Towards an efficient approach to manage graph data evolution: conceptual modelling and experimental assessments

Landy Andriamampianina, Franck Ravat, Jiefu Song, Nathalie Vallès-Parlangeau

Towards an efficient approach to manage graph data evolution: conceptual modelling and experimental assessments

Landy Andriamampianina[1,2][0000−0001−9636−7007], Franck Ravat[1][0000−0003−4820−841X], Jiefu Song[1,2][0000−0002−2066−7051], and Nathalie Vallès-Parlangeau[1][0000−0002−4463−5177]

[1] IRIT-CNRS (UMR 5505) - Université Toulouse 1 Capitole (UT1), 2 Rue du Doyen Gabriel Marty F-31042 Toulouse Cedex 09
firstname.lastname@irit.fr

[2] Activus Group, 1 Chemin du Pigeonnier de la Cépière, 31100 Toulouse
firstname.lastname@activus-group.fr

Abstract. This paper describes a new temporal graph modelling solution to organize and memorize changes in a business application. To do so, we enrich the basic graph by adding the concepts of *states* and *instances*. Our model has first the advantage of representing a complete temporal evolution of the graph, at the level of: (i) the graph structure, (ii) the attribute set of entities/relationships and (iii) the attributes' value of entities/relationships. Then, it has the advantage of memorizing in an optimal manner evolution traces of the graph and retrieving easily temporal information about a graph component. To validate the feasibility of our proposal, we implement our proposal in Neo4j, a data store based on property graph model. We then compare its performance in terms of storage and querying time to the classical modelling approach of temporal graph. Our results show that our model outperforms the classical approach by reducing disk usage by 12 times and saving up to 99% queries' runtime.

Keywords: Temporal graph · Graph snapshots · Temporal evolution · Graph data stores.

1 Introduction

In the real world, entities (i.e. objects, concepts, things with an independent existence) and the relationships between them change over time so information about them also changes. They can evolve over time in terms of (i) their topology (how entities are linked, when entities/relationships are present or absent), (ii) their inherent features (the attributes set that describes an entity or a relationship) and (iii) their status (the values of the set of descriptive attributes at a particular time). Finding and analyzing these evolutions enable to get a deeper understanding of an application notably to exploit temporal correlations and causality [9,3], to make simulations [8] or to make predictions [16]. It is therefore

necessary to be able to manage the temporal evolution of data to exploit them. The management of evolving data implies the following challenges: (i) the formalization of a conceptual model to capture the three aspects of evolution of an application that we mentioned before, (ii) the implementation of the conceptual model into a data store and (iii) the efficient querying of the temporal aspects of an application.

Regarding the first challenge, a growing part of the literature proposes conceptual models, called *temporal graphs*, based on graphs because of their flexible nature. It incorporates time dimension in graphs and captures the temporal evolution of graph data. The basic representation of temporal graphs is the *sequence of graph snapshots*[3]. However, existing snapshots-based solutions focus on specific evolution aspects according to an application's needs. Particularly, they do not capture the addition or deletion of entities and relationships' attributes over time. Regarding the second challenge, to our knowledge, no works formalize translation rules of a conceptual model of temporal graphs into a data store [27,2,24]. Regarding the third challenge, to exploit temporal graphs efficiently, some works propose optimization methods to reduce data redundancy generated by snapshots but they are not effective enough [14,26,28]. Other works focus on the performance of graph data stores supporting temporal graphs [5,10,25,19].

In response to the previous issues, our contribution is three-fold: (i) a generic conceptual model to capture a complete temporal evolution of graph data, (ii) that is directly convertible into a graph data store through formalized translation rules (iii) and that supports efficiently the querying of multiple evolution aspects of a graph. In this paper, first, we discuss the challenges posed by existing works on the management of graph data evolving over time (Section 2). Second, we propose novel concepts of the temporal graph to overcome the limits of existing concepts (Section 3). Finally, we implement our model in a graph data store, namely Neo4j, and compare its performance to the classical snapshot-based implementation and an optimized snapshot-based implementation (Section 4).

2 Related Works

In this section, we analyze existing approaches to manage the evolution of graph data at three levels: conceptual, logical, and physical levels. Then, we present our contributions in relation to existing works.

At the conceptual level, the classical approach to model graph data evolving over time is the sequence of snapshots [14]. It generally consists in sampling graph data periodically to obtain snapshots at a fixed time interval (e.g. per hour, day, month or year) [18]. The advantages of this modelling approach are that it is simple and that it represents accurately the state of the graph at a specific time instance [18]. Nevertheless, existing works based on this approach are limited in taking into account evolution. The evolution of the graph topology[4],

[3] denoted $G_1, G_2, ..., G_T$ where G_i is an image of the entire graph at the time instance i and $[1; T]$ is the timeline of the application

[4] Graph topology is the way in which nodes and edges are arranged within a graph.

i.e. the addition and deletion of edges only [27,2] or both nodes and edges [24] over time, is the most studied evolution type. To meet more complex needs, some models include attributes of nodes [6,7] or both edges and nodes' attributes [11] to capture the temporal evolution in their values. In the previous cited works, they generally consider the attribute set of nodes or edges as fixed while it can evolve over time in real-world applications. Indeed, some applications require to model evolution that happened at all levels in a graph [4]. To the best of our knowledge, there is no modelling solutions of temporal graphs including all evolution types in order to be used in any desired application. Last but not least, some works propose a modelling approach of evolving graphs completely in break with snapshots [15,17]. They attach a valid time interval to each graph component (i.e. node or edge) to track the graph evolution. However, as snapshots, they focus on specific evolution types.

At the logical level, property-graph and RDF data models are commonly used in the graph domain. Traditionally, the transformation between the conceptual level and the logical level is done in an automatic way such as in the relational databases domain. However, to our knowledge, this automation is not studied in the domain of graphs. The works that propose a conceptual model completely ignore the formalization of translation rules from the conceptual model to the logical model [27,2,24]. No standard is defined at the present time to guarantee a compliant implementation of a conceptual model of temporal graphs at the logical level.

