Analyzing Complex Data in Motion at Scale with Temporal Graphs

Thomas Hartmann*, Francois Fouquet*, Matthieu Jimenez*, Romain Rouvoy† and Yves Le Traon*
*University of Luxembourg, Luxembourg, firstname.lastname@uni.lu
†Univ. Lille / Inria / IUF, France, romain.rouvoy@inria.fr

***Abstract*—Modern analytics solutions succeed to understand and predict phenomenons in a large diversity of software systems, from social networks to Internet-of-Things platforms. This success challenges analytics algorithms to deal with more and more complex data, which can be structured as graphs and evolve over time. However, the underlying data storage systems that support large-scale data analytics, such as time-series or graph databases, fail to accommodate both dimensions, which limits the integration of more advanced analysis taking into account the history of complex graphs, for example. This paper therefore introduces a formal and practical definition of temporal graphs. Temporal graphs provide a compact representation of time-evolving graphs that can be used to analyze complex data in motion. In particular, we demonstrate with our open-source implementation, named GREYCAT, that the performance of temporal graphs allows analytics solutions to deal with rapidly evolving large-scale graphs.**

***Index Terms*—Data analytics, graph databases, large-scale graphs, time-evolving graphs**

I. INTRODUCTION

The data deluge induced by large-scale distributed systems has called for scalable analytics platforms. Modern analytics solutions succeed to understand and predict phenomenons in a large diversity of software systems, from social networks to Internet-of-Things platforms. Graphs are increasingly being used to structure and analyze such complex data [1], [2], [3]. However, most of graph representations only reflect a snapshot at a given time, while reflected data keeps changing as the systems evolve. Understanding temporal characteristics of time-evolving graphs therefore attracts increasing attention from research communities [4]—*e.g.*, in the domains of social networks, smart mobility, or smart grids [5].

Yet, state-of-the-art approaches fail to provide a scalable solution to effectively support time in graphs. In particular, existing approaches represent time-evolving graphs as sequences of full-graph snapshots [6], or they use a combination of snapshots and deltas [7], which requires to reconstruct a graph for a given time, as depicted in Figure 1. However, full-graph snapshots tend to be expensive in terms of memory requirements, both on disk and in-memory. This overhead becomes even worse when data from several snapshots need to be correlated, which is the case for most of advanced analytics [5], [6], [8]. Another challenging issue related to snapshots relates to the snapshotting frequency: regardless of changes, for any change in the graph, or only for the major changes, which results in a tradeoff between duplicating data and feeding analytics with up-to-date metrics. This is crucial when data evolves rapidly and at different paces for different elements in the graph, like it is for example the case with sensor data in domains like the *Internet-of-Things* (IoT) or *Cyber-Physical Systems* (CPS) [5].

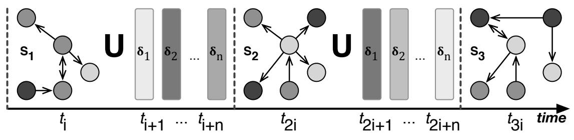


Fig. 1. Snapshots (S_i) and deltas (δ_n) of a time-evolving graph

An alternative to snapshotting consists in combining graphs with time series databases [9], by mapping individual nodes to time series. However, this becomes quickly limited when large parts of the graph evolve over time, inducing multiple time queries to explore the graph. Moreover, the description and the evolution of relationships among the nodes of the graph are rather hard to model within a time series database.

In this paper, we therefore introduce a novel *temporal graph* data model and storage, which allow analytics platforms to represent time-evolving graphs in an efficient manner. Most notably, our approach completely adopts a radically new approach by using an innovative, node-scale, and on-demand cloning approach. In temporal graphs, each node can evolve independently in time, while graphs are traversed for arbitrary timestamps.

We demonstrate with our open-source implementation of temporal graphs, named GREYCAT[1], that the performance of this data model allows analytics solutions to deal with rapidly evolving large-scale graphs. We compare our solution with a complete snapshotting approach and a combination of graph and time series databases.

The remainder of this paper is organized as follows. In Section II, we first formalize the semantics of our

[1]https://github.com/datathings/greycat

temporal graph data model. Then, in Section III, we present and discuss implementation details of this data model within GREYCAT, our open-source graph framework. We thoroughly evaluate the temporal aspects of GREYCAT in Section IV, before discussing the related work in Section V and concluding the paper in Section VI.

