GDBAlive: a Temporal Graph Database Built on Top of a Columnar Data Store

Maria Massri, Philippe Raipin, Pierre Meye

▶ **To cite this version:**

Maria Massri, Philippe Raipin, Pierre Meye. GDBAlive: a Temporal Graph Database Built on Top of a Columnar Data Store. Journal of advances in information technology, 2020. hal-02937283

HAL Id: hal-02937283

https://inria.hal.science/hal-02937283v1

Submitted on 13 Sep 2020

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

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

GDBAlive: a Temporal Graph Database Built on Top of a Columnar Data Store

Maria Massri
Orange Labs, Cesson-Sévigné, France
Email: maria.massri@orange.com

Philippe Raipin and Pierre Meye
Orange Labs, Cesson-Sévigné, France
Email: {philippe.raipin, pierre.meye}@orange.com

***Abstract*—Although graph databases have extensively found applications in the relationship-centered era, a time-version support is seldom provided. While current storage systems capture the most recently updated snapshot of the underlying graph, most real world graphs embed a dynamic behavior translating the fact that vertices or edges can join or leave the graph at any time instant. Regarding that, a graph database should faithfully maintain the state of every graph's element permitting the analysis and prediction of the underlying system's performance. Since physical deletions are forbidden in such a scenario, the outgrowing size of data is a crippling restriction steering the interest in this area towards the optimization of the persistent storage. However, capturing and storing the state of the graph as full snapshots adds a storage overhead traded by faster query responses. Accordingly, the choice of an appropriate storage engine should be adapted with the threshold of accepted query latencies and the available storage resources. This paper will recognize the anterior academic work in the era of temporal graph databases while highlighting the existing tradeoff between storage and computation time costs. The implementation of GDBAlive, a temporal graph database using two state-of-the-art techniques Copy+Log and Log, is provided relying on a robust column oriented data store. In order to optimize the responsiveness of temporal queries in terms of computation times, we will introduce two fetching strategies "AsyncFS" and "Forced Fetch" and prove their efficiency on a real dataset.**

***Index Terms*—Graph databases, Temporal graphs, Distributed storage systems, Data locality**

I. Introduction

Graphs provide the most practical tool to describe human understandable relationships. Following this logic, graph database[1]s have emerged to provide storage and information extraction capabilities in the relationship-centered Era [1], [2]. In order to faithfully analyze and predict the performance of such domains, memorization should be a first class citizen in the design of the underlying storage system which is, deceivingly, being disregarded by commercial graph databases [3]–[5]. In transportation networks, traffic is a dynamic property of roads, thus it should not be neglected in shortest paths algorithms. Such analysis will recommend accurate route planning especially in critical cases such as evacuation planning from disastrous regions to safer ones [6], [7]. Another application of temporal graphs is related to environmental sciences as exemplified by sensor networks deployed in water treatment plants. Monitoring and tracking anomalies in such systems can predict and thus prevent outgrowing hot-spots such as harmful pathogen spills [8]. In biotechnology, significant knowledge can be harvested from protein networks such as cancer prediction. In social networks, queries retrieving the most durable relationships between individuals leads to community detection [9]. Such information can be very efficient for organizing social or professional events. Another social network application is the detection of regular co-occurrence of words between community members [10].

Narrowing our survey on property graphs, one can define a graph by a set of vertices, edges representing relationships between vertices and a labelling function assigning a value to each property. That is, a property, assigned to a vertex or edge, is a key/value pair where the key is the property's name. Analogically, a temporal graph consists of the sets of vertices and edges' states and labelling function assigning a value and a timestamp to each property. In that way, it can be split into multiple snapshots where each snapshot depicts the state of the graph taken at periodic timestamps or after a modification has occurred. These modifications include the addition or deletion of a vertex or edge and the update of their properties. Now, time is not trivially defined because it depends on the perspective it falls in. Besides being periodic or continuous, a temporal flow can be defined as transaction or valid [11], [12]. The latter translates to the time period during which a fact is true with respect to the real world. However, the former translates to the time

Manuscript received 10 March 2020; revised 30 March 2020; accepted 12 April 2020.

Maria Massri is with Orange Labs, 35510, Cesson-Sévigné, France.
Philippe Raipin is with Orange Labs, 35510, Cesson-Sévigné, France.
Pierre Meye is with Orange Labs, 35510, Cesson-Sévigné, France.

when a modification has been inserted to the database. Regarding that, a transaction time does not allow out of order writes while valid time provides more flexible insertions of historical data. Knowing that, traditional graph processing queries can be migrated to support the dynamical aspect of temporal graphs such as page rank [13], finding most durable connected components [9], [14], temporal shortest paths [15], [16] and temporal pattern and event detection [17].

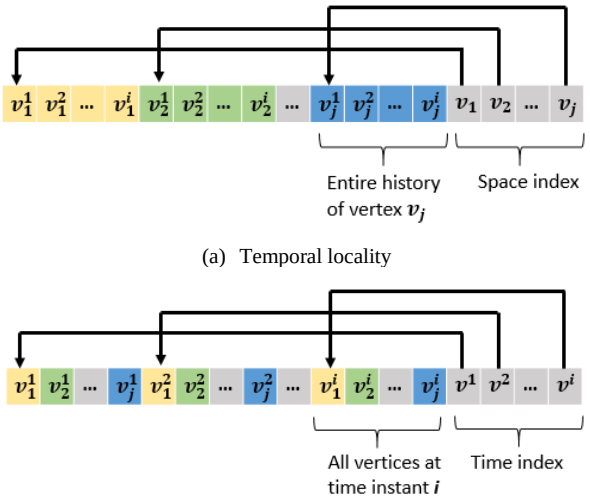
However, adding temporal dimension introduces a significant trade-off between the volume of stored data and the query's computation time. Every stored snapshot will mitigate the time needed to reconstruct the state of the graph at one time instant but induce larger volumes of data. As time elapses, the memorization of each modification imposes that a temporal graph cannot fit in memory. Regarding that, proper strategies should be applied aiming at optimizing I/O accesses to secondary storage especially when readings from sparse time points are requested in the same query.

This work will highlight the existing trade-off between storage and computation time costs by providing a survey on the-state-of-the-art approaches in section 2. We will also introduce GDBAlive, our implementation of a temporal graph database relying on a columnar data store in section 3. In section 4, our implementation is tested with a real temporal data set in both distributed and centralized architectures and provides two fetching strategies in order to accelerate the responsiveness of temporal queries. Finally, section 5 concludes the work.

