Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series

François Fouquet, Thomas Hartmann, Sébastien Mosser, Maxime Cordy

Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series

Francois Fouquet
SnT, University of Luxembourg
francois.fouquet@uni.lu

Thomas Hartmann
SnT, University of Luxembourg
thomas.hartmann@uni.lu

Sébastien Mosser
Université Côte d'Azur, I3S
mosser@i3s.unice.fr

Maxime Cordy
SnT, University of Luxembourg
University of Namur, Belgium
maxime.cordy@ext.uni.lu

ABSTRACT

Time series are commonly used to store temporal data, *e.g.*, sensor measurements. However, when it comes to complex analytics and learning tasks, these measurements have to be combined with structural context data. Temporal graphs, connecting multiple time-series, have proven to be very suitable to organize such data and ultimately empower analytic algorithms. Computationally intensive tasks often need to be distributed and parallelized among different workers. For tasks that cannot be split into independent parts, several workers have to concurrently read and update these shared temporal graphs. This leads to inconsistency risks, especially in the case of frequent updates. Distributed locks can mitigate these risks but come with a very high-performance cost. In this paper, we present a lock-free approach allowing to concurrently modify temporal graphs. Our approach is based on a composition operator able to do online reconciliation of concurrent modifications of temporal graphs. We evaluate the efficiency and scalability of our approach compared to lock-based approaches.

CCS CONCEPTS

• **Information systems** → **Stream management**; *Graph-based database models*;

KEYWORDS

Distributed computing, Time-Series, Distributed graphs, Temporal graphs, Data analytics, Lock-free workers

ACM Reference format:
Francois Fouquet, Thomas Hartmann, Sébastien Mosser, and Maxime Cordy. 2018. Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 8 pages.
https://doi.org/10.1145/3167132.3167255

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167255

1 INTRODUCTION

Technologies, such as the Internet of Things (IoT), drive our businesses to a more and more digital era. Considering the context of the IoT (or the so-called Industry 4.0), a large amount of data is collected as streams, *e.g.*, measured by sensors. These data streams have the potential to profoundly support operational decision-making, coupled to data analytics and learning algorithms. Temporal graphs have proven to be very suitable to organize and structure such data streams to empower complex analytic algorithms [9, 10, 13]. Nonetheless, analytic algorithms on large-scale graphs are usually computationally expensive. This calls for distributing and parallelizing the processing across several machines [16]. If the analysis task cannot be easily divided into independent parallel subtasks (as for example supposed by the MapReduce paradigm [6]), it is necessary that several workers read and update the graph concurrently. Infrastructure monitoring and control systems are a common example where analytics often cannot be easily split into independent parallel subtasks [11]. Inconsistency risks are a natural consequence, especially if data changes frequently. While this risk can theoretically be mitigated using distributed locks, this strongly affects performance, considering large data streams. An alternative is to use *version-aware* mechanisms to reconcile the graph, based on the order of modifications. In this context, it is important to distinguish between the *domain time* and the *logical time*. The domain time defines the temporal validity of the represented information. For instance, this time could define the moment a value has been measured by a sensor, producing a value associated usually with a timestamp. Whereas the logical time is defined by Fidge *et al.*, [8] as the partial ordering of events by their causal relationships. This is sometimes also referred to as a *version* [23]. Coming back to our temporal graph, this can be interpreted as the timestamp associated to the insertion of the data into the graph structure. It is important to notice that both times are independent: data can be inserted after it was measured, for example due to latency. It is crucial that the timeline—in terms of domain time—is respected in case of concurrent modifications. To sum up, temporal graphs have an inconsistency risk on the domain timeline when it comes to merging concurrent modifications.

Therefore, in this paper, we present a novel composition operator • able to reconcile concurrent modifications of temporal graph structures while preserving the domain timeline. Our approach is designed to work online on data streams produced by concurrent workers. This allows to distribute the processing of

temporal graphs in a lock-free manner, *i.e.*, to read and write on a shared temporal graph, over several machines in parallel, without the need for distributed locks. We detail the implementation of our approach, *e.g.*, how we handle several updates of the temporal graph per network call using inlining. We demonstrate, based on a theoretical study, the effectiveness of this operator to conserve consistency on temporal graphs while facing complex concurrent modifications. We implemented our approach into the open source graph processing framework GreyCat[1] and compare it in terms of efficiency and scalability with a classical implementation based on distributed locks.

The remainder of this paper is as follows. In Section 2 we motivate our work and present the challenges faced by our approach. Section 3 formalizes our proposed composition operator and Section 4 presents details of its implementation. We evaluate our approach in Section 5. Related work is discussed in Section 6 before the paper concludes in 7.

