* method stores time windows and snapshots. Whereas, the δ-*Copy+Log* stores time windows, deltas and snapshots. In this example, we consider a set of time buckets B where $M = 6$ which implies that the bucket contains 3 forward time windows $\{\omega^1_{\Rightarrow}, \omega^2_{\Rightarrow}, \omega^3_{\Rightarrow}\}$ and 3 backward time windows $\{\omega^4_{\Leftarrow}, \omega^5_{\Leftarrow}, \omega^6_{\Leftarrow}\}$. At the highest time instant of a forward time window, a delta is materialized resulting in 2 forward deltas $\{\delta^1_{\Rightarrow}, \delta^2_{\Rightarrow}\}$. Whereas, a delta is materialized at the lowest time instant of every backward time window except the last time window where a snapshot is materialized resulting in 2 backwards deltas $\{\delta^4_{\Leftarrow}, \delta^5_{\Leftarrow}\}$ and snapshot $\{S^6\}$. Note that, the subtractive relation $\ominus$ operating on two snapshots S and S' s.t. $S \ominus S'$ results in the minimum number of graph updates that permits the transformation of S in to S'. Half of the time windows is stored in forward fashion whereas the othe'r half is stored in a backward fashion. Suppose a query with a requested time instant t. If t falls within the time interval of time window $\omega^2_{\Rightarrow}$, we start the search in a forward fashion by fetching $\omega^1_{\Rightarrow}$, then fetching $\omega^2_{\Rightarrow}$ whose timestamp is lower than t. Whereas, if t falls within the time interval of the time window $\omega^5_{\Leftarrow}$, we construct the result in a backward fashion. That is, we start by fetching S^6 then $\delta^5_{\Leftarrow}$ and finally $\omega^5_{\Leftarrow}$. Note that, in the *Copy+Log* method all the time windows are considered as forward.

In the following, we describe the key components of the δ-*Copy+Log* approach.

Time buckets: We store the history of the graph in a sequence of temporally disjoint time buckets s.t. each time bucket is a logical container of M time windows and their corresponding checkpoints. That is, a checkpoint can be either a delta or a snapshot. Now, we store a snapshot that is valid at the highest time instant of the last time window of a each bucket, whereas we store a delta at the ending time instant of other time windows. Besides, the first $M/2$ time windows are constructed in a forward fashion and ends each with a forward delta. Whereas, the rest of the time windows are constructed in a backward fashion and ends each with a backward delta.

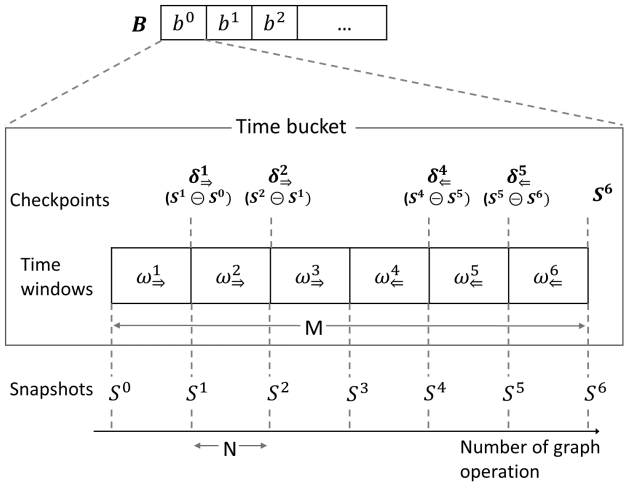


Fig. 13: The internals of the Copy+Log and δ-Copy+Log showing a time bucket (b^0) with $M = 6$, forward and backwards time windows ($\omega^i_{\Rightarrow}$, $\omega^i_{\Leftarrow}$) , deltas ($\delta^i_{\Rightarrow}$, $\delta^i_{\Leftarrow}$) and snapshot S^i

Time windows: We use time windows as physical containers for sets of N graph operations each. There are two types of time windows: forward and backward. A forward time window $\omega^i_{\Rightarrow}$ has graph operations sorted in ascending order of their timestamps, while a backward time window $\omega^i_{\Leftarrow}$ has operations sorted in decreasing order of their timestamps after they have been reversed.

Snapshots: A snapshot represents a valid state at the end of the last time window within a bucket. For node or relationship labels, the snapshot includes all existing nodes and relationships at the snapshot time. For dynamic properties, the snapshot includes all nodes and relationships with that property and their latest value before or at the snapshot time.

Deltas: A delta is defined as the minimum number of graph updates required to transform snapshot S into snapshot S'. In other words, if a graph entity is both added and subsequently deleted, these operations will cancel each other out and will not be included in the delta.

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.

