

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

Thomas Hartmann^a, Francois Fouquet^b, Assaad Moawad^b, Romain Rouvoy^{c,d}, Yves Le Traon^a

^aUniversity of Luxembourg, Luxembourg

^bDataThings, Luxembourg

^cUniversity of Lille / Inria, France

^dInstitut 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 [1]. In this domain, *predictive analytics* techniques, like sliding window analytics [2], typically extract temporal models from current and past historical facts in order to make predictions about the future [3]. 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* [4]. 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 [5–7]. However, most of graph representations only reflect a snapshot at a given time, while reflected data keeps changing as the systems evolve.

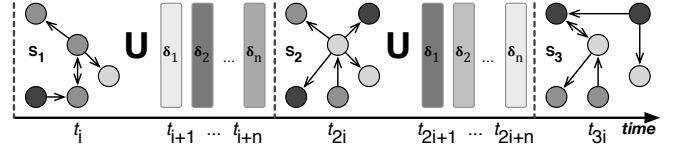


Figure 1: Snapshots (S_i) and deltas (δ_n) of a time-evolving graph

Understanding temporal characteristics of time-evolving graphs therefore attracts increasing attention from research communities [8]—*e.g.*, in the domains of social networks, smart mobility, or smart grids [9]. Yet, state-of-the-art approaches fail to provide a scalable solution to effectively support time in graphs. In particular, existing approaches represent time-evolving graphs as sequences of full-graph snapshots [10], or they use a combination of snapshots and deltas [11], which requires to reconstruct a graph for a given time, as depicted in Figure 1.

However, full-graph snapshots tend to be expensive in terms of memory requirements, both on disk and in-memory. This overhead becomes even worse when data from several snapshots need to be correlated, which is the case for most of advanced analytics [9, 10, 12]. 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 el-

Email addresses: thomas.hartmann@uni.lu (Thomas Hartmann), francois.fouquet@uni.lu (Francois Fouquet), assaad.moawad@datathings.lu (Assaad Moawad), romain.rouvoy@univ-lille.fr (Romain Rouvoy), yves.letraon@uni.lu (Yves Le Traon)

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) [9]. An alternative to snapshotting consists in combining graphs with time series databases [13], 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* [14], 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 [7, 15, 16], which are commonly used to organize massive amounts of unstructured data [7, 17]. Beyond the inclusion of temporal aspects within graphs [10, 18–20], MWG proposes an efficient exploration of many independently evolving worlds to support the requirements of what-if analysis. The main contribution of this paper 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 GitHub¹. MWG and GREYCAT refer to Everett’s [14] many-world interpretation, illustrated by Schrödinger’s cat [21].

The remainder of this article is organized as follows. Section 2 introduces a real smart grid case study, as a motivation of this research. Sections 3 and 4 introduce the main concepts of MWG and their implementation. We thoroughly evaluate GREYCAT in Section 5. The related work is discussed in Section 6 before concluding in Section 7.

2. Motivating Case Study

Intelligent load management is a critical challenge for electricity utility companies [22, 23]. 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’ homes, and regularly report to utility companies, (*e.g.*, every 15 minutes [24]). 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 [24]. 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 [25, 26], this topic is gaining popularity again with the advent of *time series* databases for the IoT, like InfluxDB [27]. 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. On the other side, graph-based storage solutions (*e.g.*, Neo4J [28]), as well as graph processing frameworks, despite some attempts to represent time dependent graphs [10, 11, 19, 29], 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 paper about temporal data, we refer to this fully

¹<https://github.com/datathings/greycat>

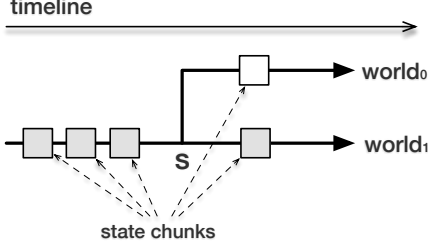


Figure 2: States of a node in two worlds

temporal aspect and not about flat time series’. Most importantly, none of these solutions supports the large-scale exploration of different alternatives.

These limitations motivates our work on MWG in order to enable large-scale what-if analysis for prescriptive analytics [4]. Haas *et al.* [4], also supports the need for large-scale what-if analysis in many other domains, *e.g.*, weather prediction. Beyond the specificities of smart grids, we introduce the concept of *Many-World Graphs* as a scalable model to explore alternative scenarios in the context of what-if analysis.