2 MOTIVATION & CHALLENGES

Graphs are suitable to represent complex data and relationships. A time-evolving graph connects multiple nodes where each node can be interpreted as a time series. Well-known examples of time-evolving graphs are social media networks and transportation routes [18]. While most of the literature approaches use some sort of snapshots or a combination of full snapshots and deltas [9, 13], some incorporate time as a first-class property into the graph representation [10], *i.e.*, making each node an independent time series.

It is no surprise that much work in the area of data analytics and machine learning focuses on large-scale graph representation and processing. In order to solve complex analytic tasks in a timely manner, the processing of large-scale graphs needs to be parallelized across several concurrent units of execution (processes or threads). Some well-known graph-processing frameworks like Pregel [18] and Giraph[2] use a BSP-inspired [28] model, others, like GraphLab [17], rely on a shared memory abstraction together with a locking strategy to parallelize the processing of complex analytic tasks. Although the latter is a more flexible programming model, which enables a broader range of analytic algorithms, as well as better support for distribution, locking often comes with severe performance bottlenecks. This is especially problematic considering rapidly changing temporal graphs, which we are focusing on in this paper.

More specifically, in this paper we address the following challenge: **how can several workers concurrently process a shared temporal graph, which rapidly changes over time, without using distributed locks, while keeping consistency**? Therefore, in the next section we will first formalize our proposed lock-free composition operator before we detail it's implementation and evaluation in Section 4 and 5 respectively.

3 DISTRIBUTED TEMPORAL GRAPHS

A temporal graph [10] is an efficient data structure for storing the history of many data that frequently change over time. It can be regarded as a pair (N, E) where N is a finite set of *nodes* and E is a finite set of edges that model relations between nodes. Each node represents the evolution of a data structure over time. In other words, it records the history of the attribute values of the data structure. New data structures (and thus nodes) can also appear at any time. Similarly, the edges represent relations between nodes that may or may not exist at a given time. Henceforth, we use a slight abuse of language and call "node" both a node and the data structure it represents. We also assume that a node n has an identifying name, written $name(n)$.

Any given node n is associated with attributes that are named and typed. Let A_n be the set of attributes of n. For an attribute $a \in A_n$, we denote its type (*i.e.*, the set of values it can take) by $\tau(a)$. We consider primitive types (*e.g.*, boolean, character, number), composite types and arrays. At any time t where n exists, each of its attributes may hold a value $v_{a,t}$ of its type, *i.e.*, $v_{a,t} \in \tau(a)$. Let T be a totally ordered time index. The values of the attributes of n at a time point $t \in T$ forms the *state* of n at this time. Given that n records all its states, it can be regarded as a function $n : A_n \to T \to \cup_{a_i \in A} \tau(a_i) : n(a,t) = v_{a,t}$, with $v_{a,t} \in \tau(a)$. This function can be partial as, *e.g.*, it is not defined for time points where the node does not exist.

The definition of an edge $e \in E$ is very similar, as the set of named relations R between nodes can be seen as time-evolving boolean attributes, that is, one can see an edge as a function $e : N \times N \times R \to T \to \mathbb{B}$ such that $e(n, n', r, t)$ holds if and only if the relation r from n to n' exists at time t. For that reason, in the rest of this paper we focus only on nodes and assume that temporal graphs have no edge, without loss of generality.

The time index being potentially infinite or even uncountable, it is infeasible to record the whole history of a given node n, *i.e.*, its state for all time points. Therefore, in practice we use a discrete representation named *timeline*, which consists of a finite subset of time-indexed states. We call an element of this set a *state chunk* (or *chunk* for short). A chunk thus defines a snapshot of the node at a given time point t; in other words, it is a function $c_{n,t} : A_n \to \cup_{a_i \in A_n} \tau(a_i) : c_{n,t}(a) = v_{a,t}$. From a timeline, the history of the node is approximated by considering that its state changes only at time points corresponding to a chunk [10]. Formally, let $\{c_{n,t_1} \dots c_{n,t_k}\}$ be the timeline of node n such that $t_i < t_{i+1}$ for all i. Then the complete history of n can be retrieved from the timeline as follows:

$$n(a,t) = \begin{cases} \bot, & t < t_1 \\ c_{n,t_{i-1}}(a), & t_{i-1} \le t < t_i \\ c_{n,t_k}(a), & t \ge t_k. \end{cases}$$

The evolution of continuous numeric values can also be better approximated by computing linear regressions [19] while still maintaining a finite representation. Nevertheless, these kinds of representations reduce the on-the-fly manipulation of temporal graphs to the computation of chunks and their recording in a storage backend. It also paves the way for distributing the computation and the storage amongst multiple workers. However, in this case each worker has only a partial view of the graph, which can be problematic, *e.g.*, if a given node history is scattered across multiple workers. This raises the question of how to consistently compose back a node history from multiple sources. If two workers make redundant computations (*e.g.*, if two sensors capture the same data

[1] https://github.com/datathings/greycat
[2] http://giraph.apache.org/

to improve accuracy), it is also essential to merge the results of these computations in a consistent way.