6.1 Space and time complexity analysis

This section analyzes the space and time complexities of δ-*Copy+Log*, *Log*, and *Copy+Log* methods, taking into account system and graph parameters such as γ, N, M, c_1, c_2, r_1, r_2, and p_d. These parameters respectively correspond to the set of all graph operations, the number of graph operations in a time window, the number of time windows in a bucket, the size of a single graph operation or element, the time taken to read a graph operation or element, and the probability of deleting a graph element. Note that due to space limitations, this section presents some formulas without detailed explanations of their derivation. A more comprehensive complexity analysis is present in our previous paper [30].

The space usage of the δ-*Copy+Log* is the sum of the space occupied by graph operations, deltas and snapshots. 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) which implies 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 simplified version of the expand operator for point-based queries. Note that point-based queries are those addressing

a single graph snapshot. The expand operator ($\uparrow_\tau (v)$) retrieves all the relationships of a node v whose validity intervals contain the time instant τ.
Execution time of the expand operator: In our analysis, we consider the worst-case execution time of the operator, which involves reading from the snapshot whose timestamp is closest to τ. This, in turn, requires reading all operations in the deltas of the selected time bucket whose time interval is before τ, resulting in the reading of $((\frac{M}{2}-1)N)$ graph operations where M and N refer to the number of time windows between snapshots and the number of graph operations in each time window, respectively. Finally, we need to read all the graph operations in the time window that follows the last selected delta. Based on this, we derive the following:

$$T_{\delta-CL}\left(\uparrow_\tau (v)\right) = \left(r_2 + \left(\frac{M}{2}-1\right) N r_1 + N r_1\right)$$

The expansion of 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 incur 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:

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

This analysis validates that δ-*Copy+Log* presents a compromise between the *Log* and *Copy+Log* methods. We specifically emphasize analyzing the time complexity of the expand operator as the basis for comparing the time complexity of the δ-*Copy+Log* approach with traditional methods such as *Log* and *Copy+Log*.

7 Overview

This section provides the details of the integration of our temporal graph query language, query processor, and storage technique into Clock-G. Hence, we present in the following the different components composing the architecture of Clock-G (Figure 14).

Request Handler The request handler is responsible for managing the read and write requests. As presented in Figure 14, the request handler comprises two functional components: Reader and Writer.

Our proposed language, T-Cypher, requires the reader to process temporal graph queries using three components: query extractor, planner, and evaluator. The query extractor converts T-Cypher queries into a system-recognized query

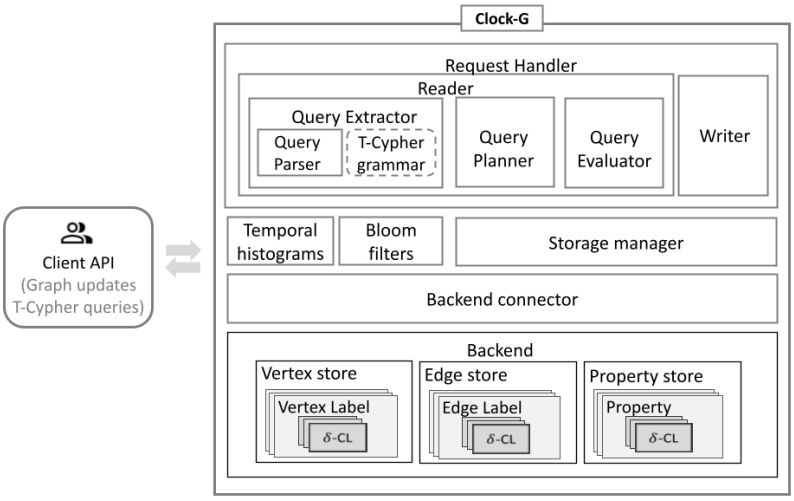


Fig. 14: Overview of the system architecture of Clock-G

object using our proposed T-Cypher grammar. The query planner uses our plan selection algorithm to convert the query object into an execution plan that minimizes estimated cardinality of sub-results. Cardinality estimation is based on a cost model and temporal histograms. The query evaluator executes operators using a pool of atomic executors that share intermediate results.

The writer is responsible for inserting graph updates, which involves informing the storage manager which holds the meta data of the δ-*Copy+Log* technique of the insertion. Then, the writer sends an insertion request to the backend connector, which translates the request into atomic write operations for the backend store.

Backend store The backend store is responsible for storing the temporal graphs following our proposed storage technique the δ-*Copy+Log*. We rely on the column-oriented database Apache Cassandra [28] 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 14 for simplicity).

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 and Bloom filters. The temporal histograms represent the evolution of the cardinality of graph elements through time.

Client API Clock-G offers a client API enabling a client to connect, ingest graph updates, or query the stored graphs. 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.

8 Evaluation

This section evaluates the performance of Clock-G, aiming to demonstrate that *δ-Copy+Log* provides a balance between the traditional methods *Copy+Log* and *Log* in terms of space usage and evaluation time. It also shows that Clock-G can be tuned to account for acceptable query latency and storage resources. This section also confirms the cost-effectiveness of our query optimizer for T-Cypher queries.

