

# GreyCat: Efficient What-If Analytics for Data in Motion at Scale

Thomas Hartmann, François Fouquet, Assaad Moawad, Romain Rouvoy, Yves Le Traon

▶ **To cite this version:**

Thomas Hartmann, François Fouquet, Assaad Moawad, Romain Rouvoy, Yves Le Traon. GreyCat: Efficient What-If Analytics for Data in Motion at Scale. Information Systems, In press, 83, pp.101-117. 10.1016/j.is.2019.03.004 . hal-02059882

**HAL Id: hal-02059882**
**https://inria.hal.science/hal-02059882**

Submitted on 26 May 2019

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

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

# GreyCat: Efficient What-If Analytics for Data in Motion at Scale

Thomas Hartmann[1,1], Francois Fouquet[1], Assaad Moawad[1], Romain Rouvoy[1,1], Yves Le Traon[1]

[a] *University of Luxembourg, Luxembourg*
[b] *DataThings, Luxembourg*
[c] *University of Lille / Inria, France*
[d] *Institut Universitaire de France (IUF)*

**Abstract**

Over the last few years, data analytics shifted from a descriptive era, confined to the explanation of past events, to the emergence of predictive techniques. Nonetheless, existing predictive techniques still fail to effectively explore alternative futures, which continuously diverge from current situations when exploring the effects of *what-if* decisions. Enabling prescriptive analytics therefore calls for the design of scalable systems that can cope with the complexity and the diversity of underlying data models. In this article, we address this challenge by combining graphs and time series within a scalable storage system that can organize a massive amount of unstructured and continuously changing data into multi-dimensional data models, called *Many-Worlds Graphs*. We demonstrate that our open source implementation, GreyCat, can efficiently fork and update thousands of parallel worlds composed of millions of timestamped nodes, such as *what-if* exploration.

*Keywords:* what-if analysis, time-evolving graphs, predictive analytics, graph processing

## 1. Introduction

The data deluge raised by large-scale distributed systems has called for scalable analytics platforms in order to guide decisions in critical cyber-physical infrastructures, such as smart grids [**?** ]. In this domain, *predictive analytics* techniques, like sliding window analytics [**?** ], typically extract temporal models from current and past historical facts in order to make predictions about the future [**?** ]. However, taking appropriate decisions rather requires *prescriptive analytics* in order to explore the impact of current and future actions on the underlying system, better known as *what-if analysis* [**?** ]. More specifically, what-if analysis is a powerful primitive to plan an optimal sequence of actions that leads to a desired target state of the underlying system. Reaching this optimization objective implies to cover all potential decision timepoints and applicable actions, thus inevitably yielding to a combinatorial explosion of alternative scenarios. In addition to that, this decision process has to deal with the continuous updates of states as time keeps flowing along.

Graphs are increasingly being used to structure and analyze such complex data [**?** **?** **?** ]. However, most of graph representations only model 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 [**?** ]—*e.g.,* in the domains of social networks, smart mobility, or smart grids [**?** ]. Yet, state-of-the-art approaches fail to deliver a scalable solution to effectively support time in graphs. In particular, existing

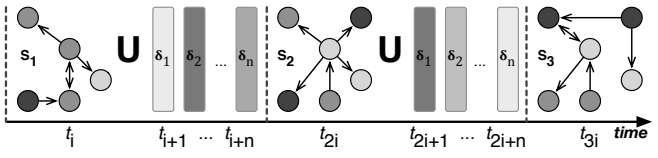



Figure 1: Snapshots ($S_i$) and deltas ($\delta_n$) of a time-evolving graph

approaches represent time-evolving graphs as sequences of full-graph snapshots [**?** ], or they use a combination of snapshots and deltas [**?** ], which requires to reconstruct a graph for a given time, as depicted in Figure **??**.

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 [**?** **?** **?** ]. 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) [**?** ]. An alternative to snapshotting consists in combining graphs with time series databases [**?** ], 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.

These challenges are even exacerbated when it comes to what-if analysis on top of such time-evolving graphs.

Therefore, in this article, we propose to adopt the theory of *many-worlds interpretation* [**?** ], where every single action can be interpreted as a divergence point, forking an alternative, independent world. In particular, we introduce the concept of *Many-Worlds Graphs* (MWG) as a versatile and scalable analytics data model supporting the evaluation of hundreds or even thousands of alternative actions on temporal graphs in parallel. MWG extends state-of-the-art graph analytics [**? ? ?** ], which are commonly used to organize massive amounts of unstructured data [**? ?** ]. Beyond the inclusion of temporal aspects within graphs [**? ? ? ?** ], MWG proposes an efficient exploration of many independently evolving worlds to support the requirements of what-if analysis. The main contribution of this article is a novel graph data model to support large-scale what-if analysis on time-evolving graphs. Related topics like graph processing, traversing, fault tolerance, and distribution are described where necessary.

We demonstrate that GreyCat, our implementation of MWG, can efficiently explore hundreds of thousands of independent worlds in parallel and we assess this capability on a real-world smart grid's workload. GreyCat is open source and available at GitHub.[1] MWG and GreyCat refer to Everett's [**?** ] many-world interpretation, illustrated by Schrödinger's cat [**?** ].

The remainder of this article is organized as follows. Section **??** introduces a real smart grid case study, as a motivation of this research. Sections **??** and **??** introduce the main concepts of MWG and their implementation. We thoroughly evaluate GreyCat in Section **??** and discuss current limitations and future work in Section **??**. The related work is discussed in Section **??** before concluding in Section **??**.

## 2. Motivating Case Study

In this section, we first introduce our motivating case study: intelligent load management in today's electricity grids. This case study is taken from our work with Creos Luxembourg,[2] the main electricity grid operator in the country. Based on this case study, we analyse the conceptual and technical requirements imposed by this use case. Then, we discuss scalability considerations, necessary operations, access patterns and argue that these are common requirements for many other case studies. In Section **??** and Section **??**, we describe how we intend to tackle these requirements with our approach and implementation, respectively. In Section **??**, we evaluate our approach and assess how valuable and suitable our design decisions are.

[1] https://github.com/datathings/greycat
[2] http://www.creos-net.lu

### *2.1. The Smart Grid Case*

Intelligent load management is a critical challenge for electricity utility companies [**? ?** ]. These companies are expected to avoid overload situations in electricity *cables* by balancing the load. The electric load in cables depends on the current and historical consumption of customers connected to a given cable within the system topology—*i.e.*, how cables are connected to each other and to power substations.

The underlying topology can be changed by opening/-closing so-called *fuses* at substations. This results in connecting/disconnecting households to different power substations, therefore impacting the electricity flow within the grid. As all of this (consumptions, decisions) changes over time, the idea behind prescriptive analytics is to continuously simulate the expected load for different topologies (what-if scenarios) with the goal to find an "optimal" one—*i.e.*, where the load in all cables is the best balanced. Smart grids are very-large-scale systems, connecting hundreds of thousands or even hundreds of millions of nodes (customers). Furthermore, most data in the context of smart grids is temporal—*i.e.*, it keeps changing over time—from consumption reports to the topology structure. This makes the simulation of different what-if scenarios very challenging and, in addition, it requires to take the temporal dimension—*i.e.*, data history—into account. Moreover, many different topologies are possible, which can easily lead to thousands of different scenarios.

To anticipate potential overload situations, alternative topologies need to be explored *a priori*—*i.e.*, before the problem actually occurs. The estimation of the electric load depends, aside from the topology, on the consumption data of customers [**?** ]. In the context of a smart grid, this data is measured by *smart meters*, which are installed at customers' houses, and regularly report to utility companies (*e.g.*, every 15 minutes [**?** ]). One can compute the electric load based on the profiles of customers' consumption behavior. These profiles are built using online machine learning algorithms, such as the ones introduced in [**?** ].

The interested reader can find more details related to this case study in [**? ? ? ? ?** ].

### *2.2. Conceptual and Technical Requirements*

However, the huge amount of consumption data quickly leads to millions of values per customer, and efficiently analyzing such large historical datasets is challenging. The temporal dimension of data often results in inefficient data querying and iteration operations to find the requested data. While this issue has been extensively discussed by the database community in the 80s and 90s [**? ?** ], this topic is gaining popularity again with the advent of *time series* databases for the IoT, like InfluxDB [**?** ]. Time series can be seen as a special kind of temporal data, which is defined as a sequence of timestamped data points, and is used to store data like ocean tides, stock values, and

weather data. It is important to note that in time series, data is "flat"—*i.e.*, time series only contain primitive tuples—like raw measurements. However, they are not able to capture complex data structures and their relationships like, for example, the evolution of a smart grid topology. Therefore, time series analysis is not sufficient to explore complex what-if analysis and prescriptive analytics. Moreover, data in real-world systems—like smart grids—evolve at very different paces: consumption data of customers changes very frequently, while the physical grid topology only changes at a much lower frequency. Correlating data coming from different time series is a complex task [**?** **?** ]. On the other side, graph-based storage solutions (*e.g.*, Neo4J [**?** ]), as well as graph processing frameworks, despite some attempts to represent time dependent graphs [**?** **?** **?** **?** ], are insufficiently addressing continuously changing data in their model: either failing to navigate through alternative versions of a given graph, or inefficiently covering this issue by generating distinct snapshots of the graph. When we speak in this article about temporal data, we refer to this fully temporal aspect and not about flat time series. Most importantly, none of these solutions supports the large-scale exploration of different alternatives. In a smart grid, many data points are measured by sensors at high frequency (data streams) and need to be processed and analyzed quickly to derive fast reactions. This is sometimes referred to *near real-time* analysis [**?** **?** ]. Classical batch analyses techniques are often too slow to be applied for such use cases.

The following conceptual and technical requirements are imposed by this use case:

- Not only flat data values, but also *complex relationships* between data need to be represented;
- Dynamic data, which *changes frequently*, needs to be efficiently represented, inserted, and read;
- Data must be able to *change at different paces*, but for each timepoint—regardless of its change frequency—it must be possible to read and correlate data;
- It must be possible to efficiently represent and explore *hundreds or thousands of hypothetical alternatives*;
- It is *not possible to keep all data in memory*, thus requiring an efficient storage concept;
- Insert and read operations must be *fast enough for near real-time analysis*.

### *2.3. Scalability Considerations and Access Patterns*

In terms of **scale**, for this case study, we have approximately 500,000 smart meters, a one year history of consumption data for each smart meter in 15 minutes intervals, 60,000 cables, and 80,000 fuses. This sums up to around 640,000 nodes in the graph and 17,520,000,000 consumption values. It is important to note that this is the scale for Luxembourg, a rather small country. Accordingly, if we think about the electricity grid of bigger countries like Germany, France, or the United States, these numbers would be even larger. For more information about the scale of a smart gird and the elements containing a smart grid, see for example [**?** ].