II. DEFINITION OF THE TEMPORAL GRAPH SEMANTICS

A graph G is commonly defined as an ordered pair consisting of a set V of nodes or vertices and a set E of edges: $G = \{V, E\}$. We define a slightly different semantics for our temporal graph by distinguishing between a node and its *state*. We define a node as a *conceptual identifier* that is mapped to its state, which we refer to as *state chunk*. It contains the values of all attributes and edges that belong to a node. Attributes are typed according to one of the following primitive types: `int`, `long`, `double`, `string`, `bool`, and `enumeration`. Formally, we define a state chunk as: The state chunk c of a node n is $c_n = (A_n, R_n)$, where A_n is the set of attribute values of n and R_n is the set of relationship values from n to other nodes. Unlike other graph models (*e.g.*, Neo4J [10]), ours does not support edge attributes (like the OO model). However, any edge attribute can be modeled leveraging an intermediate node. Next, we define the function $read(n)$ to resolve the state chunk of a node. We use this function to define a graph G as: $G = \{read(n), \forall n \in N\}$, where N is the set of nodes. Unlike common graph definitions, temporal graphs are not defined statically, but dynamically—*i.e.*, their are created as the result of the evaluation of the $read(n)$ function over all nodes.

The separation between the concept of a node and its state chunk is essential for our approach. First, it enables the implementation of a lazy loading mechanism—*i.e.*, by loading state chunks on-demand into main memory, while the graph is traversed. As further discussed in Section III, we build on key/value stores as storage backends for the temporal graphs. This mapping of a graph to keys and values is similar to what is proposed in [11]. Secondly, it allows to define different states for each node depending on the time. Therefore, we extend the previous definition of a graph with temporal semantics. We override the function $read(n)$ with $read(n, t)$, where $t \in T$ and T is a totally ordered sequence of all possible timepoints: $\forall t_i, t_j \in T : t_i \leq t_j \lor t_j \leq t_i$. Next, we extend the definition of a state chunk with a temporal version: $c_{n,t} = (A_{n,t}, R_{n,t})$, where $A_{n,t}$ and $R_{n,t}$ are the sets of resolved attributes and relationships, for the node n at time t. Then, we define a temporal graph as follows: $TG(t) = \{read(n, t), \forall n \in N\}, \forall t \in T$. Every node of the TG can evolve independently and, as timepoints can be compared, they naturally form a chronological order. We define that every state chunk belonging to a node in a TG is associated to a timepoint and can therefore be queried along this chronological order in a sequence $TP \subseteq T$. We call this ordered sequence of state chunks the *timeline* of a node. The timeline tl of a node n is defined as $tl_n = \{c_{n,t}, \forall t \in TP \subseteq T\}$. We define three node operations:

$insert(c_{n,t}, n, t)$: $(c \times N \times T) \mapsto void$ as the function that inserts a state chunk c in the timeline t of a node n, such as: $tl_n := tl_n \cup \{c_{n,t}\}$.

$read(n, t)$: $(N \times T) \mapsto c$ is the function that retrieves, from the timeline tl_n, and up until time t, the most recent version of the state chunk of n which was inserted at timepoint t_i:

$$read(n,t) = \begin{cases} c_{n,t_i} & \text{if } (c_{n,t_i} \in tl_n) \\ & \wedge (t_i \in TP) \wedge (t_i < t) \\ & \wedge (\forall t_j \in TP \rightarrow t_j < t_i) \\ \emptyset & \text{otherwise} \end{cases}$$

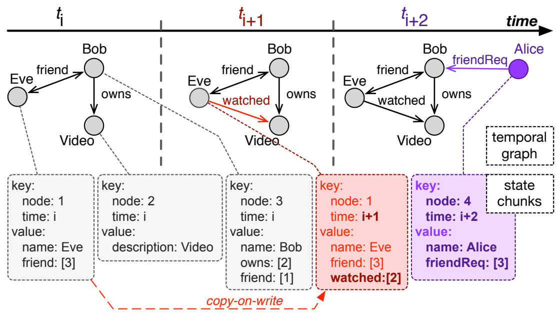
$remove(n, t)$: $(N \times T) \mapsto void$ is the function that removes a node n and the associated state chunks from the time t.