8.1 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[8] 4.4.

Datasets We evaluated our proposed methods on synthetic and real temporal graphs to validate their performance. Synthetic datasets were generated with varying probabilities of addition, resulting in three datasets, DS_{p_a}, with p_a values of 0.9, 0.75, and 0.6. These datasets allowed us to analyze the space reduction achieved by *δ-Copy+Log* through the elimination of redundant graph elements across snapshots.

[8] https://neo4j.com

We also used different real-world datasets such as DBLP dataset (DS_{DBLP} [26], Stack overflow dataset (DS_{stack}) and Wiki talk dataset (DS_{wiki}) [42]. To evaluate the sequential paths (presented in Section 4), we used the CitiBike dataset[9] (DS_{citi}) which includes information of bike trips between stations in New York city.

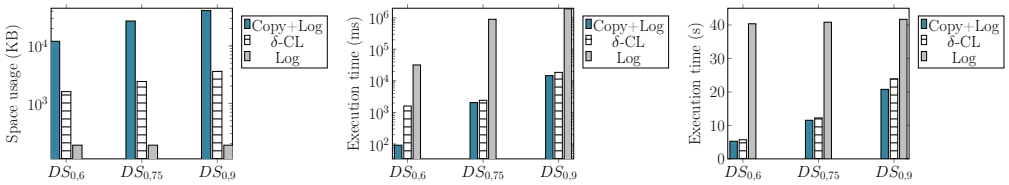
We present some of the characteristics of the generated datasets in Table 3 where $|V|$ refers to the total number of vertices, $|E|$ refers to the total number of graph operations.

Table 3: Characteristics of the generated graphs

Dataset	$\|V\|$	$\|E\|$	Space usage (GB)
DS_{pa}	500 K	10 M	0.315
DS_{stack}	2.6 M	63.4 M	1.7
DS_{DBLP}	1.8 M	29.5 M	0.831
DS_{wiki}	1.1 M	7.8 M	0.173
DS_{citi}	1 K	2.5 M	0.066
$LDBC_0$	4.2 K	12 K	0.003
$LDBC_1$	406.3 K	1.9 M	0.1
$LDBC_2$	1.1 M	3.9 M	0.3

The storage technique was evaluated using previously described datasets, but a dataset with different relationship and node labels, and time-evolving properties that change over time was required to evaluate complex T-Cypher queries using our query processor. To perform these evaluations, we used the LDBC dataset [9], which represents a temporal social graph where people know each other or like each other's posts and comments. However, the original LDBC schema did not account for dynamic properties, which were necessary for our testing requirements. Thus, we modified the schema by transforming some outgoing relationship types and target nodes into dynamic properties attached to the incident nodes. The modified schema includes nodes for people, posts, and comments, with relationships of type likes or knows. Each node and relationship has a starting and ending time instant that defines the boundaries of the validity time interval of each graph entity, as well as a set of dynamic and static properties that characterize it. For example, the property university of a person node was originally a relationship connected to that node and another university node, which 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 present the characteristics of the generated LDBC graphs ($LDBC_0$, $LDBC_1$, and $LDBC_2$) in Table 3.

[9] https://ride.citibikenyc.com/system-data



(a) Space usage (b) 5-Hop queries (c) Global queries

Fig. 15: Comparison with state-of-the-art techniques

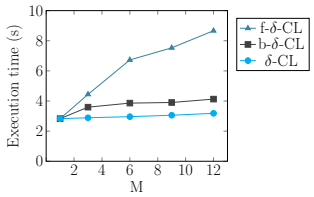


Fig. 16: Evaluation of 8 Hop queries with f-δ-CL, b-δ-CL and δ-CL methods on dataset $DS_{0,6}$ with N set to $10K$

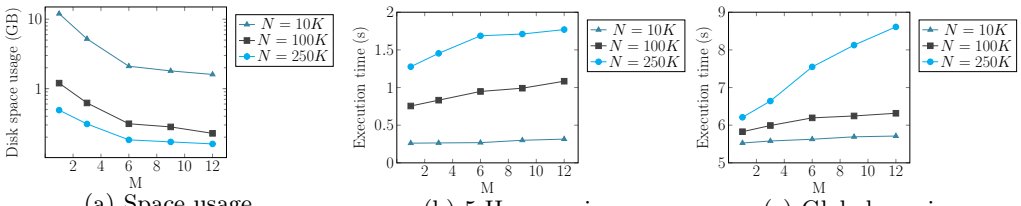
8.2 Space usage and query evaluation time of basic temporal queries

We evaluate disk space usage and query execution time with different system parameter configurations using basic temporal queries such as **local**, **global**, **point**, and **range** queries. It should be noted that **local** and **global** queries request the local neighborhood of a single query and the state of the entire graph, respectively. **Point** and **range** queries retrieve a point or global state that was valid at a single time instant and during a time range, respectively. These queries can be easily written using the T-Cypher's syntax. In this experiment, local queries start from $1k$ randomly selected vertices, while global queries retrieve snapshots of the graph at uniformly chosen time instants within the time span of the datasets. This evaluation focuses solely on storage technique performance, without using a query processor. More complex T-Cypher queries are evaluated in Section 8.3.