II. PERSISTENT STORAGE

A. *Data Locality*

Data locality implies storing logically-connected data sequentially on disk which will avoid random reads. However, in temporal graphs, temporal and structural connectivity co-exist leading to favoring one over the other in physical data layout. The challenge consists of choosing between storing sequential versions of one vertex contiguously on disk or storing neighboring vertices (forming a community), in a single snapshot, contiguously on disk. While the first case, referred to as temporal locality or entity-centric locality (illustrated in Fig. 1.a), can naturally fit with respect to the linearity of time progression, the second one, referred to as structural locality or time-centric locality (illustrated in Fig. 1.b), is challenging. Indexing these structures consists of using space index v_j with temporal locality where each data block is referenced by the vertex identifier, and a time index v^i with structural locality where each data block is referenced by the relative time range. That is, temporal locality serves queries retrieving the state of one vertex during a range of time. Meanwhile, structural locality serves queries retrieving a snapshot at one time instant. While authors of Chronos [18] favored temporal locality, their recent work ImmortalGraph [19] highlights the importance of structural locality in queries retrieving overall graphical features such as page rank [20] or graph diameter [21]. To avoid finding an optimal solution merging structural and temporal locality, ImmortalGraph enhance replication by storing the graph twice each with a different data locality format.



(a) Temporal locality

(b) Structural locality

Figure 1. Data Locality physical layouts, where vj is a space index corresponding to vertex j's history and vi is a time index corresponding to time instant i.

B. *Underlying Storage Engine*

In order to build a temporal graph database, one can rely on a pre-existing database or build his own storage engine. In the latter case, a possible approach can be the design of the file system layout [19] to fit the graph's connectivity with respect to temporal dimensionality. However, knowing that most real graphs contain heterogeneous data and are more likely to grow in size, the backend storage engine should provide flexible data modelling and scalability. These specifications are natively supported by NoSQL databases leading a large fraction of commercial graph databases to rely on key/value stores such as [5] using RocksDB [22] , and on columnar data stores such as DSE Graph [4] using Apache Cassandra [23]. Similarly, it becomes rational to use a NoSQL databases as the backend storage engine for temporal graphs as authors of [24] did in their implementation of DeltaGraph where they relied on a key/value store Kyoto Cabinet [25]. Furthermore, authors of [26] posits a temporal property graph implementation using the Key/value store LMBD [27] as their backend storage. Similarly, ChronoGraph is a temporal graph database relying on MongoDB, a document store to handle traversals in temporal graphs [28]. However, an existing graph database can be extended to include temporal dimensionality [29]–[31].

C. *Related Work*

Many work has been conducted for memorizing the data's dynamical aspect as exemplified by time series storing data as sets of key/value pairs where the key refers to the timestamp of the corresponding value.

Systems implementing a time series data store are InfluxDB [32], TSDB [33], BTrDB [34] to name a few. Furthermore, temporal dimensionality was also introduced in document stores to support IoT related data [35]. However, connectivity is naturally supported by graphs which led us to study the state of the art in temporal graph databases focusing on the persistent storage. This survey permits us to split anterior storage techniques into four categories: Copy, Copy+Log and Copy-On-Write which will be detailed next.

Copy: Consists of the explicit storage of full graph snapshots at periodic or event-driven rates. Regarding the quadratic exposure of the storage costs, this technique will not be further discussed but can be considered as the baseline solution on top of which improvements should be considered to optimize the storage.

Log: The Log approach consists of storing exactly one snapshot representing the graph's initial state, successors are implicitly stored in the form of deltas or modifications. Now, Logs are branched in two categories: First, operational logs are represented by a series of events each corresponding, normally, to one graph entity where the type of the event (addition, deletion or update), the timestamp and the entity's identifier are stored. The other approach consists of adding modifications (respectively their corresponding timestamps) as new versions directly within the entity similarly to the behavior of time series. DeltaGraph [24], an implementation of the Log with operational logs, describes their model as a rooted and directed graph whose leaves correspond to snapshots and interior nodes corresponds to a special combination of lower level nodes. In other words, it can be regarded as a k-ary tree similar to a bulk loaded B+-Tree constructed in a bottom up fashion, where nodes are not necessarily existing snapshots and edges between nodes hold their differences. A crucial concept is that nodes are not physically stored. For instance, the lowest level holds faithful operational logs. These events are chunked and supported by edges connecting two successive leaves. Provided that, a leaf can be constructed from its predecessor by following the corresponding events. Now, moving to upper levels implies the application of a differential function to the k children in order to represent the parent node. On the other hand, extending graphs with time series have been proposed by TAG [7], [10] where each vertex or edge's property can be tracked by a time series. A simple implementation of this model uses adjacency lists where each vertex points to a linked list in which each element is a direct neighbor pointing to a time series representing the evolution of the edge weights taken at periodic time points forcing the redundancy of values that remained constant.

Copy: This technique consists of chunking the time interval into buckets. For each bucket a single starting snapshot is stored, with the deltas corresponding to the time range of the bucket. It is implemented by ImmortalGraph [19] and TGI [36]. Authors of [37] introduce this approach which they called Hybrid Graph in their comparison of One Graph (a strategy used to implement the Log approach) and Snapshot Graph (a strategy used to implement the Copy approach).

Copy-on-Write: Regarding the quadratic exposure of the storage complexity invoked by the Copy storage approach, Copy on Write avoids copying full snapshots, by only copying the modified node and applying the corresponding modifications to the copy forming a new version. Authors of GreyCat [38] create a new version of a vertex which will lead to modify all the incoming edges of the source vertices. Further refinements of this technique are proposed by [39], where instead of copying a full vertex, only its modification is stored while holding a pointer to the part that remained static. LLAMA [40] is another Copy-On-Write technique mitigating data redundancy and follows a CSR (Compressed Sparse Row) representation, it stores all vertices in one vertex array and edges in multiple tables each corresponding to a snapshot. The main contribution of their work resides in the commonality discovery where changes to an edge list are only stored instead of copying an entire edge table. In addition to copying vertices upon modification, some solutions represent time by vertices where each timestamp or time interval, depending on the choice of a discrete or continuous temporal flow, is a vertex holding edges towards all existing entities during the relative time interval. This technique, followed by [30], [41] adds a storage overhead outweighed by faster snapshot retrieval related to mitigating the latencies caused by verifying the existence of a graph's entity in a snapshot.