At the physical level, existing works try to maximize the implementation and query efficiency of temporal graphs. We distinguish two research axis in existing works: data redundancy reduction and performance improvement. Regarding data redundancy reduction, snapshots inevitably introduce data redundancy since consecutive snapshots share in common nodes and edges that do not change over time [14]. There exist optimization techniques to partially reduce data redundancy. For instance, [26] proposes a strategy to determine which snapshots should be materialized based on the distribution of historical queries. [11] introduces an in-memory data structure and a hierarchical index structure to retrieve efficiently snapshots of an evolving graph. [22] proposes a framework to construct a small number of representative graphs based on similarity. These techniques are unfortunately not effective enough.

Regarding the performance improvement, some works focus on the performance of graph data stores supporting evolving graphs via experimental assessments. Some experiments are based on property-graph based NoSQL databases. For instance, [5] uses Neo4j to store time-varying networks and to retrieve specific snapshots. The authors in [10] have developed a graph database management system based on Neo4j to support graphs changing in the value of nodes and edges' properties but with a static structure. Other experiments rely on RDF triple stores, such as Virtuoso[5] or TDB-Jena[6], to store the evolution of Linked Open Data (LOD) in the Semantic Web area [25,19]. It is already known

[5] `https://virtuoso.openlinksw.com/`
[6] `https://jena.apache.org/documentation/tdb/`

that property-graph based NoSQL databases are more efficient than RDF triple stores when querying RDF data [20]. It is necessary to see if property-graph based databases are as efficient in the context of temporal graphs.

Contributions. We propose a complete solution to manage the evolution of graph data. From a conceptual point of view, we propose a graph-based modelling that does better than snapshots by capturing temporal evolution at all levels - the graph topology, the attributes' set and the value of attributes - in order to be implemented in any desired application (Section 3). From a logical point of view, we propose translation rules between our conceptual model and a graph data store to automate the implementation of our model (Section 4). We have decided to focus on a property-graph data store as it provides a more efficient environment for analytical queries than RDF triple stores. From a physical point of view, to highlight the advantage of using our model instead of snapshots, we present a comparative study of the implementation of both models using Neo4j as a property-graph data store. On the one hand, we compare the creation time and space requirements to evaluate the proportion of data redundancy in both models. On the other hand, we compare the querying performance of these implementations based on benchmark queries highlighting the temporal evolution concepts proposed by our model and using the native query language of Neo4j (Section 4).

3 Proposition

In this section, we present our modelling solution of a temporal graph. We keep the concepts of entities and relationships as in basic graphs. We incorporate the notion of time to represent the evolution of entities and their relationships. We model time as linear and discretized according to a time unit. A time unit is a partition of the timeline into a set of disjoint contiguous time intervals.

Definition 1. *A time interval defines a set of instants between two instant limits. We denote it* $T = [t_{begin}, t_{end}]$. *An instant defines a point on a timeline, that is* $T = [t_{begin}, t_{end}]$ *where* $t_{begin} = t_{end}$.

In order to respond to the current needs for capturing multiple evolution types of an application (change in the topology of entities/relationships, in the attributes set of entities/relationships or in the attribute values of entities/relationships), we cannot rely on the current works in the literature. To overcome this limitation, we propose a new model capable of capturing three types of temporal evolution of entities and relationships in an unique representation. In our model, an entity or relationship that evolves over time is modelled through three levels of abstraction: (i) the topology level to capture its presence and absence over time (ii) the state level to capture the evolution in its attributes set and (iii) the instance level to capture the evolution in the value of its attributes.

Instead of attaching time to an entire graph as in snapshots, we attach a valid time interval to each abstraction level of entities/relationships. This valid time interval expresses the validity and existence of the information associated to each abstraction level of entities/relationships inside a certain time interval. This time management method allows to keep the strict and necessary data and then avoid data redundancy.

Definition 2. *A temporal entity, called* $e_i \in E$*, is defined by* $\langle T^{e_i}, id^{e_i}, S^{e_i} \rangle$ *where* T^{e_i} *is the valid time interval of* e_i*,* id^{e_i} *is the identifier of* e_i *and* $S^{e_i} = \{s_1^{e_i}, ..., s_m^{e_i}\}$ *is the non-empty set of states of* e_i*. Each state* $s_j^{e_i} \in S^{e_i}$ *is defined by* $\langle T^{s_j}, A^{s_j}, I^{s_j} \rangle$ *where:*

- T^{s_j} *is the valid time interval of* $s_j^{e_i}$*.*
- $A^{s_j} = \{a_1^{e_i}; ...; a_n^{e_i}\}$ *is the non-empty set of attributes of* $s_j^{e_i}$ *during* T^{s_j}*. It is called the schema of* e_i *during* T^{s_j}*.*
- $I^{s_j} = \{i_1^{s_j}; ...; i_p^{s_j}\}$ *is the non-empty set of instances of* $s_j^{e_i}$ *during* T^{s_j}*. Each instance* $i_k^{s_j} \in I^{s_j}$ *is defined by* $\langle T^{i_k}, V^{i_k} \rangle$ *where:*
  - T^{i_k} *is the valid time interval of* $i_k^{s_j}$*.*
  - $V^{i_k} = \{v(a_1^{e_i}); ...; v(a_n^{e_i})\}$ *is a non-empty set of attributes' values. Each* $v(a_q^{e_i}) \in V^{i_k}$ *is the value of each attribute* $a_q^{e_i} \in A^{s_j}$ *during* T^{i_k}*.*

The highest abstraction level of a temporal entity e_i is the topology level. At this level, a temporal entity evolves only according to its presence or absence in the application reflected by the change of its valid time T^{e_i} over time.

The middle abstraction level of a temporal entity e_i is the state level under which its schema, denoted A^{s_j}, can evolve. At this level, two states of the same entity have different schemas. When a new attribute is added or removed from an entity, a new state is created instead of overwriting the old state version.