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*. Now, the implementation of *Copy+Log* in Clock-G is fairly straightforward since it consists of setting parameter M to 1, while implementing *Log* involves creating unbounded time windows.

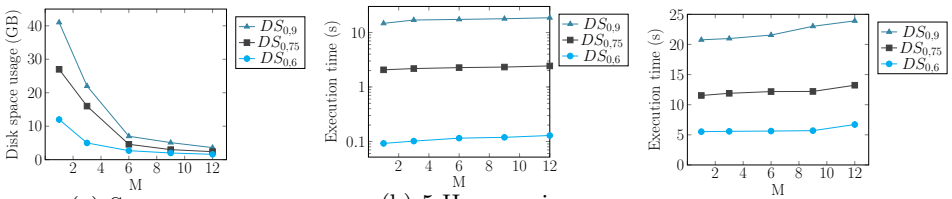
Figures 15(a), 15(b) and 15(c) display the space usage, the execution time of 5-Hops and global queries on datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$. Note that, we set the system parameters N and M to $10k$ and 12, respectively.

The results clearly demonstrate that the proposed δ-*Copy+Log* method provides a balance between the *Log* and *Copy+Log* approaches. It reduces space



(a) Space usage (b) 5-Hop queries (c) Global queries

Fig. 17: 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 $DS_{0,6}$

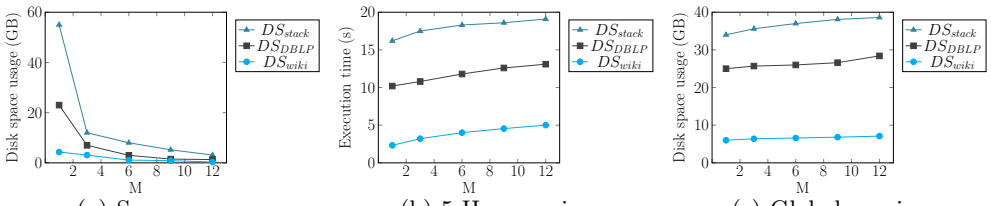


(a) Space usage (b) 5-Hop queries (c) Global queries

Fig. 18: 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

usage by a factor of 12 compared to *Copy+Log*, and query execution time by a factor of 340 compared to *Log* for $DS_{0.75}$.

Validating the use of Bloom filters. In this evaluation test, we compare the execution time using 3 methods namely: f-δ-CL, b-δ-CL and δ-CL. The f-δ-CL method 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 the δ-*Copy+Log* method, hence, consists of adding forward and backward time windows and deltas to the b-δ-CL. Comparing these methods emphasizes the gain of adding Bloom filters and that of storing backward time windows and deltas, separately. Figure 16 shows the execution time of traversal queries with fixed depth 8 on dataset $DS_{0,6}$ while increasing the system parameter M from 1 to 12. The f-δ-CL method significantly increases the execution time with increasing M, but adding Bloom filters to the b-δ-CL reduces the execution time, with a speedup of 52% for $M = 12$. Adding forward and backward time windows and deltas to the δ-CL speeds up traversals by 23% compared to the b-δ-CL. The f-δ-CL method has an overhead of 206% when M is increased from 1 to 12, which is reduced to 12.5% when using the δ-CL.



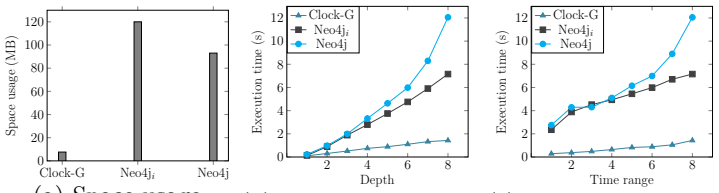
(a) Space usage (b) 5-Hop queries (c) Global queries

Fig. 19: 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}

Variation of N ***and*** M***.*** In this evaluation test, we study the effect of system parameters N and M on disk space usage and query execution time. Figure 17(a) shows that increasing M while fixing N significantly reduces space usage compared to the *Copy+Log* method. Smaller values of N result in higher space usage of checkpoints. Increasing M induces more significant disk space gain for smaller values of N. The execution time of 5-Hop and global queries is also evaluated for different configurations of N and M, with results shown in Figures 17(b) and 17(c). A higher value of N results in higher execution time because fewer checkpoints are created.