As for necessary **operations** and **access patterns**: Intelligent load management makes it necessary to anticipate—*i.e.,* predict the electric load in cables and specific regions. This prediction is, amongst others, based on historical data. In a first step, it is necessary to *find* all smart meters *connected* to a cable (around 20 to 30 meters). After this, the consumption *time series* of the meters have to be *navigated* and the required data *collected*. For example, if the cable load has to be predicted for a winter sunday in January, consumption data for similar days have to be collected. This information can be used to estimate the anticipated load for the cables. Depending on the anticipated load, it might become necessary to reconfigure the grid—*i.e.,* change the state of fuses in order to better balance the load. The electricity grid is divided into logical sections, so that the number of different combinations of fuse states is reduced. Nonetheless, hundreds or thousands of different configurations need to be explored, making it necessary to retrieve the data of smart meters and fuse states in different worlds.

Today's electricity grid is built to withstand short-term overloads. Therefore, reconfiguration **times of minutes** are in most cases acceptable. The following general requirements are imposed by this case study:

- A scale of *millions of connected nodes* is realistic. Therefore, inserting and reading nodes must be efficient;
- A scale of *thousands of parallel worlds* is realistic. Thus, forking worlds and retrieving data from different worlds must be efficient;
- It must be possible to *find a subset of nodes*;
- It must be easy to *navigate from a node to connected nodes*;
- Every node can have an independent time series, *millions of timestamped values per node* is realistic;
- It must be efficient to *insert and read time series data—i.e.,* to navigate in time;
- It must be possible to *explore thousands of parallel worlds—i.e.,* finding data in different worlds must be efficient.

In the context of this article, we therefore consider the following access patterns:

1. inserting nodes,
2. reading a node (defined by an id or attribute),
3. traversing the edges from a node to connected nodes,
4. retrieving the values of a node for a certain timestamp,
5. retrieving the values of a node for a certain world.

### *2.4. Contribution*

The above considerations motivate our work on MWG in order to enable large-scale what-if analysis for prescriptive analytics [**?** ]. Haas *et al.* [**?** ] also supports the need for large-scale what-if analysis in many other domains, *e.g.*

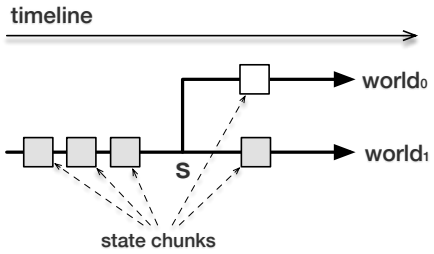



Figure 2: States of a node in two worlds

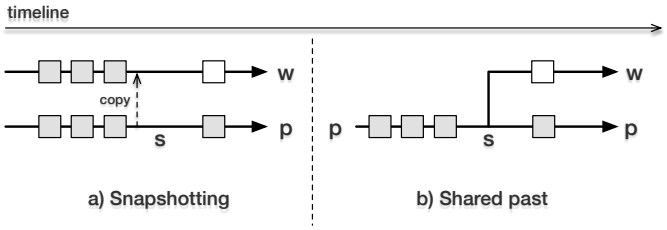



Figure 3: Types of many-worlds.

weather prediction and biology. Others include social networks [**?** ], smart cities and buildings, Industry 4.0, IoT, and the banking sector. Beyond the specificities of smart grids, we introduce the concept of *Many-World Graphs* (MWG) as a scalable, generic model to explore alternative scenarios in the context of what-if analysis. This article does not aim at solving a specific problem in the smart grid domain, but rather uses this case study to exemplify the requirements imposed by efficient what-if analytics for data in motion at scale. In this article, we aim at demonstrating that our MWG concept and our open source implementation, GreyCat, can efficiently fork and update thousands of parallel worlds composed of millions of timestamped nodes.

## 3. Introducing Many-World Graphs

### *3.1. Key Concepts*

This article introduces the notion of *Many-World Graphs* (MWG), which are directed, attributed hypergraphs with structures and properties that can evolve along time and parallel worlds. In particular, MWG build on the following core concepts:

**Timepoint** is an event, encoded as a timestamp;

**World** is a parallel universe, used as an identifier;

**Node** reflects a domain-specific concept, which exists across worlds, and is used as an identifier;

**State** is a node's value for a given world and timepoint, including attributes and relationships;

**Timeline** is a sequence of states for a given node and a given world.

Depending on the timepoint ($t$) and world ($w$), different states can be fetched from a given node ($n$). This is illustrated in Figure **??**.

Therefore, states are organized into chunks ($c$), which are uniquely mapped from any viewpoint: $\langle n, t, w \rangle$ : $read(n, t, w) \mapsto c_t$.

We associate each state chunk with a timepoint ($c_t$) and we define a timeline ($t_{n,w} = [c_0, \ldots, c_n]$) as an ordered sequence of chunks belonging to a given node ($n$) from a given world ($w$). Alternative state chunks in different worlds therefore form alternative timelines. Hence, a resolution function *read* returns a chunk ($c_t$) for an input viewpoint as the "closest" state chunk in the timeline.

Therefore, when a MWG is explored, state chunks of every node have to be resolved according to an input world and timepoint. The storage and processing of MWG made of billions of nodes cannot be done in memory, thus requiring to efficiently store and retrieve chunks from a persistent data store. For this purpose, we decompose state chunks into keys and values to leverage existing key/value stores to persist the data on disk. The mapping of nodes to state chunks (including attributes and references to other nodes) and their persistent storage is detailed in Section **??**.

While prescriptive analytics tends to explore new worlds along time, two techniques can be employed when forking worlds: *snapshotting* and *shared past* (cf. Figure **??**).

**Snapshotting** consists in copying all state chunks of all timepoints from a *parent world* $p$ to the *child world* $w$, thus leaving both worlds to evolve completely independently, from past to future. Although this approach is simple, it is very inefficient in terms of time and storage to clone all state chunks of all historical records.

**Shared past** proposes to adopt an alternative approach, which makes it unnecessary to copy past state chunks. Instead, a new world $w$ is diverged from a parent $p$ at a point $s$ in time. Before $t_s$, both worlds share the same past, thus resolving the same state chunks. After $t_s$, world $w$ and $p$ *co-evolve*, which means that each have their own timeline for $t_n \geq t_s$. Therefore, both worlds share the same past before the divergent point (for $t < s$), but each evolves independently after the divergent point for $t \geq s$.

### *3.2. Many-World Graph Semantics*

With MWG, we seek to efficiently organize and analyze data that can evolve along time and alternative worlds. We define such a complex topology as a graph $G = N \times T \times W$, where $N$ is the set of nodes, $T$ the set of timepoints, and $W$ the set of worlds. However, what-if analysis needs to explore many different actions, which usually do not affect all data in all worlds and all timepoints. To address this combinatorial problem of world and timepoint alternatives, we define our MWG so that values of each node are resolved on-demand, based on a reference world and timepoint. In this section, we formalize the semantics of our MWG by starting with a base graph definition, which we first extend with temporal semantics and then with the many-worlds semantics.

The intention behind the way we formalize the semantics is twofold. First, to provide an abstract and clear defini-

tion of the main concepts of our proposed MWG. Second, we choose a formalization that reflects the design concepts of our MWG implementation in order to make it easier to map the implementation choices to the concepts and *vice versa*.

### *3.3. Base Graph (BG)*

A graph $G$ is commonly defined as an ordered pair $G = \{V, E\}$ consisting of a set $V$ of nodes and a set $E$ of edges. In order to distinguish between nodes and their states, we define a different semantics. First, we define a node as a conceptual identifier that is mapped to a "state chunk". A state chunk 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`, or `enumerations`.

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 node $n$ and $R_n$ is the set of relationship values from $n$ to other nodes.

From now on, we refer to edges as directed relationships or simply as *relationships*. Unlike other graph models (*e.g.*, Neo4J [**?** ]), our model does not support edge attributes. Nevertheless, any edge attribute can be easily modeled as an intermediate node within such graphs, without compromising the expressiveness and the efficiency. Besides being simple, this also makes our graph data model similar to the object-oriented one, which today is the dominating data model of many modern programming languages, like Java, C#, Scala and Swift. This straightforward—even direct—mapping supports a seamless integration of MWG into applications written in such languages. We decided to keep the formalization as close to the conceptual and technical level as possible and therefore reflect this design decision, to join nodes and edges, already on the formal level. In addition, as we will see in the next subsections, joining edges and nodes together in state chunks makes the definition of the temporal and many-world dimensions much more straightforward.

Then, we introduce the function $read(n)$ to resolve the state chunk of a node $n$. It returns the state chunk of the node, which contains the relationships—or edges—to other nodes. Thus, we define a base graph $BG$ as: $BG = \{read(n), \forall n \in N\}$.

Unlike common graph definitions, our base graph is not statically defined, but dynamically created as the result of the evaluation of the $read(n)$ function over all nodes $n$. Implicitly, all state chunks of all nodes are dynamically resolved and the graph aggregates the nodes accordingly to the relationships defined within the resolved state chunks. This definition forms the basis for the semantics of our proposed data model.

In this way, only the destination nodes need to be listed in the set, since all the directed edges start from the same node $n$, thus making it redundant to list the source node. For example, if we have: $c_n = \{\{att1\}, \{m, p\}\}$, where $m, p \in N$, this means that the node $n$ has one attribute and two relationships (one to node $m$ and another one to node $p$). Two directed edges can be implicitly constructed: $n \to m$ and $n \to p$. In the next sections, we incrementally override the function *read* to integrate, step by step, the time and many-world semantic.

It is important to note that this principle of "dynamic creation" is different from state-of-the-art approaches, which often rely on sequences of graph snapshots [**?** ] or a combination of snapshots and deltas [**?** ] for decomposing a graph. This usually requires to reconstruct the full graph for any given time and world, which we argue makes difficult to implement efficient what-if analytics for data in motion. As we will see in Sections **??** and **??**, the "dynamic creation" of the MWG forms the basis for an efficient graph decomposition, thus supporting an efficient what-if analytics for data in motion.

### *3.4. Temporal Graph (TG)*

To extend our $BG$ with temporal semantics, we override the function $read(n)$ with a function $read(n, t)$, with $t \in T$. $T$ is a totally ordered sequence of all possible timepoints: $\forall t_i, t_j \in T : t_i \leq t_j \vee t_j \leq t_i$.

We also extend the state chunk with a temporal representation: $c_{n,t} = (A_{n,t}, R_{n,t})$, where $A_{n,t}$ and $R_{n,t}$ are the sets of resolved values of attributes and relationships, for the node $n$ at time $t$.

Then, we define the 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. As timestamps are natural numbers, this means that they form a total order per node. We define that every state chunk belonging to a node in a $TG$ is associated to a timepoint and can therefore be organized according to 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 $tl_n = \{c_{n,t}, \forall t \in TP \subseteq T\}$.