3. Introducing Many-World Graphs

3.1. Key Concepts

This paper introduces the notion of *Many-World Graphs* (MWG), which are directed, attributed hypergraphs which structure and properties 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 2. 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** ($t_{n,w} = [c_0, \dots, c_n]$) 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

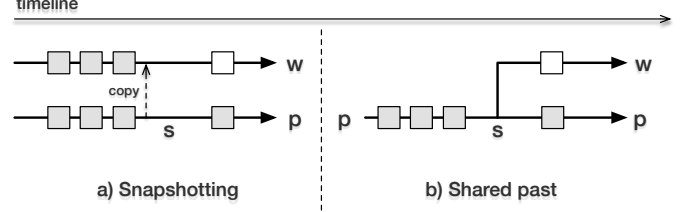


Figure 3: Types of many-worlds.

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 4.1.

While prescriptive analytics tends to explore new worlds along time, two techniques can be employed when forking worlds: *snapshotting* and *shared past* (cf. Figure 3).

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: we propose 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 does 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.

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 [17]), 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 mapping leverages the integration of MWG within an application layer.

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: $n \rightarrow m$ and $n \rightarrow p$. In the next sections, we overload the function $read$ incrementally to integrate step by step the time and many-world semantic.

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: $\forall t_i, t_j \in T : t_i \leq t_j \vee t_j \leq t_i$.

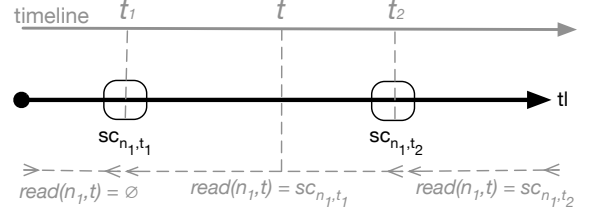


Figure 4: TG node timeline

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. 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: $tl_n := tl_n \cup \{c_{n,t}\}$.

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 :

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

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 4.

Since state chunks with this semantics have temporal validities, relationships between nodes also have temporal validities. This leads to *temporal relationships* between TG nodes and forms a natural extension of relationships in

the time dimension. 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 3.4, 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, or the worlds that are created from different parents, cannot be compared (in terms of order) to each other. 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. 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 3), we define the global timeline of a world per node as

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

The global timeline of a world is therefore the recursive aggregation of the local timeline of the world w with the

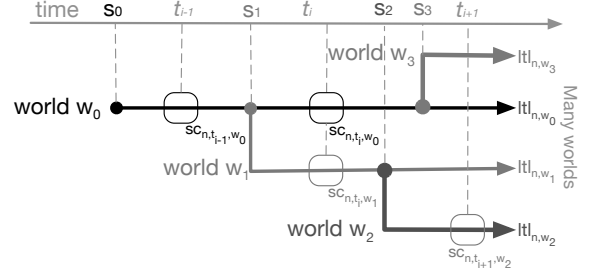


Figure 5: Many worlds example

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:

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

The function **insert** always operates on the local timeline $l_{n,w}$ 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 $l_{n,w}$, as defined in Section 3.4. 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 5 shows an example of a 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 $w_0 < w_3$. But no 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 l_{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 l_{n,w_2} if $t \geq s_2$, from l_{n,w_1} if $s_1 \leq t < s_2$, from l_{n,w_0} if $s_0 \leq t < s_1$, and \emptyset if $t < s_0$.

It is important to note that this semantics goes beyond copy-on-write strategies [30] as a world is never copied, even if data is modified. Instead, only modified nodes are

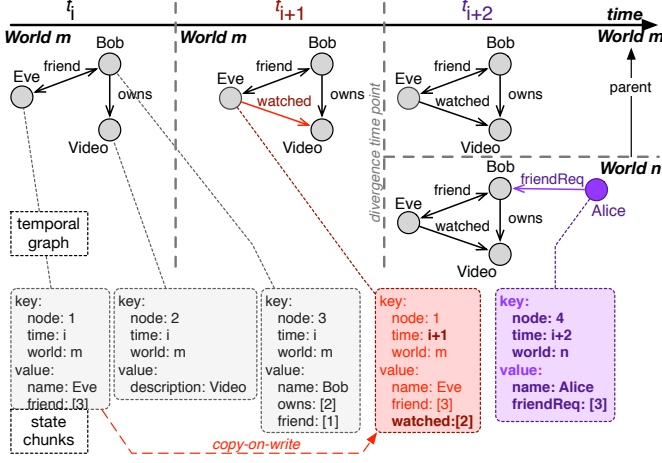


Figure 6: Mapping of nodes to storable state chunks

copied and transparently loaded. In fact, this is similar to the concept how we organise and represent the temporal data aspect of the graph.