III. GDBALIVE INTRODUCTION

A. Motivations

Inspired by the previous work, we define a new temporal graph database GDBAlive implementing two state of the art techniques: Log and Copy+Log. Our system relies on Cassandra, a column oriented data store taking advantage of its robustness, fault tolerance and scalability. Furthermore, in most real world dynamic graphs, parts of the graph are static such as topological aspects, while others are dynamic such as volatile properties. Therefore, a physical separation of the static part from the dynamic part becomes rational by applying structural locality to the former and temporal locality to the latter. In such a way, GDBAlive combines structural and temporal localities. In order to enhance the responsiveness of temporal queries, some optimizations are implemented namely AsyncFS and Forced fetch strategies (which will be detailed in section IV.D). The first approach consists of batching fetching requests from remote servers while the second denotes loading chunks of data as pages instead of retrieving voluminous data blocks such as full time windows all at once.

B. Architecture

GDBAlive (as shown in Fig.2) provides a temporal graph translation layer taking the role of adapting the functionalities of a column oriented data store to support temporal graph databases. The constructing bricks of this layer are **Query Builder** and **Result set Processor** receiving temporal queries from clients and interacting

with Cassandra nodes. For instance, a Query builder will translate a temporal query that can be an insertion, update,

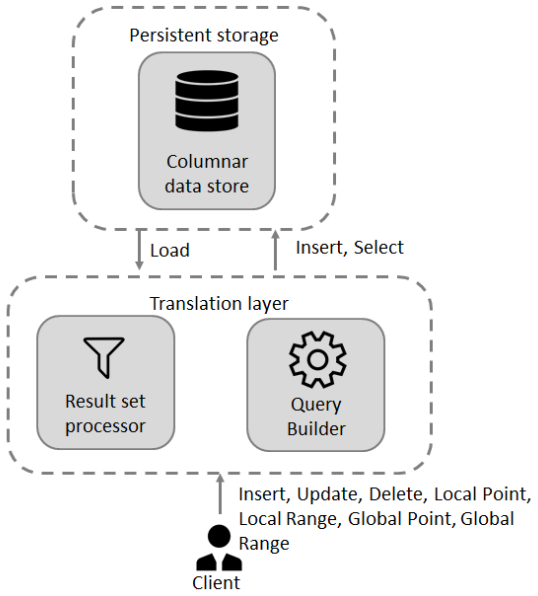


Figure 2. GDBAlive architecture

deletion or temporal query to CQL (Cassandra Query Language) queries. Now, a temporal query can be either global or local depending on whether it retrieves information about the entire graph or one vertex respectively. Furthermore, the requested data can be relative to one time instant or a range of time which refers to point or range queries. The persistent storage will store data, as modeled by graphs, in column families (more details on Cassandra's storage architecture is provided in section C and can have a distributed or centralized architecture. In order to construct the query's response, the Result set Processor will filter the returned data according to the requested time instant or time range.

C. Models

Before introducing how the Log and Copy+Log are implemented, a proper definition of Cassandra's primary key, decomposed of partition and clustering key should be provided. For instance, Cassandra stores data in column families where each table depicts a column family, rows are identified by their primary key and are restricted with the predefined columns of the table. However, an inserted row does not need to set all the columns since Cassandra supports semi-structured data. Now, every inserted row is sharded according to the hashed value of the partition key. Finally, rows can be stored in a predefined order according to the clustering key. In GDBAlive, the data model is based on the concept of separating entities, where each graph's entity notably vertex, edge and dynamic property will be placed in a Cassandra table. One table will be created for each vertex's label having start time (creation time) and end time (deletion time) columns to indicate their existence interval time with the rest of static properties. A table will also be created for each edge's label having source identifier, destination identifier, start time and end time columns with a list of static properties. Finally, each dynamic property acting as a time series will be stored in a table having a vertex identifier indicating the vertex to which we refer the corresponding property with timestamp and value columns.

1. *Log*

The Log model Stores a vertex's related data on a single node and creates a log storage approach by contiguously storing states of a vertex. As shown in Table 1., each

TABLE I. PARTITION AND CLUSTERING KEYS OF THE LOG MODEL

Table	Partition Key	Clustering key
vertex	vertexID	start
edge	sourceID	start
Dynamic property	vertexID	timestamp

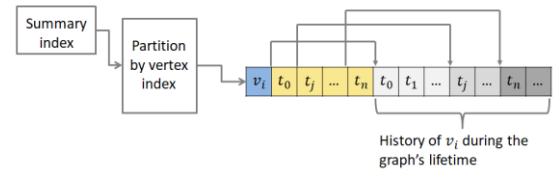
partition refers to one vertex, states of a dynamic property will be sorted by timestamp and each vertex will have its dynamic properties and edges stored in a single server. The Reading Path of Log approach is as follows: First, a binary search on an In-Memory summary index is performed in order to position the range of partition keys to which the searched key belongs. If found, the vertex partition will be referenced by the partition by vertex index. A second binary search is applied to Rows Index, in order to point to the rows block having the possible searched time instant. Next, the entire rows block will be read in order to find the desired timestamp. The physical data structure of the Log approach is presented in Fig.3 (a) it should be mentioned that each shade of grey block corresponds to a time range in the history of the vertex, where each sub-block refers to one Cassandra row.

2. *Copy+Log*

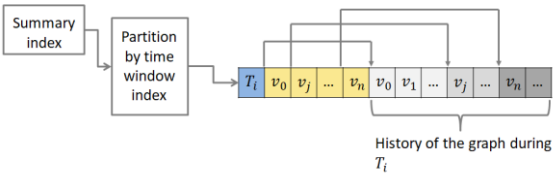
Table 2. presents the Primary key composition for each of vertex, edge and Dynamic property table in the case of Copy+Log model. The model goals consist of grouping time related data into the same server and creating Copy+Log by copying the state of the graph at the beginning of each time window that is represented by one Cassandra partition. The Reading Path of Copy+Log approach is as follows: Similarly to the read path in the case of Log model, a time window partition will be positioned after passing two indexing levels: Summary and Partition by time window indexes. Next, a binary search on the Row Index will be performed to position the rows block in which the state of the vertex at the desired time instant can be found. Fig.3 (b) presents the physical data structure of the Copy+Log model. It should be mentioned that each shade of grey block corresponds to a range of vertices with their corresponding histories during the range time T_i where each sub-block corresponds to a collection of rows, holding each data corresponding to vertex's state at one time instant t_i that falls within the time range T_i.