3.1 On the Composition of Temporal Graphs

Let us first study what it means to compose two temporal graphs regardless of their actual representation. Given the execution context of this operator (*i.e.*, merging data held by multiple workers), the composition operator should ideally satisfy the following three properties:

Idempotence. Composing a temporal graph with itself should not corrupt it in any way. It allows a worker to freely perform a synchronisation even if the origin of the temporal graph to synchronize with is unknown.

Commutativity. The commutativity of the composition operator allows the synchronization process between workers to be order-independent. This requires logical time information to be present in t_1 and t_2 to ensure commutative reconciliation.

Associativity. By being associative, the composition operator supports multiple workers working together and exchanging data in an asynchronous way.

We now give an intuition of how composition should work. We know that temporal graphs capture data values that change over time. By composing two temporal graphs, one should get a new temporal graph containing more information than the two individual graphs taken separately. By more information, we mean not only that every information stored in the separate graphs should also appear in their composition, but also that the composition should provide more accurate data about a node whose related information is scattered across the two graphs. This leads us to three requirements for our composition operator. First, a node contained in only one graph is also part of the composition. Second, if both graphs contain information about the same node[3] on different time points, their composition contains all these information. Third, if the graphs contain (possibly contradictory) information about the same node on the same time point, their composition contains a composed version of this information.

Remember that we consider temporal graphs as sets of nodes without edges. Formally, the composition of two temporal graphs N_1 and N_2 yields the smallest set $N_1 \bullet N_2$ satisfying, for all $n_1 \in N_1$ and $n_2 \in N_2$:

$$name(n_1) \notin N_2 \Rightarrow n_1 \in (N_1 \bullet N_2) \tag{1}$$

$$name(n_2) \notin N_1 \Rightarrow n_2 \in (N_1 \bullet N_2) \tag{2}$$

$$name(n_1) = name(n_2) \Rightarrow n_1 \oplus n_2 \in (N_1 \bullet N_2) \tag{3}$$

where $name(n) \in N$ is equivalent to $\exists n' \in N : name(n) = name(n')$, and $\oplus$ is the node composition operator. The composition of two identically-named nodes by $\oplus$ yields a new node with the same name and defined as follows for any attribute a:

$$n_1 \oplus n_2(a) = \begin{cases} n_1(a), & n_2(a) = \perp \\ n_2(a), & n_1(a) = \perp \\ \mu(n_1(a), n_2(a)), & \text{otherwise.} \end{cases} \tag{4}$$

That is, if values of a are contained in only one node then the composition keeps these values. Otherwise, the values known in the two nodes are merged together by the merging operator μ. The definition of μ is critical and case specific. Indeed, it follows from Equations 1 to 4 that the composition operator $\bullet$ is idempotent, commutative, and associative if and only if the merging operator μ also is, respectively.

Note that these definitions do not impede the merging operator to modify the value of a at a given time point t when only one temporal graph has a defined value for a at time t. In other words, it may *not* hold that

$$n_2(a,t) = \perp \Rightarrow n_1(a,t) = (\mu(n_1(a), n_2(a)))(t). \tag{5}$$

Attributes with discrete values will satisfy this property. However, it may not be the case, *e.g.*, when the history of continuous values is approximated by polynomial functions [19]. Indeed, the composition of two functions over successive time frames may yield a new function that defines different values for the concerned attributes. If Equation 5 holds, the merging operator μ can be written as:

$$\mu(n_1(a), n_2(a))(t) = \begin{cases} n_1(a,t), & n_2(a,t) = \perp \\ n_2(a,t), & n_1(a,t) = \perp \\ \varphi(n_1(a,t), n_2(a,t)), & \text{otherwise} \end{cases}$$

for a given attribute a and a time point t. Again, the case where two nodes contain colliding information is delegated to a new merging operator φ and the idempotence, commutativity and associativity of $\bullet$ and μ will be that of φ.

The properties of $\bullet$ thus strongly rely on how the attributes are merged. Concretely, this raises the question of choosing a strategy to merge data gathered by multiple workers at the same time for the same attributes. For numeric values, a common way consists in computing the average, *i.e.*, $\varphi(n_1(a,t), n_2(a,t)) = 0.5 * (n_1(a,t) + n_2(a,t))$. The average operator guarantees idempotence and commutativity, but is not associative. This implies that the result of synchronizing more than two workers is order-dependent. A practical solution to this issue is to generalize φ to any number of nodes, and to perform the synchronization for all workers at the same time. An alternative is the *max* operator, which is idempotent, commutative and associative. The case of boolean values is not problematic since conjunction and disjunction are idempotent, commutative and associative. However, moving from propositional to multi-valued logics (*e.g.*, to represent uncertainty and imperfect information) may break these properties. For complex types (*e.g.*, strings, composite structure, arrays), an even more careful attention must be given [21]. Nevertheless, if φ breaks these properties, our definition allows one to pinpoint which attributes and time points are problematic and negotiate a mediation strategy with the workers (*e.g.*, via a master server).