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 3, we map the conceptual nodes (and relationships) of a MWG to *state chunks*. State chunks are the 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 6 depicts, in form of a concrete example, how nodes are mapped to state chunks in accordance with the semantic definitions of Section 3.2.

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 valid. Then, at time t_{i+2} , 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. Only Alice changes so that GREYCAT only creates one state chunk for Alice from time t_{i+2} and world n . Here, we see the benefit of the MWG and its semantics: while we are able to represent complex graph topologies, which evolve in time and in 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. Whenever the MWG is traversed, the correct state chunks are retrieved with the right time and world. The resolution algorithm behind this is presented in Section 4.2.3.

State chunks are the storage units of GREYCAT. They are stored on disk and loaded into main memory while the MWG is traversed or when nodes are explicitly retrieved. Loading state chunks can be qualified as lazy, because only attributes and sets of identifiers are loaded. This theoretically allows to process unbounded MWGs. For persistent storage of state chunks, we rely on 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 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 3.3, 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

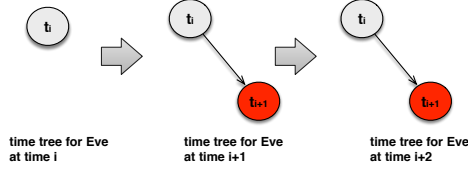


Figure 7: Example ITTs for node *Eve* of Figure 6

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 use red-black trees [31] 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 and 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 [32]. Every conceptual node of a MWG can evolve independently in time. For scalability reasons, we decided to use one red-black tree, further called *index time tree* (ITT), per conceptual node to represent its timeline. Figure 7 depicts how the ITT looks like and evolves for the node *Eve* introduced in Figure 6.

As it can be seen, at time t_i , one conceptual version of node *Eve* exists and therefore the ITT has only one entry. 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 5, the divergent point is therefore not enough to identify the parent relationship. 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

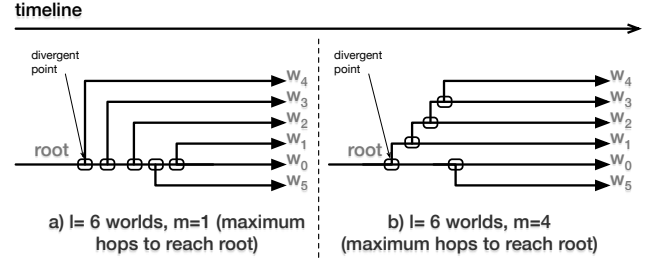


Figure 8: Example of different configurations of the same number of worlds l , but with a different m

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 3.5).

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 8 reports on 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 8-a). The worst case complexity is $O(m) \leq O(l)$, like for the **stair-shaped** case shown in Figure 8-b, where we might 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 than regular state chunks.

4.2.3. Chunk Resolution Algorithm

To illustrate the resolution algorithm of MWG, let us consider the example of Figure 6, 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 1.

Algorithm 1 State chunk resolution

```

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

```

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.

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 selectable 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 [33].

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 4.5 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 various computers. However, the current implementation of GREYCAT does not define a specific sharding mechanism to distribute homogeneously data stream on 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 basically only relies on the implementation of a **get** and **put** methods. For example, if performance is the most critical requirement, but fault tolerance, availability, distribution, and replication are less important, GREYCAT provides drivers for ROCKSDB [34] and LEVELDB [35]. On the other hand, if availability, scalability, and replication are more critical, GREYCAT provides also implementations for drivers for CASSANDRA [36] and HBASE [37].

To interconnect heterogeneous platforms, such as Android mobile devices, browser environments, and Java-powered servers, we use buses 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 use an optimistic approach using *Conflict-Free replicated Data Types* (CRDT). 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 studying 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. 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 2 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 3 depicts how GREYCAT's API can be used to do this, by extending the example of Listing 2.

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, from 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 4. 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 5.

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) {
                    //...
                }
            }
        });
    }
});
```

Since most operations in GREYCAT are non-blocking and therefore asynchronous, deep navigations inside a graph can lead to many nested callbacks (*cf.* Listing 5). Therefore, GREYCAT offers what we call 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 one to specify if a given number of `Tasks` can be executed in 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 5 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 6.

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

These examples are showing only the main concepts of

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
});
```

GREYCAT’s API’s to work with MWG. More advanced queries and navigation methods are available. The complete API can be found online on GitHub².

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 5.2), to comparing it against the closest solutions in the state of the art (cf. Sections 5.3–5.4), and to stressing it against micro-benchmarks (cf. Sections 5.5–5.7).

5.1. Experimental Setup

For all these experiments, we report on the throughputs 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. All the experiments have been executed on the *high performance computer* (HPC) of the University of Luxembourg (Gaia cluster) [38]. 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 rational behind this is that we want to evaluate our GREYCAT implementation 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. All our experiments are available on GitHub [39].