The lowest abstraction level of a temporal entity e_i is the instance level. It captures the evolution in the value V^{i_k} of its attributes A^{s_j}. At this level, between two instances of the same entity, the schema is the same but the values of its attributes are different. When the values of an entity's attributes change, a new instance is created instead of overwriting the old instance version.

The changes at the topology and state levels impact the instance level. At the topology level, when an entity is present/absent at a particular time, this translates by the presence/absence of its states and the instances that composed them at this particular time. At the state level, when there is a change in the schema of an entity, a new state is created. Moreover, at least one instance of this state is created. At each change on an abstraction level, this ends the valid time of the last instance at the time of the change and starts the valid time of the new created instance at the time of the change. So the valid times of the abstraction levels higher than the instance level are deduced by calculation as described in following definition.

Definition 3. *The valid time interval of each instance of a temporal entity* $i_k^{s_j} \in I^{s_j}$ *is defined by* $T^{i_k} = [t_{begin}, t_{end}]$ *where* $t_{begin} \neq \emptyset$ *and* $t_{end} \neq \emptyset$*. There is only*

one case where an instance has not yet got a pre-defined ending time: if some instances under the current state $s_m^{e_i}$ are current in the application.

The valid time interval of each state of a temporal entity $s_j^{e_i} \in S^{e_i}$ is obtained by calculation:

$$T^{s_j} = \cup_{k=1}^{k=p} T^{i_k} \text{ where } i_k \in I^{s_j} \quad (1)$$

The valid time interval of each temporal entity $e_i \in E$ is obtained by calculation :

$$T^{e_i} = \cup_{j=1}^{j=m} T^{s_j} \text{ where } s_j \in S^{e_i} \quad (2)$$

The temporal evolution of relationships includes the evolution in the graph topology, in their attribute set and in the value of their attributes. We use the same evolution mechanisms as in temporal entities for relationships. In fact, a temporal relationship is also modelled through the concepts of states and instances.

Definition 4. *A temporal relationship, called r_i, is defined by $\langle T^{r_i}, S^{r_i} \rangle$ where T^{r_i} is the valid time interval of r_i and $S^{r_i} = \{s_1^{r_i}, ..., s_u^{r_i}\}$ is the non-empty set of states of r_i. A state $s_b^{r_i} \in S^{r_i}$ is defined by $\langle T^{s_b}, A^{s_b}, I^{s_b} \rangle$ where:*

- *T^{s_b} is the valid time interval of $s_b^{r_i}$.*
- *$A^{s_b} = \{a_1^{r_i}; ...; a_w^{r_i}\}$ is the non-empty set of attributes of $s_b^{r_i}$ during T^{s_b}. It is called the schema of r_i during T^{s_b}.*
- *$I^{s_b} = \{i_1^{s_b}; ...; i_x^{s_b}\}$ is the non-empty set of instances of $s_b^{r_i}$ during T^{s_b}. Each instance $i_c^{s_b} \in I^{s_b}$ is defined by $\langle T^{i_c}, V^{i_c} \rangle$ where:*
  - *T^{i_c} is the valid time interval of $i_c^{s_b}$.*
  - *$V^{i_c} = \{v(a_1^{r_i}); ...; v(a_w^{r_i})\}$ is a non-empty set of attributes' values. Each $v(a_d^{r_i}) \in V^{i_c}$ is the value of each attribute $a_d^{r_i} \in A^{s_b}$ during T^{i_c}.*

Definition 5. *A temporal relationship is defined based on the same concepts as a temporal entity. The valid time of each state $s_b^{r_i} \in S^{r_i}$, denoted T^{s_b}, is obtained by calculation as in Definition 3. The valid time of each temporal relationship $r_i \in R$, denoted T^{r_i}, is obtained by calculation as in Definition 3. The particularity of a temporal relationship is that it does not have an independent existence contrary to temporal entities. This implies in our modelling that:*

- *An instance $i_c^{s_b} \in I^{s_b}$ is a relationship between a couple of instances (i_h, i_l) belonging respectively to entities e_i and e_j.*
- *The valid time of each instance $i_c^{s_b} \in I^{s_b}$ is defined by $T^{i_c} = [t_{begin}, t_{end}]$ where $t_{begin} \neq \emptyset$ and $t_{end} \neq \emptyset$. There is only one case where it has not yet got a pre-defined ending time: if some instances under the current state $s_u^{r_i}$ are current in the application and both connected entities' instances i_h and i_l do not have yet a pre-defined ending time.*
- *$T^{i_c} d (T^{i_h} \cap T^{i_l})$[7] where T^{i_h} is the valid time of an instance i_h of e_i and T^{i_l} is the valid time of an instance i_l of e_j. $(T^{i_h} \cap T^{i_l}) \neq \emptyset$ because $(T^{i_h} \circ T^{i_l})$[8].*

[7] d is an ALLEN temporal operator to express that a time interval X occurs "during" a time interval Y, i.e. XdY [1].

[8] $\circ$ is an ALLEN temporal operator to express that a time interval X "overlaps" a time interval Y, i.e. $X \circ Y$ [1].

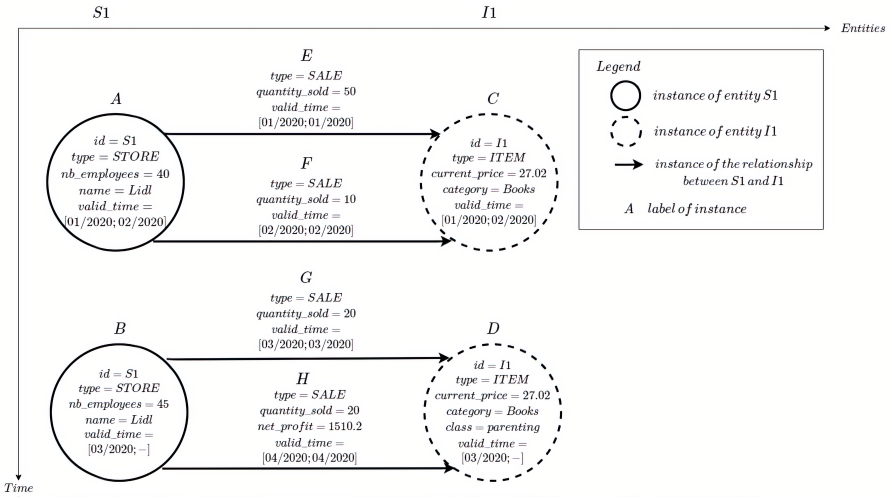

Fig. 1: Management of the temporal evolution of the application in Example 1 with our modelling solution.