3.2 Chunk-Based Representation

We mentioned at the beginning of Section 3 that a timeline, *i.e.*, a set of chunks, is a practical discrete representation for uncountable numbers of values in temporal graphs. Having given the theoretical foundations for composing temporal graphs, we now define concrete composition operators over timelines that can lead to efficient implementations.

Given that timelines are sets, we naturally consider set union as the composition operator. Let $C_1 = \{c_{n,t_1} \dots c_{n,t_k}\}$ and $C_2 =$

[3] We use an abuse of language: by "same node", we actually mean nodes with the same name, although these can be different functions.

$\{c'_{n,t'_1} \dots c_{n,t'_j}\}$ be two timelines over node n. Consider first the case where they have no contradictory information, that is, $\nexists(n,t) : c_{n,t} \in C_1 \wedge c'_{n,t} \in C_2$. Then, the composition of C_1 and C_2 is a new timeline given by $C_1 \cup C_2$. Note that this is not equivalent to the union of the temporal graphs that C_1 and C_2 represent. Indeed, since the value of a in n at time t is given by the left-closest chunk in the time-ordered sequence, say $c_{n,t'}$, this value can change by inserting a new chunk c_{n,t^*} such that $t' < t^* \leq t$. This is actually a particular implementation of φ that selects the chunk whose time occurrence is left-closest to t. This implementation guarantees commutativity and associativity, since the selection of the chunk is independent of their insertion order.

Let us now consider the case of merging chunks related to the same node and time point. This can happen, *e.g.*, when workers redundantly capture the same data at the same time, or when predictive workers infer the same data using different algorithms. In the composed timeline, we need to merge these chunks into a single one in order to preserve the conciseness of the representation. Let c_1 and c_2 be the chunks to merge. On the one hand, if $c_1 = c_2$, then the merge operator returns an identical chunk. On the other hand, if the attributes associated to c_1 and c_2 differ, the merged chunk must satisfy the following requirements: *(i)* the attributes valued in c_1 that are not valued in c_2 are in c (and vice-versa); and *(ii)* the attributes valued in both chunks have a value in c that results from the application of the merge operator φ. Therefore, the properties of timeline composition in this case depend on the definition of φ, just as previously discussed for temporal graphs.

4 IMPLEMENTATION & PROTOCOL

In this section, we detail our implementation of a lock-free distributed temporal graph data structure, based on the formal definition of the merge operator $\bullet$, described in Section 3. For the rest of this paper, we refer to this lock-free distributed temporal graph data structure short as *TGraph*. First, in Section 4.1, we present the architecture of TGraph. Then, in Section 4.2, we detail how a TGraph can be distributed in a client/server environment. Finally, in Section 4.3 and 4.4 respectively, we explain how our implemented protocol can reorder concurrent updates and optimize the network usage.

4.1 Memory Architecture

The contribution of this paper targets temporal graphs that can be efficiently shared and used concurrently among different processes. Unlike other approaches that follow a fork-and-join paradigm, our workers can arbitrarily read and update any part of the temporal graph. Nonetheless, we ensure global consistency at the momentum changes are saved (*i.e.*, committed) using our proposed merge operator (*cf.*, Section 4.2 and 4.3 for more details).

In our implementation, every worker hosts an instance of the TGraph as depicted in Figure 1. This includes a graph API to create, update, delete and modify nodes of the graph and their associated attributes. Our graph API is currently available for Java and JavaScript. Temporal graphs are then mapped to an underlying memory structure, which decomposes the temporal graphs into a sequence of chunks (*e.g.*, as proposed in [10]), following the semantics defined in Section 3. Implementation details of this mapping

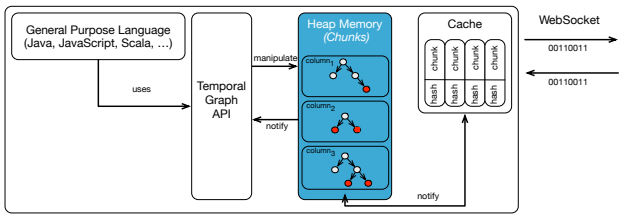

Figure 1: TGraph memory architecture

are out of the scope of this paper and omitted here due to space limitations.

The memory architecture of our TGraph implementation is organized as a heap structure that organizes fragments of a temporal graph. These fragments correspond to the *chunks*. The memory heap zone is divided into a number of columns, where each column stores all chunks belonging to one node. Each column thus hierarchically organizes the temporal evolution of a node. In other words, a column encodes the timeline of one node and each chunk stores a slice of a node. It is important to note that in our implementation chunks themselves are hierarchical, *i.e.*, a chunk can contain another chunk and so on. This inlining strategy is discussed in more detail in Section 4.4. This means that, for instance, a chunk can hierarchically contain all attribute values of a node from a time t_n to a time t_m. Similar to the organization of red black trees [19], our columns are balanced to ensure the following property: if a chunk c_{n,t_p} is the parent of chunk c_{n,t_i}, then $t_p < t_i$.