The two core operations `insert` and `read` are defined as follows: $insert(c_{n,t}, n, t) : (c \times N \times T) \mapsto Void$, as the function that inserts a state chunk in the timeline of a node $n$, such as: $tl_n := tl_n \cup \{c_{n,t}\}$. If, for the same timestamp, a new state chunk is inserted, the former state chunk for this timestamp will be replaced. This means, there can be maximum one state chunk per timestamp, not multiple ones.

The operation $read(n, t) : (N \times T) \mapsto c$, 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 $t_i$:

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

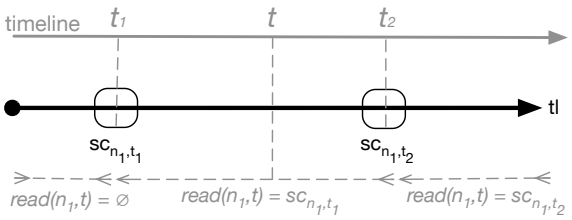



Figure 4: TG node timeline

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 $n_1$. The corresponding time validities are depicted in Figure **??**.

As 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. Once the time resolution returns the correct timepoint $t_i$, the temporal graph can be reduced to a base graph, therefore a TG for a particular $t$ can be seen as a base graph: $TG(t) \equiv BG_{t_i}$.

### *3.5. Many-World Graph (MWG)*

To extend the $TG$ with a many-world semantics, we refine the definition of the function $read(n, t)$ by considering, in addition to time, the different worlds. The function $read(n, t, w)$, with $t \in T$ and $w \in W$, where $W$ is the set of all possible worlds, which resolves the state chunk of node $n$ at timepoint $t$ in world $w$. In analogy to Section **??**, the state chunk definition is extended as follows:

$c_{n,t,w} = (A_{n,t,w}, R_{n,t,w})$, where $A_{n,t,w}$ and $R_{n,t,w}$ are the sets of resolved values of attributes and relationships, for the node $n$ at time $t$, in world $w$.

From this definition, a MWG is formalized as:

$MWG(t, w) = \{read(n, t, w), \forall n \in N\}, \forall (t, w) \in T \times W$, where $W$ is a partially ordered set of all possible worlds.

The partial order $<$ on the set $W$ is defined by the **parent** ordering, with $(p < w) \equiv (p = parent(w))$. Intuitively, the set $W$ is partially ordered by the generations of worlds. However, worlds that are created from the same parent world, or the worlds that are created from different parent worlds, cannot be compared (in terms of order) to each other. Meaning, the parent relation is not transitive. We define the first created world as the **root world**, with $parent(root) = \emptyset$. Then, all other worlds are created by diverging from the root world, or from any other existing world in the world map set $WM$ of our MWG. The divergence function is defined as:

$w = diverge(p) : World \mapsto World$, the function that creates world $w$ from the parent world $p$, with $p < w$ and $p \in WM \subseteq W$. Upon divergence, we therefore obtain $WM := WM \cup \{w\}$. According to this definition, we consider the world $w$ as the **child** of world $p$ and it is added to the world map of our MWG.

Our world structure corresponds to an unordered tree, therefore we can define the parent function as follows: for the unordered world tree $WT$, by $V(WT)$, we denote the set of all worlds in $WT$, and by $\leq_{WT}$ the hierarchical order $\leq$ of $WT$. In the following, we write $x \in WT$ instead of $x \in V(WT)$. Then, the parent of a *non-root world* $x$, denoted by $parent(x)$, is the minimum world $y$ in the set $\{z \in V \mid z > x\}$. Following this, the world $x$ is called a *child* of $parent(x)$ and the set of all children of a world $x$ is denoted by $child(x)$, *i.e.,* $child(x) = \{y \in V \setminus \{root(WT)\} \mid parent(y) = x\}$. A world with no children is called a *leaf world*.

For the MWG, we define the **local timeline of a world and a node** as $ltl_{n,w} = \{c_{n,t,w}, \forall t \in TP_{n,w}\}$, with $TP_{n,w} \subseteq T$, which is the ordered subset of timepoints for node $n$ and world $w$. As $TP_{n,w}$ is ordered, there exists a timepoint $s_{n,w}$, which is the smallest timepoint in $TP_{n,w}$, defined as $s_{n,w} \in TP_{n,w}, \forall t \in TP_{n,w}, s_{n,w} \leq t$. We call this timepoint a **divergent timepoint**—*i.e.*, where the world $w$ starts to diverge from its parent $p$ for node $n$. Following the shared-past concept between a world and its parent (cf. Section **??**), we define the global timeline of a world per node as:

$$tl(n, w) = \begin{cases} \emptyset \text{ if } w = \emptyset \\ ltl(n, w) \cup subset\{tl(n, p), \forall t < s_{n,w}, p < w\} \end{cases}$$

The global timeline of a world is therefore the recursive aggregation of the local timeline of the world $w$ with the subset of the global timeline of its parent $p$, up until the divergent point $s_{n,w}$.

Finally, we extend the functions `insert` and `read` as:

$insert(c_{n,t,w}, n, t, w) : (c \times N \times T \times W) \mapsto Void$, the function that inserts a state chunk in the local timeline of node $n$ and world $w$, such as $ltl_{n,w} := ltl_{n,w} \cup \{c_{n,t,w}\}$.

$read(n, t, w) : (N \times T \times W) \mapsto c$ is the function that retrieves a state chunk from a world $w$, at time $t$. It is recursively defined as:

$$read(n, t, w) = \begin{cases} read_{ltl_{n,w}}(n, t) & \text{if } (t \geq s) \wedge (ltl_{n,w} \neq \emptyset) \\ read(n, t, p) & \text{if } (t < s) \wedge (p < w, p \neq \emptyset) \\ \emptyset & \text{otherwise} \end{cases}$$

As a reminder, except the root world, which has no parent, every other world has exactly one direct parent. Meaning, the parent relationship is hierarchical. In this

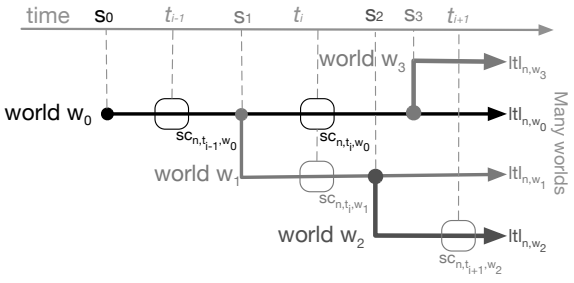



Figure 5: Many worlds example

sense, the parent relationship is *not transitive* in the general case, *i.e.,* if world $w_0$ is parent of world $w_1$ and world $w_1$ is parent of world $w_2$, does not imply that world $w_0$ is parent of world $w_2$. Following the above definitions, only in the case that world $w_1$ would have no changes, the *read* function would hierarchically resolve $w_0$ from $w_2$.

The function `insert` always operates on the local timeline $ltl_{n,w}$ of the requested node $n$ and world $w$. For the function `read`, if the requested time $t$ is higher or equal to the divergent point in time $s_{n,w}$ of the requested world $w$ and node $n$, the read is resolved on the local timeline $ltl_{n,w}$, as introduced in Section **??**. Otherwise, we recursively resolve on parent $p$ of $w$, until we reach the corresponding parent to read from.

Once the world resolution is completed, a MWG state chunk can be reduced to a temporal graph state chunk, which in turn can be reduced to a base graph state chunk once the timepoint is resolved. Similarly, over all nodes, a MWG can be reduced to a temporal graph, then to a base graph, once the read function dynamically resolves the world and time.

Figure **??** depicts an example of MWG with several worlds. $w_0$ is the root world. In this figure, $w_1$ is diverged from $w_0$, $w_2$ from $w_1$, and $w_3$ from $w_0$. Thus, we have the following partial order: $w_0 < w_1 < w_2$ and $w_0 < w_3$. However, the parent relationship does not define any order between $w_3$ and $w_2$ or between $w_3$ and $w_1$. $s_i$, for $i$ from 0 to 3, represent the divergent timepoint for world $w_i$, respectively. An insert operation on the node $n$ and in any of the worlds $w_i$, will always insert in the local timeline $ltl_{n,w_i}$ of the world $w_i$. However, a read operation on the world $w_2$ for instance, according to the shared-past view, will resolve a state chunk from $ltl_{n,w_2}$ if $t \geq s_2$, from $ltl_{n,w_1}$ if $s_1 \leq t < s_2$, from $ltl_{n,w_0}$ if $s_0 \leq t < s_1$, and $\emptyset$ if $t < s_0$.

To support with world modifications, we apply copy-on-write strategies [**?** ], but at the granularity of nodes. This means that a world is never copied, even if data is modified. Instead, only modified nodes are copied and transparently loaded. Thus, we apply the same concept we introduced to manage the temporal data aspect of the graph.

The above definitions characterize the shared past concept depicted in Figures **??** and **??**. As a consequence, all changes (or inserts) made in a parent world before the divergence point are automatically made visible in the derived worlds. Hence, any modification (node, attribute or relationship) in a parent world before a divergence point can impact derived worlds. This model suggests that modifications (or inserts) in the past—*i.e.,* before a divergence point—can potentially have unintended side-effects. To illustrate this issue, we suppose a child world in which some kind of calculations are inserted—for simplicity reasons let us just take a simple sum over data inherited from the parent world—and then data is modified or inserted in the parent world. In such a situation, state-of-the-art approaches would fail to update the calculations in the child world, thus leading to outdated states or, at best, induce complex computations to propagate the introduced modifications. The same considerations also applies with only a single world when modifying or inserting data in the past. However, having a deeply nested world-order hierarchy makes it more difficult to track and understand the impacts. Although, one valid solution could forbid modifications and inserts in the past (before a divergence point), we believe that for many use cases this would be a crippling restriction. For example, when considering data imports from various sources (*e.g.,* large CSV files, webservice interfaces), data might not be imported consistently along time. The same applies in the domain of IoT and smart grids (cf. Section **??**), we often get the consumption data from smart meters with a certain delay, which happens, for example, when a meter is not reachable for a certain amount of time due to disturbances. In this cases, the data is buffered and sent later, *e.g.,* the next day. These complex systems have to continuously deal with old and new data that have to integrated consistently in order to deliver accurate prescriptions. For these reasons, we believe that forbidding data insertion in the past and before divergence points is not a viable solution.

However, we leave it to the application developers to handle the potential side-effects. In our example, the statistics—like the average consumption of a meter per day, week, etc.—must be updated. We leave it to future work to investigate if and how such cases could be supported by MWG implementation.

## 4. GreyCat: Implementing MWG

Our MWG concept is supported by an implementation, named GreyCat, to create, read, update, fork and delete graphs and nodes along time. In particular, the following sections dive into the implementation details of GreyCat to expose the design choices we made to outperform the state-of-the-art.