TABLE 2. PARTITION AND CLUSTERING KEYS OF THE COPY+LOG MODEL

Table	Partition Key	Clustering key
vertex	Time window	vertexID+start
edge	Time window	sourceID+start
Dynamic property	Time window	vertexID+timestamp



(a) Log Model



(b) Copy+Log model

Figure 3. Physical data structure

D. *Example*

For the sake of illustration, we provide in this section an example of the storage internals of a simple graph using the aforementioned models: Log and Copy+Log. We point out in this example to an IoT application domain, in which devices are connected to each other with different communication protocols. In such a scenario, the states of these devices and their connections are dynamic and should be faithfully maintained by the underlying graph database. For our example, we consider two vertices X and Y of type "Device" linked by an edge of type "Communicates with". Each of the devices has a static property "Model" and only device Y has the dynamic property "Measurement" that can refer to temperature, pressure, sound, power consumption, etc. The edge type has a static property "Protocol" depicting the communication protocol between the devices. Considering the temporal validity, vertices are valid during a range of time whilst edges are valid at one timestamp. It should be mentioned that this example is a special case where edges refer to instantaneous events, hence the start time and end time of the validity interval collapse. Now, the dynamic property is considered to be valid between two successive timestamps. Let's consider the case of Log model, Fig. 4 shows the graph's history starting at timestamp 10 and ending at timestamp 20 where the entire history of each graph's element is attached to it. For each of the graph's entities: vertex type, edge type and dynamic property, a Cassandra table is created as presented in Table 3. Furthermore, the partition and clustering keys are chosen based on Table 1. in order to store the consecutive states of each graph's entity sequentially on disk.

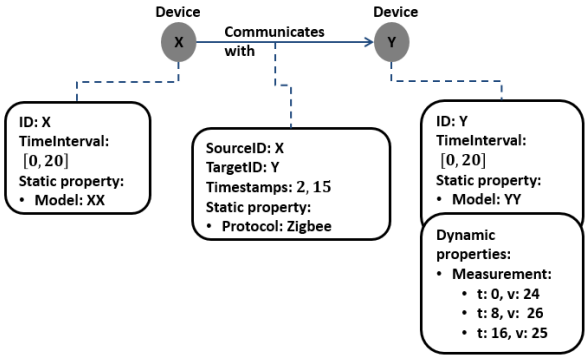


Figure 4. An example of temporal graph following the Log model

TABLE 3. INTERNAL STORAGE OF A TEMPORAL GRAPH FOLLOWING THE LOG MODEL

(a) Device

ID	Start	End	Model
X	0	20	XX
Y	0	20	YY

(b) Measurement

ID	Timestamp	Value
Y	0	24
	8	26
	16	25

(c) Communicates with

SourceID	Start	TargetID	End	Protocol
X	2	Y	2	Zigbee
X	15	Y	15	Zigbee

Considering the Copy+Log model, the history is supposed to be divided between two time windows with the respective time intervals: [0, 10] and [10, 20]. As time elapses, the first time window is closed and the valid entities are copied at the beginning of the second time window as illustrated in Fig. 5. Since devices X and Y are valid (not yet deleted) at timestamp t= 10, they must be copied in the next time window with their respective validity intervals and static properties. However, the dynamic property of device Y should be copied as the last updated value before timestamp t=10. As with the Log model, the Copy+Log model stores each graph's entity in a separate Cassandra table as presented in Table 4. We choose the partition and clustering keys based on Table 2. in order to store each time window as a physical partition. Now, each partition embeds the graph element's states that were valid at the beginning of the time window as copies with every valid state during the lifespan of the time window as a log.

TABLE 4. INTERNAL STORAGE OF EACH GRAPH'S ENTITY FOLLOWING THE COPY+ LOG MODEL.

(a) Device

Time Window	ID	Start	End	Model
10	X	0	10	XX
	Y	0	10	YY
20	X	10	20	XX
	Y	10	20	YY

(b) Measurement

Time Window	ID	Timestamp	Value
10	Y	0	24
		8	26
20	Y	10	26
		16	25

(c) Communicates with

Time Window	SourceID	Start	TargetID	End	Protocol
10	X	2	Y	2	Zigbee
20	X	15	Y	15	Zigbee

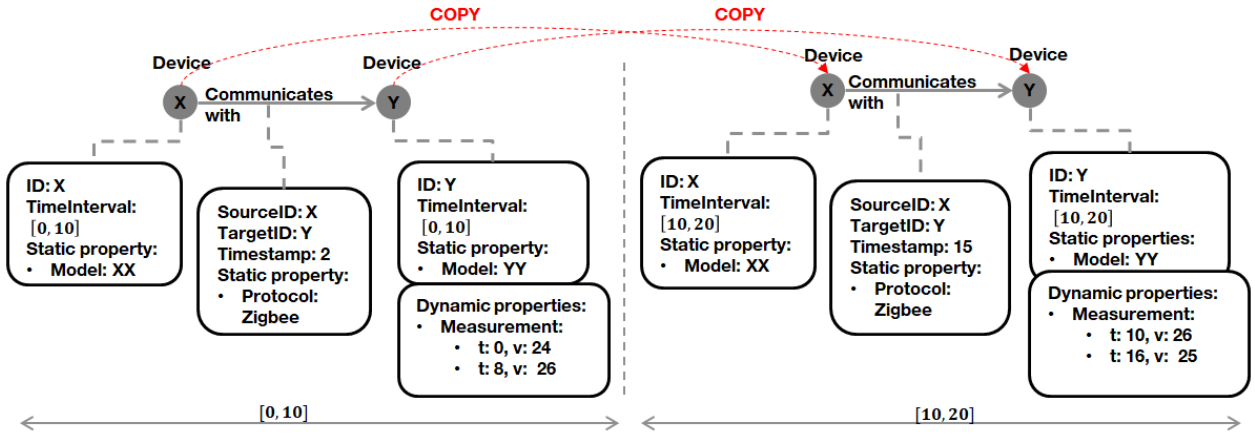


Figure 5. An example of a temporal graph following the Copy+Log model

IV. RESULTS

A. *Experimental Environment*

In order to, trustworthy, extract results from our solution, we decided to use a real dataset from which we have extracted metadata notably: the spanned years during the graph's history. Hence, we have chosen ASKUbuntu a temporal graph dataset found on [42] containing 160 000 vertices and 1 million edges spanning 8 years from 2009 until 2016. Vertices have only one label (type) which is user, meanwhile edges have three different labels: a2q (answer to question), c2q (comment to question) and c2a (comment to answer). It should be mentioned that we added one year at the beginning of the graph's lifetime in order to create all the vertices in 2008. Hence the resulting timespan is [2008: 2016]. As depicted in section III.B, the experimentation is implemented in two modes: Single representing a centralized architecture and Cluster representing a distributed architecture. All cluster nodes, namely CAS1, CAS2 and CAS3 are similar and the client can define his graph by specifying the labels of vertices and edges with their corresponding static and dynamic properties and the granularity of time windows. After the schema creation, any desired temporal dataset can be inserted. Once the data is populated into Cassandra nodes, the client can insert, delete, modify values and send temporal queries. Cassandra nodes will take the charge of creating tables, splitting the history of the inserted data into time windows, adding a full snapshot at the beginning of each in the case of Copy+Log approach and serving client's query requests.