Variation of p_a ***and*** M***.*** In this evaluation test, we study the effect of varying the linkage probability of datasets DS_{p_a} and the system parameter M on the space usage and query execution time. We display in Figure 18(a) the space occupied by checkpoints for datasets $DS_{0,6}$, $DS_{0,75}$, and $DS_{0,9}$ for different system configurations where M ranges from 1 to 12. Our results indicate that increasing M leads to a decrease in space usage, and graphs with higher probability of additions provide better space gains. This is because snapshots of such graphs consume more space, making the replacement with deltas more significant in terms of space gain. We also analyze the impact of varying p_a and M on the execution time of 5-Hop and global queries, as shown in Figures 18(b) and 18(c). We find that increasing p_a leads to an increase in query execution time, as higher node degrees result in more computations to evaluate query results.

Evaluation on real datasets. We assess the space efficiency of ingesting real-world datasets using the δ-*Copy+Log* method, with results shown in Figure 19(a). The space usage of checkpoints created by ingesting datasets DS_{stack}, DS_{DBLP}, and DS_{wiki} into Clock-G reduces significantly when increasing the value of M from 1 to 12. Furthermore, we evaluate 5-Hop traversal and global queries on these real-world datasets, and the results in Figures 19(b) and 19(c) demonstrate that our solution significantly reduces space usage while adding only a slight query execution time overhead, as compared to the *Copy+Log* method.



(a) Space usage (b) Varying the depth (c) Varying the range

Fig. 20: Evaluation of the space usage and query execution time of Clock-G, Neo4j and Neo4j$_i$

Comparison with a non-temporal graph database. In this study, we compare the performance of Clock-G with a commercial graph database Neo4j. To enable the storage and evaluation of temporal graphs in Neo4j, we created a temporal layer by adding validity intervals to each node and relationship occurrence. We tested two implementations: Neo4j without indexes and Neo4j$_i$ with indexes where we add indexes to the starting and ending time instants (***tStart*** and ***tEnd***) of nodes, relationships, and properties [10]. We ingested dataset DS_{citi} in all three systems and evaluated a time increasing path query for each node and for depths 1 to 8 and time ranges 1 hour to 8 hours.

Figure 20(a) compares the space usage of Clock-G with those of Neo4j and Neo4j$_i$. It is evident from the figures that Clock-G consumes less space than Neo4j and Neo4j$_i$.

The figures 20(b) and 20(c) show the performance of time increasing path queries with varying depth and time range, comparing the execution time of Clock-G with that of Neo4j and Neo4j$_i$. Results show that Clock-G performs better than the alternative solutions, especially with increasing depth and time range. Clock-G uses parallelism to compute query results and trims the search space to the requested time interval, which is not possible with Neo4j and Neo4j$_i$.

The experiment results emphasize the importance of developing v a graph management system that natively supports temporal data, rather than relying on a non-temporal commercial system.

8.3 Query execution time of complex temporal queries

In this section, we evaluate the performance of our query processor by presenting the execution time of T-Cypher queries with the best, random, and worst plan selection. The best execution plan is selected using a greedy algorithm presented in Algorithm 1. For the worst execution plan, a modified version of this algorithm is used where the most expensive algebraic operator is selected at each iteration. Similarly, a random plan is computed using the same algorithm, but the algebraic operator is chosen randomly at each iteration.

[10] We use the built-in Neo4j's indexing utility to include indexes on the properties *tStart* and *tEnd*

Queries We ran these tests with several T-Cypher queries listed below. Note that all these queries apply to a time interval covering the full history of the LDBC datasets.

```
Query Q0
RANGE_SLICE [2009-01-01T08:00:00Z; 2020-01-01T10:00:00Z]
MATCH (p1:person) -[k1:knows]-> (p2:person)
RETURN p1, p2

Query Q1
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 Q2
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 Q3
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 Q4
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 Q5
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
```

Q_0 returns the pairs of persons who knew each other in the time interval. Q_1 returns the person's 2-hop friendship paths. Q_2 returns the person's 3-hop friendship paths. 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. Q_4 returns all friends of friends of each person such that the intermediate person likes a post. Q_5 returns the friends who like the same post. Some of these queries include graph entities with varying levels of granularity. For example, the *knows* relationships are more selective than the *likes* relationships. In such cases, it is reasonable for the query processor to prioritize loading the "knows"

relationships prior to the "likes" relationships during evaluation in queries $Q4$ and $Q5$.

Plan selection This section shows the best, random, and worst execution plans of Query Q_3 due to space constraints. Notably, the dataset exhibits a lower cardinality for the *person* label compared to the *post* label, and a higher cardinality for the *like* relationship than the *know* type.