Memory chunks are designed to be loaded and saved independently. We use chunks also as our "unit of transportation" for network communication. In our current implementation we use a WebSocket channel for this. It is important to note that the previously defined merge operator (*cf.*, Section 3) is also used to merge a remote chunk with its local version in the local heap. Using such merge, a central server can act as a referee for chunk versions. In addition, we use a classic cache layer to enhance performance. This cache hosts a fixed number of binary representations of chunks in order to reduce unnecessary network communication.

In this section, we discussed how the memory architecture of a TGraph looks on a single machine/process. In the next section, we describe how our TGraph implementation can be distributed over several machines/processes and how the interaction between these works.

4.2 Distribution

Our current TGraph implementation supports distribution in form of a client/server architecture, as depicted in Figure 2. Every client and server maintains its own TGraph instance, where each instance corresponds to the architecture shown in Figure 1. Our implementation supports several servers, which are synchronized using the RAFT [22] consensus algorithm. It is important to note that only the synchronization of the servers is done with a consensus algorithm, merging concurrent modifications from several clients on a server is, as discussed in Section 3, lock-free. The server is also responsible for persisting the chunks of a TGraph to disk. In our current

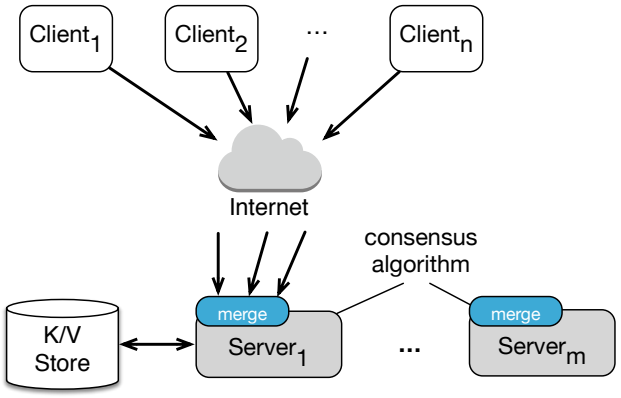

Figure 2: Distribution of a TGraph in a client/server architecture

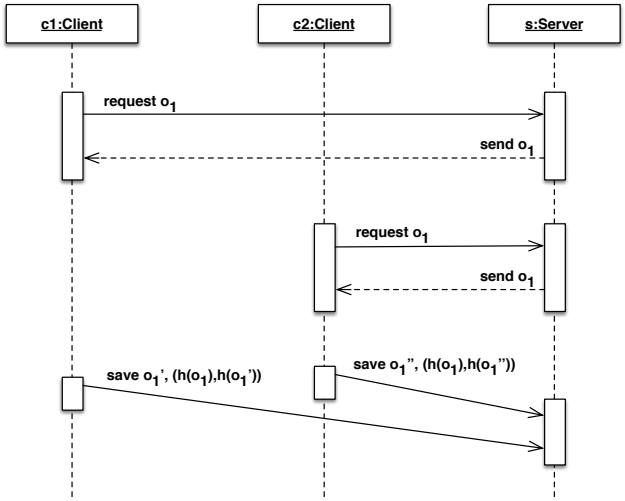

Figure 3: Reconciling the order of concurrent modifications in TGraph using double hashes

implementation, we use RocksDB[4] for persisting our TGraph to disk. We save/load each chunk of the TGraph as a key/value pair.

4.3 Reconciling Concurrent Modifications

To be able to reconcile the order of concurrent modifications, we use a strategy based on *double hashes* of the content of the changed chunk. Basically, we hash the content of a chunk before it gets changed and after. Then, we send both hashes together with the changed chunk from a client to the server. This allows the server to reconcile the order of concurrent modifications and to detect potential conflicts by comparing the origin hashes (before the change) with the hash of the chunk on the server. Then, the chunks are merged according to the semantics defined in Section 3. Figure 3 exemplifies how this works with two clients and one server. First, a client c_1 requests, *i.e.*, loads, a chunk o_1 from the server s. While c_1 uses this chunk, another client, c_2 loads the same chunk o_1 from the server s. Now, c_1 and c_2 are both modifying chunk o_1 concurrently and sending their changes to s. If o_1' denotes the chunk changed by c_1, o_1'' the chunk changed by c_2, and $h(x)$ the hash function, then c_1 sends together with the changed chunk o_1' the hashes $(h(o_1), h(o_1'))$ and c_2 sends together with the chunk o_1'' the hashes $(h(o_1), h(o_1''))$. Regardless of o_1' or o_1'' reaches the server s first, by comparing the origin hashes of the changes with the hash of the chunk on the server s, the merge can be executed, respecting the priority rules defined in Section 3.