TABLE 5. CASSANDRA AND CLIENT NODES SPECIFICATIONS

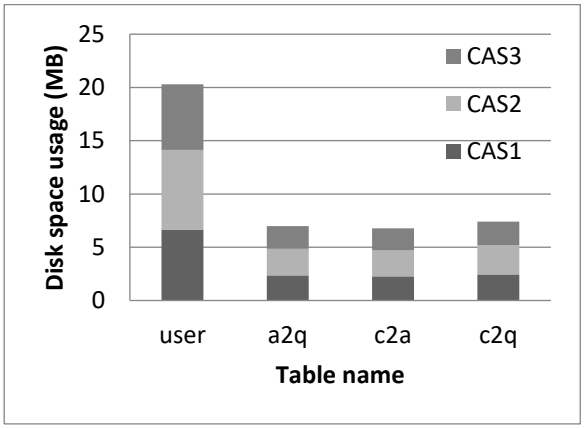
Specifications	Client	Single-mode server	Cluster-mode server
Number of vCPUs	1	2	2
Memory size (GB)	4	4	8
Disk space (GB)	10	40	40
OS	Ubuntu 16.04.4 LTS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS

The specifications of the client and Cassandra nodes in terms of: Number of virtual CPUs, memory size, disk space and operating system are provided by Table 5.

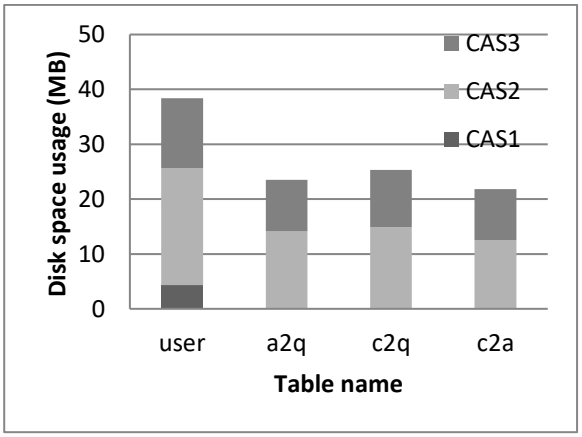
B. *Disk Space Usage Analysis*

One of the major limitations of the Copy+Log storage approach is its costly space usage as compared with LOG storage approach. Hence, a comparison between the space usages of the two approaches becomes crucial to prove

the precedent statement. It can be noticed from Fig. 6 that the space cost of the Copy+Log is 2x higher than the space cost of the Log approach considering the user table and 3x higher considering an edge table (a2q, c2q and c2a). Fig.6.a presenting the Disk space usage of the Log approach, shows an equal distribution of table's partitions between the three nodes. Meanwhile, Fig.6.b presenting the disk usage of the Copy+Log approach, shows an equitable distribution of the tables across CAS2 and CAS3 nodes while CAS1 holds only data belonging to the user table. In fact, a justification of that is the creation of all users during the year 2008 that is equivalent to one partition (time window), apparently this year (with no edge activity) is assigned to CAS1 which indeed hold exactly one partition.



(a) Log Model



(b) Copy+Log Model

Figure 6. Disk space usage comparison

C. Local Queries Results

Local point query: This query retrieves the state of the vertex at one instant, the search will differ according to the data storage model. In case of the Log approach, one partition corresponding to the history of one vertex will be searched while adding filtering clauses on start times, end times and timestamps in order to limit the search to only one time instant. The query is executed for each year in the graph's history, resulting in a fixed query response time (≅40 ms average) for both approaches in Single and Cluster modes. The performances of our models being undistinguishable for local point queries, the local range query, as described next, presents a noticeable differentiation.

Local range query: This query will extract the entire state of one vertex during a time range, starting from the smallest possible time range until reaching the full lifespan of the graph. Fig.7 present the response times of local range query with different time ranges for the Copy+Log and Log approaches in single and cluster modes respectively. It can be extracted from the two charts that the Log approach is faster than the Copy+Log approach. Actually, the Log approach will simply search in one vertex partition containing the entire history of a vertex. Meanwhile, the Copy+Log will search the state of the vertex in every spanned time window during the query time range. After the partition key is found, a binary search will be applied in order to find the Rows Block in which one clustering key is actually searched. Since the Copy+Log approach manage wide partitions, narrowing the search to one vertex will be a costly task. Now, regarding the use of Single and Cluster modes, it can be noticed that response times are faster in the case of a single Cassandra server (Single mode can improve responsiveness by a factor of 1.4 as compared to Cluster mode), because of the elimination of network latencies serving the coordination between nodes. Furthermore, this statement is only applied to Copy+Log where Log will not cause fetching from different nodes since the desired data corresponds to only one vertex partition.

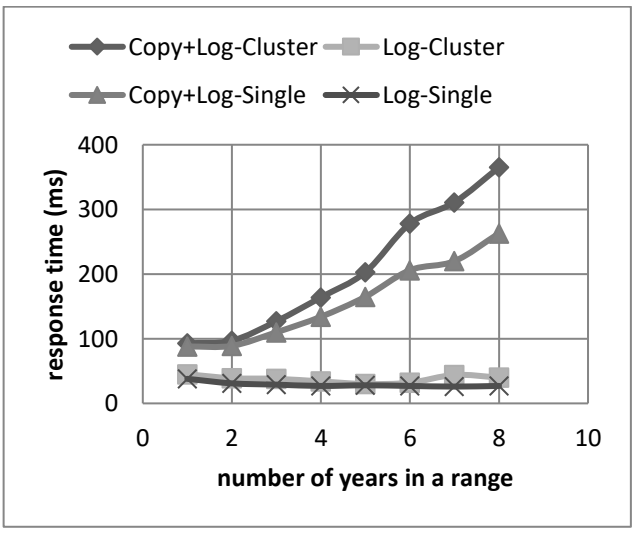


Figure 7. Local Range query response times

D. Global Queries Results

Global point query: will retrieve the state of the entire graph at one time instant. In order to achieve better responsiveness of such queries, we introduced two fetching optimizations: AsyncFS and Forced Fetch. Since each graph's entity that can be a vertex, edge or dynamic property is stored in a separate Cassandra table, several read requests are sent to serve one global query,

