Clock-G: A temporal graph management system with space-efficient storage technique

Maria Massri, Zoltan Miklos, Philippe Raipin, Pierre Meye

Clock-G: A temporal graph management system with space-efficient storage technique

Maria Massri
Orange Labs
France
maria.massri@orange.com

Zoltan Miklos
University of Rennes CNRS IRISA
France
zoltan.miklos@irisa.fr

Philippe Raipin
Orange Labs
France
philippe.raipin@orange.com

Pierre Meye
Orange Labs
France
pierre.meye@orange.com

***Abstract*—IoT applications can be naturally modeled as a graph where the edges represent the interactions between devices, sensors, and their environment. Thing'in [1] is a platform, initiated by Orange[2]. The platform manages a graph of millions of connected and non-connected objects using a commercial graph database. The graph of Thing'in is dynamic because IoT devices create temporary connections between each other. Analyzing the history of these connections paves the way to new promising applications such as object tracking, anomaly detection, and forecasting the future behavior of devices. However, existing commercial graph databases are not designed with native temporal support which limits their usability in such use cases.**

In this paper, we discuss the design of a temporal graph management system Clock-G and introduce a new space-efficient storage technique δ-*Copy+Log*. Clock-G is designed by the developers of the Thing'in platform and is currently being deployed into production. It differentiates from existing temporal graph management systems by adopting the δ-*Copy+Log* technique. This technique targets the mitigation of the apparent trade-off between the conflicting goals of the reduction of space usage and acceleration of query execution time. Our experimental results demonstrate that the δ-*Copy+Log* presents an overall better performance as compared to traditional storage methods in terms of space usage and query evaluation time.

***Index Terms*—Temporal Graphs, Graph management systems, Temporal graph traversal**

I. Introduction

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

In this work, we mainly focus on designing a temporal graph management system and integrating it into the Orange initiated platform Thing'in. This platform manages a graph of objects that can be either connected such as IoT devices (machines, traffic lights, cameras, etc.) or non-connected (doors, roads, shelves, etc.). It equally maintains a thorough structural and semantic description of the environments of these objects, such as cities or buildings, in which they are located. The clients of this platform are companies and public administrations developing services around smart cities, building or factories, as well as private object owners and developers building analytical IoT applications. This graph of objects is maintained by a commercial graph database that do not account for the temporal dimension. However, there has been an extensive and recent demand by the clients of Thign'in for preserving the past states and connections of the graph for the interest of tracking objects, anomaly detection and forecasting the future behaviour. For instance, Mo.Di.Flu, a project whose main goal targets the Industry 4.0 and BIM2TWIN [3] European project focusing on the integration of Digital Twins [1] in smart buildings (e.g., smart factories), represents a concrete use case that motivated the integration of the temporal dimension into the graph of Thing'in. Mo.Di.Flu collaborates with Thing'in in order to keep track of the different positions of a product throughout a manufacturing pipeline, hence, detect the causes of manufacturing delays or products loss. Another requirement, also applied on the use-ase of smart factories, consists of reconstructing the surroundings of a machine at a given time instant or the states of all machines right before a system failure. These querying capabilities are only enabled by a system that persists the historical states and interactions recorded in the factory. These requirements have been the incentive of maintaining the history of the graph of Thing'in which we have addressed by designing the temporal graph management system Clock-G. This system is developed by the team of Thing'in and is currently being deployed into production.

Storing and querying temporal graphs are possible by exploiting a commercial graph database with temporal metadata [2]–[4]. However, these systems do not natively offer time-version support which might lead to unpredictable performances. Hence, we argue that time should be considered as a first-class citizen rather than a simple add-on.

A critical point in the design of such a system is defining a formal model of temporal graphs. Hence, we propose OPGM **O**peration-based **P**roperty **G**raph **M**odel that we refer to throughout the paper to define key concepts of our proposal.

Available temporal graph management systems focus on the storage model in order to deal with the outgrowing size of data while being able to query it efficiently. In this context, several storage approaches have been proposed in literature such as the *Log* and the *Copy+Log* methods.

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

[2]Orange is a French multinational telecommunication operator

[3]https://bim2twin.eu/

The *Log* approach [5], [6] consists of preserving all the graph updates as a series of timestamped logs. Whereas the *Copy+Log* [7] approach consists of storing the graph updates in time windows such as each time window contains a fixed number of graph operations. These time windows are stored along with snapshots that represent the state of the graph at the end of each time window and are used as starting points for query evaluation.

Despite the advantage of the *Copy+Log* method in pruning the search space, storing full graph snapshots is space consuming, particularly in the case of growth-mostly graphs. The limitation of this method is that unchangeable graph entities will be copied across snapshots causing a large volume of redundant graph entities. Whereas the *Log* approach has a detrimental impact on the query evaluation time. Hence, a compromise between both methods is needed which is mainly addressed in this paper by proposing the δ-*Copy+Log* method.

The δ-*Copy+Log* method stores all graph updates in time windows. Instead of storing full snapshots along with these time windows, it stores deltas which contain only the difference between two successive snapshots. To clarify, a delta contains all the graph updates that are contained in a time window and are not canceled by another graph update. For instance, an addition of a graph element is cancelled by a deletion of the same graph element, thus, not stored in a delta. Besides deltas, a snapshot is stored for each M time windows that is considered as a starting point for query evaluation. That is, M represents as a configurable parameter that can tune the performance of the δ-*Copy+Log* method. Furthermore, we define two types of deltas: forward and backward deltas such that we store half of the time windows and their corresponding deltas between successive snapshots in a forward fashion, whereas the other half is stored in a backward fashion. Having this, based on the positioning of the requested time instant of a query, we choose between a forward or backward construction of the result. Indeed, this has the advantage of reducing the maximum execution time of queries by a factor of 50%. Besides, we also propose an optimization technique to mitigate the query execution time overhead caused by storing deltas instead of snapshots. This technique consists of assigning each delta with a Bloom filter that will be checked whenever data is requested from that delta.

We developed Clock-G: a system that manages temporal property graphs and adopts the δ-*Copy+Log* storage method. The system Clock-G supports several types of temporal queries such as point-based and range-based local and global queries. Clock-G also supports a special type of range-based traversal queries referred to as time-increasing paths [8]–[11]. The queries supported by Clock-G belongs to the category of OLTP queries which cover most of the needs of our industrial use case.

We also conducted experiments to evaluate the performance of Clock-G. The obtained results are inline with previous findings on the apparent trade-off between space and query execution time of the *Copy+Log* and *Log*. Furthermore, a comparison between these traditional methods and the δ-*Copy+Log* validates that the δ-*Copy+Log* offers a good compromise between the performances of the *Log* and *Copy+Log* methods. Besides, we showcase how the parameters of Clock-G can be calibrated in order to tune the overall performance with an adequate configuration that adheres most with the acceptable threshold of query latency and available storage resources. We also compared the performance of Clock-G with a non-temporal commercial graph database (Neo4j [4]) to motivate the choice of creating a temporal graph management system with a native temporal support instead of simply adding a temporal layer. The results show that Clock-G outperforms Neo4j by orders of magnitude.