Based on these definitions, although timestamps are discrete, they logically define intervals in which a state chunk can be considered as *valid* within its timeline. When executing $insert(c_{n_1,t_1}, n_1, t_1)$ and $insert(c_{n_1,t_2}, n_1, t_2)$, we insert 2 state chunks c_{n_1,t_1} and c_{n_1,t_2} for the same node n_1 at two different timepoints with $t_1 < t_2$. We define that c_{n_1,t_1} is valid in the open interval $[t_1, t_2[$ and c_{n_1,t_2} is valid in $[t_2, +\infty[$. Thus, an operation $read(n_1, t)$ resolves $\emptyset$ if $t < t_1$, c_{n_1,t_1} when $t_1 \leq t < t_2$, and c_{n_1,t_2} if $t \geq t_2$ for the same node n_1. After executing $remove(n_1, t_3)$, $read(n_1, t)$ resolves $\emptyset$ if $t \geq t_3$. Since state chunks with this semantics have temporal validities, relationships between nodes also have temporal validities. This leads to *temporal relationships* between TG nodes and forms a natural extension of relationships in the time dimension. This temporal validity definition follows and extends our previous work [5]. It enables to transparently navigate inside the graph without considering time for every navigation step, by always loading the last valid version relative to the node the navigation started from.

III. GREYCAT: A TEMPORAL GRAPH IMPLEMENTATION

A. Mapping nodes to state chunks

The proposed temporal graph data model is a conceptual view of data to represent and analyze time-evolving complex data. Internally, we structure the data of a temporal graph as an unbounded set of *state chunks*. Therefore, we map the conceptual nodes (and relationships) of a temporal graph to *state chunks*. State chunks are the internal data structures reflecting a temporal graph and, at the same



t_i
t_{i+1}
t_{i+2}
time
Bob
Eve
friend
owns
Video
watched
Alice
friendReq
temporal graph
state chunks
key: node: 1 time: i value: name: Eve friend: [3]
key: node: 2 time: i value: description: Video
key: node: 3 time: i value: name: Bob owns: [2] friend: [1]
key: node: 1 time: i+1 value: name: Eve friend: [3] watched:[2]
key: node: 4 time: i+2 value: name: Alice friendReq: [3]
copy-on-write

Fig. 2. Mapping of temporal graphs to state chunks

time, also used for storing the temporal graph data to persistent storage. A state chunk contains, for every attribute of a node, the name and value of the attribute and, for every outgoing relationship, the name of the relationship and a list of identifiers of the referenced state chunks. Figure 2 depicts a concrete example of the mapping of nodes to state chunks, according to the semantic definitions of Section II.

At time t_i (start of the temporal graph), GREYCAT maps the nodes and the relationships to 3 state chunks: Bob, Eve, and Video. At time t_{i+1}, the graph evolves to declare a relationship watched from Eve to Video. Since this evolution only affects Eve, GREYCAT only creates an additional state chunk for Eve at time t_{i+1} by cloning and modifying the previous version of Eve's state chunk (using copy-on-write). All other nodes are therefore kept unchanged at time t_{i+1}. At time t_{i+2}, Bob meets Alice, who sends a friend request to Bob. As only Alice's status changes, GREYCAT only needs to create a new state chunk for Alice from time t_{i+2}.

In this example, the graph contains 10 different conceptual nodes and 12 relationships (counting each bidirectional relation as two edges) and evolves along 3 different timestamps, but GREYCAT only stores 5 state chunks to model the whole temporal graph. Whenever the temporal graph is explored, the correct state chunks are retrieved depending on the requested time.