### *4.1. Mapping Nodes to State Chunks*

The MWG is a conceptual view of data to work with temporal data and to explore many different alternative worlds. Internally, we structure the data of a MWG as an unbounded set of *state chunks*. Therefore, as discussed in Section **??**, we map the conceptual nodes (and relationships) of a MWG to *state chunks*. State chunks are the

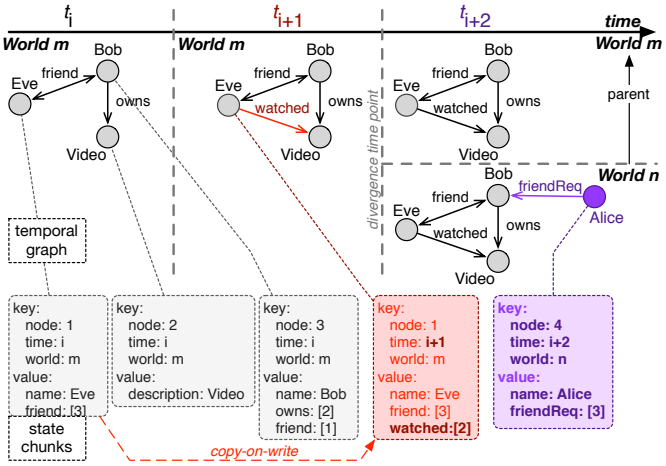



Figure 6: Mapping of nodes to storable state chunks

internal data structures reflecting the MWG and at the same time also used for storing the MWG data. 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 **??** depicts, as a concrete example, how nodes are mapped to state chunks in accordance with the semantic definitions of Section **??**.

At time $t_i$ (the starting time of the MWG), GreyCat maps the nodes and the relationships to 3 state chunks: one for Eve, one for Bob, and one for Bob's video. At time $t_{i+1}$, the MWG evolves to include a relationship watched from Eve to Bob's video. Since this evolution only affects Eve, GreyCat only creates an additional state chunk for Eve from time $t_{i+1}$. All other nodes are kept unchanged at time $t_{i+1}$ and thus still up-to-date. Then, at time $t_{i+2}$, the world $m$ of the MWG diverges into two worlds: $m$ and $n$. While world $m$ remains unchanged, in world $n$ Bob meets Alice, who sends a friend request to Bob. As only Alice requires to be updated, GreyCat only creates one state chunk for Alice from time $t_{i+2}$ and world $n$. Here, we notice the benefit of the MWG and its semantics: while we are able to represent complex graph topologies, which evolve over time and many worlds, we only need to store a fraction of this structure. In this example, the graph contains 13 different nodes and 16 relationships (counting each bidirectional relation as two) and evolves along 2 different worlds and 3 different timestamps, but we only have to create 5 state chunks to represent all of this structure. Whenever the MWG is traversed, the correct state chunks are retrieved with the right time and world. The resolution algorithm behind this approach is detailed in Section **??**.

State chunks are the storage units of GreyCat, which are stored on disk and loaded into main memory whenever the MWG is traversed or when nodes are explicitly retrieved. State chunks are lazy loaded, as only attributes and sets of identifiers are loaded upon request. This theoretically allows to process unbounded MWGs. For persistent storage of state chunks, we rely on state-of-the-art key/value stores by using the 3-tuple of $\{node; time; world\}$ as key and the state chunk as value. We serialize chunk states into Base64-encoded `blobs`. Despite being simple, this format can be used to distribute state chunks over networks. Moreover, it reduces the minimal required interface to insert state chunks into, and read from, a persistent data store to `put` and `get` operations. This allows GreyCat 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.

This mapping approach copies state chunks only on-demand—*i.e.*, on-write (per time and world). This delivers very efficient read and write operations at any point in time. Using diffs instead of our proposed on-demand forking concept could—in some cases—save disk space, but it would come with a much higher cost for inserting and reading.

### *4.2. Indexing and Resolving Chunks*

This section focuses on the index structures used in GreyCat and the state chunk resolution algorithm. In particular, GreyCat combines two structures for the indexes of the MWG: *time trees* and *many-world maps*.

#### *4.2.1. Index Time Tree (ITT)*

As discussed in Section **??**, timepoints are chronologically ordered. This creates implicit intervals of "validities" for nodes in time. Finding the right "position" in a timeline of a node has to be very efficient. New nodes can be inserted at any time—*i.e.*, not just after the last one. Besides, ordered trees (*e.g.*, binary search trees) are suitable data structures to represent a temporal order, since they have efficient random insert and read complexities. If we consider $n$ to be the total number of modifications of a node, the average insert/read complexity is $O(log(n))$ and $O(n)$ in the worst case (inserting new nodes at the end of a timeline). Given that inserting new nodes at the end of a timeline and reading the latest version of nodes is the common case, we adopt red-black trees [**?** ] for the implementation of our time tree index structure. The self-balancing property of red-black trees avoids the tree to only grow in depth, while it improves the worst case of insert/read operations to $O(log(n))$. Furthermore, we used a custom Java implementation of red-black trees, using primitive arrays as a data backend to minimize garbage collection times, as garbage collection can be a severe bottleneck in graph data stores [**?** ]. Every conceptual node of a MWG can evolve independently in time. For scalability reasons, we decided to use one red-black black tree, further called *index time tree* (ITT), per conceptual node to represent its timeline. Figure **??** depicts how the ITT looks like and evolves for the node *Eve* introduced in Figure **??**.

As it can be seen, at time $t_i$, one conceptual version of node *Eve* exists and therefore the ITT has only one entry.

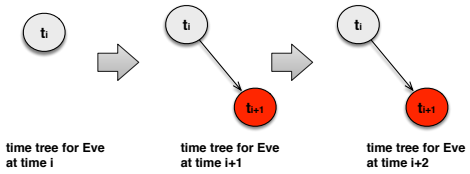



Figure 7: Example ITTs for node *Eve* of Figure **??**

At time $t_{i+1}$, *Eve* changes, a new conceptual version of this node is created and the ITT is updated, accordingly. Then, at time $t_{i+2}$, there are additional changes on the MWG, which do not impact *Eve*: the ITT of *Eve* remains unchanged.

ITTs are special state chunks and stored/loaded in the same way to/from key/value stores than any other state chunk. More specifically, as key, we use the *id* of the corresponding conceptual node, together with the *world identifier* and the *type* (time tree in this case). The value is a serialized and Base64 encoded blob of the tree's values.

### *4.2.2. World Index Maps (WIM)*

Since new worlds can diverge from existing worlds at any time and in any number, the hierarchy of worlds can arbitrarily grow both in depth and width. As it can be observed in Figure **??**, the divergent point is therefore not enough to identify the parent relationship. In our many-world resolution, we use a global hash map that stores, for every world $w$, the corresponding parent world $p$ from which $w$ is derived: $w \rightarrow p$. We refer to it as the *global world index map* (GWIM). This allows GreyCat to insert the parent $p$ of a world $w$, independently of the overall number of worlds, in average in constant time $O(1)$ and in the worst case in $O(l)$, where $l$ is the total number of worlds. We also use a custom Java hash map implementation built with primitive arrays to minimize garbage collector effects.

In addition to the GWIM, GreyCat defines one local index map, called *local world index map* (LWIM), per conceptual node to identify different versions of the same conceptual node in different worlds. In this map, we link every world in which the node exists with its "local" divergent time, meaning the time when this node was first modified (or created) in this world and therefore starts to diverge from its parent: $w \rightarrow t_{local\ divergence}$. As we will see, this information is needed to resolve state chunks. The LWIM is the core allowing nodes to evolve independently in different worlds. When a conceptual node is first modified (or created) in a world, its state chunk is copied (or created) and the LWIM of the node is updated—*i.e.*, the world in which the node was modified is inserted (and mapped to its local divergence time). Both the GWIM and the LWIM must be recursively accessed for every read operation of a node (see the semantic definitions in Section **??**).

Other than the total number of worlds $l$, we define another notation: $m$, as the maximum number of hops necessary to reach the root world ($m \leq l$). Figure **??** reports on

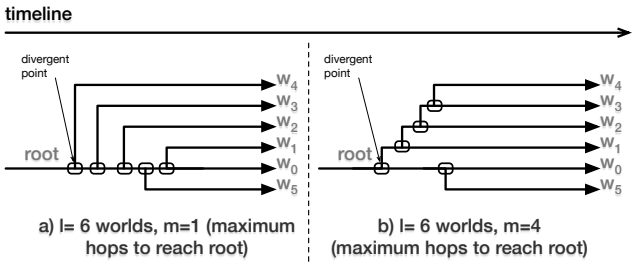



Figure 8: Example of different configurations of the same number of worlds $l$, but with a different $m$

an example of 2 MWGs with the same number of worlds $l = 6$ but, in the first case we can always reach the root world in $m = 1$ hop, while in the second case, we might need $m = 4$ hops in the worst case (from world $w_4$ to $w_0$).

The recursive world resolution function has a minimum complexity of $O(1)$ in the best case, where all worlds are directly derived from the root world (shown in Figure **??**-a). The worst case complexity is $O(m) \leq O(l)$, like for the **stair-shaped** case shown in Figure **??**-b, where we might to have to go several hops down before actually resolving the world.

Like it is the case for ITTs, WIMs are special state chunks and stored/loaded in the same way as regular state chunks.

### *4.2.3. Chunk Resolution Algorithm*

To illustrate the resolution algorithm of MWG, let us consider the example of Figure **??**, assuming we want to resolve node *Bob* at time $t_{i+2}$ in world $n$. We first check the LWIM of *Bob* and see that there is no entry for world $n$, since *Bob* has never been modified in this world. Therefore, we resolve the parent of world $n$ with the GWIM, which is world $m$. A glance in the LWIM of *Bob* reveals that world $m$ diverged (or started to exist in this case) for *Bob* at time $t_i$. This indicates MWG that world $m$ is the correct place to lookup state chunks, since we are interested in *Bob* at time $t_{i+2}$, which is after time $t_i$ where world $m$ for *Bob* becomes valid. World $m$ is the "closest" where *Bob* has been actually modified. Otherwise, it would have been necessary to recursively resolve the parent world of $m$ from the GWIM until we find the correct world. In a last step, we look at the ITT of *Bob* to find the "closest" entry to time $t_{i+2}$, which is time $t_i$. Finally, this index indicates GreyCat to resolve the state chunk for *Bob* (id 3) with the following parameters: $\{node\ 3; time\ i; world\ m\}$. This state chunk resolution is summarized in Algorithm **??**.

The full resolution algorithm has a complexity of $O(1)$ for insert, and a complexity of $O(1) + O(m) + O(n) \leq O(l) + O(n)$ for read operations, where $l$ is the number of worlds, and $n$ the number of time points, and $m$ the maximum depth of worlds.