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

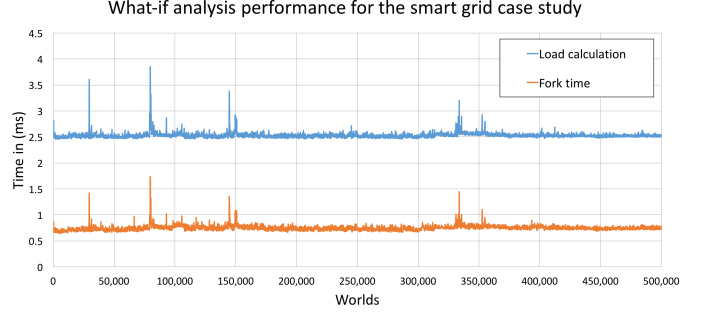


Figure 9: Performance of load calculation in a what-if scenario for the smart grid case study

are out of scope of this paper. We focus on the evaluation of *create*, *read*, and *update* operations since the contributions of this paper 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 paper.

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 2. 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 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 [40]. 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 [41]. 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 [41] 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 underlying physical properties of the grid, which we neglect in the following experiment.

Figure 9 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

²<https://github.com/datathings/greycat>

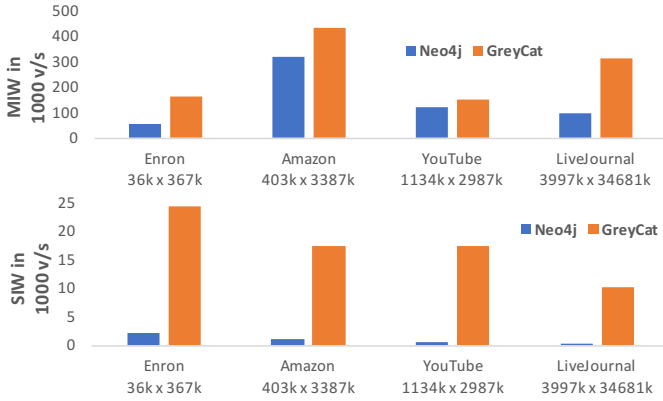


Figure 10: MIW and SIW benchmark throughput in 1,000 values/second. Bigger numbers mean better results. Numbers beneath dataset names mean number of nodes and edges, *i.e.* nodes \times edges.

the latency (in *ms*) of the load calculations and world creation (fork time) per world. As depicted in Figure 9, 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 [42] provided by Beis *et al.* [43]. 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* [44]. This dataset collection contains sets from “social network and ground-truth communities” [45], 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 [28], which was the best performing base graph in [43]. Figure 10 reports on the results of MIW and SIW, achieved by GREYCAT and Neo4J (both in-memory and not persisting data), along the different datasets. For all benchmarks, GREYCAT outperforms Neo4J by factors ranging from 1.3x to 20x.

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

(n) in millions	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

5.4. Temporal Graph Benchmarks

This experiment aims to evaluate the complexity of the ITT (cf. Section 4.2.1). 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 1 reports on 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 [46]. 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 is to create 250,000 nodes where they insert 1,000 values in each, 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.

The main difference with the experiment above 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 [34] as our key/value backend. Despite the fact that our MWG is not limited to flat time series, but a full temporal graph, we are able to outperform InfluxDB by completing the MacBook test in 388 seconds compared

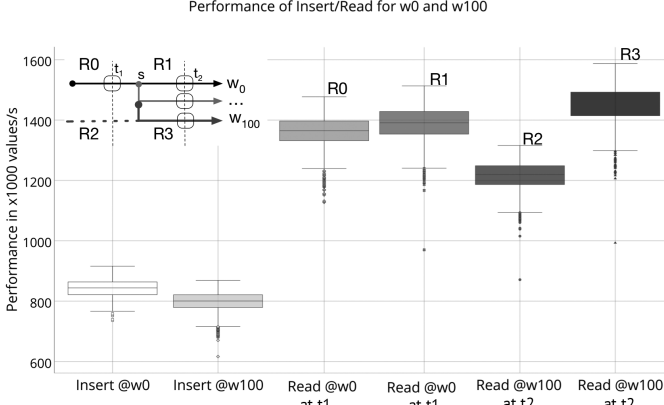


Figure 11: *Insert* and *read* performance before and after the divergence timepoint s

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 1). This is due to the increased complexity of balancing the ITT of one node. The experiment therefore assess 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.

5.5. Node-scale Benchmarks

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 11 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 4.2.2.