Example 1. We propose in this example an application to show the implementation of our modelling concepts. We consider two entities: a store and an item. At the creation of the application at the month 01/2020, the store and the item are characterized by the following set of attributes: an identifier (denoted *id*) and a type (denoted *type*). In addition, the store is described by the number of its employees (denoted *nb_employees*) and a name (denoted *name*). The item is also described by a price (denoted *current_price*) and the category to which it belongs (denoted *category*). These two entities are linked by the sale of the item by the store. This sale is characterized by a type (denoted *type*) and the quantity sold of the item by the store (denoted *quantity_sold*). The store is identified by "$S1$" and named "Lidl". Its type is "Store". It has 40 employees. The item is identified by "$I1$". Its type is "Item". It has a price of 27.02$. Its category is "Books". The type of the sale between of $I1$ by $S1$ is "Sale". The quantity sold of the latter is 50.

Since the application creation, both entities have not evolved until 02/2020. However, the quantity sold of $I1$ by $S1$ has decreased by 40 at the month 02/2020. At the month 03/2020, several evolutions took place. The store $S1$ has recruited 5 employees. The item $I1$ is described by a new attribute called *class*. The item $I1$ has been affected to the class "parenting". The quantity sold of $I1$ by $S1$ has increased by 10. Since the month 03/2020, both entities have not evolved anymore. At the month 04/2020, the sale of $I1$ by $S1$ is described by the new attribute net_profit. It is the net profit made by $S1$ on $I1$. Its value is 1510.2$. Finally, there is no sale of $I1$ by $S1$ since the month 05/2020.

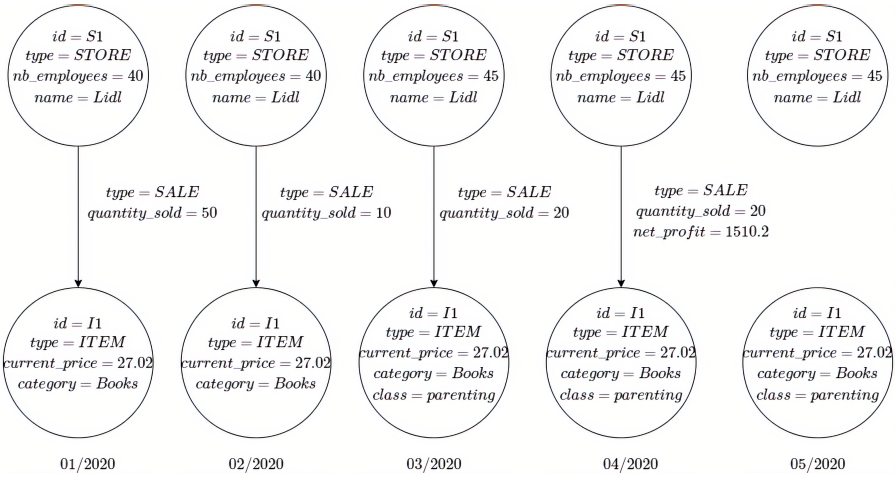

Fig. 2: Management of the temporal evolution of the application in Example 1 with the snapshots-based solution.

To answer the need of capturing these evolutions, we show in Figure 1 the implementation of our modelling concepts to manage the temporal evolution of the application. The sale of $I1$ by $S1$ corresponds to a relationship between the two entities. An instance is illustrated graphically by a node for entities and by an edge for relationships in Figure 1. Similarly, a state is illustrated graphically by a set of nodes for entities and a set of edges for relationships in Figure 1.

Both entities have not changed from 01/2020 to 02/2020 neither in terms of their presence/absence nor their schema nor their attributes' value. At the level of our model, this is translated by one state of $S1$ with the schema $\{id, type, nb\_employees, name\}$. One instance of this state is created to initialize the value of $S1$'s attributes. The latter is illustrated by the node A in Figure 1. Moreover, one state of $I1$ is created with the schema $\{id, type, current\_price, category\}$. One instance of this state is created to initialize the value of $I1$'s attributes. It is illustrated by the node C in Figure 1. Instances A and C have a valid time starting from 01/2020 and ending at 02/2020 during which they did not change.

The first sale of $I1$ by $S1$ generates at the level of our model one state of the relationship between both entities with the schema $\{type, quantity\_sold\}$. One instance of this state, illustrated by the edge E in Figure 1, is created to initialize the value of the sale's attributes. At the month 02/2020, the value of the attribute $quantity\_sold$ of the sale of $I1$ by $S1$ has decreased under the same schema of the instance E. This generates in our model a new instance, labelled F in Figure 1, with the updated value of $quantity\_sold$ under the same state in which the instance E belongs. The valid time of F begins at the time of the change it captures i.e. 02/2020.

At the month 03/2020, $S1$ experienced an increase in the value of the attribute $nb\_employees$ under the same schema of its instance A. This creates at the level of our model a new instance of $S1$, labelled B in Figure 1, with the updated value of $nb\_employees$ under the state in which the instance A belongs. As $S1$ does not change anymore since 03/2020, the valid time of instance B is starting from 03/2020 but its ending date is not specified. It is worthy to notice that $S1$ has never experienced a change in its schema. It has only one state with the schema $\{id, type, nb\_employees, name\}$ and which gathers the instances A and B.

Moreover, at the month 03/2020, the attribute $class$ has been added to the schema of $I1$. From our modelling point of view, this generates a new state with the new schema $\{id, type, current\_price, category, class\}$. A new instance of this state is created to specify the value of $I1$'s attributes. It is illustrated by the node D in Figure 1. As $I1$ does not change anymore since 03/2020, the valid time of instance D is starting from 03/2020 but its ending date is not specified. To sum up, $I1$ has two states: one with the schema $\{id, type, current\_price, category\}$ and composed of the instance C and another one with the schema $\{id, type, current\_price, category, class\}$ and composed of the instance D.