The main contributions of this work reduce to the following:

- Defining a formal model of temporal property graphs.
- Proposing δ-*Copy+Log* as a space-efficient variant of the traditional *Copy+Log* method.
- Implementing the δ-*Copy+Log* in a temporal graph management system Clock-G with optimization techniques that accelerates the query evaluation time.
- Evaluating the performance of Clock-G with large scale real-world and synthetic temporal graphs and validate the gain of using δ-*Copy+Log* technique as compared to traditional techniques.

Outline Section II discusses the related work. Section III introduces key definitions of our temporal graph model. Section IV introduces and formalises key concepts of the δ-*Copy+Log* storage approach. Section V introduces the key design features of Clock-G such as its architecture and querying optimizations. Section VI evaluates the work through experiments conducted on real and synthetic datasets. Finally, Section VII concludes the paper.

II. Related work

The challenge of versioning data is one of the classical problems in data management and there is a large literature on this subject, however it is not extensively studied in the context of graph management systems. Indeed, such a system must incorporate first a formal model for underlying temporal graphs. Since these graphs are not as intuitive as their static counterparts [12], several formal models have been posited in literature [13]–[16]. The main limitation of these models is that they do not support the property graph model and assign only edges with timestamps. In this context, we provide a formal model of temporal property graphs presented in Section III.

Narrowing the survey on the storage of temporal graphs, two main concerns should be addressed: the backend storage engine and the storage technique. Since temporal graphs can grow very large in size, the questions of secondary storage are crucial. In this context, one can develop a temporal graph management system by designing a dedicated storage engine such as a file system layout [17] or a temporal key/value store [18]. However, regarding the fact that most real-world temporal graphs are more likely to exponentially grow in size, the backend storage engine should offer scalability.

[4]https://neo4j.com/

Such a specification is natively offered by NoSQL databases which has been already exploited in the design of temporal management systems. Examples of such an implementation are SystemG [19] using a key/value store LMBD [5] and ChronoGraph [20] using a document store MongoDB [21]. Clock-G relies on Apache Cassandra [22] as a backend storage engine. In this paper, we choose Cassandra for the reasons of scalability, fault-tolerance and engineering maturity.

Available temporal graph storage techniques can be categorized as follows: *Log*, $Copy$, $Copy\text{-}on\text{-}write$ and *Copy+Log*. These methods are mainly motivated by concepts of logging and *checkpointing* which reflects on lessons learned from classical techniques of database state recovery [23], [24]. The Log approach followed by Raphtory [25], ChronoGraph [26] and TAG [5], [6] consists of storing graph updates as a series of timestamped logs such that any graph state can be recovered by reloading all logs with a relative timestamp lower than or equal to the requested one. Whereas the Copy approach consists of materializing and persisting graph snapshots. As discussed by authors of [27], the *Log and* Copy represents two extremes for storing temporal graphs. That is, these methods favor either the optimization of the space usage at the expense of the query's computation time or vice-versa.

Instead of storing full graph copies, the Copy-On-Write [28]–[32] consists of copying the state of a single graph entity that can be a vertex or edge whenever the latter gets updated. The main limitation of this technique is that one cannot control the copying frequency as it depends on the granularity of graph updates.

The Copy+Log storage method [7], [17], [33]–[37] consists of storing graph updates in temporally disjoint chunks known as time windows along with valid states of the graph known as snapshots. These snapshots represent a materialization of the graph states that are valid between the boundaries of two successive time windows. The advantage of this solution relies in the fact that recovering the state of the graph at a given time instant requires at most reading a single snapshot and all graph updates recorded in a time window. This method reflects on lessons learned from the area of database recovery such as Oracle's RMAN [6].

The critical limitation of the *Copy+Log* that is mainly tackled in this work relates to the fact that snapshots are cost prohibitive in terms of space consumption because they materialize the state of every existing vertex or edge at a single time instant. Now, in a lot of use cases some parts of the graph are static especially in growth mostly graphs where additions are more probable than deletions. In this case, vertices and edges tend to survive for longer duration and hence are copied over and over again in several snapshots. To overcome this limitation, we proposed the δ-*Copy+Log* storage approach to mitigate the space usage caused by the storage of full graph snapshots. In this method, we consider deltas representing the difference between snapshots instead of snapshots. However, we preserve a number of snapshots to serve as starting points for query evaluation such that after a fixed number of delta, a full snapshot is materialized. The main difference between the δ-*Copy+Log* and the methods that follow the *Copy+Log* technique such as RMAN is that storage of full snapshots or full database backups is critical and space consuming. Hence, we propose in this paper to keep the time windows (incremental backups) but to replace full backups (snapshots) by deltas which contain only the difference between two snapshots. If a large number of incremental updates cancel each other between two successive snapshots, leading to few differences between successive snapshots, the δ-*Copy+Log* keeps only those differences to preserve space. Now, to account for the evaluation time overhead, we add bloom filters to the deltas to verify the existence of an element in a delta.

[5]https://symas.com/lmdb/

[6]https://www.oracle.com/database/technologies/high-availability/rman.html

III. Formal definitions

A. Time domain

In this section, we provide a definition of the time domain. Indeed, defining the time domain is needed in data management systems when temporal ontologies are to be defined in order to assign data items with temporal validity information [38]. In this work, we choose a discrete temporal flow by considering a time axis that is quantified by time granules [39]. A time granule, also referred to as a chronon, is the smallest non-decomposable unit of time defined by a specific temporal granularity (for example, a second or a millisecond, etc.). Hence, we consider $D^T = \{t^i | i \in \mathbb{N}\} \bigcup \{Now, \infty\}$ to be the time domain defined as a totally ordered set of instants such that the duration between consecutive instants is equal to a chronon. Besides, we consider a transactional time of graph updates that is assigned by the system. Hence, we do not encounter the classical challenges of valid time considerations such as out-of-order insertions, clock shifts, different time zones or granularity that are imposed by having independent data spouts assigning a valid timestamp for each event.

B. Temporal property graph model

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

We define the notations used to formalize this model as follows: Let V and E denote finite sets of vertex and edge identifiers (ids), respectively. Vertices and edges can have a single label and a set of dynamic property keys denoted as P. Let R denote an infinite set of atomic values that can have any type from a finite set of data types D (e.g., string), L denote a finite set of strings, id denote an identifier, 2^X denote the set of all finite subsets of domain X. Finally let a, d and u denote an addition, deletion and update respectively. We define a graph operation ϵ^i as an action applied on a graph entity which translates to either an addition/deletion of a vertex/edge or the update of a dynamic property. We define finite sequence of temporally ordered graph operations

$\Upsilon = \{\epsilon^i, i \in \mathbb{N}\}$. Following the OPGM, a temporal property graph model is a tuple:

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

We explain the functions below:

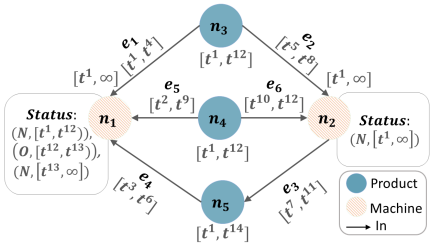
- $\rho : (V \cup E) \rightarrow 2^L$ maps each vertex and edge to a finite set of labels from L.
- $\alpha : E \rightarrow (V \times V)$ maps each edge to its source and target ID.
- $f^G : \Upsilon \rightarrow (V \cup E \cup ((V \cup E) \times R \times P))$ maps a graph operation to its corresponding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

Example III.1. To clarify, we provide in the following an example of a toy graph modelling the use case of Mo.Di.Flu presented in Section I. Figures 1a and 1b illustrate a temporal property graph and its representation based on the OPGM model, respectively. The temporal graph models the dynamic connections between the machines and products and the different values recorded by the sensors monitoring the state of the machines. We show how, for a sample of vertices and edges, the history of this graph can be represented using the above definitions:

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

As presented in Figure 1a, the validity intervals of vertices indicate the time during which the corresponding machine or product was recorded into the database. The validity intervals of the 'In' edges indicate the time during which a product was in a machine. The property 'Status' indicates the value and the time interval of the status of a machine.