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

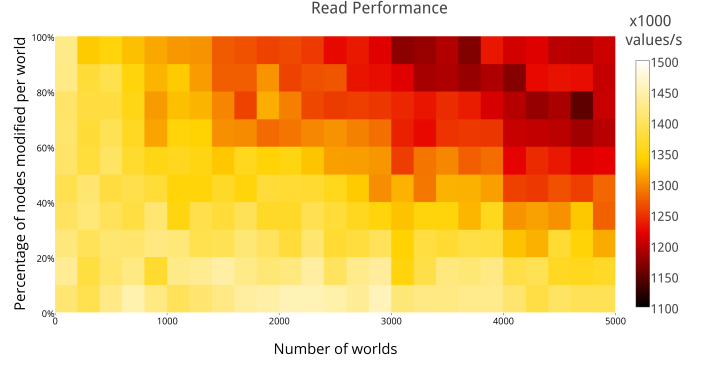


Figure 12: Read performance, over several worlds and several percentage of modified nodes

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 8-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 12 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 4.2.2. The performance drop is linear in $O(m)$ and 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

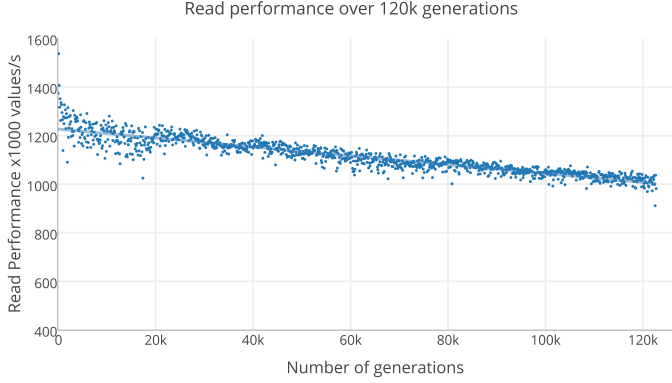


Figure 13: Average *read* performance over 120,000 generations with 3% mutations

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% [47]). 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 [47]. 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 13 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 4.2.2, 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. Related Work

Over the years, data management systems have pushed the limits of data analytics for huge amounts of data further and further. In the 1990’s Codd *et al.* [48] 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.* [49] 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 [50], Heron [51], and Spark [52] drove the development of new and powerful data analytics. Despite this, today data analytics is still predominantly *descriptive*. However, as Haas *et al.* [4] 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 [7], Giraph, Neo4j [28], GraPS [53], SNAPLE [54], and GraphLab [15]. While many of them require the graph to be completely in-memory while processing [55], others, like Roy *et al.* [56] or Shao *et al.*, [57], 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 use a rather standard graph data model and focus on graph computation and processing, the focus of our work is rather 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.*, [25] and also Ariav [58] provide a formal semantic for historical databases. In a similar direction goes the work of Ariav [58]. 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 [26], Segev and Shoshani discuss the semantics of temporal data and corresponding operators independently from a specific data model. Salzberg *et al.*, [59] discuss different temporal indexing techniques. Although most of this work is relatively old, such temporal databases are not very widespread. Google [60] 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, tempo-

ral 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 [61], which was developed by Netflix to manage dimensional time series data for near real-time operational insights, and OpenTSDB [62]. RRDtool [63] is another data logging and graphing system for time series. The same counts for Atlas [61], OpenTSDB [62], and RRDtool [63]. 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.*, [64] discuss the problem of finding the shortest path in a temporal graph. Bahmani *et al.*, [18] show how to compute PageRank on evolving graphs. Khurana and Deshpande [20] present with *Historical Graph Store* (HGS) a system for managing and analyzing large historical traces of graphs. T-SPARQL [65] is a temporal extension for the SPARQL [66] 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 similar. This simplifies the retrieval of historical data without the need to rebuild it from stored deltas. GraphTau [10], Kineograph [29], and Chronos [11] also extend graph processing to time-evolving graphs. While Neo4j itself does not provide any support for temporal data, Cattuto *et al.*, [19] 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 complicated data and query model. Chronos [11] 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.*, [67] 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 [68] focus on queries with form $Q \text{ when } \{U\}$ 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 [69] and [70], 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.

7. 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. 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 [71], smart cities, and biology [4].

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 [24]. We also look at the integration of GREYCAT with existing graph processing systems, like Giraph [7]. Finally, beyond the what-if analysis, the coverage of alternative prescriptive analytics based on GREYCAT is a direction we are aiming for.

Acknowledgments

The research leading to this publication is supported by the National Research Fund Luxembourg (grant 6816126) and Creos Luxembourg S.A. under the SnT-Creos partnership program.