Last but not least, also at the month 03/2020, the value of the attribute $quantity\_sold$ describing the sale of $I1$ by $S1$ has increased under the same schema of instances E and F. This is translated in our model by the creation of a new instance G in Figure 1 with the updated value of $quantity\_sold$. This instance belongs to the state composed of instances E and F. The valid time of G starts at the time of the change it captures i.e. 03/2020. At 04/2020, the attribute net_profit has been added to the schema of the relationship. From our modelling point of view, this generates a new state of the relationship with the new schema $\{type, quantity\_sold, net\_profit\}$. A new instance of this state is created to specify the value of the relationship's attributes. It is illustrated by the edge H in Figure 1. To sum up, the relationship between $I1$ and $S1$ has two states: one with the schema $\{type, quantity\_sold\}$ and composed of three instances (E, F and G) and another one with the schema $\{type, quantity\_sold, net\_profit\}$ and composed of one instance (H). Finally, there is no sale of $I1$ by $S1$ since the month 05/2020. This corresponds in our model to the absence of relationship between $I1$ and $S1$ at the month 05/2020. Consequently, no instance or state of this relationship with a valid time including the month 05/2020 is created.

In a nutshell, our model translates the different evolutions of the application into 1 graph with 4 nodes and 4 edges. We present in Figure 2 the representation of the same application if we would have adopt the snapshot-based approach to manage the temporal evolution. We would have 5 graph snapshots with 10 nodes and 4 edges.

As a result of the previous definitions, our *temporal graph* is defined as follows:

Definition 6. *A Temporal Graph, called G, is defined by* $\langle L, E, R \rangle$ *where:*

- L *is the timeline of the temporal graph,*
- $E = \{e_1, ..., e_g\}$ *is a non-empty set of temporal entities,*
- $R = \{r_1, ..., r_h\}$ *is a non-empty set of temporal relationships.*

Definition 7. *The timeline L of a temporal graph G only depends on the valid times of temporal entities as they have an independent existence. L is obtained by calculation:*

$$L = \cup_{i=1}^{i=g} T^{e_i} \text{ where } e_i \in E \tag{3}$$

4 Experimental evaluation

In this section, we present an experimental comparison of three approaches for modelling an evolving graph: the classical sequence of snapshots, an optimized sequence of snapshots and our temporal graph. As we have discussed in Section 2, the classical snapshots consists in sampling of graph data at a regular time period (here we chose a month). Our optimized snapshots approach consists in creating snapshots only if they differ. In other terms, we create a snapshot only if it includes a change compared to a previous snapshot.

This experiment has two goals: (i) to study the feasibility of our model, i.e. illustrate if our modelling is easily implementable (stored and queried) in a graph-oriented data store and (ii) to study the efficiency of our model by comparing its storage and query performance to the classical sequence of snapshots and the optimized sequence of snapshots.

In Section 4.1, we present the technical environment of our experiment. Then, in Section 4.2, we present the datasets we used for the three implementations. We stored these datasets in Neo4j based on defined translation rules of our model presented in Section 4.3. Finally, we query the three approaches according to different querying criteria and compare their runtime in Section 4.4. We present the details of our experiment at the following web page `https://gitlab.com/2573869/temporal_graph_modelling`.

4.1 Technical environment

The hardware configuration is as follows: PowerEdge R630, 16 CPUs x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40Ghz, 63.91 GB. One virtual machine is installed on this hardware. This virtual machine has 6GB in terms of RAM and 100GB in terms of disk size. We installed on this virtual machine Neo4j (community version 4.1.3) as a data store for our datasets.

4.2 Datasets

To run our experimental comparison, we needed a dataset that reflect realistic applications with temporal evolutions. We therefore used a dataset from a reference benchmark namely TPC-DS[9]. Temporal evolutions exist in this benchmark

[9] `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.13.0.pdf`

Implementations	**# Nodes**	**# Edges**	**# Snapshots**
Temporal graph	112.897	1.693.623	N/A
Classical snapshots	7.405.461	4.207.657	60
Optimized snapshots	5.347.477	4.044.481	53

Table 1: Characteristics of datasets.

and allow us to find all the three types of evolution mentioned in Section 3. We transformed the dataset provided by TPC-DS into three datasets having the temporal graph, the classical snapshots and the optimized snapshots representations. All transformation details of the TPC-DS dataset into the three representations are available on the website `https://gitlab.com/2573869/temporal_graph_modelling`. In Table 1, we found as a result of the transformation steps, the number of nodes, edges and snapshots of the dataset used for each implementation.

4.3 Translation of temporal graph into Neo4j

To evaluate the feasibility of our modelling solution, we searched for translation rules to map our conceptual model of temporal graph into the graph data model supported by Neo4j, the property graph model [23]. In Table 2, we define the translation rules between our model and the property graph. The concepts of our temporal graph are directly translatable into the property graph of Neo4j. An instance of an entity or relationship in our model can be represented respectively by a node and an edge in Neo4j. The value of the attributes set and valid times of instances correspond to the key-value properties in Neo4j. An entity, relationship or state is composed of instances. Then, they are represented by a set of nodes or edges in Neo4j. Schemas and valid times of a state, an entity or relationship can be retrieved by query in Neo4j.

For each implementation, we stored the relative dataset in a database instance of Neo4j. Table 3 shows the size (in GB) and the creation time (in seconds) of each database instance in Neo4j. The two snapshots approaches use a different time management method than our model which leads to larger sizes of their database instances. Our model reduces respectively 12 times and 9 times the size of database instance storing classical snapshots and optimized snapshots. To load the datasets into Neo4j, we designed a program based on the CSV importing system of Neo4j. Again, the datasets based on snapshots approaches require more time to be imported since they contain more nodes and edges than our model (Table 1).