(a) Temporal property graph



(b) OPGM representation

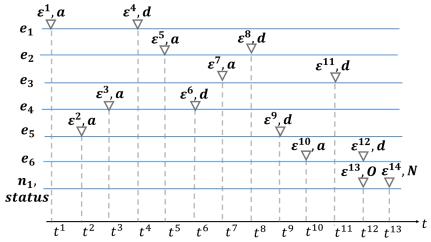


Fig. 1: Example illustrating a temporal property graph and its representation based on the OPGM model. We used (a), (d), (u), (N), (O) to denote addition, deletion, update of a dynamic property, normal and out-of-service, respectively

IV. δ-Copy+Log

The δ-*Copy+Log* is a variant of the Copy+Log storage approach that we propose to mitigate the space cost induced by storing full snapshots. Recall that the Copy+Log consists of storing snapshots that are valid between the boundaries of a time window s.t. each time window contains a fixed number of graph operations. Now, the δ-*Copy+Log* follows a similar mechanism with the main difference that consists of storing deltas instead of snapshots. A critical point is that a delta differs from a time window. That is, a time window contains every graph operation that exists between two snapshots whereas a delta contains the only the minimum number of graph operations that morph a snapshot into another one. Indeed, an addition of an element in cancelled by a deletion of the same element, hence, both operations are stored in time window but omitted from the delta. We store a snapshot after a number of time windows in order to serve as a starting point for query evaluation. Having this, we store graph operations in consecutive time buckets containing each a number M of time windows such that the first $M-1$ time windows ends with a delta, whereas the final time window ends with a snapshot. A critical optimization is the forward and backward data storage and retrieval. That is, half of the deltas and time windows in a bucket is constructed in a forward fashion whereas the other half is constructed in a backward fashion. The rationale behind this choice is the acceleration of the query execution time. That is, we choose the closest snapshot from which to start the retrieval then compute the result in a forward or backward fashion whether the time instant of that snapshot is lower or greater than the requested one.

Figure 2 illustrates the storage internals of the δ-*Copy+Log* and *Copy+Log* methods. It shows that the *Copy+Log* method stores time windows and snapshots. Whereas, the δ-*Copy+Log* stores time windows, deltas and snapshots. In this example, we consider a set of time buckets B where $M = 6$ which implies that the bucket contains 3 forward time windows $\{\omega^1_{\Rightarrow}, \omega^2_{\Rightarrow}, \omega^3_{\Rightarrow}\}$ and 3 backward time windows $\{\omega^4_{\Leftarrow}, \omega^5_{\Leftarrow}, \omega^6_{\Leftarrow}\}$. At the highest time instant of a forward time window, a delta is materialized resulting in 3 forward deltas $\{\delta^1_{\Rightarrow}, \delta^2_{\Rightarrow}, \delta^3_{\Rightarrow}\}$. Whereas, a delta is materialized at the lowest time instant of every backward time window except the last time window where a snapshot is materialized resulting in 2

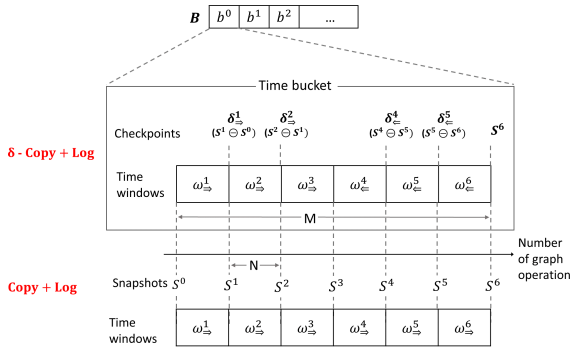


Fig. 2: The internals of the Copy+Log and δ-Copy+Log showing a time bucket (b^1) with $M = 6$, forward and backwards time windows ($\omega^i_{\Rightarrow}$, $\omega^i_{\Leftarrow}$) , deltas ($\delta^i_{\Rightarrow}$, $\delta^i_{\Leftarrow}$) and snapshot S^i

backwards deltas $\{\delta^4_{\Leftarrow}, \delta^5_{\Leftarrow}\}$ and snapshot $\{S^6\}$. Note that, the subtractive relation $\ominus$ operating on two snapshots S and S' s.t. $S \ominus S'$ results in the minimum number of graph updates that permits the transformation of S in to S'. That is, $(M-2)$ time windows are stored with a delta, whereas a single time window is stored with a snapshot. Half of the time windows is stored in forward fashion whereas the other half is stored in a backward fashion. Suppose a query with a requested time instant t. If t falls within the time interval of time window $\omega^2_{\Rightarrow}$, we start the search in a forward fashion by fetching $\omega^1_{\Rightarrow}$, then fetching $\omega^2_{\Rightarrow}$ whose timestamp is lower than t. Whereas, if t falls within the time interval of the time window $\omega^5_{\Leftarrow}$, we construct the result in a backward fashion. That is, we start by fetching S^6 then $\delta^5_{\Leftarrow}$ and finally $\omega^5_{\Leftarrow}$. Note that, in the *Copy+Log* method all the time windows are considered as forward.

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

Time buckets: The concept of time buckets stems from the choice of storing a fixed number of time windows and deltas between two snapshots. Indeed, we store the history of the graph in a sequence of temporally disjoint time buckets s.t. each time bucket is a logical container of M time windows and their corresponding checkpoints. That is, a checkpoint can be either a delta or a snapshot. Now, we store a snapshot that is valid at the highest time instant of the last time window of a each bucket, whereas we store a delta at the ending time instant of other time windows. Having this, M time windows exist between the snapshots of two successive buckets. Besides, the first $M/2$ time windows are constructed in a forward fashion and ends each with a forward delta. Whereas, the rest of the time windows are constructed in a backward fashion and ends each with a backward delta. To formalize, we consider the sequence of time buckets $B = \{b^i | i \in \mathbb{N}\}$ s.t. a time bucket is defined as the tuple $b^i = \{\Omega^i, \Gamma^i\}$:

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

TABLE I: Symbols and their descriptions used in the formalisation of the δ-*Copy+Log* approach

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

s.t.:

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

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

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

Time windows: A time window is a physical container of N graph operations. A forward time window $\omega^i_{\Rightarrow}$ contains graph operations that are sorted following an ascending order of their timestamps. However, a backward time window $\omega^i_{\Leftarrow}$ contains graph operations that are first reversed, meaning that an addition is stored as a deletion and vice versa then sorted following a decreasing order of their timestamps.

Snapshots: A snapshot is only persisted at the ending time instant of the last time window of a bucket and represents a state that is valid at that time instant. If the bucket corresponds to a vertex or edge label, then a snapshot contains all vertices and edges that exist at the time instant of the snapshot. Whereas, if the bucket corresponds to a dynamic property, then a snapshot contains all vertices and edges that has that property with the last updated value before or at the time instant of a snapshot.