the purpose behind AsyncFS (Asynchronous fetching strategy) is the acceleration of the global query retrieval times by prohibiting the client's waiting states. Normally, the client will send read requests from multiple tables and later perform a filtering operation, on the retrieved graph's elements, keeping the "alive" ones within the desired time point or time range. Thus, the client can send one request → wait for response → filter → send the next request and so on. This implementation is referred to as SyncFS (Synchronous fetching strategy). Meanwhile the client can batch all his requests, and whenever a returned value is ready, a filtering will be performed at the client side in a multi-threaded fashion. This implementation is referred to as AsyncFS. Further accelerations of this strategy, can be obtained if the requested data is sharded between different nodes. In this perspective, multiple nodes will be engaged at once in the data fetching. In order to test the efficiency of our optimization approach, we implemented the global point query, with the SyncFS and AsyncFS in both centralized and distributed architectures. Fig.8 presents the resulting response times for different timestamps starting from 2008 till 2016 for the Copy+Log approach. It can be noticed that both AsyncFS and SyncFS curves have the same response times considering the first 3 years in either Single mode or Cluster mode. Starting from 2010, the curves start diverging, while AsyncFS presents a noticeable acceleration in the query response times. Furthermore, the gap between the two fetching strategies performances gets larger with higher timestamps due to the increase of the graph's size with the elapsed time. However, the four curves share the same allure, that is to say they have three phases: first, from 2008 till 2010 where a slight variation can be detected. Second, from 2010 till 2015 where the responses are continuously increasing and third from 2015 till 2016 where a drop in the Query response time can be noticed caused by Cassandra's caching techniques.

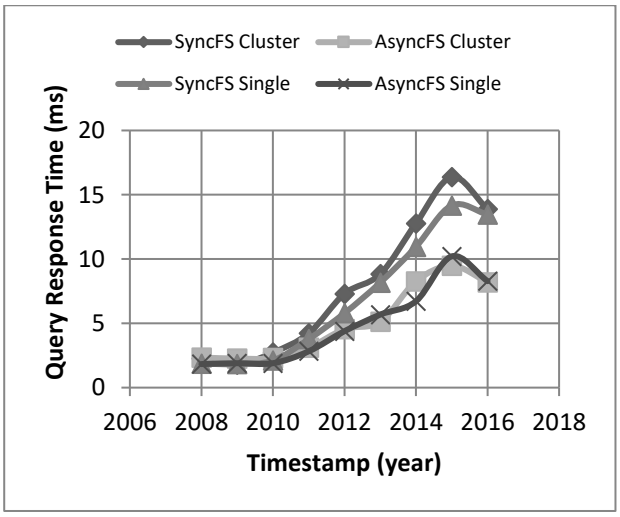


Figure 8. Global point query response times

It can be noticed that the goal of getting a better performance from AsyncFS in Cluster mode faster than that of AsyncFS in Single mode is not yet achieved, since global point retrieves data from one time window partition , that is to say that it retrieves data from one node in the cluster. However, the mentioned goal is achieved in the Global Range query described in the next section. Moreover, the measured results present the total response time thus a separation of the time taken by the client for filtering the time partition to keep alive data for a single timestamp and the time taken by the Cassandra cluster to fetch and send back the results is a must. Considering that, Fig. 9 shows the percentage distribution of the time taken by the client and that of Cassandra nodes. We can see that Cassandra is, in most cases, taking the highest time averaging in 55% of the query response time.

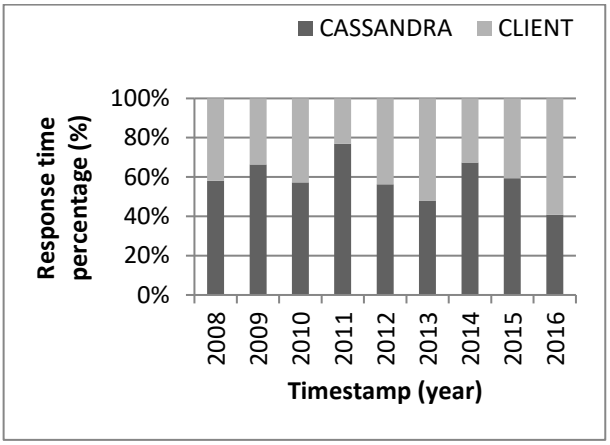


Figure 9. Global point query's response time distribution between the user and Cassandra nodes

After proving the efficiency of AsyncFS, further improvements can be obtained by following a Forced Fetch strategy. For instance, Forced Fetch is a strategy which fetches the data when needed, and Fetch All is a strategy which fetches the data all at once. However, data is retrieved as pages having each a number of rows Fetch Size α that can be specified with each request. The previous test was implemented with pages of 5000 rows ($\alpha = 5000$), and the client had to wait for the entire result set to be retrieved from Cassandra nodes before start filtering. In order to eliminate waiting states, the client can start filtering with available loaded pages while requesting more data as a future result set. In order to accomplish that, we specified a threshold (τ) for the size of the data, at the client size, that is not filtered yet. When the client is left with a number of rows equal to τ a request will be sent to Cassandra nodes asking for more data. Finally, the returned data will be added to the result set as it is retrieved and sent back to the client. In order to test the efficiency of Forced Fetch we had to run a test, 10 times with averaging, comparing the performance of Forced Fetch and Fetch All in terms of response times of the Global Point query with the SyncFS approach in Cluster mode. Furthermore, different values of α in the following set {5000, 50000, 100000} were tested. It is to be mentioned that the threshold τ is considered to represent 50 \% of the Fetch Size α. Fig.10 shows the percentage of improvement in response times in case of Forced Fetch as compared with Fetch All. It can be noticed that a Fetch Size $\alpha = 50000$ presented best

performance improvement in most years. In fact, 50000 is a compromise between a very small page size which will cause a higher number of accesses to the Cassandra cluster and a large page size which will impose waiting states.

Global range query: Tracking the graph's evolution relies on reconstructing the state of the graph during a range of time which refers to global range queries. Fig. 11 presents the global range query's response times in Single and Cluster modes for the Copy+Log model with both fetching strategies SyncFS and AsyncFS. As expected, AsyncFS in both single and cluster modes is faster than SyncFS.

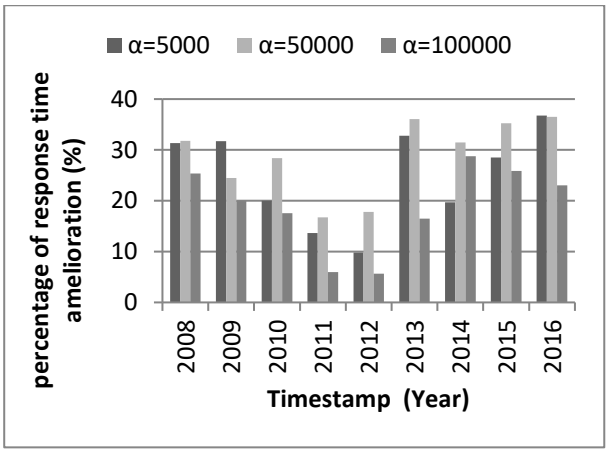


Figure 10. Forced Fetch versus Fetch All for the Global point query's response time

It can be seen that AsyncFS when applied to a Cassandra cluster can be 24.7 \% faster than when applied to a single server. In fact, this is explained by the sharded requested data which will be fetched from multiple nodes since the global range will be retrieving time window partitions from different nodes in the cluster. Knowing that, network latencies induced in Cluster mode can be compensated by utilizing the distribution of data across the cluster and simultaneously distributing the work between nodes.