4.4 Query performance

To evaluate the efficiency of our model, we compare its querying performance with the classical snapshots and the optimized snapshots based implementations.

Our model's concepts	Neo4j's concepts
an instance of an entity state $i_k^{s_j}$	a node
an instance of a relationship state $i_c^{s_b}$	an edge
valid time of an entity instance T^{i_k}	two properties*
valid time of a relationship instance T^{i_c}	two properties*
a state of an entity $s_j^{e_i}$	a set of nodes (with different valid times)
a state of a relationship $s_b^{r_i}$	a set of edges (with different valid times)
valid time of an entity state T^{s_j}	by query
valid time of a relationship state T^{s_b}	by query
a schema of an entity state A^{s_j}	by query
a schema of a relationship state A^{s_b}	by query
an entity e_i	a set of nodes (with different valid times)
a relationship r_i	a set of edges (with different valid times)
an attribute of an entity $a_q^{e_i}$	a property
an attribute of a relationship $a_d^{r_i}$	a property
a temporal entity's identifier id^{e_i}	a property
valid time of an entity T^{e_i}	by query
valid time of a relationship T^{r_i}	by query

Table 2: Translation rules of our model into Neo4j. *$start\_valid\_time$ and end_valid_time.

Implementations	Size (in GB)	Creation time (in sec)
Temporal graph	0,3	15,795
Classical Snapshots	3,7	56,529
Optimized snapshots	2,8	45,827

Table 3: Size and creation time of graph database instances.

To do so, we created benchmark queries that cover a large range of scenarios to get insights about the temporal aspects of a graph (Table 4). We classified them into the following criteria: the entity scope, the time scope, the temporal evolution type and the operations type [12,13]. Then, we translated these benchmark queries in the native query language of Neo4j: Cypher. Finally, we recorded for each benchmark query its execution time which is the elapsed time in seconds for processing the query (Figure 3). We run each query ten times and take the mean time of all runs as final execution time. To avoid any bias in the disk management and querying performance, we do not use any customized optimization techniques but rely on default tuning of Neo4j.

Observations. In Figure 3, we observe that queries Q1-Q6 are instantaneous (close to 0) for all three implementations. Q17-Q21 and Q27 record execution spikes for the two snapshots implementations. Moreover, we notice that the runtimes of Q28 explode for the snapshots and the temporal graph implementations. The rest of benchmark queries (Q7-Q16 and Q22-Q26) does not exceed 6 seconds for the three approaches. Overall, the execution query times of the temporal graph are more stable than both snapshot-based approaches.

		Graph component	Evolution type	Time scope	Operation type
Q1	The descriptive attributes of a store at the month X	SE	S	SP	
Q2	The descriptive attributes of a store at the months X and Y	SE	S	MP	
Q3	The changes that occurred on the descriptive attributes of a store between the months X and Y	SE	S	MP	C
Q4	The descriptive attributes of a store from the month X to the month Y	SE	S	SI	
Q5	The descriptive attributes of a store every year of a period	SE	S	MI	
Q6	The changes that occurred on descriptive attributes of a store from the month X to the month Y	SE	S	SI	C
Q7	The price of an item at the month X	SE	I	SP	
Q8	The price of an item at the months X and Y	SE	I	MP	
Q9	Measure the change in the price of an item between the months X and Y	SE	I	MP	C
Q10	The price(s) of an item from the month X to the month Y	SE	I	SI	
Q11	Measure the average price of an item every year of a period	SE	I	MI	A
Q12	The customers that shopped in a store at the month X	SU	T	SP	
Q13	The customers that shopped in a store at the months X and Y	SU	T	MP	
Q14	Count the number of customers that shopped in a store at the month X	SU	T	SP	A
Q15	Count the number of customers that shopped in a store at the months X and Y	SU	T	MP	A
Q16	The customers that shopped in a store from the month X to the month Y	SU	T	SI	
Q17	The customers that shopped in a store every year of a period	SU	T	MI	
Q18	Count the number of customers in a store every year of a period	SU	T	MI	A
Q19	The household attributes of a customer at the month X	SU	S	SP	
Q20	The household attributes of a customer at the months X and Y	SU	S	MP	
Q21	The changes that occurred on the household characteristics of a customer between the months X and Y	SU	S	MP	C
Q22	The household attributes of a customer from the month X to the month Y	SU	S	SI	
Q23	The changes that occurred on the household characteristics of a customer from the month X to the month Y	SU	S	SI	C
Q24	The sold quantity of an item by a store at the month X	SU	I	SP	
Q25	The sold quantity of an item by a store at the months X and Y	SU	I	MP	
Q26	The sold quantity of an item by a store from the month X to the month Y	SU	I	SI	
Q27	Measure the average sold quantity of an item by a store every year of a period	SU	I	MI	A
Q28	The historical state of the store sales at the month X	EG		SP	

Table 4: Benchmark queries. *SE = Single Entity, SU = Subgraph, EN = Entire Graph, S = Schema, I = Instance, T = Topology, SP = Single Point, MP = Multiple Points, SI = Single Interval, MI = Multiple Intervals, C = Comparison, A = Aggregation.*

Discussion. The gap between the temporal graph and the two snapshots based implementations is partly due to difference in the volume of data involved in queries. As we can see in Table 1, the classical snapshots based implementation results in 66 times more nodes and 2 times more edges than the temporal graph. The optimized snapshots based implementation enables to reduce significantly the number of nodes but still it counts 47 times more nodes than the temporal graph. Inevitably, both snapshots approaches use more disk space and require more time to process during querying.

Queries' runtime are also impacted by their types. First, we analyze the querying performance of the temporal graph according to the entity scope, that is requesting information about the history of the graph at the level of a single entity (Q1-Q11) or a set of entities (subgraph) (Q12-Q27) or the entire graph (Q28). Most queries on a single entity are instantaneous as they involve the least data for the three approaches. The temporal graph approach outperforms the classical and the optimized snapshots approaches on querying a subgraph by respectively reducing their runtimes by 58%-99% and 74%-99%. The same trend is observed on querying the entire graph. The temporal graph saves 35% of the classical snapshots' runtime. Neo4j was not able to process Q28 for the optimized snapshots approach due to main memory limitations.