Deltas: A delta between two snapshots S and S' contains the minimum number of graph updates that permit the transformation of S into S'. That is, if an addition is followed by a deletion of the same graph entity, these graph operations cancel each other and will not be added to the corresponding delta. Having this we can formally define a forward delta $\delta^i_{\Rightarrow}$, using constructs from the OPGM model, as follows:

$$\delta^i_{\Rightarrow} = \{\ \epsilon^k | \forall \epsilon^k \in \omega^i_{\Rightarrow} (\forall \epsilon^l \in \omega^i_{\Rightarrow} - \{\epsilon^k\} (f^G(\epsilon^k) = f^G(\epsilon^l) \implies f^T(\epsilon^k) > f^T(\epsilon^l))) \ \} \quad (3)$$

Equation (3) states that a graph operation is contained in a forward delta if the former is not followed by any other graph

operation in the same time window that maps to the same graph entity. A formal definition of a backward delta can be derived from Equation (3) by replacing $\delta^i_{\Rightarrow}$ and $\omega^i_{\Rightarrow}$ by $\delta^i_{\Leftarrow}$ and ω_i and $(f^T(\epsilon^k) > f^T(\epsilon^l))$ by $(f^T(\epsilon^k) < f^T(\epsilon^l))$.

A. Space and time complexities

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

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

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

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

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

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

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

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

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

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

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

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

The space usage of the *Copy+Log* method (χ_{CL}) is equal to the space occupied by graph operations and snapshots ($\chi_o + \chi_s$) where $M = 1$. Having this, we derive the following:

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

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

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

We analyze the time complexity of a unary query evaluation operator for point-based traversal queries. The expand operator ($\uparrow_\tau (v)$) retrieves all the edges of a vertex v whose validity intervals contain the time instant τ. Note that the expand operator was first proposed in [40] to define a graph algebra. In this complexity analysis, we use a simplified temporal variant of this operator.

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

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

The expansion of a vertex using the Log method might incur loading all graph operations in γ. Having this, we derive the following:

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

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

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

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

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

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

V. Clock-G

A. Architecture

In the following, we provide the description of the key elements composing the architecture of Clock-G (Figure 3).

1) Client API: Clock-G offers a Client API enabling a client to connect, ingest graph updates or query the stored graphs. For instance, users can define the graph space's schema that includes labels of vertices and edges and a set of static and dynamic properties for each vertex/edge label. That is, we differentiate between static and dynamic properties for the interest of reducing the disk space usage such that static properties will be stored along with the corresponding vertex or edge whereas dynamic properties are stored separately as depicted in Section V-A3. Users can insert graph operations individually or in batches into the system. In both cases, graph operations are attached with a transactional time based on the system's internal clock. Besides, users can query the temporal graph with local and global point queries.

2) Request Handler: The request handler manages a pool of a fixed number of workers s.t. each worker is responsible of serving a single client request. Requests are first sent to the gateway and then buffered until being served by a free worker.

3) Storage: Clock-G uses the column-oriented database Apache Cassandra [22] as a backend store. We choose Cassandra for the reasons of robustness, engineering maturity and scalability. Besides, Cassandra sorts blocks of data according to a given column or combination of columns. We utilize this feature in order to sort graph updates according to their chronological order which accelerates their sequential read.

For instance, the storage is separated based on the graph entity type. That is, we store vertex operations in vertex store, edge operations in edge store and dynamic property operations in property store. In each store, the storage is separated based on the type of graph element. The vertex and edge stores separate the storage of different vertex and edge labels, respectively. Whereas, property store separates the storage of different property names. The rationale behind the separation is that graph elements are more likely to be queried independently.

For each vertex/edge label or dynamic property name we partition the storage based on a Hash partitioning strategy. Each of these partitions corresponds to a storage unit and is stored following the δ-*Copy+Log* method (denoted δ-CL in Figure 3 for simplicity).

4) Bloom Cache: In order to mitigate the execution time overhead of graph traversals induced by the storage of deltas instead of snapshots, we assign each edge delta with a Bloom filter. Indeed, for each graph operation in a delta, we add the ID of the source vertex to the Bloom filter. Having this, we can accelerate graph traversals by skipping the retrieval of the neighborhood of a vertex if the *id* of the latter is not found in the Bloom filter. It should be noted that we keep Bloom filters in main memory s.t. we fix the threshold of memory usage as a design parameter to limit the creation of bloom filters. That is, whenever the space occupied by Bloom filters reaches the specified threshold, we follow a FIFO (**F**irst **I**n **F**irst **O**ut) policy to evict the oldest Bloom Filters (corresponding to the oldest delta) from cache. Further description of the functionality of Bloom filters in the acceleration of query evaluation is provided in section V-B.

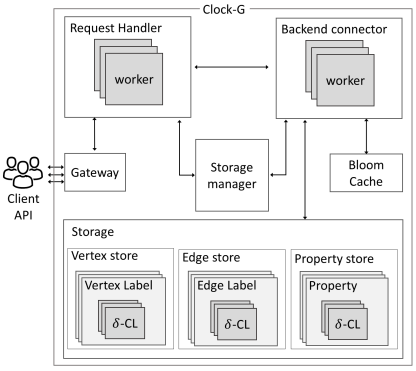


Fig. 3: System architecture

5) Storage manager: The storage manager is responsible of applying the rules of the δ-*Copy+Log* method. It directs newly inserted graph operations to an open time window. However, if the size of the time window reaches the threshold N, it considers it as a closed time window, and sends a creation request of a delta to the backend connector.

6) Backend connector: The backend connector manages a pool of workers whose main functionality is to connect to and execute requests against the backend store. It is critical to mention that a backend worker is responsible for a range of partitions s.t. all requests directed to a worker correspond to elements of its local partitions. A write request consists of the insertion of a graph operation in a forward or backward time window, creation of a forward or backward delta, or creation of a snapshot. In order to create a delta, a worker loads the time window in memory, then removes any pair of graph operations that cancel each other. Whereas, it creates a snapshot by loading the snapshot of the previous time bucket on which it applies the graph updates contained in forward deltas and the inverse of the graph updates contained in backwards deltas of the current time bucket. If a read request consists of finding the neighborhood of a source vertex in a delta, then the worker checks the corresponding Bloom filter from the Bloom cache. Having this, the backend store is only fetched in case the bloom filter returns a positive response.

B. Querying

Clock-G supports several types of temporal graph queries as defined below.