B. Lazy-loading state chunks

State chunks are the units of storage in GREYCAT. They are stored on disk and loaded into main memory while the graph is explored or when nodes are explicitly requested. The loading of state chunks is achieved lazily by GREYCAT, because only attributes and sets of identifiers are loaded. This theoretically allows GREYCAT to process temporal graphs of unbounded size even with restricted main memory. For persistent storage of state chunks, we rely on key/value stores by using the tuple of $(node, time)$ as key and the state chunk as value. We serialize chunk states into Base64 encoded `blobs`. This format reduces the required interface to insert, read, and remove state chunks to a persistent data store. It allows to use different storage backends depending on the requirements of an application: from in-memory key/value stores up to distributed and replicated NoSQL databases. For fault tolerance and concurrency, we use a per-node lock policy and rely on the underlying storage technology to ensure concurrency and distribution. With a node we also lock the index structures associated to it (cf. Section III-D). Another consistency approach, based on consistent global checkpoints, is discussed in [12]. All clients reading the node (at the same time) see updates on it. This mapping approach copies state chunks only on-demand—*i.e.*, copy-on-write—and ensures efficient read and write operations at any point in time. Basically, it enables analytics algorithms to be executed with constant memory requirements.

C. Caching state chunks

Despite of having a positive effect on memory, lazy loading significantly increases input/output (I/O) operations. Therefore, we rely on caching mechanisms to reduce I/O operations, which means that some state chunks are kept in memory, based on their probability of being reused. In contrary to pure key/value storages, our state chunks have semantic relationships, which can be leveraged by the caching mechanism. For instance, if contiguous timepoints of the same node are loaded, the temporal index has a high probability to be reused. We build our caching mechanism as a *Least-Recently-Used* (LRU) cache, where each read operation is taken into account for the cache victim eviction computation. To reflect semantic relationships, we count in temporal index LRU scores: the number of state chunks using them. This way, we are giving chances for temporal indexes to be reused to access other timepoints.

D. Indexing state chunks using red-black trees

Temporal graphs can be composed by highly volatile nodes, characterized by a very long timeline of millions of timepoints. Such volatile nodes are for example needed to store sensor data collected by IoT devices. In order to efficiently execute temporal queries, these timepoints must be indexed. For scalability reasons, GREYCAT relies on a lazy loading mechanism for retrieving nodes [5] . In a similar way, we define temporal indexes with a semantic to avoid loading millions of timepoints, *i.e.*, the full temporal index, if only parts of the timepoints are actually matching a query. Queries in GREYCAT are always precise.

Temporal characteristics of timepoints—*i.e.*, regularity, periodicity—make some indexing approaches more suitable than others [13]. For non-monotonic measurements, balanced trees offer one of the best compromises between read and insert performance. In particular, *Red-Black Trees* (RBT) are one of the most adopted structures to index non-monotonic time-series [14]. However, RBTs are in-memory structures that cannot be partially loaded due to their balanced hierarchy. To workaround this limitation, we defined an adaptive multi-layer on top of RBTs in order to split temporal indexes in pieces that can be lazily loaded.

Our approach, named *Adaptive Multi-Layered Red-Black Tree* (AMT), is based on the same balancing principle than RBTs are, but at a coarser granularity to support efficient lazy loading. Therefore, we defined the notion of a *supertree* that can index fixed-sized subtrees, based on their oldest timepoint. For every read and insert operation, the supertree is first loaded and used to locate the relevant subtrees, which in turn are used to load an updated index. In addition, to compact the size of supertrees, we define an adaptive strategy to reconfigure the size of subtrees according to the size of the supertree. This adaptive scattering mechanism offers an efficient approach for short timelines to use lazy loading, which is automatically relaxed for longer ones, as illustrated by Algorithm 1.

Algorithm 1. Inserting t_i in the AMT for node n

```
procedure INSERT($n$,$t_i$)
    $tr_{sup} \leftarrow$ LOADTREE($n, t_0$)
    $tr_r \leftarrow$ CLOSESTTIME($tr_{sup}, t_i$)
    $tr_{sub} \leftarrow$ LOADTREE($n, t_r$)
    if SIZE($tr_{sup}$) $< step_1$ then
        $max \leftarrow step_1$
    else if SIZE($tr_{sup}$) $< step_2$ then
        $max \leftarrow step_2$
    else
        $max \leftarrow step_3$
    end if
    if SIZE($tr_{sub}$) $< max$ then
        INSERT($tr_{sub}, t_i$)
    else
        $\langle tr_{left}, tr_{right} \rangle \leftarrow$ SPLIT($tr_{sub}$)
        if $t_i >$ LOWEST($tr_{right}$) then
            INSERT($tr_{right}, t_i$)
        else
            INSERT($tr_{left}, t_i$)
        end if
    end if
end procedure
```

IV. Evaluation of GreyCat

In this section, we evaluate our reference implementation of the temporal graph model against two potential alternative implementations: *i)* a plain graph stored in a time series database and *ii)* a plain graph versioned with checkpoints. More specifically, we focus on read and write throughput, the elementary operations of data analytics.