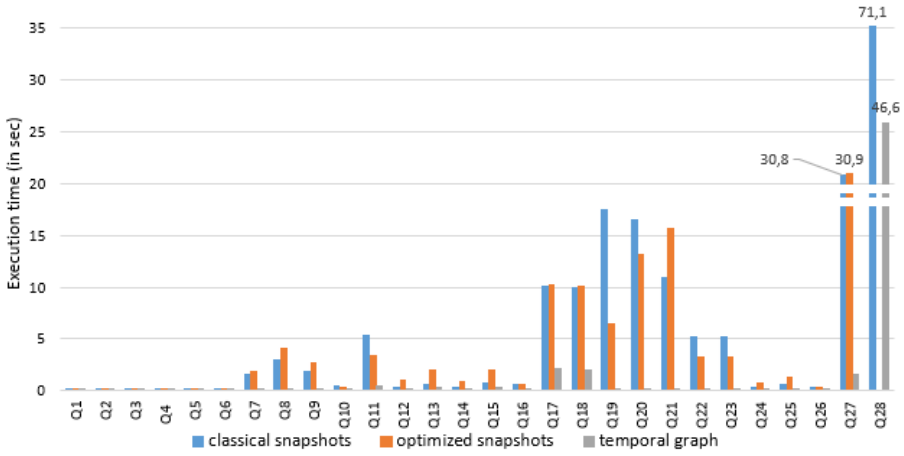

Fig. 3: Execution times of 28 benchmark queries.

Second, we analyze the impact of evolution type over queries' runtime: the evolution of schemas (Q1-Q6 and Q19-Q23), instances (Q7-Q11 and Q24-Q27) or topology (Q12 to Q18). The retrieval tasks relative to this scope enable to evaluate the cost of retrieving a specific information about graph changes. Excluding instantaneous queries, the gap between the two snapshots and the temporal graph implementations is the largest when querying schema. Indeed, queries on schema involve one specific operator[10]. Particularly, the temporal graph decreases the runtimes of Q19-Q23 in both classical and optimized snapshots by 98%-99%.

Third, we focus on the impact of time scope over queries' runtime to evaluate the cost of time travelling in the graph: querying a single time point (Q1, Q7, Q12, Q14, Q19, Q24 and Q28), a single interval (Q4, Q6, Q10, Q16, Q22, Q23 and Q26), multiple time points (Q2, Q3, Q8, Q9, Q13, Q15, Q20, Q21 and Q25) or multiple time intervals (Q5, Q11, Q17, Q18 and Q27). On the one hand, we observe that for queries with a complex time view (i.e. concerning multiple time points or multiple time intervals) the execution time explodes for both snapshots approaches while our model is still effective. Particularly, queries Q17-Q18 and Q27, involving multiple time intervals, reach respectively 10s and 31s for the two snapshots approaches. Our temporal graph allows to reduce those runtimes by 79%-95%. Moreover, queries Q20-Q21, involving multiple time points, exceed 10s for the two snapshots approaches. Our temporal graph enables to save 99% of both snapshots approaches on those queries. On the other hand, surprisingly, we notice that some queries' runtimes with the optimized snapshots such as Q7, Q8, Q9 or Q21 are higher than the classical snapshots even if they involve less

[10] The operator *keys* allows to extract the schema of a node or an edge. `https://neo4j.com/docs/cypher-manual/current/functions/list/`

data. Indeed, the translation of these queries into Cypher differs between the three implementations. As the time management method differs in the three models, the predicate on valid times differs even if they answer to the same business insights. Particularly, the translation of queries on a single time and multiple time points in Cypher for the optimized snapshots is more complex. These queries imply a sub-query to search for the snapshot that is the closest to the requested time instance.

Last but not least, we focus on the impact of operation types over queries' runtime: (i) comparison queries aiming at evaluating how does a graph component change over time with respect to a temporal evolution type (Q3, Q6, Q9, Q21 and Q23), (ii) aggregation queries aiming at evaluating an aggregate function (Q11, Q14, Q15, Q18 and Q27). Excluding instantaneous queries, both snapshots approaches perform the worst runtimes on aggregation and comparison queries. The temporal graph based implementation allows to save 61%-99% of classical snapshots' runtimes and 80%-99% of optimized snapshots' runtimes.

Implications. The choice of a data model to manage evolving graph data impacts significantly the storage and querying efficiency. Our model has a double advantage. First, it allows to get rid of data redundancy. So it saves a significant amount of space on the disk compared to snapshots. Second, it supports efficiently a wide range of queries while keeping the query runtime low and stable. In particular, querying an entire graph snapshot (Q28), which is the basic task in the literature, is costly in snapshots approaches. The implementation with our model allows to save 35% of runtime compared to the classical snapshots implementation.

5 Conclusion and future works

This paper has presented a complete solution to manage graph data evolution. The power of our solution lies on the proposition of a conceptual modelling and experimental assessments to illustrate its feasibility and efficiency.

Our conceptual modelling proposes concepts allowing representing the evolution of a graph at different levels: the graph topology, the attributes' set and the attributes' value of entities and relationships. Thus, it is generic enough to be compatible with any desired applications. Moreover, it does not introduce data redundancy thanks to a different time method from snapshot-based approaches. Time is attached to each individual graph component while it is attached to the entire graph in snapshots.

To validate the feasibility of our model, we implemented it in Neo4j based on a dataset containing temporal evolution. We showed that our model is directly convertible to the data model of Neo4j based on a set of translation rules we formalized. Then, we implemented several queries. We were able to query the evolution types proposed by our model using the native querying language of Neo4j.

To highlight the efficiency of our model, we made a comparative study of its implementation with the traditional sequence of snapshots and an optimized

version of snapshots based on the same dataset. We observed that our model performs better than the sequence of snapshots by reducing 12 times disk usage and by saving up to 99% on queries' execution time. In comparison to the optimized sequence of snapshots, our model reduces 9 times disk usage and saves until 99% on queries' runtime. In a nusthell, our model is an efficient solution for storing and querying a dataset with temporal evolution.