1) Point-based local queries: This type of queries retrieve the N-Hop neighborhood of a vertex given several predicates. These predicates are used to express constraints on the *id*, label, or values of properties of the starting vertex, length of the traversal, type of the edges and the time instant at which all the vertices and edges must be valid. For the industrial use-case presented in this paper, a local query (N-Hop queries, i.e.) can retrieve the surroundings of a malfunctioning machine including sensors, other machines, products up to a fixed length, given a set of property predicates at a given time instant. For example, the property predicates can express that the property representing the status of the machine should be

equal to 'Out-of-service' at the requested time instant. Clock-G returns the traversal in the form of a Bag of records s.t. each record u is a tuple mapping a field name k to a value v s.t. $u(k) = v$. The notation $dom(u)$ is used to denote the domain of u. Now, each tuple represents a valid path that maps a depth d', s.t. $0 \leq d' \leq d$ where d is the depth of the traversal, to the *id* of the vertex that belongs to the path at d'. Towards introducing the algorithm of the traversal, we define the following unary operators:

- **Select:** $\sigma_c(r)$ This operator selects the tuples of a bag r for which the propositional formula c holds and returns a new bag with the selected tuples.
- **Distinct:** $\vartheta(r)$ This operator returns a bag containing all distinct tuples of a bag r s.t. it de-duplicates tuples sharing the same set of fields that are mapped to same values.
- **Projection:** $\pi_{f_0,f_1,...,f_n}(r)$ This operator keeps a specific set of the names of the fields $(f_0, f_1, ..., f_n)$ of a bag r. Note that the tuples are not de-duplicated, thus the result bag can have the same number of tuples as the input bag.
- **Expand:** $\xrightarrow[i,(x,y)]{(r)}$ This operator expands a tuple u, if u is contained in a bag r and maps the field i to the value x s.t. $u(i) = x$, by adding the field $i + 1$ to $dom(u)$ s.t. $u(i + 1) = y$ and keeps u unchanged otherwise. Then, the operator adds the tuple u to the returned bag.
- **Shrink** $\xleftarrow[i,(x,y)]{(r)}$ This operator shrinks a tuple u, if u is contained in a bag r and maps fields i and $i+1$ to values x and y s.t. $u(i) = x$ and $u(i+1) = y$, by removing the field $i + 1$ from dom(u) and keeping u unchanged otherwise. Then, the operator adds the tuple u to the returned bag.

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

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

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

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

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

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

2) Range-based local queries: are similar to the point-based local queries. While point-based queries take a time instant as an input, range queries take a time interval with which the validity intervals of the returned vertices and edges should overlap. Furthermore, we implemented a special type of range-based local queries that we refer to as **time-increasing paths**. For this type of temporal paths, every outgoing edge of a vertex should have occurred after the incoming edges to the same node. In the case of Thing'in, this type of temporal paths finds applicability in logistic chains. For example, a product starting from a given station can reach another station if there exists a sequential path representing product transfer between the stations.

3) Global queries: Point-based global queries retrieve the state of a sub-graph given the labels of the vertices, types of edges and a time instant at which all the returned vertices and edges must be valid. Similarly, range-based global queries retrieves a sub-graph that was valid during a time range meaning that the validity interval of the returned nodes and edges should intersect with the requested time interval. Note that, one can choose to return a full snapshot of the graph which translates to returning all the nodes and edges without any constraints on the nodes and edges. In the case of Mo.Di.Flu, a global query can recover the status of all the machines and their connections with other devices at a given time instant or follow their evolution during a time interval. Note that we omit the pseudo-code of the range-based point and global queries for the lack of space.

VI. Evaluation

In this section, we present the evaluation of the overall performance of Clock-G. Our main goal is to validate that the δ-*Copy+Log* produces a compromise between the performances of traditional methods *Copy+Log* and *Log*. Another goal is to show that one can tune the performance of Clock-G by

Algorithm 1: N-Hop traversal

Input: id of the source vertex s; Number of hops d; Time instant t
Output: Traversal T

$T \leftarrow InitBag(0, s)$;// Initialize traversal
// Find checkpoints
$ids \leftarrow GetCheckpointIDs(t)$;
for $n \leftarrow 0, 1, 2, ..., (d-1)$ **do**
  $nextVertices \leftarrow \vartheta(\pi_n(T))$;
  for $id \in ids$ **do**
    for $i \in nextVertices$ **do**
      if *id corresponds to a snapshot* **then**
        $neighbors \leftarrow GetNeighbors(i, id)$ **for** $j \in neighbors$ **do**
          // Expand traversal
          $T \leftarrow \xrightarrow[n:(i,j))]{(T)}$;
      else if *id corresponds to a delta* **then**
        if *CheckBloomFilters(i, id)* **then**
          $ops \leftarrow GetOperations(i, id)$;
      else
        $ops \leftarrow GetOperationsT(i, id, t)$;
      for $\epsilon \in ops$ **do**
        // Get event from operation
        $e \leftarrow f^E(\epsilon)$;
        // Get target vertex from operation
        $j \leftarrow Target(\alpha(f^G(\epsilon)))$;
        if $e == a$ **then**
          // Expand traversal
          $T \leftarrow \xrightarrow[n:(i,j))]{(T)}$;
        else
          // Shrink traversal
          $T \leftarrow \xleftarrow[n:(i,j)]{(T)}$;
  // Select d hops paths in Traversal
  $T \leftarrow \sigma_{d \in dom(u)}(T)$;

choosing the adequate configuration of the system parameters. Based on this evaluation, one should be able to configure the parameters of Clock-G in order to account for the threshold of accepted query latency and available storage resources. Furthermore, we want to compare Clock-G with a non-temporal graph database to highlight that available commercial systems are not optimized for handling temporal graphs.

A. Experimental setup

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

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

Although Clock-G is introduced in this paper to answer the requirements of a specific use case, it can also be deployed for other categories of temporal graphs. Having this, we used different real-world datasets such as DBLP dataset (DS_{DBLP} [41], Stack overflow dataset (DS_{stack}) and Wiki talk dataset (DS_{wiki}) [16]. We assume that these graphs are growth-only graphs in the sense that once an edge is added, it will not be deleted. To evaluate the time increasing paths, we used the CitiBike dataset[7] (DS_{citi}) which includes information of bike trips between stations in New York city. We transformed 3 months of data into a series of timestamped graph updates.

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

TABLE II: Characteristics of the generated graphs

Dataset	$\|V\|$	$\|E\|$	p_a	Time span (Days)
DS_{p_a}	500K	10 M	0.9, 0.74, 0.6	116
DS_{stack}	2.6 M	63.4 M	_	2774
DS_{DBLP}	1.8 M	29.5 M	_	29930
DS_{wiki}	1.1 M	7.8 M	_	2320
DS_{citi}	1K	2.5M	_	90

B. Space usage and query execution time

We evaluate the disk space usage and query execution time with different configurations by tuning the system parameters. We compute local queries, detailed in Section V-B1, by randomly choosing $1k$ vertices to be the starting nodes of the traversals. Global queries, detailed in Section V-B3, retrieve a snapshot of the graph that was valid at a requested time instant. It should be highlighted that time instants used in queries are uniformly chosen within the time span of the datasets in order to avoid a biased distribution that favors only time instants that are closer to checkpoints.