Availability

The source code of GREYCAT is available under:

<https://github.com/datathings/greycat>

Bibliography

References

- [1] P. Kathiravelu, L. Sharifi, L. Veiga, Cassowary: Middleware Platform for Context-Aware Smart Buildings with Software-Defined Sensor Networks, in: Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT, M4IoT 2015, ACM, New York, NY, USA, 2015, pp. 1–6. doi:10.1145/2836127.2836132. URL <http://doi.acm.org.proxy.bnl.lu/10.1145/2836127.2836132>
- [2] P. Bhatotia, U. A. Acar, F. P. Junqueira, R. Rodrigues, Slider: incremental sliding window analytics, in: Proceedings of the 15th International Middleware Conference, ACM, 2014, pp. 61–72.
- [3] S. Rusitschka, C. Doblander, C. Goebel, H.-A. Jacobsen, Adaptive Middleware for Real-time Prescriptive Analytics in Large Scale Power Systems, in: Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference, Middleware Industry '13, ACM, New York, NY, USA, 2013, pp. 5:1–5:6. doi:10.1145/2541596.2541601. URL <http://doi.acm.org.proxy.bnl.lu/10.1145/2541596.2541601>
- [4] P. J. Haas, P. P. Maglio, P. G. Selinger, W. C. Tan, Data is Dead... Without What-If Models, PVLDB 4 (12) (2011) 1486–1489. URL <http://www.vldb.org/pvldb/vol14/p1486-haas.pdf>
- [5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 17–30. URL <http://dl.acm.org/citation.cfm?id=2387880.2387883>
- [6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727. doi:10.14778/2212351.2212354. URL <https://doi.org/10.14778/2212351.2212354>
- [7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A System for Large-scale Graph Processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146.
- [8] J. Leskovec, J. Kleinberg, C. Faloutsos, Graphs over time: Densification laws, shrinking diameters and possible explanations, in: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, ACM, New York, NY, USA, 2005, pp. 177–187. doi:10.1145/1081870.1081893. URL <http://doi.acm.org/10.1145/1081870.1081893>
- [9] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, Y. Le Traon, Reasoning at runtime using time-distorted contexts: Amodels@run.time based approach, in: (SEKE'14) 26th International Conference on Software Engineering and Knowledge Engineering, 2014.
- [10] A. P. Iyer, L. E. Li, T. Das, I. Stoica, Time-evolving Graph Processing at Scale, in: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, GRADES '16, ACM, New York, NY, USA, 2016, pp. 5:1–5:6. doi:10.1145/2960414.2960419. URL <http://doi.acm.org/10.1145/2960414.2960419>
- [11] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, E. Chen, Chronos: A Graph Engine for Temporal Graph Analysis, in: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, ACM, New York, NY, USA, 2014, pp. 1:1–1:14.
- [12] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, W. Chen, Immortalgraph: A system for storage and analysis of temporal graphs, Trans. Storage 11 (3) (2015) 14:1–14:34. doi:10.1145/2700302. URL <http://doi.acm.org/10.1145/2700302>
- [13] J. Lin, E. Keogh, S. Lonardi, B. Chiu, A symbolic representation of time series, with implications for streaming algorithms, in: Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '03, ACM, New York, NY, USA, 2003, pp. 2–11. doi:10.1145/882082.882086. URL <http://doi.acm.org/10.1145/882082.882086>
- [14] H. Everett, "relative state" formulation of quantum mechanics, Rev. Mod. Phys. 29 (1957) 454–462. doi:10.1103/RevModPhys.29.454. URL <http://link.aps.org/doi/10.1103/RevModPhys.29.454>
- [15] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, J. Hellerstein, Graphlab: A new framework for parallel machine learning, arXiv preprint arXiv:1408.2041.
- [16] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, Graphx: A

- resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, ACM, New York, NY, USA, 2013, pp. 2:1–2:6.
- [17] J. J. Miller, Graph Database Applications and Concepts with Neo4j, in: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd–24th, 2013.
 - [18] B. Bahmani, R. Kumar, M. Mahdian, E. Upfal, PageRank on an Evolving Graph, in: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, ACM, New York, NY, USA, 2012, pp. 24–32.
 - [19] C. Cattuto, M. Quaggiotto, A. Panisson, A. Averbuch, Time-varying Social Networks in a Graph Database: A Neo4J Use Case, in: First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, ACM, New York, NY, USA, 2013, pp. 11:1–11:6.
 - [20] U. Khurana, A. Deshpande, Storing and analyzing historical graph data at scale, arXiv preprint arXiv:1509.08960.
 - [21] E. Schrödinger, The present status of quantum mechanics, *Die Naturwissenschaften* 23 (48) (1935) 1–26.
 - [22] S. M. Amin, B. F. Wollenberg, Toward a smart grid: power delivery for the 21st century, *IEEE Power and Energy Magazine* 3 (5) (2005) 34–41. doi:10.1109/MPAE.2005.1507024.
 - [23] H. Farhangi, The path of the smart grid, *IEEE Power and Energy Magazine* 8 (1) (2010) 18–28. doi:10.1109/MPE.2009.934876.
 - [24] T. Hartmann, A. Moawad, F. Fouquet, Y. Reckinger, T. Mouelhi, J. Klein, Y. L. Traon, Suspicious electric consumption detection based on multi-profiling using live machine learning, in: 2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, FL, USA, November 2–5, 2015, IEEE, 2015, pp. 891–896. doi:10.1109/SmartGridComm.2015.7436414. URL <http://dx.doi.org/10.1109/SmartGridComm.2015.7436414>
 - [25] J. Clifford, D. S. Warren, Formal semantics for time in databases, *ACM Transactions on Database Systems (TODS)* 8 (2) (1983) 214–254.
 - [26] A. Segev, A. Shoshani, Logical modeling of temporal data, in: *ACM Sigmod Record*, Vol. 16, ACM, 1987, pp. 454–466.
 - [27] influxdb - TIME-SERIES DATA STORAGE DOCS. [Online], Available: <https://influxdata.com/time-series-platform/influxdb>.
 - [28] neo4j. [Online], Available: <http://neo4j.com>.
 - [29] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen, Kineograph: Taking the Pulse of a Fast-changing and Connected World, in: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, ACM, New York, NY, USA, 2012, pp. 85–98. doi:10.1145/2168836.2168846. URL <http://doi.acm.org/10.1145/2168836.2168846>
 - [30] F. J. T. Fábrega, F. Javier, J. D. Guttman, Copy on Write (1995).
 - [31] L. J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, in: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, SFCS '78, IEEE Computer Society, Washington, DC, USA, 1978, pp. 8–21. doi:10.1109/SFCS.1978.3. URL <http://dx.doi.org/10.1109/SFCS.1978.3>
 - [32] L. Veiga, R. Bruno, P. Ferreira, Asynchronous Complete Garbage Collection for Graph Data Stores, in: Proceedings of the 16th Annual Middleware Conference, Middleware '15, ACM, New York, NY, USA, 2015, pp. 112–124. doi:10.1145/2814576.2814813. URL <http://doi.acm.org.proxy.bnl.lu/10.1145/2814576.2814813>
 - [33] Z. Chen, Y. Zhou, K. Li, Eviction-based cache placement for storage caches., in: *USENIX Annual Technical Conference, General Track, USENIX*, 2003, pp. 269–281.
 - [34] RocksDB. [Online], Available: <http://rocksdb.org>.
 - [35] Leveldb, <http://leveldb.org>, accessed: 2015-03-02.
 - [36] Cassandra. [Online], Available: <http://cassandra.apache.org/>.
 - [37] L. George, HBase: the definitive guide, " O'Reilly Media, Inc.", 2011.
 - [38] HPCUni.lu. [Online], Available: <https://hpc.uni.lu>.
 - [39] GreyCat Experiments. [Online], Available: <http://datathings.com/experiments/greycat-mwg-benchmark/>.
 - [40] London DataStore. [Online], Available: <http://data.london.gov.uk/dataset/smartmeter-energy-use-data-in-london-households>.
 - [41] T. Hartmann, F. Fouquet, J. Klein, Y. L. Traon, A. Pelov, L. Toutain, T. Ropitault, Generating realistic Smart Grid communication topologies based on real-data, in: 2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3–6, 2014, IEEE, 2014, pp. 428–433. doi:10.1109/SmartGridComm.2014.7007684. URL <http://dx.doi.org/10.1109/SmartGridComm.2014.7007684>
 - [42] graphdb-benchmarks. [Online], Available: <https://github.com/socialsensor/graphdb-benchmarks>.
 - [43] S. Beis, S. Papadopoulos, Y. Kompatsiaris, Benchmarking graph databases on the problem of community detection, in: *New Trends in Database and Information Systems II*, Springer, 2015, pp. 3–14.
 - [44] Stanford Large Network Dataset Collection. [Online], Available: <http://snap.stanford.edu/data>.
 - [45] J. Yang, J. Leskovec, Defining and Evaluating Network Communities based on Ground-truth, CoRR abs/1205.6233. URL <http://arxiv.org/abs/1205.6233>
 - [46] influxDB. [Online], Available: <http://tinyurl.com/influxblog>.
 - [47] M. Srinivas, L. M. Patnaik, Genetic algorithms: A survey, *Computer* 27 (6) (1994) 17–26.
 - [48] E. F. Codd, S. B. Codd, C. T. Salley, Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate, E. F. Codd and Associates (1993).
 - [49] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, C. Welton, Mad skills: new analysis practices for big data, *Proceedings VLDB Endowment* 2 (2) (2009) 1481–1492. URL <http://dl.acm.org/citation.cfm?id=1687553.1687576>
 - [50] Apache Hadoop. [Online], Available: <https://hadoop.apache.org>.
 - [51] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja, Twitter Heron: Stream Processing at Scale, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, ACM, New York, NY, USA, 2015, pp. 239–250.
 - [52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
 - [53] C. Cañas, E. Pacheco, B. Kemme, J. Kienzle, H.-A. Jacobsen, GraPS: A Graph Publish/Subscribe Middleware, in: Proceedings of the 16th Annual Middleware Conference, Middleware '15, ACM, New York, NY, USA, 2015, pp. 1–12. doi:10.1145/2814576.2814812. URL <http://doi.acm.org.proxy.bnl.lu/10.1145/2814576.2814812>
 - [54] A.-M. Kermarrec, F. Taiani, J. M. Tirado, Scaling Out Link Prediction with SNAPLE: 1 Billion Edges and Beyond, in: Proceedings of the 16th Annual Middleware Conference, Middleware '15, ACM, New York, NY, USA, 2015, pp. 247–258. doi:10.1145/2814576.2814810. URL <http://doi.acm.org.proxy.bnl.lu/10.1145/2814576.2814810>
 - [55] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: graph processing at Facebook-scale, *Proceedings of the VLDB Endowment* 8 (12) (2015) 1804–1815.