While this double hash strategy works well in client/server architectures, it cannot be directly applied to other architectures, like peer-to-peer. This is because we only keep two hashes, which is sufficient to do the merge of n concurrent modifications on one server (or potentially m servers synchronized with a consensus algorithm) but it is not sufficient to reconcile the order of concurrent modifications along several peers. For peer-to-peer architectures, it would be necessary to keep x hashes, *i.e.*, a path of all changes x.

[4]http://rocksdb.org/

4.4 Inlining

As discussed in Section 4.1, we use chunks as the main unit for sending data over the network. However, often the size of a (low level) chunk is not optimal in terms of network throughput. Indeed, we found that chunks tend to be too small in many cases (although they occasionally can also be too big). We found by experience that a "good" size for a chunk to be transferred over the network is in the order of magnitude of around 100 KBs per chunk. Therefore, as briefly mentioned in Section 4.1, we implement a hierarchical inlining strategy to embed—or inline—a chunk into another one. Given a hierarchy of chunks, we speak of *uplifting* when a chunk from one level is inlined into a chunk of the level directly above it, and vice versa of *splitting*, when a chunk from a higher level is divided into two consecutive chunk levels. Uplifting and splitting make the implementation of our proposed merge operator slightly more complicated. In fact, the server needs to potentially rebuild and reorganize the hierarchy in order to merge concurrently modified chunks.

5 EVALUATION

In this section, we experimentally evaluate the efficiency of our proposed merge operator. As consistency is guaranteed by definition (*cf.*, Section 3), we put the emphasis of this evaluation on the performance. We benchmark updates using our proposed merge operator against implementations relying on distributed locks or consensus algorithms. The following Section 5.1 first details the experimental setup. Then, Section 5.2 presents and discusses the experimental results. Finally, in Section 5.3 we discuss threats to validity.

5.1 Experimental setup

We integrated our merge operator into the network synchronization layer of the temporal graph framework: GreyCat. In addition, we extended this synchronization layer with two additional protocols

that implement a per-node lock mechanism to ensure consistency. In a first alternative implementation, we rely on the Atomix framework[5] to leverage a state-of-art consensus protocol: Raft. Using Raft and the distributed lock of Atomix, we enforce temporal consistency by acquiring a lock before any temporal modification of a node. Thus, other graph peers are blocked until we release the lock on this particular node. Raft ensures a strong consensus, allowing to synchronize chunks without relying on additional mechanisms. In a second alternative implementation, we implement a centralized lock mechanism using Java's atomic primitives on a central server. This second alternative implementation is built for client/server architectures, as described in Section 4.2. While being slightly weaker in terms of consistency compared to consensus algorithms, this implementation reuses the WebSocket protocol of GreyCat. This ensures a fair comparison with our proposed merge operator, which is integrated into the same low-level synchronization protocol.

We evaluate the three alternative implementations in terms of updates of a temporal graph. These updates simply change an attributed field of different nodes. We performed 1.000.000 of these updates, synchronized (to the server) every 1.000 modifications. The Raft and centralized lock-based implementations require a network call at every node update, while our merge operator only communicates for the synchronization points over the network, *i.e.*, every 1.000 elements. Updates are done on different physical computers. Client and server are deployed on a Java Virtual Machine 8 on two identical MacBook i7 computers, using SSDs and a Gigabit network. Persistence, *i.e.*, file system accesses and disk storage, is managed by RocksDB.[6] Finally, we ran the Raft experiment twice, once with 2 and once with 7 peers to evaluate the scalability of this solution. The evaluation of the centralized lock and of our proposed merge operator are only executed in a client/server mode, because they are by construction not impacted by the number of peers.

5.2 Performance Results

Every experiment is exectued five times. We first measured the elapsed time for every synchronization batch, *i.e.*, every 1,000 updates. Then, in a second step, we calculated the throughput per second from these values – for every step between 0 and 1,000,000 updates. The results (in operations per second) are depicted in Figure 4, using a logarithmic scale due to the big difference. We obtained the following average values: (1) lock-free merge operator=23,635 op/s, (2) central server lock=687 op/s, (3) Raft with 2 peers=512 op/s, and (4) Raft with 7 peers=43 op/s). Our lock-free merge operator offers a significantly bigger throughput: above 20,000 op/s in average and never less than 4,000 op/s. The central server lock solution is slower by a significant factor but still performs much better than a Raft-based implementation. We can also notice that Raft is, without surprise, significantly impacted by the number of peers involved in the update of the temporal graph. As a result, performance drops to less than 50 op/s with the Raft consensus and 7 peers. Using a Box-and-Whisker chart, we can also see that the performance is less stable for the lock-free merge operator as the minimum and maximum varies—up to a factor of 10. This can be explained by the fact that chunks need to be in memory while being updated and merged. Since we use an LRU-cache implementation, if a chunk is already in memory, this accelerates the merge operation significantly. On the other hand, chunks have to be loaded from disk. These performance variations can thus be explained by the statistical effect of the cache.