A. Experimental Protocol

All the reported experiments were executed on a Linux server with a 12-core Intel Xeon E5-2430 processor with 128 GB of memory. Experiments have been executed 10 times and the reported numbers refer to mean values. Experiments are made available on GitHub[2] for the sake of reproducibility. To compare these approaches, we consider a synthetic graph of $100,000$ nodes. The generated graph corresponds to a k-ary tree where each node includes a unique identifier, an integer value, a character and, if necessary, a link to the parent node as well as links to the children nodes. In a second step, we update $15\,\%$ of the nodes of the graph repetitively, until reaching an history of $5,000$ changes. This ratio is extracted from a smart grid topology generator, which is based on a realistic smart grid dataset [15]. A node change consists in randomly setting a new value and a new character to the node. Our evaluation aims at testing the scalability of each solution for a growing temporal graph history. Therefore, we considered the following *Key Performance Indicators* (KPI): *read throughput* and *write throughput*. Throughput indicators are reported in *nodes read or written per second*.

[2]https://github.com/electricalwind/greycatBench

B. Alternative Implementations of Temporal Graphs

We compare GreyCat with 2 alternative implementations of temporal graphs: using time series and using checkpoints.

1) Storing Graphs as Time Series: To build this candidate solution we leveraged a plain graph whose values are versioned in InfluxDB [16] (version 1.1.2). InfluxDB is one of the newest and fastest time series databases that received much attention lately. This category of databases is heavily used for data mining and forecasting [17], [18]. While many time series databases provide interesting features, like SQL-like query languages, their data model is essentially flat—usually integer or double values—and does not support complex relationships between data. In a time series based temporal graph, each node of the graph maintains its own corresponding time series. Thus, to retrieve the version of a node at time t, one must first fetch its time series and then retrieve the value at that time. However, a major limitation of such an approach in the context of temporal graphs is that it is not possible to directly store relations. In our evaluation, we chose to represent the relationship of a node A to a node B and C as a field in node A, containing the identifiers of B and C.

2) Storing Graph with Checkpoints: Another possibility—and the most common one [7], [8]—is to take a snapshot of the graph at regular intervals. A snapshot is a complete copy of the full graph. Thus, conversely to time series, to retrieve a node at time t, one will first have to load the closest previous snapshot and then find the node in the graph of this snapshot. Despite being very simple, such solution comes at the price of *i)* redundancy, *ii)* possible missing values if a node were updated twice along one interval, and *iii)* the impossibility to create new nodes in the past. Techniques exist to mitigate the expensive cost of a full copy, such as chained immutable trees, where every update consists in a wrapper of the previous version plus a delta. Such