- [56] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, Chaos: Scale-out Graph Processing from Secondary Storage, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 410–424.
- [57] B. Shao, H. Wang, Y. Li, Trinity: A Distributed Graph Engine on a Memory Cloud, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, ACM, New York, NY, USA, 2013, pp. 505–516. doi:10.1145/2463676.2467799.
URL <http://doi.acm.org/10.1145/2463676.2467799>
- [58] G. Ariav, A temporally oriented data model, ACM Transactions on Database Systems (TODS) 11 (4) (1986) 499–527.
- [59] B. Salzberg, V. J. Tsotras, Comparison of Access Methods for Time-evolving Data, ACM Comput. Surv. 31 (2) (1999) 158–221.
- [60] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A Distributed Storage System for Structured Data, ACM Trans. Comput. Syst. 26 (2) (2008) 4:1–4:26.
- [61] Netflix Atlas. [Online], Available: <https://github.com/Netflix/atlas/wiki/Graph>.
- [62] OpenTSDB. [Online], Available: <http://opentsdb.net>.
- [63] RDDTool. [Online], Available: <http://oss.oetiker.ch/rddtool>.
- [64] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, Y. Xu, Path problems in temporal graphs, Proceedings of the VLDB Endowment 7 (9) (2014) 721–732.
- [65] F. Grandi, T-SPARQL: A TSQL2-like Temporal Query Language for RDF, in: ADBIS (Local Proceedings), Citeseer, 2010, pp. 21–30.
- [66] J. Pérez, M. Arenas, C. Gutierrez, Semantics and Complexity of SPARQL, ACM Trans. Database Syst. 34 (3) (2009) 16:1–16:45.
- [67] A. Balmin, T. Papadimitriou, Y. Papakonstantinou, Hypothetical queries in an olap environment, in: Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 220–231.
URL <http://dl.acm.org/citation.cfm?id=645926.672016>
- [68] T. Griffin, R. Hull, A framework for implementing hypothetical queries, in: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, SIGMOD '97, ACM, New York, NY, USA, 1997, pp. 231–242. doi:10.1145/253260.253304.
URL <http://doi.acm.org.proxy.bnl.lu/10.1145/253260.253304>
- [69] M. Arenas, L. Bertossi, Hypothetical temporal queries in databases, in: Proceedings “ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98): Innovative Application Programming and Query Interfaces, Citeseer, 1998.
- [70] M. Arenas, L. Bertossi, Hypothetical Temporal Reasoning in Databases, Journal of Intelligent Information Systems 19 (2) 231–259. doi:10.1023/A:1016524013831.
URL <http://dx.doi.org/10.1023/A:1016524013831>
- [71] A. Kermarrec, F. Taïani, J. M. Tirado, Scaling Out Link Prediction with SNAPLE: 1 Billion Edges and Beyond, in: Middleware, ACM, 2015, pp. 247–258.