5.3 Threats to validity

The major threat to validity of this validation is the quality of the implementation of the Raft protocol and its inclusion into the GreyCat framework. Indeed, due to the numerous network calls that have to be performed to obtain a consensus, any issue in the protocol would have a major impact on the results. To mitigate such risk, we implemented the central server lock strategy, which uses the exact same WebSocket protocol than our lock-free merge operator implementation, but with mechanism closer to Raft. Using these three implementations, we highly mitigate the risk of a bias. Finally, garbage collector times and disk speed are also threats to validity. To mitigate these, we initially executed the experiments and monitored these effects to ensure that none of these factors impacts more than 5% of final results.

6 RELATED WORK

A lot of work has been done in the area of structured data merging for XML files. Different approaches for three-way merging of XML documents together with sets of merge rules are proposed in [15, 26]. While most of these techniques are built for human-authored documents that are naturally structured as ordered trees, Abdessalem *et al.*, [1] proposes semi-structured document integration as probabilistic trees. Al-Ekram *et al.*, [3] propose with *diffX* an alternative to three-way merging - diffX is an algorithm to detect changes in multi-version XML documents. These approaches have in common that they are designed to find changes in only 2 XML documents, where each document is usually represented as one big tree data structure, which negatively impacts the performance. This is quite different from our use cases, where potentially a large number of workers concurrently modify a shared data structure. Moreover, in the case of comparing XML documents, while performance is still important, it is not a dominating and critical factor, like it is the case in on our scenarios.

The work of Blanc *et al.*, [4, 5] goes into a similar direction. They discuss interoperability of different modelling services and propose an architecture as well as a prototype to connect these different services together. In this context, when it comes to merging information from the different services, they face similar problems and propose similar solutions as for collaboratively working on XML documents.

Distributed transactions, as for example discussed in [2, 20, 27], are another way to handle concurrent modifications of shared data. Strict distributed transactions provide ACID properties and therefore rely on locking mechanisms, which are problematic in the context of frequent small changes, as in our case. Similarly, compensating transactions [14] and lock-free distributed transactions [7] rely on some sort of snapshot isolation, which again is problematic when it comes to frequent small changes, which are performed concurrently.

[5] http://atomix.io/atomix/
[6] http://rocksdb.org/

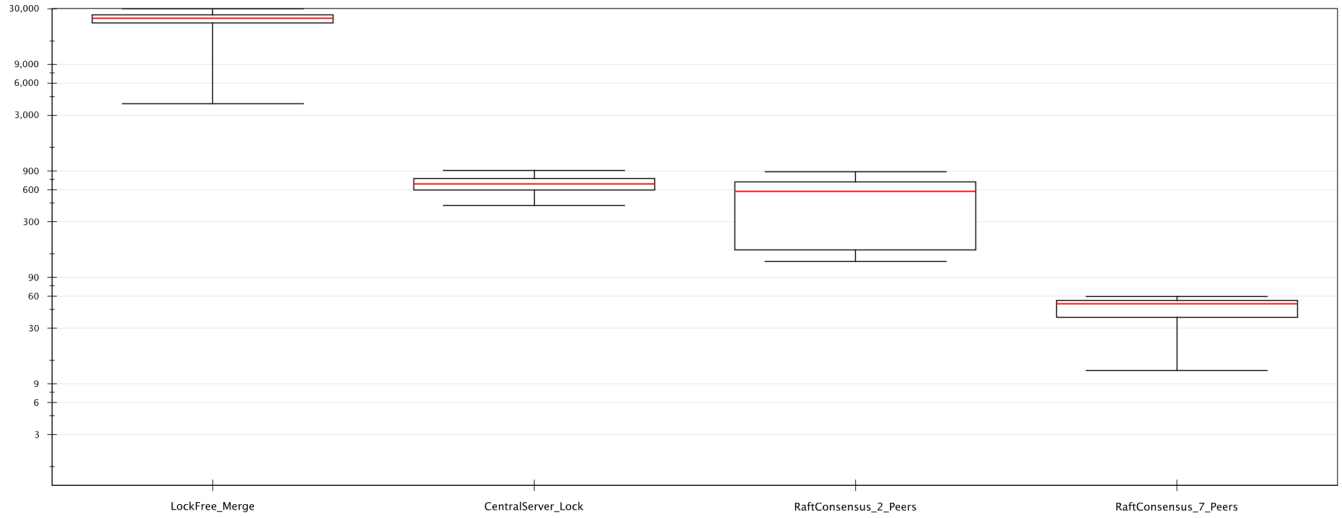

Figure 4: Box-and-Whisker chart representing the obtained update throughput using our proposed lock-free merge, a central server lock, and a Raft consensus with 2 and 7 peers. Results are given in operations per second, from 0 to 30,000 in a logarithmic scale.