Comparison with state-of-the-art methods. We compare the results of the proposed method δ-*Copy+Log* with those of the traditional methods *Copy+Log* and *Log*. Now, the implementation of *Copy+Log* in Clock-G is fairly straightforward since it consists of setting parameter M to 1. However, the

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

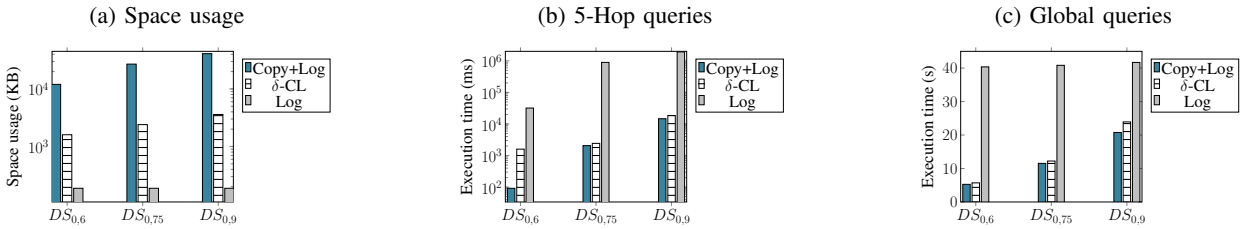


Fig. 4: Comparison with state-of-the-art techniques

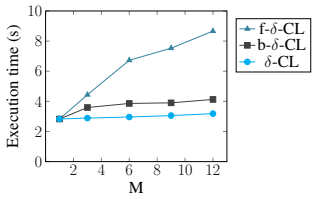


Fig. 5: Evaluation of 8 Hop queries with f-δ-CL, b-δ-CL and δ-CL methods on dataset $DS_{0,6}$ with N set to $10K$

implementation of the *Log* method consists of creating time windows of size N that are not bounded by any checkpoint. That is, the evaluation of a query with a requested time instant t implies reading from time windows whose time intervals fall before or contains t. Figures 4a, 4b and 4c display the space usage, the execution time of 5-Hops and global queries on datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$. Note that, we set the system parameters N and M to $10k$ and 12, respectively. The results are directly inline with previous findings on the apparent trade-off between the space usage and execution time of the *Copy+Log* and *Log* methods. Such that the *Copy+Log* results in a space usage that is 144 times higher than that resulting from the *Log* approach whereas it results in a query execution time of 5-Hop queries that is 433 times faster then that resulting from the *Log* approach. It is clear from the results that the proposed δ-*Copy+Log* method offers a compromise between the *Log* and *Copy+Log* as it reduces the storage obtained by the *Copy+Log* by a factor of 12 whereas it reduces the query execution time offered by the *Log* approach by a factor of 340. For the interest of space, we limit the discussion here to the results obtained from the evaluation on the $DS_{0.75}$.

Validating the use of Bloom filters. As previously discussed, storing deltas instead of snapshots induces a query execution time overhead. Hence, we developed optimization techniques to reduce the induced query latency. We evaluated the execution time of queries with 3 methods namely: f-δ-CL, b-δ-CL and δ-CL. The f-δ-CL method, standing for forward-δ-*Copy+Log*, follows the same approach as the δ-*Copy+Log* with the difference of storing only forward time windows and deltas and omitting the use of Bloom filters. The b-δ-CL, standing for bloomed-δ-*Copy+Log*, consists of adding Bloom filters to the f-δ-CL. Finally, the δ-CL refers the δ-*Copy+Log* method, hence, consists of adding forward and backward time windows and deltas to the b-δ-CL. Indeed, comparing the aforementioned methods emphasizes the gain of adding Bloom filters and that of storing backward time windows and deltas, separately. Figure 5 shows the average execution time of traversal queries with a fixed depth equals to 8 on the dataset $DS_{0,6}$ while increasing the system parameter M from 1 to 12. Note that the system parameter N is set to $10k$. It is clear that using the f-δ-CL significantly increases the execution time with the increase of M. Now, adding Bloom filters to the b-δ-CL method reduces the execution time as compared to the f-δ-CL s.t. the speedup can reach 52% for $M = 12$. Furthermore, adding forward and backward time windows and deltas to the δ-CL speeds up the traversals by a factor of 23% as compared to the b-δ-CL. The execution time overhead of the f-δ-CL is equal to 206% when the value of M is increased from 1 to 12. Indeed, this overhead is reduced to $12,5\%$ when using the δ-CL. The rest of evaluations depicted in this section are executed using the δ-CL method s.t. the f-δ-CL and b-δ-CL methods are used as proof-of-concept implementations and will not be further discussed.

Variation of N and M. We evaluate the disk space usage in function of system parameters N and M. Figure 6a shows the disk space usage of checkpoints for different configurations s.t. each configuration is set with a value of N in $\{10k, 100k, 250k\}$ and a value of M in $\{1, \ldots, 12\}$. We ingest the dataset $DS_{0,6}$ in Clock-G with every combination of the values of parameters N and M. It is notable that the smaller is the value of N, the higher is the space usage of checkpoints. Now, increasing M while fixing the value of N significantly reduces the space usage as compared to the *Copy+Log* method (corresponding to $M = 1$). Besides, the disk space gain obtained by increasing the value of M is more significant for smaller values of N which is intuitively justified by the fact that lower values of N cause the creation of more time windows. Consequently, a larger number of snapshots will be substituted by deltas which leads to a more significant overall space reduction as compared to that obtained with higher values of N. We also evaluate the variation of N and M on the execution time of 5-Hop and global queries with the same system configuration whose results are given in Figures 6b and 6c. For these results, it is notable that the higher is the value of N, the higher is the execution time. That is, less checkpoints are created in configurations tuned with larger time windows (higher values of N) which increases, in general, the duration between requested time instants and the time instant of the closest snapshots that are used as starting points for query evaluation.

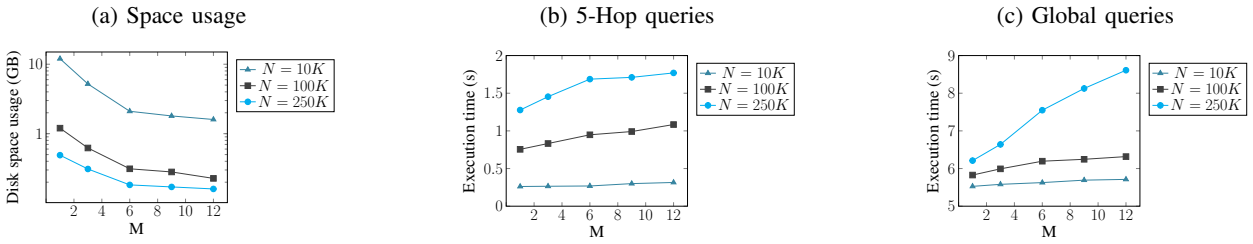


Fig. 6: Evaluation of the disk space usage and execution time of queries while varying the system's configuration parameter N. The evaluation is conducted on the synthetic dataset $DS_{0,6}$

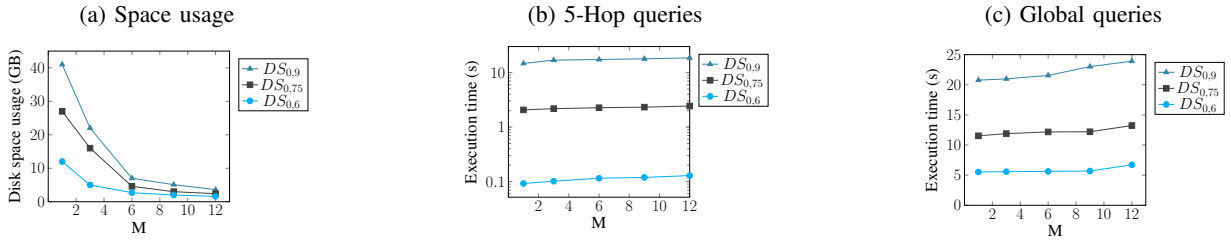


Fig. 7: Evaluation of the disk space usage and execution time of queries with $N = 10K$. The evaluation is conducted on synthetic datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$ having each a different value of parameter p_a