**Algorithm 1** State chunk resolution

```
1: procedure RESOLVE(id, t, w)
2:     lwim ← getLWIM(id)
3:     s ← lwim.get(w)
4:     if t >= s then
5:         itt ← getITT(id)
6:         return itt.get(t, w)
7:     else
8:         p ← GWIM.getParent(w)
9:         return resolve(id, t, p)
10:    end if
11: end procedure
```

### *4.3. Scaling the Processing of Graphs*

Memory management and transactions or "units of work" are closely related. In GreyCat, we first connect our graph to a database. This connection, further called *unit of work* (UoW), marks the beginning of what can be seen in a broader sense as a long-living transaction. While working with this connection, the state chunks representing the MWG are loaded on-demand into main memory. All modifications of the MWG are performed in memory. When saving, the modified and new state chunks (internally marked as *dirty*) are written from memory into (persistent) key/value stores. Then, the allocated memory is tagged as eligible for eviction, which marks the end of the UoW. This, together with the on-demand loading of state chunks into main memory, allows to work with graphs of unlimited sizes, in theory. To increase the read performance, GreyCat uses local eviction-based LRU caches [**?** ].

### *4.4. Concurrency and Consistency*

In this section, we discuss concurrency and consistency properties of GreyCat. This section focuses on multi-core architectures, while Section **??** describes these properties for distributed deployments.

**Concurrency**: in order to ensure concurrency, GreyCat uses a per-node locking strategy for every operation that impacts the timeline of a particular node—*i.e.,* new time or world insertion. For such a major operation, it is important to notice that we lock the whole conceptual node rather than only one state chunk, belonging to one precise world and time. This means that all data structures—*i.e.,* all state chunks and index structures belonging to the locked node, are locked for temporal or many-world write operations. This way, temporal indexes are always consistent. Nonetheless, for other write operations, such as attribute value modification or relationship insertion, we adopt a lock per state chunk. Therefore, parallel writes are allowed if they do not modify the node timeline or the same time-point. To keep achieving a high level of parallelism, only write operations are blocked while concurrent read operations remain—with some restrictions—allowed. More specifically, GreyCat uses a *compare-and-swap* mechanism on the LWIM ensuring that a read operation of a specific node (for a given time and given world) is blocked in case that the requested node is concurrently modified in the same world. Otherwise, in the case of read operations for the same node in another world, are fully concurrent without locking.

**Consistency**: the previously-defined locking strategy of GreyCat aims at ensuring consistency per node under parallel read and write operations. In addition, GreyCat indexes are created as node attributes, therefore following the locking scheme indexes are consistent by using the standard locks like for write operations. Based on this decomposition of GreyCat indexes, no global consistency strategy is necessary.

### *4.5. Distribution*

GreyCat defines a data access layer that can be distributed over several machines. However, the current implementation of GreyCat does not define a specific sharding mechanism to homogeneously distribute data streams across a pool of computers. Instead, GreyCat relies on the underlying key/value store for distributed storage of our graph. Depending on the application requirements, different key/value stores can be plugged via a simple interface, which essentially only relies on the implementation of a `get` and `put` method. For example, when performance is the most critical requirement, but fault tolerance, availability, distribution, and replication are less important, GreyCat provides drivers for RocksDB [**?** ] and LevelDB [**?** ]. On the other hand, when availability, scalability, and replication are of importance, GreyCat provides also implementations of drivers for Cassandra [**?** ] and HBase [**?** ].

To interconnect heterogeneous platforms, such as Android devices, web (browser) applications, and Java-based servers, GreyCat uses a bus concept based on WebSocket or MQTT protocols. This implies a distribution consistency mechanism. To avoid the use of distributed locks, which could drastically decrease performance, we decided to consider an optimistic approach using *Conflict-Free replicated Data Types* (CRDT). Advantages, limitations, and alternatives to CRDT are extensively discussed in [**?** ]. By using CRDT structures for every chunk, we ensure the ability to merge every distributed concurrent modification in a consistent manner. In addition, we are considering the extension of such mechanism by using a Raft algorithm to offer, on-demand, a consensus primitive to ensure distributed consistency for a dedicated zone of the temporal graph.

### *4.6. Federation versus Distribution*

The computational model of GreyCat relies on a lazy loading mechanism that loads compressed data chunks, from a generic key-value store into main memory. From a data perspective, this key-value store interface can be implemented using many established methods to enable data consistency and security. For instance, a cluster of computers, driven by a sharding mechanism that replicates

data consistently over the computers of the cluster can guarantee a desired level of replication. Sharding techniques, together with peer-to-peer protocols, have been used for several years now in domains like Big Data [**?** ] and Cloud storage [**?** ].

However, despite distributed key-value stores with their variants using Gossip or consensus-based protocols, which can offer guaranties regarding data consistency, safety, and availability, they do not distribute the computational load once data chunks are loaded into memory. In this context, two approaches are often discussed in literature: distributed synchronized workers [**?** ], as for example used in many Hadoop clusters, and federated computational nodes, divided accordingly to a domain separation [**?** ]. The key distinguishable point of these approaches is that federation relies on domain expertise to distribute data to enable computations close to the data source, while pure distributed models scale-up horizontally without any pre-assignment of workers to computers.

GreyCat is designed to empower live analytics over temporal graphs. Therefore, latency and data freshness are key. In addition, past values are often necessary, for example to detect recurring patterns. This advocates for a federated model of GreyCat where the temporal graph is sliced and each slice is associated with one or several computational nodes. A federation protocol would then map requests to the corresponding computational node which is responsible for the requested slice of the distributed graph. An efficient federation protocol is part of future work. The main idea is to slice a temporal graph based on nodes IDs. This slicing mechanism would lead to a routing table, which can be replicated over all federated instances. We are currently experimenting with several slicing strategies.

### *4.7. Working with MWGs*

In order to work with MWGs—*i.e.,* to create, navigate, and analyse them—GreyCat provides Java APIs for developers. For example, Listing **??** illustrates how a graph can be created using GreyCat.

**Listing 2** Java API to create a MWG

```
public static final long TIME = 0;
public static final long WORLD = 0;

Graph g = new GraphBuilder().build();
g.connect(new Callback<Boolean>() {
  @Override
  public void on(Boolean result) {
    Node eve = graph.newNode(WORLD, TIME);
    node.set("name", Type.STRING, "Eve");
    Node bob = graph.newNode(WORLD, TIME);
    node.set("name", Type.STRING, "Bob");
    eve.addToRelation("friend", bob);
    bob.addToRelation("friend", eve);
  }
});
```

The listing creates a graph consisting of two nodes, `eve` and `bob`. Both are created in world `0` and for time `0`. Node `eve` has one attribute `name`, which is of type `String` and is set to `Eve`. In addition, it has a relation named `friend` to node `bob`.

Any graph can be manipulated and evolve over time. Listing **??** depicts how GreyCat's API can be used to do this, by extending the example of Listing **??**.

**Listing 3** Java API to change a graph over time

```
public static final long TIME = 0;

Graph g = new GraphBuilder().build();
g.connect(new Callback<Boolean>() {
  @Override
  public void on(Boolean result) {
    //... code from Listing 2
    long newTime = TIME + 100;
    eve.travelInTime(newTime, node -> {
      node("age", Type.INTEGER, 18);
    });
  }
});
```

The listing moves the node `eve` to `TIME + 100` and changes its attribute `age` to `18`. This means, form time `TIME + 100` on, `age` will be resolved as `18`, before this time, it will be resolved to `17`.

Now, let us consider several different worlds in Listing **??**. We are interested in diverging a world `1` from world `0` and change the `name` of Eve in this divergent world to `Alice`.

**Listing 4** Java API to change a graph in several different worlds

```
Graph g = new GraphBuilder().build();
g.connect(new Callback<Boolean>() {
  @Override
  public void on(Boolean result) {
    //... code from Listing 2
    long newWorld = 1;
    eve.travelInWorld(newWorld, node -> {
      node("name", Type.STRING, "Alice");
    });
  }
});
```

In order to process and analyse such a temporal graph, GreyCat provides an API to efficiently navigate and query the content of the graph. From any given node, the graph can be easily traversed, as shown in Listing **??**.

Since most operations in GreyCat are non-blocking and therefore asynchronous, deep navigations inside a graph can lead to many nested callbacks (*cf.* Listing **??**). Therefore, GreyCat offers a *Task API*, which is comparable to `Promises` and `Futures` and allows the developer to chain several navigation operations without the need to nest them. In addition, this Task API allows the developer to specify if a given number of `Tasks` can be executed in

**Listing 5** Java API to traverse a graph

```
Graph g = new GraphBuilder().build();
g.connect(new Callback<Boolean>() {
  @Override
  public void on(Boolean result) {
    //... code from Listing 2
    eve.relation("friend", new Callback<Node[]>()
        {
      @Override
      public void on(Node[] friends) {
        //...
      }
    });
  }
});
```

parallel and also if they should be executed on a specific machine, *e.g.,* local or on a remote machine. Therefore, tasks, their parameters, and results must be serializable.

While Listing **??** shows how the graph can be navigated from a specific node, the question remains how we find this starting point in the first place. To solve this issue, GreyCat uses indexes, which can be created and queried as shown in Listing **??**.

**Listing 6** Java API to query a graph

```
//...
// creating index "nameIndex" from attribute "
    name"
 g.index(WORLD, Constants.BEGINNING_OF_TIME, "
     nameIndex", new Callback<NodeIndex>() {
   @Override
   public void on(NodeIndex index) {
     index.addToIndex(self, "name");
   }
 });

// using indexes
indexNode.find(nodes -> {
  // filter here
});
```

As it can be noticed in the listing, the index itself is a regular node and can also evolve over time.

While, the above examples are illustrating the core concepts of GreyCat's API to work with MWG, more advanced queries and navigation methods are available. The complete API can be found online on GitHub.[3]

## 5. Experiments

This section reports on the extensive experiments we performed to assess the performance of GreyCat along several perspectives, from reporting on a real industrial case study (cf. Section **??**), to comparing it against the closest solutions in the state of the art (cf. Sections **??**–**??**), and to stressing it against micro-benchmarks (cf. Sections **??**–**??**). The goal of these experiments is not to exhaustively compare GreyCat against Neo4J and others, but rather to evaluate GreyCat from different perspectives and to report how it performs in different settings. The comparisons with Neo4J and others are mainly used to report numbers of closest approaches for the corresponding experiments. The experiments are performed to stress the limits of our implementation and observe if the performance and scalability is sufficient with regards to the requirements we identified in Section **??**.