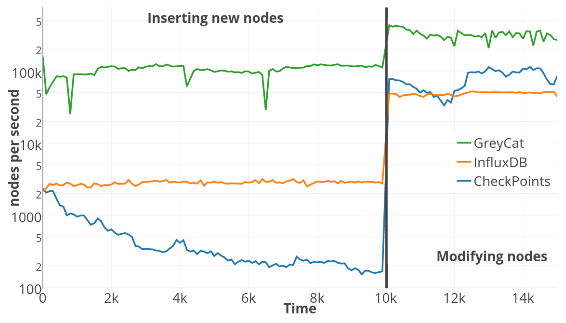


Fig. 3. Write throughput when creating and updating nodes

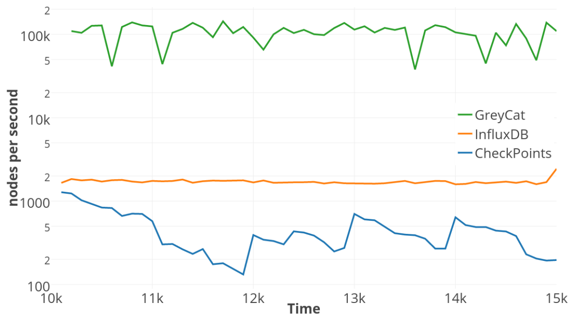


Fig. 4. Read throughput: *sum of children*

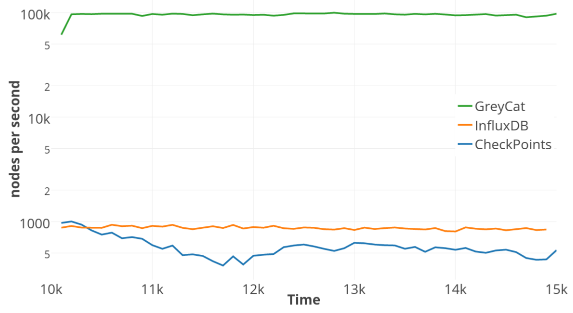


Fig. 5. Read throughput: *string building*

techniques are used to implement transactional storages. In our evaluation, we use the *checkpoints* mechanism offered by ROCKSDB [19], to store graph data with a minimum amount of redundancy, thanks to the use of hard links from a new version to the previous stored one.

C. Empirical Evaluation

To measure the efficiency of each solution, we consider 3 steps. First, we load the temporal graph and explore its history—the throughput is measured for every step of the graph evolution. Then, we compute the recursive sum of all children's integer values of a node recursively at a given time. This requires to read a large number of nodes (up to a tenth of the total number of nodes) and a large number of graph traversals. Finally, we simulate the construction of a synthetic state vector by building a string with the character value of the n^{th} child of a node recursively at a given time. This requires fewer reads (up to log_{10} of the total number of nodes), but emphasizes on the ability to handle large relationships. These benchmarks are evaluated against the 3 temporal graph approaches, which adopt different design choices. To improve the overall readability, graphs have been smoothen by gathering measure per group of 100 timestamps and then averaged.

1) Write Throughput: Fig. 3 demonstrates that GREYCAT outperforms the other solutions, with a write throughput close to $100,000$ nodes per second for step 1 and $400,000$ for step 2. The differences for the two steps can be explained by the fact that creating a node is costlier than updating one (for all solutions). Another interesting observation is that snapshotting performs worse than INFLUXDB for large-scale graphs for the insertion, but slightly better for the update step.

2) Read Throughput: Fig. 4 and 5 depict the read throughput when trying to access nodes in one of their previous states for the two use cases presented above. Note that the result starts at $10,000$ as we wait for step 2 to start to perform read measurements.

Similarly to the write throughput, GREYCAT performs significantly better than the two other solutions in both scenarios for reading, with around $100,000$ nodes per second. Time series are also stable over time in both cases, but as expected, the read throughput of traversing specific relationships, is twice slower than traversing all the nodes of the relationship. In the case of snapshots, performance is quite similar in both situations, but decreases over time.

V. RELATED WORK

The need to deal with temporal data has been discussed across several research communities. Early works in database communities [20], [21] delivered formal semantics for historical relational databases. Some of these temporal features are integrated into SQL:2011 or ISO/IEC 9075:2011 [22]. The necessity to reason about time-evolving data has also been discussed in the area of the semantic web, *e.g.*, MOTIK [23]. With the emergence of big data and the IoT, temporal aspects of data, in form of time series databases [16], [24], gained again visibility in research communities. However, the data model of time series databases is essentially flat and does not support complex relationships between data.

Temporal graph processing frameworks go a step further and consider time-evolving graphs. CHRONOS [7], and its extension IMMORTALGRAPH [8], are storage and execution engines for graph computations on temporal graphs. They define a temporal graph as a sequence of graph snapshots at specific points in time. To store temporal graph data on disk, they use so-called snapshot groups. A snapshot group is valid for a time interval and comprises a complete snapshot for the beginning of the interval and a number of deltas until the end of the interval. GRAPHTAU [6], *Historical Graph Store* (HGS) [25], G* [26], and KINEOGRAPH [27] are other temporal graph processing frameworks. They all represent time-evolving graphs as series of consistent graph snapshots. Furthermore, for none of these solutions the source code is available. Some of them optimize storage by using a combination of complete snapshots at specific timepoints and deltas in-between these [7]. Nonetheless, in some form or another, data models of existing approaches represent time-evolving graphs as sequences of full graph snapshots. This comes with severe limitations: First of all, full-graph snapshots are expensive in terms of memory requirements (both on disk and in-memory). Secondly, for every small change in the graph it would be necessary to snapshot the graph (and/or the delta) to keep track of the change history. Thirdly, the continuous semantics of time is lost by the discretisation in snapshots. Thus, navigating in the time and space dimensions of the graph is problematic, which complicates analytics algorithms.