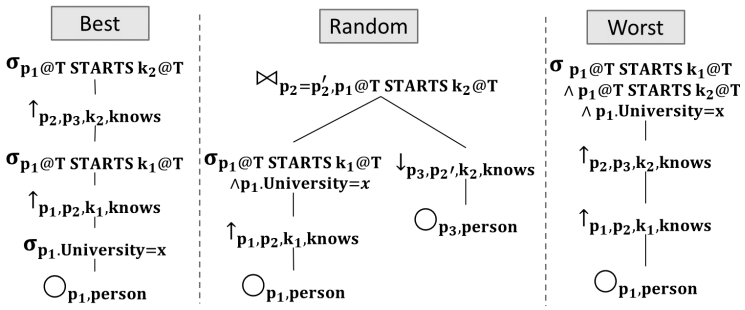


Fig. 21: Best, random, and worst evaluation plan of Query Q_3

We present the evaluation plans for Query Q_3 in Figure 21. The optimal plan begins by retrieving nodes p_1 and selecting those who attended the specified *university*. It then expands nodes p_1 with relationship k_1, selects those who began at the *university*, expands again with relationship k_2, and selects those who began with p_1. As predicted, the worst plan postpones selections until the end. The depicted random plan computes two subparts of the query and joins the results. The first subpart retrieves nodes p_1 and their direct neighbors p_2 connected via relationship k_1, selecting only those who studied at the given *university*. The second subpart retrieves nodes p_3 and their direct neighbors p_2' connected via relationship k_2. The two subparts are then joined based on the condition that p_2 should equal p_2' and the temporal condition that k_2 should be started by p_1.

Figure 22 displays the average execution time resulting from the best, random, and worst plan selection strategies for computing queries $\{Q_0, \ldots, Q_5\}$ on datasets LDBC_0, LDBC_1, and LDBC_2. Note that these results correspond to the average computation time of 10 repetitions.

The execution time of queries Q_0, Q_1, and Q_2 increases with the number of traversed hops for all evaluated datasets, but the best, random, and worst evaluation plans show negligible differences. This difference is expected, as all node and relationship variables in these queries share the same label and type, resulting in the same cardinality. In contrast, the execution time for queries Q_3, Q_4, and Q_5 differs significantly between the best and worst evaluation plans since these queries present graph entities of different granularity.

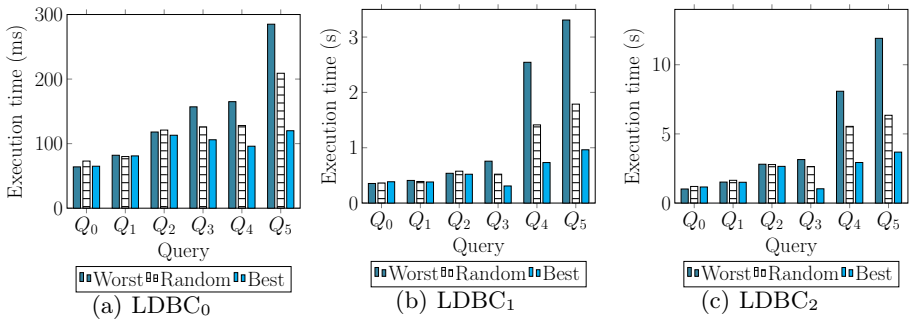


Fig. 22: Comparison between the execution time of T-Cypher queries with worst, random, and best execution plans

The worst plan selection strategy delays the selection process until the end and begins with the nodes having the highest cardinalities. This negatively impacts the cost of query evaluation. On the other hand, selecting plans randomly provides a balance between the best and worst plan selection strategies in terms of query execution time. This is because there are several alternative query plans that fall between the best and worst plan selections. Therefore, these results demonstrate the effectiveness of our plan selection algorithm and cost model.

Comparison with Neo4j We compared Clock-G with a non-temporal graph system by introducing a temporal layer on top of Neo4j. This layer handles the temporal dimension by storing time instants for graph updates and converting T-Cypher queries into Cypher queries.

We compared the performance of Neo4j and Clock-G in executing queries $\{Q_0, \ldots, Q_5\}$ using Algorithm 1 and present the results in Figure 23. Clock-G outperforms Neo4j by up to 80% due to its ability to prune the search space and directly search within selected time windows, snapshots, and deltas.

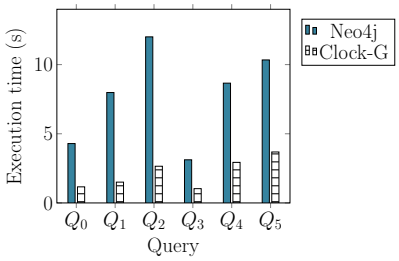


Fig. 23: Comparison between the execution time of T-Cypher queries with Neo4j and Clock-G on LDBC$_2$

9 Conclusion

In this paper, we presented Clock-G, a temporal graph management system with a holistic approach to covering query language, processing, and storage. T-Cypher is our user-friendly query language that allows for temporal constraints on graph pattern matching and navigational queries. Our query processor evaluates T-Cypher queries and targets the minimization of the processing cost. To address this, our processor uses a graph algebra that defines the algebraic operators of a plan, cost model that defines the cost of each operator, and temporal histograms that preserve the cardinality's evolution. Our storage technique, *δ-Copy+Log*, targets the reduction of the space usage of the traditional *Copy+Log* technique by storing deltas instead of full graph snapshots. Tests on synthetic and real-world graphs show that *δ-Copy+Log* significantly reduces space usage and execution time compared to traditional methods and validates the efficiency of our query processor.