### *5.1. Experimental Setup*

For all these experiments, we report on the throughput of **insert** and **read** operations as key performance indicators. We executed each experiment 100 times to assess the reproducibility of our results. Unless stated otherwise, all the reported results are the average of the 100 executions and all experiments have been executed on the *high performance computer* (HPC) of the University of Luxembourg (Gaia cluster) [**?** ]. We used a Dell FC430 with 2 Intel Xeon E5-2680 v3 processors, running at 2.5 GHz clock speed and 128 GB of RAM. The experiments were executed with Java version 1.8.0_73. All experiments (except the comparison to InfluxDB) have been executed in memory without persisting the results. The rationale behind this choice is to isolate the performance of GreyCat, and not the performance of 3rd-party key/value stores, which we use for persisting the data. For similar reasons, caches have been deactivated for all experiments in order to report worst-case performances. All our experiments are available on Bitbucket.[4]

All experiments—for GreyCat, Neo4j, and InfluxDB—are conducted in a non-clustered way, and are comparable, since the goal of the evaluation is to show the capabilities and limits of GreyCat. Data sharding and distribution are out of scope of this article. The current limitations and future work regarding these and other topics, such as cluster deployments, are discussed in Section **??**. We focus on the evaluation of *create*, *read*, and *update* operations since the contributions of this article are a MWG data model and the associated storage support rather than processing. Graph processing algorithms, which are based on these primitives, are therefore considered as out of scope of this article.

### *5.2. Smart Grid Case Study*

In this experiment, we evaluate GreyCat on a real-world smart grid case study, which we introduced in Section **??**. In particular, we leverage MWG to optimize the electric load in a smart grid. Therefore, we build profiles for the consumption behavior of customers. Based on the

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

[4] `git@bitbucket.org:thhartmann/greycat-experiments.git`

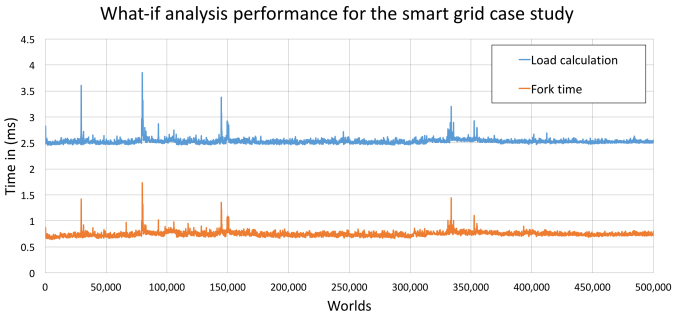



Figure 9: Performance of load calculation in a what-if scenario for the smart grid case study

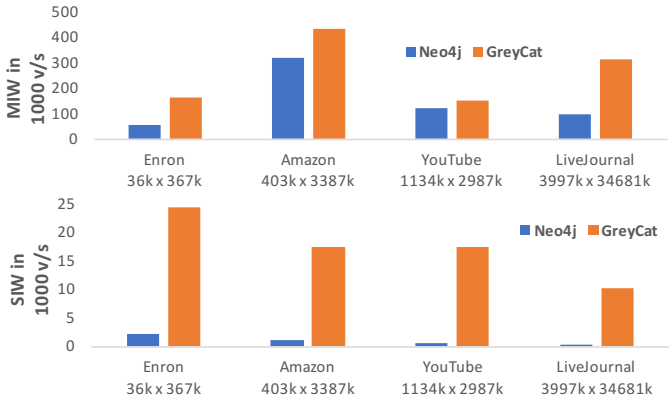



Figure 10: MIW and SIW benchmark throughput in 1,000 values/second. Bigger numbers mean better results. Numbers beneath dataset names mean number of nodes and edges, *i.e. nodes x edges*.

profiles, we simulate different hypothetical what-if scenarios for different topologies, compute the expected electric load in cables, and derive the one with the most balanced load in all cables. This allows to anticipate which topology is likely to be the best for the upcoming days.

For this experiment, we use an in-memory configuration, without a backend storage, because we do not need to persist all the different alternatives. We use the publicly available smart meter data from households in London [**?** ]. As the dataset from our industrial partner CREOS is confidential, we use this publicly available dataset for the sake of reproducibility. The grid topology used in our experiments is based on the characteristics of the CREOS smart grid deployment [**?** ]. We consider 5,000 households connected to the smart grid, including 4,000 consumption reports per customer. This leads to a large-scale graph with 20,000,000 conceptual nodes used to learn the consumption profiles. As described in [**?** ] around 100 customers are connected to one transformer substation. We simulate 50 power substations for our experiments and we suppose that every household can be connected to every power substation. This is a simplification of the problem, since which household can be connected to which power substation depends on the underlaying physical properties of the grid, which we neglect in the following experiment.

Figure **??** reports on the what-if analysis performed over 500,000 worlds where, in each world, we mutate 3 % of the power substations connections to smart meters. We plot the latency (in *ms*) of the load calculations and world creation (fork time) per world. As depicted in Figure **??**, both curves are quite constant, with some peaks due to garbage collection. Based on this experiment, we can conclude that GREYCAT is scalable and can apply to large-scale systems, such as smart grids.

### *5.3. Base Graph Benchmarks*

The objective of this benchmark is to evaluate the performance of GREYCAT as a standard graph storage by neglecting time and many-worlds. Therefore, this section compares the performance of GREYCAT to state-of-the-art graph databases. For this comparison, we use the graph database benchmark [**?** ] provided by Beis *et al.* [**?** ]. This benchmark is based on the problem of community detection in online social networks. It uses the public datasets provided by *Stanford Large Network Dataset Collection* [**?** ]. This dataset collection contains sets from "social network and ground-truth communities" [**?** ], which are samples extracted from Enron, Amazon, YouTube, and LiveJournal. The benchmark suite defines several metrics, among which:

**Massive Insertion Workload (MIW)** creates the graph database for massive loading, then populates it with a dataset. The creation throughput of the whole graph is reported;

**Single Insertion Workload (SIW)** creates the graph database and loads it with a dataset. Every insertion (node or edge) is committed directly and the graph is constructed incrementally. The insertion throughput is reported.

We compare the performance of GREYCAT to Neo4J [**?** ], which was the best performing base graph in [**?** ]. Figure **??** reports on the results of MIW and SIW, achieved by GREYCAT and Neo4J—both in-memory and not persisting data—along the different datasets. The goal of this benchmark is not to exhaustively compare GREYCAT against Neo4J. The comparisons with Neo4J are made to put the numbers in relation to something known. The reason why data is not persisted in the above experiments is that the persistence mechanisms of Neo4J and GREYCAT are quite different. Neo4J uses a fully transactional system (supporting ACID transactions [**?** ]), whereas GREYCAT has no transactional guarantees and directly serialises and flushes the data to a K/V store. GREYCAT has also no special optimizations for textual data and is not yet efficient for storing large texts.

Therefore, in order to evaluate the respective overhead of persisting data, the following experiment uses a dataset provided by the University of Standford, which contains movie reviews from amazon [**?** ]. More specifically, the data span a period of more than 10 years, including all (around 8 million) reviews up to October 2012. Reviews include product and user information, ratings, and a plain-

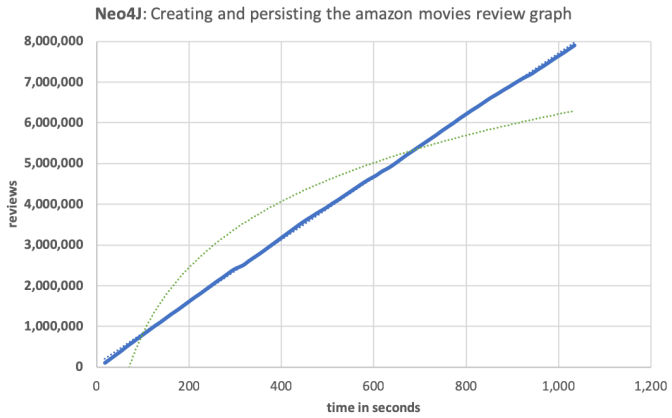



Figure 11: Time (in seconds) to create and persist the amazon movies review graph to disk using Neo4J. The x-axis shows the time in seconds and the y-axis the number of reviews.

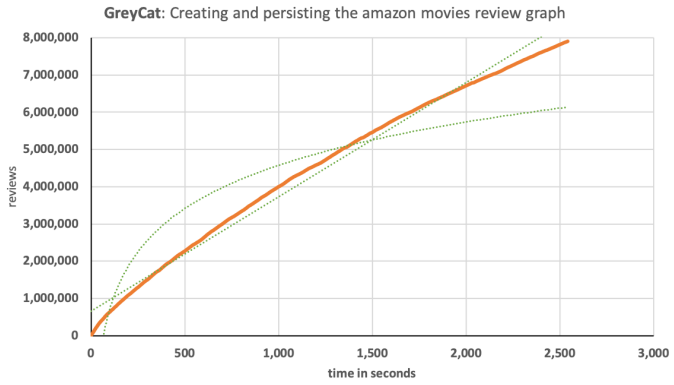



Figure 12: Time (in seconds) to create and persist the amazon movies review graph to disk using GreyCat. The x-axis shows the time in seconds and the y-axis the number of reviews.

text review. We create a graph representing this information and persisting it to disk. For GreyCat, we rely for the storage on the K/V store RocksDB [**?** ]. RocksDB is a derivate of Google's LevelDB developed by Facebook. It is important to note that the creation of of this graph does not only contain insert operations but also plenty of read operations and navigations in the graph, as the users and products for a certain review need to be retrieved. The dataset contains different types of data, especially lot's of text (review text, review summary, profile name, user id, product id). This experiment reflects a more generic use case. Figure **??** and **??** respectively shows the time to populate and persist the amazon movies review graph to disk using Neo4J and GreyCat. Looking at the total time to create and persist the graph, **Neo4J takes 1,041 seconds**, **whereas GreyCat takes 2,542 seconds** to insert the nearly 8 million reviews. This corresponds to a **factor of 2.5 in favour of Neo4J**. Both figures show in addition a linear and a logarithmic trend line (dashed green lines). As can be seen from the figures, Neo4J is almost exactly linear, whereas GreyCat is somewhere between linear and logarithmic. A main challenge for GreyCat in this experiment is that the amazon movies review graph contains lot's of text, which GreyCat is not yet optimized for.

This experiment shows that the base graph of GreyCat is still comparable (in terms of magnitudes of order) to a commercial state-of-the-art graph database, despite the overhead coming from time and many-world management.