In our future works, we will extend our experiments to other types of data stores such as relational data stores since they can outperform both NoSQL graph stores and RDF triples stores [21]. This will require to extend the translation rules between the conceptual and logical level to be applicable to relational data stores. Then, we will compare the performance of these data stores in terms of storage and querying with different query languages than Cypher.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM **26**(11), 832–843 (1983). https://doi.org/10.1145/182.358434
2. Aslay, C., Nasir, M.A.U., De Francisci Morales, G., Gionis, A.: Mining Frequent Patterns in Evolving Graphs. In: Proceedings of the 27th ACM International Conference on Information and Knowledge Management. pp. 923–932. ACM (Oct 2018)
3. Beheshti, S.M.R., Motahari-Nezhad, H.R., Benatallah, B.: Temporal Provenance Model (TPM): Model and Query Language. arXiv:1211.5009 [cs] **abs/1211.5009** (Nov 2012)
4. Brunsmann, J.: Semantic Exploration of Archived Product Lifecycle Metadata under Schema and Instance Evolution. In: SDA. pp. 37–47. Citeseer (2011)
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. pp. 1–6. GRADES '13, Association for Computing Machinery (2013). https://doi.org/10.1145/2484425.2484442
6. Desmier, E., Plantevit, M., Robardet, C., Boulicaut, J.F.: Cohesive co-evolution patterns in dynamic attributed graphs. In: International Conference on Discovery Science. pp. 110–124. Springer (2012)
7. Fournier-Viger, P., He, G., Lin, J.C.W., Gomes, H.M.: Mining Attribute Evolution Rules in Dynamic Attributed Graphs. In: Song, M., Song, I.Y., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) Big Data Analytics and Knowledge Discovery, vol. 12393, pp. 167–182. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-59065-9_14
8. Hartmann, T., Fouquet, F., Moawad, A., Rouvoy, R., Le Traon, Y.: GreyCat: Efficient what-if analytics for data in motion at scale. Information Systems **83**, 101–117 (Jul 2019). https://doi.org/10.1016/j.is.2019.03.004
9. Holme, P., Saramäki, J.: Temporal networks. Physics reports **519**(3), 97–125 (2012), publisher: Elsevier
10. Huang, H., Song, J., Lin, X., Ma, S., Huai, J.: TGraph: A Temporal Graph Data Management System. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. pp. 2469–2472. ACM (2016)
11. 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 (Apr 2013). https://doi.org/10.1109/ICDE.2013.6544892

12. Khurana, U., Deshpande, A.: Storing and Analyzing Historical Graph Data at Scale. arXiv:1509.08960 [cs] (Sep 2015)
13. Koloniari, G., Souravlias, D., Pitoura, E.: On Graph Deltas for Historical Queries. arXiv:1302.5549 [cs] (2013)
14. Kosmatopoulos, A., Giannakopoulou, K., Papadopoulos, A.N., Tsichlas, K.: An Overview of Methods for Handling Evolving Graph Sequences. In: Karydis, I., Sioutas, S., Triantafillou, P., Tsoumakos, D. (eds.) Algorithmic Aspects of Cloud Computing, vol. 9511, pp. 181–192. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-29919-8_14
15. Kosmatopoulos, A., Gounaris, A., Tsichlas, K.: Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data. Computing **101**(12), 1885–1908 (Dec 2019). https://doi.org/10.1007/s00607-019-00715-6
16. Li, J., Han, Z., Cheng, H., Su, J., Wang, P., Zhang, J., Pan, L.: Predicting path failure in time-evolving graphs. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 1279–1289. KDD '19, Association for Computing Machinery (2019)
17. Maduako, I., Wachowicz, M., Hanson, T.: STVG: an evolutionary graph framework for analyzing fast-evolving networks. Journal of Big Data **6**(1), 55 (2019). https://doi.org/10.1186/s40537-019-0218-z
18. Moffitt, V.Z., Stoyanovich, J.: Towards sequenced semantics for evolving graphs (2017). https://doi.org/10.5441/002/EDBT.2017.41
19. Pernelle, N., Saïs, F., Mercier, D., Thuraisamy, S.: RDF data evolution: automatic detection and semantic representation of changes. In: SEMANTiCS (2016)
20. Ravat, F., Song, J., Teste, O., Trojahn, C.: Improving the performance of querying multidimensional RDF data using aggregates. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 2275–2284. SAC '19, Association for Computing Machinery (2019)
21. Ravat, F., Song, J., Teste, O., Trojahn, C.: Efficient querying of multidimensional RDF data with aggregates: Comparing NoSQL, RDF and relational data stores. International Journal of Information Management **54**, 102089 (2020)
22. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. Proceedings of the VLDB Endowment **4**(11), 726–737 (2011)
23. Rodriguez, M.A., Neubauer, P.: Constructions from Dots and Lines. arXiv:1006.2361 [cs] (2010)
24. Rossi, R.A., Gallagher, B., Neville, J., Henderson, K.: Modeling dynamic behavior in large evolving graphs. In: Proceedings of the sixth ACM international conference on Web search and data mining - WSDM '13. pp. 667–676. ACM Press (2013)
25. Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G., Stavrakas, Y.: A flexible framework for understanding the dynamics of evolving RDF datasets. In: International Semantic Web Conference. pp. 495–512. Springer (2015)
26. 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). https://doi.org/10.1109/ACCESS.2020.2994242
27. Yang, Y., Yu, J.X., Gao, H., Pei, J., Li, J.: Mining most frequently changing component in evolving graphs. World Wide Web **17**(3), 351–376 (May 2014)
28. Zaki, A., Attia, M., Hegazy, D., Amin, S.: Comprehensive Survey on Dynamic Graph Models. International Journal of Advanced Computer Science and Applications **7**(2) (2016). https://doi.org/10.14569/IJACSA.2016.070273