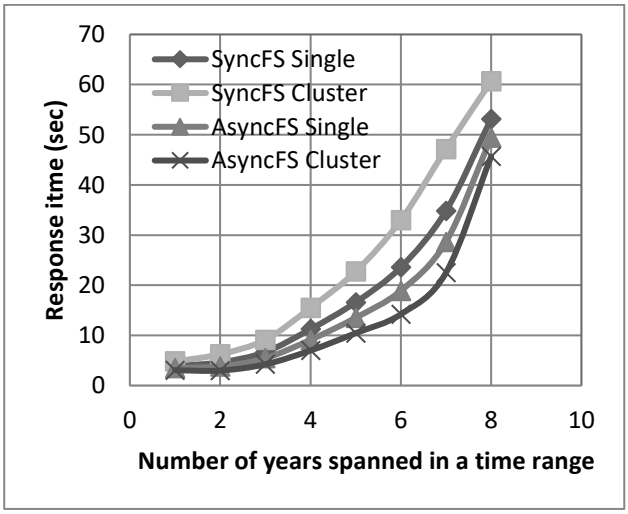


Figure 11. Global range query response times

In this section, we provided the results of four types of temporal queries. Local queries, tested in both Log and Copy+Log storage approaches, proved to be more efficient in terms of computation times with the Log storage technique. However, global queries proved efficiency with Copy+Log approaches after applying fetching optimizations such as AsyncFS and Forced Fetch. For AsyncFS, the coordination between the nodes in distributed architecture can be compensated by fully utilizing the distribution of data across the cluster. Meanwhile, the Forced Fetch strategy permitted the data processing without the need to store entire time windows in memory.

V. Conclusion

In this work we have provided a survey on the anterior work focusing on the persistent storage of temporal graph databases. Furthermore, we have provided a detailed implementation of state of the art techniques Log and Copy+Log relying on a column oriented database Cassandra, taking advantage of its engineering maturity. Our implementation, GDBAlive, is tested with four types of temporal queries: local point, local range, global point and global range. With the obtained results, we permit ourselves to conclude the necessity of Copy+Log approach for global queries, and Log approach for local queries. An asynchronous fetching strategy served as an optimization taking advantage of horizontal scalability by distributing the work between cluster nodes. Finally, we have provided a forced fetching strategy permitting us the reconstruction of a snapshot by reading a full time window without the need of loading it entirely in memory.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Maria Massri wrote the manuscript under the supervision of Philippe Raipin and Pierre Meye.

References

[1] R. Angles and C. Gutierrez, 'Survey of Graph Database Models', *ACM Comput Surv*, vol. 40, no. 1, pp. 1:1–1:39, Feb. 2008, doi: 10.1145/1322432.1322433.

[2] R. kumar Kaliyar, 'Graph databases: A survey', in *International Conference on Computing, Communication Automation*, May 2015, pp. 785–790, doi: 10.1109/CCAA.2015.7148480.

[3] 'Neo4j Graph Platform – The Leader in Graph Databases', *Neo4j Graph Database Platform*. https://neo4j.com/ (accessed Oct. 16, 2019).

[4] 'DSE Graph | DSE 6.0 Dev guide'. https://docs.datastax.com/en/dse/6.0/dse-dev/datastax_enterprise/graph/graphTOC.html (accessed Oct. 16, 2019).

[5] 'Multi-model highly available NoSQL database - ArangoDB'. https://www.arangodb.com/ (accessed Oct. 21, 2019).

[6] Z. Ding and R. Hartmut Güting, 'Modeling Temporally Variable Transportation Networks', in *Database Systems for Advanced Applications. DASFAA*, 2004, pp. 154–168, doi: https://doi.org/10.1007/978-3-540-24571-1_13.