### *5.4. Temporal Graph Benchmarks*

This experiment aims to evaluate the complexity of the ITT (cf. Section **??**). We compare the performance of temporal data management of our approach with plain time series databases. Therefore, we consider only **one world** and **one node id** and we benchmark the throughput of *insert* and *read* operations over a varying size of timepoints, from 1 million to 256 million. Table **??** reports on

Table 1: Average *insert* and *read* time, for different timepoints, for one node and in the same world.

| **(n) in million** | **Insert speed** (1,000 **val./s**) | **Read speed** (1,000 **val./s**) | **Insert / log(n)** | **Read / log(n)** |
|---|---|---|---|---|
| 1 | 589.17 | 605.30 | 42.6 | 43.8 |
| 2 | 565.05 | 564.11 | 38.9 | 38.8 |
| 4 | 554.40 | 544.23 | 36.4 | 35.8 |
| 8 | 537.22 | 528.18 | 33.8 | 33.2 |
| 16 | 520.98 | 516.26 | 33.2 | 31.1 |
| 32 | 515.05 | 485.73 | 29.8 | 28.1 |
| 64 | 489.55 | 458.32 | 27.2 | 25.5 |
| 128 | 423.53 | 400.49 | 22.7 | 21.5 |
| 256 | 391.56 | 378.50 | 20.2 | 19.5 |

the measured results under progressive load, to check the complexity according to the expected one.

As one can observe, *read* and *insert* performance follows an $O(log(n))$ scale as $n$ increases from 1 million to 256 million. The performance deterioration beyond 32 million can be explained due to a 31 bit limitation in the hash function of the ITT. This comes from the fact that our ITT is implemented as a red-black tree backed by primitive Java arrays. These are limited to 31 bit indexes. At these large numbers, collisions become very recurrent. For instance, for the 256 million case, there are around 8 % of collisions. This compares to less than 0.02 % of collisions for 1 million. To address this problem, we plan for future work an off-heap memory management implementation (based on Java's unsafe operations), which would allow us to solve the limitation of 31 bit indexes for primitive arrays and to use hash functions with more than 31 bits.

To compare with a time series database, namely InfluxDB, we use the InfluxDB benchmark [**?** ]. It consists of creating 1,000 nodes (time series) where they insert 1,000 values in each node on MacBook Pro, resulting in a graph with conceptually 1,000,000 million nodes. The second test creates 250,000 nodes and inserts 1,000 values in each of the created nodes, on an Amazon EC2 i2.xlarge instance. This results in a large-scale graph with conceptually 250,000,000 nodes. For both experiments, data is persisted to disk. Unlike in our experiments for the base

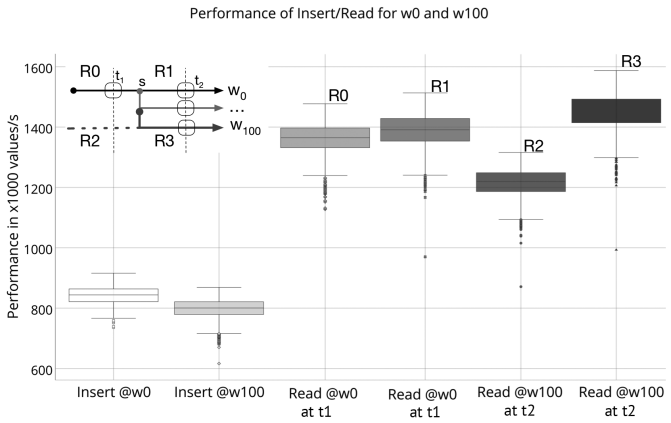



Figure 13: *Insert* and *read* performance before and after the divergence timepoint **s**

graph, in Section **??**, we persist data for these experiments due to two main reasons: 1) the persistence model of InfluxDB is much closer to the one of GreyCat, so that a direct comparison is reasonable, and 2) we use as a basis for the experiments the benchmark published by InfluxDB, where the data is persisted.

The main difference with the above experiment is that the ITT of each node does not grow the same way in terms of complexity as an ITT of 250 million elements in a single node does. For the sake of comparison, we applied the same benchmarks using the same types of machines. We use RocksDB [**?** ] as our key/value backend. Despite the fact that our MWG is not limited to flat time series, but full temporal graphs, we are able to outperform InfluxDB by completing the MacBook test in 388 seconds compared to their 428 seconds (10 % faster), and by getting an average speed of 583,000 values per second on the amazon instance, compared to their 500,000 values per second (16 % faster). We note that, when all elements are inserted in the same ITT, the speed drops to 391,560 inserts per second on average (cf. Table **??**). This is due to the increased complexity of balancing the ITT of one node. The experiment therefore assesses that GreyCat is able to manage full temporal graphs as efficiently (on a comparable scale) as time series databases are able to manage flat sequences of timestamped values. It is important to note that the experiments were performed on a single node (computer) deployment and that one cannot automatically conclude that GreyCat also outperforms time series databases like InfluxDB in a cluster deployment. Cluster deployments are discussed in short in Section **??**. A deeper investigation of cluster deployments and distribution of GreyCat is part of future work (see also Section **??**).

### *5.5. Node-scale Benchmarks*

In the following experiments, we evaluate the performance of our MWG structure and therefore do not persist data to disk, as this would pollute the evaluation with the time needed to store data on the disk. The overhead of persisting data to disk is shown in Section **??**. In this experiment, we demonstrate the effect on *insert* and *read* performance of creating many worlds from one node. Diverging only one world from the root world is not enough to measure a noticeable performance difference. Therefore, we created 100 nested parallel worlds from root world $w_0$. We first measure the *insert* performance for the worlds $w_0$ and $w_{100}$. Then, we measure—for the root world—the *read* performance $R_0$ at a shared past timepoint $t_1 = 5000 < s$ and $R_1$ at timepoint $t_2 = 15000 > s$ (after the divergence). We repeat the experiments for the same timepoints $t_1$ and $t_2$, but from the perspective of world $w_{100}$, to get *read* performance $R_2$ and $R_3$. The results are depicted in Figure **??** as box plots over 100 executions. We can conclude that the *insert* performance is similar for both worlds. The *read* performance for the root world is not affected by the divergence $R_0 = R_1$, while the *read* performance of world $w_{100}$ depends on the timepoint—*i.e.*, it is faster to read after the divergence point than before it ($R_3 > R_2$). This is due to the recursive resolution algorithm of GreyCat, as explained in Section **??**.

In this experiment, we validated that the write and *read* performance on the GreyCat are not affected by the creation of several worlds. In particular, we showed that the *read* speed is kept intact after the divergence for the child worlds.

### *5.6. Graph-scale Benchmarks*

To stress the effect of recursive world resolution, we consider the **stair-shaped** scenario presented in Figure **??**-b. In this benchmark, we create a graph of $n = 2000$ nodes, each having an initial fixed timeline of 10,000 timepoints in the main world. This results in an initial graph of 20,000,000 conceptual nodes. Then, we select a fixed $x$ % amount of these nodes to go through the process of creating the shape of stairs of $m$ steps across $m$ worlds. In each step, we modify one timepoint in the corresponding world of the corresponding node. For this experiment, we vary $m$ from 1 to 5,000 worlds by steps of 200 and $x$ from 0 to 100 % per steps of 10 %. This generates 250 different experiments. We executed each experiment 100 times and averaged the *read* performance of the whole graph before the divergence point, from the perspective of the last world. Figure **??** depicts the results as a heat map of the average *read* performance for the different combinations of number of worlds and percentage of nodes changed. The brightest area (lower left) represent the best performance (low number of worlds or low percentage of nodes changed in each world). The darkest area (upper right) represent up to 26 % of performance drop (when facing an high percentage of changes and an high number of worlds).

This benchmark is the worst case for the MWG, since for $m^{th}$ worlds, a *read* operation might potentially require $m$ hops on the WIM, before actually resolving the correct state (*e.g.,* reading the first inserted node from the perspective of the last created world), as discussed in Section **??**. The performance drop is linear in $O(m)$ and

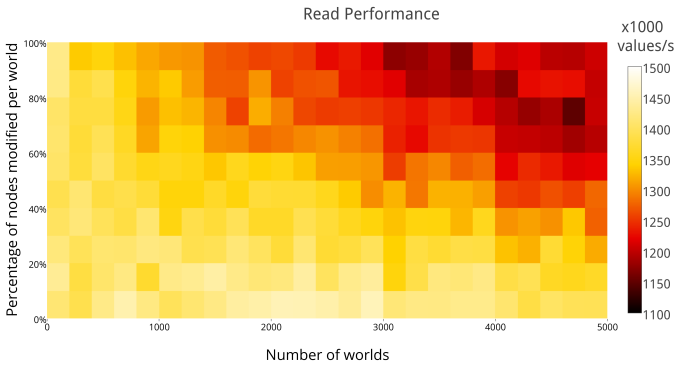



Figure 14: Read performance, over several worlds and several percentage of modified nodes

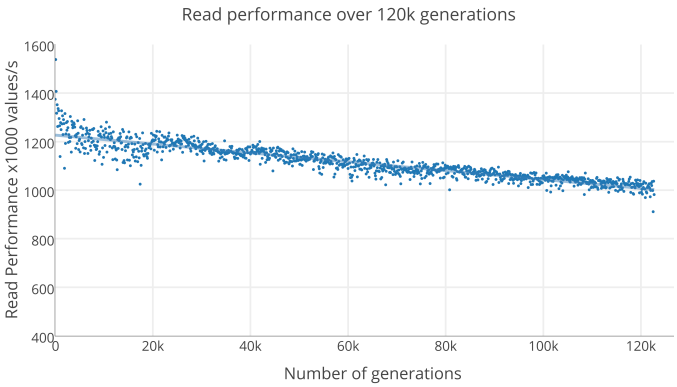



Figure 15: Average *read* performance over 120,000 generations with 3 % mutations

according to the percentage of nodes changed from one world to another. For less than 20 % of changes, the performance drop is hardly noticeable even at an high number of worlds (lower right). We note that our solution only stores the modifications for the different worlds and rely on the resolution algorithm to infer the past from the previous worlds. Any snapshotting technique, cloning the whole graph of 2,000 nodes, each including 10,000 timepoints—*i.e.*, a graph with 20,000,000 nodes—5,000 times would be extremely costly to process. To sum up, we show in this section that our index structure allows independent evolution of nodes at scale. The performance decreases linearly with the percentage of nodes changed and the maximum of worlds reached.

### *5.7. Deep What-if Simulations*

As the motivation of our work is to enable deep what-if simulations, we benchmark in this section the *read* performance over a use-case similar to the ones we can find in this domain. We use a setup similar to the previous section: a graph of $n = 2,000$ nodes with initially 10,000 timepoints in the root world. The difference is that we fixed the percentage of changes between one world to another to $x = 3\,\%$ (similar to a nominal mutation rate in genetic algorithms of 0.1–5 % [**?** ]). The second difference is that changes only randomly affect 3 % of the nodes for each step. This is unlike the previous experiment, where the target was to reach a maximum depth of worlds for the same amount of $x\,\%$ of nodes. We executed this simulation in steps of 1,000 to 120,000 generations (120 experiments, each repeated 100 times). The number of generations is similar to the ones in genetic algorithms [**?** ]. In each generation, we create a new world from the previous one and randomly modify 3 % of the nodes. At the end of each experiment, we measure the performance of reading the whole graph of 1,000 nodes. Figure **??** reports on the results MWG achieves. In particular, one can observe that the *read* performance drops linearly, 28 % after 120,000 generations. This validates the linear complexity of the world resolution, as presented in Section **??**, and the usefulness of our approach for what-if simulation when a small percentage of nodes change, even in a huge amount of deep nested worlds.