An interesting concept is so-called conflict-free replicated data types (CRDT) [24, 25]. The main idea of CRDTs is that updates do not require synchronisation and that CRDT replicas provably converge to a correct common state. Most work in this direction focuses on collaborative document editing. An exception is *ChainVoxel*, a CRDT for collaborative editing of 3D Models proposed by Imae and Hayashibara [12]. While CRDTs are very interesting when it comes to collaborative and concurrent modifications of a shared data type, in many real-world applications, designing data types which provably converge to a correct common state, is practically impossible.

7 CONCLUSIONS & PERSPECTIVES

Computationally expensive tasks often need to be distributed among several workers. For problems that cannot be split into independent parts, these workers have to concurrently read and update a shared data structure. Complex data analytics and machine learning algorithms often rely on temporal graph data structures for this purpose. However, concurrently modifying a shared temporal graph entails severe consistency risks, especially when these changes happen at a fast pace. Distributed locks or consensus algorithms mitigate such consistency risks but often at an important performance penalty. Therefore, in this paper, we proposed a lock-free approach, relying on a merge operator to concurrently modify a shared temporal graph. We formalized this operator • and detailed its implementation into the open source graph processing framework GreyCat. Based on this implementation, we showed that this approach outperforms implementations relying on distributed locks or consensus algorithms. Currently, our approach of logically reordering concurrent updates targets *client/server* architectures. In future work, we plan to extend the proposed double hashing protocol to a vector-based one in order to enable peer-to-peer architectures without central servers. Also, this merge operator will be extended to manage specific node attributes such as tensors, which are used as a multidimensional storage for machine learning algorithms. Furthermore, despite we focused in our work on temporal graphs, we believe that this approach can be generalized for transactional systems.

8 ACKNOWLEDGMENTS

This work is partially funded by the CNRS (INS2I PEPS JCJC program) through the *M4S* project, by the European Commission (FEDER IDEES/CO-INNOVATION), and Creos Luxembourg S.A through the UL/SnT partnership programme.

REFERENCES

[1] T. Abdessalem, M. L. Ba, and P. Senellart. A probabilistic xml merging tool. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 538–541, New York, NY, USA, 2011. ACM.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, Cambridge, MA, USA, 1999. AAI0800775.

[3] R. Al-Ekram, A. Adma, and O. Baysal. diffx: An algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 1–11. IBM Press, 2005.

[4] X. Blanc, M.-P. Gervais, and P. Sriplakich. *Model Bus: Towards the Interoperability of Modelling Tools*, pages 17–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[5] X. Blanc, P. Sriplakich, and M. Gervais. Modeling services and web services: Application of modelbus. In *Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2005, Las Vegas, Nevada, USA, June 27-29, 2005, Volume 2*, pages 557–563, 2005.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*, pages 676–687, March 2014.

[8] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[9] 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*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[10] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. L. Traon. Analyzing complex data in motion at scale with temporal graphs. In *The 29th International Conference on Software Engineering and Knowledge Engineering, July 5-7, 2017.*, pages 596–601, 2017.

[11] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, Y. Le Traon, and J.-M. Jezequel. Model-Driven Analytics: Connecting Data, Domain Knowledge, and Learning. *ArXiv e-prints*, Apr. 2017.

[12] K. Imae and N. Hayashibara. Chainvoxel: A data structure for scalable distributed collaborative editing for 3d models. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 344–351, Aug 2016.

[13] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 5:1–5:6, New York, NY, USA, 2016. ACM.

[14] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 88–97, New York, NY, USA, 1991. ACM.

[15] T. Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[16] 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.*, 5(8):716–727, Apr. 2012.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *ArXiv e-prints*, June 2010.

[18] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[19] A. Moawad, T. Hartmann, F. Fouquet, G. Nain, J. Klein, and Y. L. Traon. Beyond discrete modeling: A continuous and efficient model for iot. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 90–99, 2015.

[20] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 76–88, New York, NY, USA, 1983. ACM.

[21] W. Ng. Repairing Inconsistent Merged XML Data. In V. Marík, W. Retschitzegger, and O. Stepánková, editors, *Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings*, volume 2736 of *Lecture Notes in Computer Science*, pages 244–255. Springer, 2003.

[22] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[23] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[24] M. Shapiro and N. Preguiça. Designing a commutative replicated data type. *ArXiv e-prints*, Oct. 2007.

[25] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] C. Thao and E. V. Munson. Using versioned tree data structure, change detection and node identity for three-way xml merging. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, pages 77–86, New York, NY, USA, 2010. ACM.

[27] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[28] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.