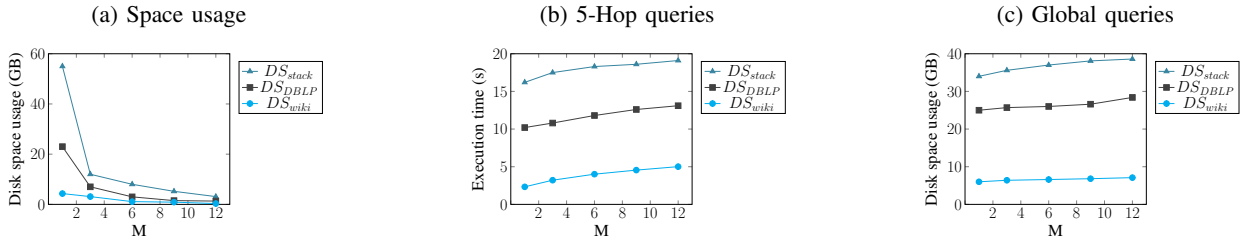


Fig. 8: Evaluation of the disk space usage and execution time of queries with $N = 250K$. The evaluation is conducted on real datasets DS_{stackO}, DS_{DBLP} and DS_{wiki}

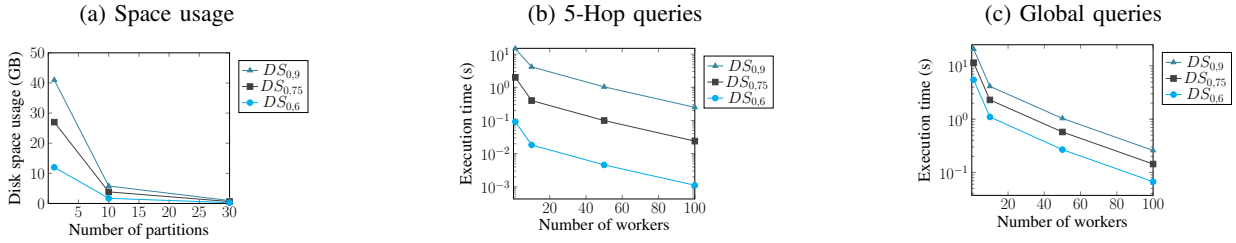


Fig. 9: Evaluation of the effect of varying the number of partitions and backend connector workers on the disk space usage and execution time of queries with $N = 10K$. The evaluation is conducted on synthetic datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$

Variation of p_a and M. We study the effect of varying the system parameter M and the dataset characteristic p_a on the disk space. Figure 7a displays the space usage occupied by the checkpoints of datasets $DS_{0,6}$, $DS_{0,75}$ and $DS_{0,9}$ with different system configurations corresponding each to a value of M in $\{1, \dots 12\}$. The obtained results demonstrate that the space usage strictly decreases with the increase of the value of M. Besides, superior space gains are obtained for graphs with higher probability of additions. This is due to the fact that snapshots of such graphs are more space consuming which implies that replacing them with deltas emphasizes more significantly the space gain.

We also evaluate the effect of varying p_a and the system parameter M on the execution time of 5-Hop queries and Global queries. Figures 7b and 7c show that the execution time of queries increases with the increase of parameter p_a. That is, the average degree of a node increases with the value of p_a which results in a higher number of computations to evaluate the result of a query.

Evaluation on real datasets. We evaluate the space gain obtained from ingesting real-world datasets following the δ-*Copy+Log* method. Figure 8a displays the disk space usage

of checkpoints created by the ingestion of datasets DS_{stack}, DS_{DBLP} and DS_{wiki} into Clock-G while increasing the value of the system parameter M from 1 to 12. It can be noticed that the δ-*Copy+Log* approach markedly reduces the space usage occupied by the dataset when increasing the value of M from 1 to 12. We also evaluate 5-Hop traversal and global queries on these real-world datasets. The obtained results, given in Figures 8b and 8c, validate that our solution gives clearly good results as compared to the *Copy+Log* method such as it significantly reduces the space usage while adding a slight query execution time overhead.

Variation of the number of partitions and backend workers. We evaluate the space usage by changing the total number of partitions. As depicted in Section V-A3, we separate each graph element store into a number of partitions. It is clear from Figure 9a that the space gain decreases with the increase of the total number of partitions. Figure 9b and 9c validate that the query execution time is reduced with the increase of the number of backend connector workers. Indeed, this is due to the fact that fetching tasks are executed by a pool of backend workers in parallel.

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

Figures 10a and 10b show the ingestion throughput and space usage of Clock-G, Neo4j, Neo4j$_i$. It can be derived from the plots that Clock-G significantly outperforms Neo4j and Neo4j$_i$. This difference in the ingestion throughput is due to the fact that performing deletes incur the update the property *tEnd*, which induces a read operation to match the graph element before setting the new property value. Another key factor is the parallelism of Clock-G. That is, a number of backend workers are delegated to batch each, in parallel, a partition of the received graph operations. However, inserting graph updates in parallel into Neo4j is not possible since the chronological order of the updates should be guaranteed.

Figures 10c and 10d show the execution time of time increasing path queries while varying the depth and time range of the queries. It should be noted that for each depth and time range, we run the query on all the nodes of the graph and plot the average of the obtained results. It can be noticed from the plots that Clock-G outperforms Neo4j and Neo4j$_i$ such

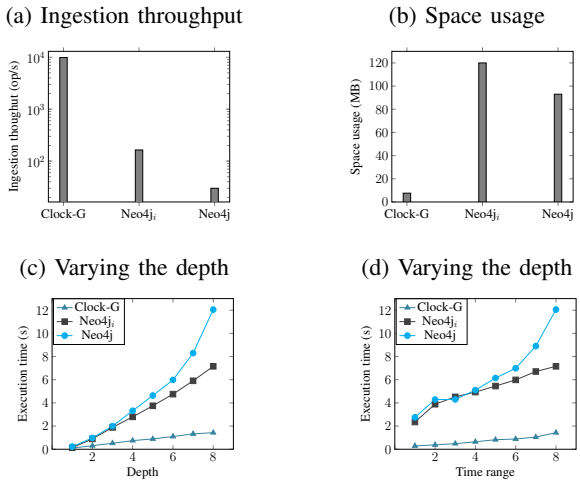


Fig. 10: Evaluation of the ingestion throughput and space usage of Clock-G, Neo4j and Neo4j$_i$

that the difference is more significant with the increase of the depth and time range of the query. Indeed, when evaluating a time range query in Clock-G, the search space is trimmed to a number of selected time windows whose time interval intersects with the time range of the query. Furthermore, backend connector workers compute the result of the query in parallel such that each worker computes a sub-result that contains vertices and edges belonging to its local partitions. This, however, is not possible with Neo4j and Neo4j$_i$ even when indexes are used to accelerate the traversals.

The results obtained from this experiment highlight the need to develop a graph management system with a native temporal support instead of using an existing non-temporal commercial system.