## 6. Limitations and Future Work

While we focused in this article on single-node deployments and mostly (except the experiments in Section **??**) on in-memory experiments, we plan to investigate distribution and federation for horizontal scalability in future work. This includes a thorough performance and storage space comparison using different—among others distributed—key/value stores and different scenarios in order to measure the impact of different data distribution mechanisms on the analysis performance. Since the impact of distribution and cluster deployments not just heavily depend on the underlying distribution mechanism—*i.e.*, key/value store—but also on the use case and data to be analyzed, various different analytic algorithms need to be compared. This is why we focused in this article on single-node deployments and in-memory analysis. In addition, we are working on our own append-only key/value store based on direct memory mapping techniques. Other topics we are currently investigating are sharding and federation. These are closely linked to distribution and cluster deployments. As mentioned before, we currently do not support the latter and support data distribution only via distributed key/value stores. We also plan to investigate how default graph querying languages, *e.g.*, Gremlin [**?** ]/TinkerPop, can be integrated into GreyCat. As discussed in details at the end of Section **??**, we plan to investigate if and how modifications and insertions in the past and before divergent points can be better supported by the framework, *e.g.*, to track dependent data and indicate if something has to be potentially updated. For now, we leave it to the application developers to handle potential side-effects, like updating dependent data.

## 7. Related Work

Over the years, data management systems have pushed the limits of data analytics for huge amounts of data fur-

ther and further. In the 1990's Codd *et al.* [**?** ] presented a category of database processing, called *online analytical processing* (OLAP). It addressed the lack of traditional database processing to consolidate, view, and analyze data according to multiple dimensions. In a more recent work about best practices for big data analytics, Cohen *et al.* [**?** ] present what they call "MAD Skills". They highlight the practice of *magnetic, agile, and deep* (MAD) data analysis as a departure from traditional data warehousing and business intelligence. For example, the Hadoop stack [**?** ], Heron [**?** ], and Spark [**?** ] drove the development of new and powerful data analytics. Despite this, today data analytics is still predominantly *descriptive*. However, as Haas *et al.* [**?** ] suggested, what enterprises really need is *prescriptive* analytics to identify optimal business decisions. They argue that this requires what-if analysis. In accordance with this idea, we propose a graph data model, which is able to efficiently evolve in time and in many worlds to simulate different decisions.

Recently, much work focuses on large-scale graph representation, storage, and processing for analytics. Well-known examples are Pregel [**?** ], Giraph, Neo4J [**?** ], GraPS [**?** ], SNAPLE [**?** ], and GraphLab [**?** ]. While many of them require the graph to be completely in-memory while processing [**?** ], others, like Roy *et al.* [**?** ] or Shao *et al.* [**?** ], suggest to process graphs from secondary storage. Similarly, we allow to store graph data on secondary storage, since even with big clusters at a certain point the limit of in-memory only solutions is reached. Given that our MWG is built to evolve extensively in time and many worlds, the need for secondary storage is even more underlined, since many different versions of nodes can coexist, making graphs even bigger. While most of this work uses a rather standard graph data model and focuses on graph computation and processing, the focus of our work is to support large-scale what-if analysis.

The need to represent and store the temporal dimension of data has been comprehensively discussed in the database community in the 80s and 90s. For example, Clifford *et al.* [**?** ] and also Ariav [**?** ] provide a formal semantics for historical databases. In a similar direction goes the work of Ariav [**?** ]. They all suggest, in some way or another, to directly integrate temporal structures in the data model itself, rather than at the application level. In [**?** ], Segev and Shoshani discuss the semantics of temporal data and corresponding operators independently from a specific data model. Salzberg *et al.* [**?** ] discuss different temporal indexing techniques. Although most of this work is relatively old, such temporal databases are not very widespread. Google [**?** ] embeds versioning at the core of its BigTable implementation by enabling each cell in a table to contain multiple versions of the same data.

With the emergence of cyber-physical systems, temporal aspects of data evolved again in form of time series management. As mentioned before, InfluxDB is one of the newer time series databases, which received much attention lately. They position themselves as an IoT and sensor database for real-time analytics. While it provides many interesting features, like a SQL-like query language, their data model is essentially flat and does not support complex relationships between data—*i.e.,* it provides very little support for richer data models—like graphs. The same counts for Atlas [**?** ], which was developed by Netflix to manage dimensional time series data for near real-time operational insights, and OpenTSDB [**?** ]. RRDtool [**?** ] is another data logging and graphing system for time series. All of this work has in common that it provides high performance storage and management specialized for time series data. However, these solutions provide very little support for richer data models, like graphs.

Lately, an increasing amount of work deals with the need of temporal aspects of graph data. Finally, an increasing interest in time-evolving graphs appeared. For example, Huanhuan *et al.* [**?** ] discuss the problem of finding the shortest path in a temporal graph. Bahmani *et al.* [**?** ] show how to compute PageRank on evolving graphs. Khurana and Deshpande [**?** ] present with *Historical Graph Store* (HGS) a system for managing and analyzing large historical traces of graphs. T-SPARQL [**?** ] is a temporal extension for the SPARQL [**?** ] RDF query language. HGS consists of two major components, the *Temporal Graph Index* (TGI) and a *Temporal Graph Analysis Framework* (TAF). Their proposed TGI stores the complete history of a graph in form of partitioned deltas and rebuilds the graph from these deltas while querying graph data. With TAF, they provide a library to specify a wide range of temporal graph analysis tasks. With the index time tree, we pursue similar goals as they do with their indexing strategies, however we do not save deltas for the whole graph, but instead treat all versions of nodes similarly. This simplifies the retrieval of historical data without the need to rebuild it from stored deltas. GraphTau [**?** ], Kineograph [**?** ], and Chronos [**?** ] also extend graph processing to time-evolving graphs. While Neo4J itself does not provide any support for temporal data, Cattuto *et al.* [**?** ] present a pattern on how to use Neo4J for analyzing time-varying social networks. They suggest to associate nodes and edges with time intervals (frames) and to represent both logical graph nodes and edges as Neo4J nodes. This lack of a native support leads to a rather complex data and query model. Chronos [**?** ] is another interesting storage and execution engine, however it is designed specifically for in-memory iterative graph computation. These approaches have in common that they represent time-evolving graphs, in some form or another, as a sequence of snapshots and use a rather standard graph data model. In addition, most of these approaches requires to keep a full graph snapshot in memory and they have some limits when data is changing at a very high pace. Our approach suggests an efficient temporal graph data model for large-scale what-if analysis on rapidly changing data.

The idea of what-if analysis with hypothetical queries

has been discussed in database communities. Balmin *et al.* [**?** ] proposed an approach for hypothetical queries in OLAP environments. They enable data analysts to formulate possible business scenarios, which then can be consequently explored by querying. Unlike other approaches, they use a "no-actual-update" policy—*i.e.,* the possible business scenarios are never actually materialized but only kept in main memory. In a similar approach, Griffin and Hull [**?** ] focus on queries with form *Q when {U}* where Q is a relational algebra query. This paper develops a framework for evaluating hypothetical queries using a "lazy" approach, or using a hybrid of eager and lazy approaches. They present an equational theory and family of rewriting rules that is analogous and compatible with the equational theory and rewriting rules used for optimizing relational algebra queries. In [**?** ] and [**?** ], Arenas *et al.* developed an approach for hypothetical temporal queries of form *"Has it always been the case that the database has satisfied a given condition C"*. Despite there is no explicit time in these queries, they call them "temporal" due to a similarity with dynamic integrity constraints. Although these approaches have a similar goal than our approach, they differ in many major points. First, they mainly aim at data analysts which perform selective queries on a modest number of possible business scenarios to investigate impacts of decisions. In contrary, we aim at intelligent systems and complex data analytics, which need to explore a very large number of parallel actions (*e.g.*, as for genetic algorithms or the presented smart grid case study), which can be highly nested. Moreover, these systems usually face significantly higher demands regarding performance. In addition, most of these approaches do not support (or only in a limited manner) the co-evolution of worlds, which is an essential feature of the MWG. To the best of our knowledge there is no approach allowing graphs to evolve in time and in many worlds for efficient what-if analysis. Another major difference is that the MWG is a fully temporal graph supporting both the exploration of different hypothetical worlds and the temporal evolution of data. Our proposed MWG can be used in arbitrary analytics and is independent of the concrete underlying database whereas most of the work on hypothetical queries has been done on relational databases.

## 8. Conclusion

We proposed a novel graph data model, called *Many-World Graph* (MWG), which allows to efficiently explore a large number of independent actions—both in time and many worlds—on a massive amount of data. We validated that GreyCat, our MWG implementation, follows the theoretical time complexity of $O(log(n))$ for the temporal resolution and $O(m)$ for the world resolution, where $m$ is the maximum number of nested worlds.

Our experimental evaluation showed that even when used as a base graph—without time and many-worlds—GreyCat outperforms a state-of-the-art graph database, Neo4J, for both mass and single inserts. A direct comparison with a state-of-the-art time series database, InfluxDB, showed that although the MWG is not just a simple time series, but a fully temporal graph, the temporal resolution performance of MWG is comparable or in some cases even faster than time series databases—at least in single node deployments. The experimental validation showed that the MWG is very well suited for what-if analysis. Regarding the support for prescriptive analytics, we showed that GreyCat is able to handle efficiently hundreds of millions of nodes, timepoints, and hundreds of thousands of independent worlds.

Beyond the specific case of smart grids, we believe that GreyCat can find applications in a large diversity of application domains, including social networks [**?** ], smart cities, and biology [**?** ].

Aside of potential applications of this approach, our perspectives also include the extension of GreyCat to consider different *laws of evolution* for the stored graphs, thus going beyond the application of machine learning [**?** ]. We also look at the integration of GreyCat with existing graph processing systems, like Giraph [**?** ] and the support of standard graph querying languages, like TinkerPop.[5] Beyond the what-if analysis, the coverage of alternative prescriptive analytics based on GreyCat is a direction we are aiming for. Finally, we are working on data sharding and cluster deployments of GreyCat.


## Acknowledgments

The research leading to this publication is supported by the Luxembourg National Research Fund (grant 12490856) under the Industrial Fellowship program and Creos Luxembourg S.A. under the SnT-Creos partnership program. This work has been achieved within the framework of the CPER Data project. CPER Data is co-financed by the European Union with the financial support of European Regional Development Fund (ERDF), French State and the French Region of Hauts-de-France.


## Availability

The source code of GreyCat is available under:

`https://github.com/datathings/greycat`

## Bibliography

[5] http://tinkerpop.apache.org