VI. CONCLUSION AND DISCUSSION

In this paper, we presented a temporal graph data model and GREYCAT, its open source reference implementation. Most notably, our approach is able to model large-scale, time-evolving graphs without relying on snapshotting, like the current state-of-the-art does [7], [8]. Moreover, GREYCAT is one of the only open source frameworks for time-evolving graphs. We demonstrated that our temporal graphs pave the way for analyzing complex data in motion at scale. In particular, we illustrate that this data model is especially efficient when analyzing large-scale graphs with partial changes along time, which is typical for many real world analytics [5], [6].

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'10, 2010.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, Apr. 2012.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012.

[4] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proc. of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD'05, 2005.

[5] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, and Y. L. Traon, "Model-based time-distorted contexts for efficient temporal reasoning," in *Proc of the 26th International Conference on Software Engineering and Knowledge Engineering*, 2014.

[6] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proc. of the 4th International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES'16, 2016.

[7] 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 *Proc. of the 9th European Conference on Computer Systems*, ser. EuroSys'14, 2014.

[8] 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," *Trans. Storage*, Jul. 2015.

[9] J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *Proc. of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, ser. DMKD'03, 2003.

[10] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proc. of the Southern Association for Information Systems Conference*, vol. 2324, 2013.

[11] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516.

[12] Y.-M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 456–468, Apr 1997.

[13] R. Elmasri, Y.-J. Kim, and G. T. Wuu, "Efficient implementation techniques for the time index," in *Proc. 7th International Conference on Data Engineering*. IEEE, 1991.

[14] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Foundations of Computer Science, 1978., 19th Annual Symposium on*. IEEE, 1978.

[15] T. Hartmann, F. Fouquet, J. Klein, Y. L. Traon, A. Pelov, L. Toutain, and T. Ropitault, "Generating realistic Smart Grid communication topologies based on real-data," in *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*.

[16] "influxdb: Time-Series Data Storage," https://influxdata.com/time-series-platform/influxdb.

[17] E. Keogh, S. Lonardi, and B. Y.-c. Chiu, "Finding surprising patterns in a time series database in linear time and space," in *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 550–556.

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994.

[19] "Use Checkpoints for Efficient Snapshots," http://rocksdb.org/blog/2015/11/10/use-checkpoints-for-efficient-snapshots.html.

[20] J. Clifford and D. S. Warren, "Formal semantics for time in databases," *ACM Trans. Database Syst.*, vol. 8, no. 2, Jun. 1983.

[21] E. Rose and A. Segev, "Tooa: A temporal object-oriented algebra," in *Proc. of the 7th European Conference on Object-Oriented Programming*, ser. ECOOP'93, 1993.

[22] K. Kulkarni and J.-E. Michels, "Temporal features in sql:2011," *SIGMOD Rec.*, vol. 41, no. 3, Oct. 2012.

[23] B. Motik, "Representing and querying validity time in rdf and owl: A logic-based approach," *Web Semant.*, vol. 12-13, Apr. 2012.

[24] "OpenTSDB: The Scalable Time Series DB," http://opentsdb.net.

[25] U. Khurana and A. Deshpande, "Storing and analyzing historical graph data at scale," *CoRR*, vol. abs/1509.08960, 2015.

[26] A. G. Labouseur, J. Birnbaum, P. W. Olsen, Jr., S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The g* graph database: Efficiently managing large distributed dynamic graphs," *Distrib. Parallel Databases*, vol. 33, no. 4, Dec. 2015.

[27] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. of the 7th ACM European Conference on Computer Systems*, ser. EuroSys'12, 2012.