A promising direction for future work is the incorporation of the spatial dimension into Clock-G, as it has been the subject of research in the area of spatio-temporal databases [14,35]. To further enhance the capabilities of the Thing'in platform, we can explore spatio-temporal queries, such as expressing a geographic region in which objects should be located during a specified time interval.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM **26**(11), 832–843 (1983)
2. Arenas, M., Bahamondes, P., Aghasadeghi, A., Stoyanovich, J.: Temporal regular path queries. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE). pp. 2412–2425 (2022). https://doi.org/10.1109/ICDE53745.2022.00226
3. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry, pp. 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/978-3-662-03427-9_1, https://doi.org/10.1007/978-3-662-03427-9_1
4. Castelltort, A., Laurent, A.: Representing history in graph-oriented nosql databases: A versioning system. In: Eighth International Conference on Digital Information Management (ICDIM 2013). pp. 228–234. IEEE (2013)
5. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: 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, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2484425.2484442, https://doi.org/10.1145/2484425.2484442
6. Clifford, C.S.J.J., Elmasri, R., Dyreson, C., Kline, F.G.W.K.N., Lorentzos, N., Mitsopoulos, Y., Montanari, A., Pernici, D.N.E.P.B., Sarda, J.F.R.N.L., Segev, M.R.S.A.: A consensus glossary of temporal database concepts. SIGMOD record **23**(1) (1994)
7. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. The VLDB Journal **30**(5), 825–858

(Sep 2021). https://doi.org/10.1007/s00778-021-00675-4, https://doi.org/10.1007/s00778-021-00675-4
8. Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., Lindaaker, T., Marsault, V., Martens, W., Michels, J., Murlak, F., Plantikow, S., Selmer, P., van Rest, O., Voigt, H., Vrgoč, D., Wu, M., Zemke, F.: Graph pattern matching in gql and sql/pgq. In: Proceedings of the 2022 International Conference on Management of Data. p. 2246–2258. SIGMOD '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3514221.3526057, https://doi.org/10.1145/3514221.3526057
9. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The ldbc social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. p. 619–630. SIGMOD '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2742786, https://doi.org/10.1145/2723372.2742786
10. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data. pp. 1433–1445 (2018)
11. George, B., Kang, J.M., Shekhar, S.: Spatio-temporal sensor graphs (stsg): A data model for the discovery of spatio-temporal patterns. Intelligent Data Analysis **13**(3), 457–475 (2009)
12. Ghrab, A., Romero, O., Skhiri, S., Vaisman, A.A., Zimányi, E.: GRAD: on graph database modeling. CoRR **abs/1602.00503** (2016), http://arxiv.org/abs/1602.00503
13. Gubichev, A.: Query processing and optimization in graph databases. Ph.D. thesis, Technische Universität München (2015)
14. Hadjieleftheriou, M., Kollios, G., Gunopulos, D., Tsotras, V.J.: On-line discovery of dense areas in spatio-temporal databases. In: Hadzilacos, T., Manolopoulos, Y., Roddick, J., Theodoridis, Y. (eds.) Advances in Spatial and Temporal Databases. pp. 306–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
15. Haeusler, M., Trojer, T., Kessler, J., Farwick, M., Nowakowski, E., Breu, R.: Chronograph: A versioned tinkerpop graph database. In: International Conference on Data Management Technologies and Applications. pp. 237–260. Springer (2017)
16. Han, W., Li, K., Chen, S., Chen, W.: Auxo: a temporal graph management system. Big Data Mining and Analytics **2**(1), 58–71 (2018)
17. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: a graph engine for temporal graph analysis. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–14 (2014)
18. Hartig, O., Pérez, J.: Semantics and complexity of graphql. In: Proceedings of the 2018 World Wide Web Conference. p. 1155–1164. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2018). https://doi.org/10.1145/3178876.3186014, https://doi.org/10.1145/3178876.3186014
19. Hartmann, T., Fouquet, F., Jimenez, M., Rouvoy, R., Le Traon, Y.: Analyzing complex data in motion at scale with temporal graphs (2020)
20. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: Proceedings of the 26th International Conference on Scientific and Statistical Database Management. SSDBM '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2618243.2618282, https://doi.org/10.1145/2618243.2618282