VII. Conclusion

In this paper, we introduced a temporal graph management system designed with a space-efficient storage technique in order to keep pace with the requirement of querying the history of the graphs managed by the platform Thing'in. Our proposed storage technique δ-*Copy+Log* differentiates from the *Copy+Log* by storing deltas instead of snapshots. Besides, we referred to forward and backward data storage and retrieval in order to accelerate the query evaluation time. We also provided a detailed description of the architecture of Clock-G and the special implementation of Bloom filters that permits the acceleration of traversal queries. We conducted an evaluation test on synthetic and real-world graphs that validate the efficiency of the δ-*Copy+Log* method as compared to traditional methods. The results demonstrate that our solution reduces significantly the space usage of the *Copy+Log* method and the execution time of the *Log* method, hence mitigates the space-execution time tradeoff limiting the efficiency of these methods. In a future perspective, we plan to integrate a temporal querying language into Clock-G. Despite the existing work on temporal query languages [42]–[46], this problem has not been fully addressed in the context of graph databases.

REFERENCES

[1] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2019.

[2] A. Debrouvier, E. Parodi, M. Perazzo, V. Soliani, and A. Vaisman, "A model and query language for temporal graph databases," *The VLDB Journal*, pp. 1–34, 2021.

[3] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch, "Time-varying social networks in a graph database: A neo4j use case," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2484425.2484442

[4] H. Huang, J. Song, X. Lin, S. Ma, and J. Huai, "Tgraph: A temporal graph data management system," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2469–2472. [Online]. Available: https://doi.org/10.1145/2983323.2983335

[5] B. George and S. Shekhar, "Time-aggregated graphs for modeling spatio-temporal networks," in *Journal on Data Semantics XI*. Springer, 2008, pp. 191–212.

[6] B. George, J. M. Kang, and S. Shekhar, "Spatio-temporal sensor graphs (stsg): A data model for the discovery of spatio-temporal patterns," *Intelligent Data Analysis*, vol. 13, no. 3, pp. 457–475, 2009.

[7] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 997–1008.

[8] T. Zhang, Y. Gao, L. Qiu, L. Chen, Q. Linghu, and S. Pu, "Distributed time-respecting flow graph pattern matching on temporal graphs," *World Wide Web*, vol. 23, no. 1, pp. 609–630, Jan. 2020. [Online]. Available: https://doi.org/10.1007/s11280-019-00674-0

[9] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820–842, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000002918295

[10] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Phys. Rev. E*, vol. 84, p. 016105, Jul 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.84.016105

[11] U. Redmond and P. Cunningham, "Subgraph isomorphism in temporal networks," *CoRR*, vol. abs/1605.02174, 2016. [Online]. Available: http://arxiv.org/abs/1605.02174

[12] P. Holme, "Modern temporal network theory: a colloquium," *The European Physical Journal B*, vol. 88, no. 9, p. 234, 2015.

[13] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 145–156.

[14] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 601–610. [Online]. Available: https://doi.org/10.1145/3018661.3018731

[15] K. Semertzidis and E. Pitoura, "A hybrid approach to temporal pattern matching," *arXiv preprint arXiv:2001.01661*, 2020.

[16] D. Wen, Y. Huang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently answering span-reachability queries in large temporal graphs," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1153–1164.

[17] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *ACM Transactions on Storage (TOS)*, vol. 11, no. 3, pp. 1–34, 2015.

[18] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, "Chronograph: A versioned tinkerpop graph database," in *Data Management Technologies and Applications*, J. Filipe, J. Bernardino, and C. Quix, Eds. Cham: Springer International Publishing, 2018, pp. 237–260.

[19] W. D. Vijitbenjaronk, J. Lee, T. Suzumura, and G. Tanase, "Scalable time-versioning support for property graph databases," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 1580–1589.

[20] J. Byun, S. Woo, and D. Kim, "Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 3, pp. 424–437, 2020.

[21] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[22] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[23] J. S. M. Verhofstad, "Recovery techniques for database systems," *ACM Comput. Surv.*, vol. 10, no. 2, p. 167–195, Jun. 1978. [Online]. Available: https://doi.org/10.1145/356725.356730

[24] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, p. 287–317, Dec. 1983. [Online]. Available: https://doi.org/10.1145/289.291

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

[26] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, "Chronograph: A versioned tinkerpop graph database," in *International Conference on Data Management Technologies and Applications*. Springer, 2017, pp. 237–260.

[27] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys (CSUR)*, vol. 31, no. 2, pp. 158–221, 1999.

[28] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon, "Analyzing complex data in motion at scale with temporal graphs," 2020.

[29] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The g* graph database: efficiently managing large distributed dynamic graphs," *Distributed and Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2015.

[30] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[31] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch, "Time-varying social networks in a graph database: a neo4j use case," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[32] 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)*. IEEE, 2013, pp. 228–234.

[33] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[34] G. Koloniari, D. Souravlias, and E. Pitoura, "On graph deltas for historical queries," *arXiv preprint arXiv:1302.5549*, 2013.

[35] U. Khurana and A. Deshpande, "Storing and analyzing historical graph data at scale," *arXiv preprint arXiv:1509.08960*, 2015.

[36] W. Han, K. Li, S. Chen, and W. Chen, "Auxo: a temporal graph management system," *Big Data Mining and Analytics*, vol. 2, no. 1, pp. 58–71, 2018.

[37] L. Xiangyu, L. Yingxiao, G. Xiaolin, and Y. Zhenhua, "An efficient snapshot strategy for dynamic graph storage systems to support historical queries," *IEEE Access*, vol. 8, pp. 90 838–90 846, 2020.

[38] A. Montanari and J. Chomicki, *Time Domain*. Boston, MA: Springer US, 2009, pp. 3103–3107. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_427

[39] C. S. J. J. Clifford, R. Elmasri, C. Dyreson, F. G. W. K. N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. N. E. P. B. Pernici, J. F. R. N. L. Sarda, and M. R. S. A. Segev, "A consensus glossary of temporal database concepts," *SIGMOD record*, vol. 23, no. 1, 1994.

[40] J. Hölsch and M. Grossniklaus, "An algebra and equivalences to transform graph patterns in neo4j," in *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016)*, ser. CEUR Workshop Proceedings, T. Palpanas and K. Stefanidis, Eds., no. 1558, 2016. [Online]. Available: http://ceur-ws.org/Vol-1558/paper24.pdf

[41] J. Kunegis, "Konect: The koblenz network collection," ser. WWW '13 Companion. New York, NY, USA: Association for Computing Machinery, 2013, p. 1343–1350. [Online]. Available: https://doi.org/10.1145/2487788.2488173

[42] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev,

M. D. Soo, and S. M. Sripada, "Tsql2 language specification," *SIGMOD Rec.*, vol. 23, no. 1, p. 65–86, Mar. 1994. [Online]. Available: https://doi.org/10.1145/181550.181562

[43] F. Grandi, "T-sparql: A tsql2-like temporal query language for rdf." in *ADBIS (local proceedings)*. Citeseer, 2010, pp. 21–30.

[44] M. Perry, P. Jain, and A. P. Sheth, *SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries*. Boston, MA: Springer US, 2011, pp. 61–86. [Online]. Available: https://doi.org/10.1007/978-1-4419-9446-2_3

[45] J. Tappolet and A. Bernstein, "Applied temporal rdf: Efficient temporal querying of rdf data with sparql," in *The Semantic Web: Research and Applications*, L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, R. Mizoguchi, E. Oren, M. Sabou, and E. Simperl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–322.

[46] F. Rizzolo and A. A. Vaisman, "Temporal xml: modeling, indexing, and query processing," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 17, no. 5, pp. 1179–1212, 2008.