[7] B. George and S. Shekhar, ‘Time-Aggregated Graphs for Modeling Spatio-temporal Networks’, *J. Data Semant. XI*, vol. 5383, no. Springer-Verlag Berlin Heidelberg, pp. 191–212, 2008, doi: https://doi.org/10.1007/978-3-540-92148-6_7.
[8] B. George, J. M. Kang, and S. Shekhar, ‘Spatio-Temporal Sensor Graphs (STSG): A data model for the discovery of spatio-temporal patterns’, *Intell. Data Anal.*, vol. 13, no. 3, pp. 457–475, 2009.
[9] K. Semertzidis, E. Pitoura, E. Terzi, and P. Tsaparas, ‘Best Friends Forever (BFF): Finding Lasting Dnese Subgraphs’, *CoRR*, vol. abs/1612.05440, 2017.
[10] S. Shekhar and D. Oliver, ‘Computational Modeling of Spatio-temporal Social Networks: A Time-Aggregated Graph Approach’, 2010.
[11] et al. Jensen Christian S., ‘A Consensus Glossary of Temporal Database Concepts’, *SIGMOD Rec.*, vol. 23, no. 1, pp. 52–64, Mar. 1994, doi: 10.1145/181550.181560.
[12] J. A. Gohil and P. M. Dolia, ‘Checking and verifying temporal data validity using valid time temporal dimension and queries in oraccle12C’, *International journal of database management systems (IJDMS)*, Aug. 2015.
[13] L. Yang, L. Qi, Y.-P. Zhao, B. Gao, and T.-Y. Liu, ‘Link Analysis Using Time Series of Web Graphs’, in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, New York, NY, USA, 2007, pp. 1011–1014, doi: 10.1145/1321440.1321598.
[14] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, and J.-P. Onnela, ‘Community Structure in Time-Dependent, Multiscale, and Multiplex Networks’, *Science*, vol. 328, no. 5980, pp. 876–878, 2010, doi: 10.1126/science.1184819.
[15] W. Huo and V. J. Tsotras, ‘Efficient Temporal Shortest Path Queries on Evolving Social Graphs’, in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, New York, NY, USA, 2014, pp. 38:1–38:4, doi: 10.1145/2618243.2618282.
[16] R. K. Pan and J. Saramäki, ‘Path lengths, correlations, and centrality in temporal networks’, *Phys Rev E*, vol. 84, no. 1, p. 016105, Jul. 2011, doi: 10.1103/PhysRevE.84.016105.
[17] H. N. Chaudhry, ‘FlowGraph: Distributed Temporal Pattern Detection over Dynamically Evolving Graphs’, in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2019, pp. 272–275, doi: 10.1145/3328905.3332303.
[18] W. Han *et al.*, ‘Chronos: A Graph Engine for Temporal Graph Analysis’, in *Proceedings of the Ninth European Conference on Computer Systems*, Amsterdam, The Netherlands, Apr. 2014, doi: 10.1145/2592798.2592799.
[19] Y. MIAO *et al.*, ‘ImmortalGraph: A System for Storage and Analysis of Temporal Graphs’, *Trans. Storage TOS*, Jul. 2015, doi: 10.1145/2700302.
[20] N. Duhan, A. Sharma, and K. K. Bhatia, ‘Page Ranking Algorithms: A Survey’, in *2009 IEEE International Advance Computing Conference*, Mar. 2009, pp. 1530–1537, doi: 10.1109/IADCC.2009.4809246.
[21] B. Bollobás* and O. Riordan, ‘The Diameter of a Scale-Free RandomGraph’, *Combinatorica*, vol. 24, no. 1, pp. 5–34, Jan. 2004, doi: 10.1007/s00493-004-0002-2.
[22] ‘RocksDB | A persistent key-value store’, *RocksDB*. http://rocksdb.org/ (accessed Apr. 30, 2020).
[23] ‘Apache Cassandra’. https://cassandra.apache.org/ (accessed Apr. 30, 2020).
[24] U. Khurana and A. Deshpande, ‘Efficient snapshot retrieval over historical graph data’, in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, Apr. 2013, pp. 997–1008, doi: 10.1109/ICDE.2013.6544892.
[25] ‘Kyoto Cabinet: a straightforward implementation of DBM’. https://fallabs.com/kyotocabinet/ (accessed Oct. 21, 2019).
[26] W. D. Vijitbenjaronk, J. Lee, T. Suzumura, and G. Tanase, ‘Scalable Time-Versioning Support for Property Graph Databases’, Boston, MA, USA, Dec. 2017, doi: 10.1109/BigData.2017.8258092.
[27] ‘Symas Lightning Memory-mapped Database’, *Symas Corporation*. https://symas.com/lmdb/ (accessed Nov. 04, 2019).
[28] J. Byun, S. Woo, and D. Kim, ‘ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time’, *IEEE Trans. Knowl. Data Eng.*, pp. 1–1, 2019, doi: 10.1109/TKDE.2019.2891565.
[29] M. A. Rodriguez, *Gremlin’s Time Machine*. 2016.
[30] C. Cattuto, A. Panisson, M. Quaggiotto, and A. Averbuch, ‘Time-Varying Social networks in Graph Databases’, in *First International Workshop on Graph Data Management Experiences and Systems*, New York, NY, USA, Jun. 2013, doi: 10.1145/2484425.2484442.
[31] H. Haixing, S. Jinghe, L. Xuelian, M. Shuai, and H. Jinpeng, ‘TGraph: A temporal Graph Data Management System’, Indianapolis, IN, USA, Nov. 2016, doi: http://dx.doi.org/10.1145/2983323.2983335.
[32] ‘InfluxDB: Purpose-Built Open Source Time Series Database’, *InfluxData*, Jul. 16, 2019. https://www.influxdata.com/ (accessed Jul. 16, 2019).
[33] L. Deri, S. Mainardi, and F. Fusco, ‘tsdb: A Compressed Database for Time Series’, in *Traffic Monitoring and Analysis*, Berlin, Heidelberg, 2012, pp. 143–156.
[34] M. P Andersen and D. E. Culler, ‘BTrDB: Optimizing Storage System Design for Timeseries Processing’, *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST ’16)*, 2016.
[35] K. Yong-Shin, P. Il-Ha, R. Jongtae, and L. Yong-Han, ‘MongoDB-Based Repository Design for IoT-Generated RFID/Sensor Big Data’, *IEEE Sensors Journal*, pp. 485–497, 2016.
[36] U. Khurana and A. Desphande, ‘Storing and Analyzing Historical Graph Data at Scale’, in *Proc. 19th International Conference on Extending Database Technology (EDBT)*, Bordeaux, France, 2016.
[37] V. Z. Moffitt and J. Stoyanovich, ‘Towards a Distributed Infrastructure for Evolving Graph Analytics’, in *Proceedings of the 25th International Conference Companion on World Wide Web*, Republic and Canton of Geneva, Switzerland, 2016, pp. 843–848, doi: 10.1145/2872518.2889290.
[38] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon, ‘Analyzing Complex Data in Motion at Scale with Temporal Graphs’, in *International Conference on Software Engineering & Knowledge Engineering*, Pittsburgh, USA, Jul. 2017, doi: 10.18293/SEKE2017-048.
[39] A. G. Labouseur *et al.*, ‘The G* graph database: efficiently managing large distributed dynamic graphs’, *Distrib. Parallel Databases*, vol. 33, no. 4, pp. 479–514, Dec. 2015.
[40] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, ‘LLAMA: Efficient graph analytics using Large Multiversioned Arrays’, Seoul, South Korea, Apr. 2015, doi: 10.1109/ICDE.2015.7113298.
[41] A. Castelltort and A. Laurent, ‘Representing history in graph-oriented NoSQL databases: A versioning system’, in *Eighth International Conference on Digital Information Management (ICDIM 2013)*, Islamabad, Pakistan, Sep. 2013.
[42] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014.

Maria Massri has received the M.S degree in Embedded Real Time Systems from the Ecole Centrale De Nantes, Nantes, France, the Dipl.Ing. degree in telecommunication engineering from the Lebanese University: Faculty of Engineering, Lebanon, in 2019 and is currently working toward the Ph.D. degree in temporal graph databases optimized for the internet of things (IoT) at Orange Labs, Cesson-Sévigné, France.

Her research interests include distributed systems, temporal graphs and graph databases.

Philippe Raipin has received his PhD from the University of Rennes 1, Rennes, France, in 2004.

He was a temporary lecturer and research assistant in the University of Rennes 1 from 2004 to 2006 and a post doc researcher in Orange from 2006 to 2007. Since 2007, he has been a research engineer in Orange. He is currently the research program manager of the Web of Things Platform in Orange. His

research interests include dependability, large systems and graph databases.

Pierre Meye has received a PhD degree in cloud storage dependability from the University Rennes 1 (France) in 2016.
He is currently a Research Engineer at Orange Labs Rennes. His research interests comprise storage systems, distributed systems, graph databases, and semantic web technologies for the Web of Things.