21. Hölsch, J., Grossniklaus, M.: An algebra and equivalences to transform graph patterns in neo4j. In: Palpanas, T., Stefanidis, K. (eds.) Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016). No. 1558 in CEUR Workshop Proceedings (2016), http://ceur-ws.org/Vol-1558/paper24.pdf
22. Kempe, D., Kleinberg, J., Kumar, A.: Connectivity and inference problems for temporal networks. Journal of Computer and System Sciences **64**(4), 820–842 (2002). https://doi.org/https://doi.org/10.1006/jcss.2002.1829, https://www.sciencedirect.com/science/article/pii/S0022000002918295
23. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 997–1008. IEEE (2013)
24. Khurana, U., Deshpande, A.: Storing and analyzing historical graph data at scale. arXiv preprint arXiv:1509.08960 (2015)
25. Koloniari, G., Souravlias, D., Pitoura, E.: On graph deltas for historical queries. arXiv preprint arXiv:1302.5549 (2013)
26. Kunegis, J.: Konect: The koblenz network collection. p. 1343–1350. WWW '13 Companion, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2487788.2488173, https://doi.org/10.1145/2487788.2488173
27. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J.H., Han, W.S.: The g* graph database: efficiently managing large distributed dynamic graphs. Distributed and Parallel Databases **33**(4), 479–514 (2015)
28. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
29. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: Llama: Efficient graph analytics using large multiversioned arrays. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 363–374. IEEE (2015)
30. Massri, M., Miklos, Z., Raipin, P., Meye, P.: Clock-g: A temporal graph management system with space-efficient storage technique. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE). pp. 2263–2276. IEEE Computer Society, Los Alamitos, CA, USA (may 2022). https://doi.org/10.1109/ICDE53745.2022.00215, https://doi.ieeecomputersociety.org/10.1109/ICDE53745.2022.00215
31. Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., Chen, W.: Immortalgraph: A system for storage and analysis of temporal graphs. ACM Transactions on Storage (TOS) **11**(3), 1–34 (2015)
32. Moffitt, V.Z., Stoyanovich, J.: Temporal graph algebra. DBPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3122831.3122838, https://doi.org/10.1145/3122831.3122838
33. Montanari, A., Chomicki, J.: Time Domain, pp. 3103–3107. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-39940-9_427, https://doi.org/10.1007/978-0-387-39940-9$_$427
34. Pan, R.K., Saramäki, J.: Path lengths, correlations, and centrality in temporal networks. Physical Review E **84**(1), 016105 (2011)
35. Perry, M., Jain, P., Sheth, A.P.: SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries, pp. 61–86. Springer US, Boston, MA (2011). https://doi.org/10.1007/978-1-4419-9446-2_3, https://doi.org/10.1007/978-1-4419-9446-2_3

36. Ramesh, S., Baranawal, A., Simmhan, Y.: A distributed path query engine for temporal property graphs. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). pp. 499–508 (2020). https://doi.org/10.1109/CCGrid49817.2020.00-43
37. Redmond, U., Cunningham, P.: Subgraph isomorphism in temporal networks. CoRR **abs/1605.02174** (2016), http://arxiv.org/abs/1605.02174
38. Rizzolo, F., Vaisman, A.A.: Temporal xml: modeling, indexing, and query processing. The VLDB Journal—The International Journal on Very Large Data Bases **17**(5), 1179–1212 (2008)
39. Rost, C., Gomez, K., Täschner, M., Fritzsche, P., Schons, L., Christ, L., Adameit, T., Junghanns, M., Rahm, E.: Distributed temporal graph analytics with gradoop. The VLDB Journal **31**(2), 375–401 (Mar 2022). https://doi.org/10.1007/s00778-021-00667-4, https://doi.org/10.1007/s00778-021-00667-4
40. Semertzidis, K., Pitoura, E., Lillis, K.: Timereach: Historical reachability queries on evolving graphs. In: EDBT (2015)
41. Semertzidis, K., Pitoura, E., Terzi, E., Tsaparas, P.: Best friends forever (BFF): finding lasting dense subgraphs. CoRR **abs/1612.05440** (2016), http://arxiv.org/abs/1612.05440
42. Wen, D., Huang, Y., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Efficiently answering span-reachability queries in large temporal graphs. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE). pp. 1153–1164 (2020). https://doi.org/10.1109/ICDE48307.2020.00104
43. Wu, H., Huang, Y., Cheng, J., Li, J., Ke, Y.: Reachability and time-based path queries in temporal graphs. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE). pp. 145–156 (2016). https://doi.org/10.1109/ICDE.2016.7498236
44. Xiangyu, L., Yingxiao, L., Xiaolin, G., Zhenhua, Y.: An efficient snapshot strategy for dynamic graph storage systems to support historical queries. IEEE Access **8**, 90838–90846 (2020)
45. Zhang, T., Gao, Y., Qiu, L., Chen, L., Linghu, Q., Pu, S.: Distributed time-respecting flow graph pattern matching on temporal graphs. World Wide Web **23**(1), 609–630 (Jan 2020). https://doi.org/10.1007/s11280-019-00674-0, https://doi.org/10.1007/s11280-019-00674-0