Visualization of Electronic Health Record Sequences at Scale

Ambre Assor, Mickael Sereno, Jean-Daniel Fekete

Visualization of Electronic Health Record Sequences at Scale

A. Assor, Mickael Sereno and J.-D. Fekete

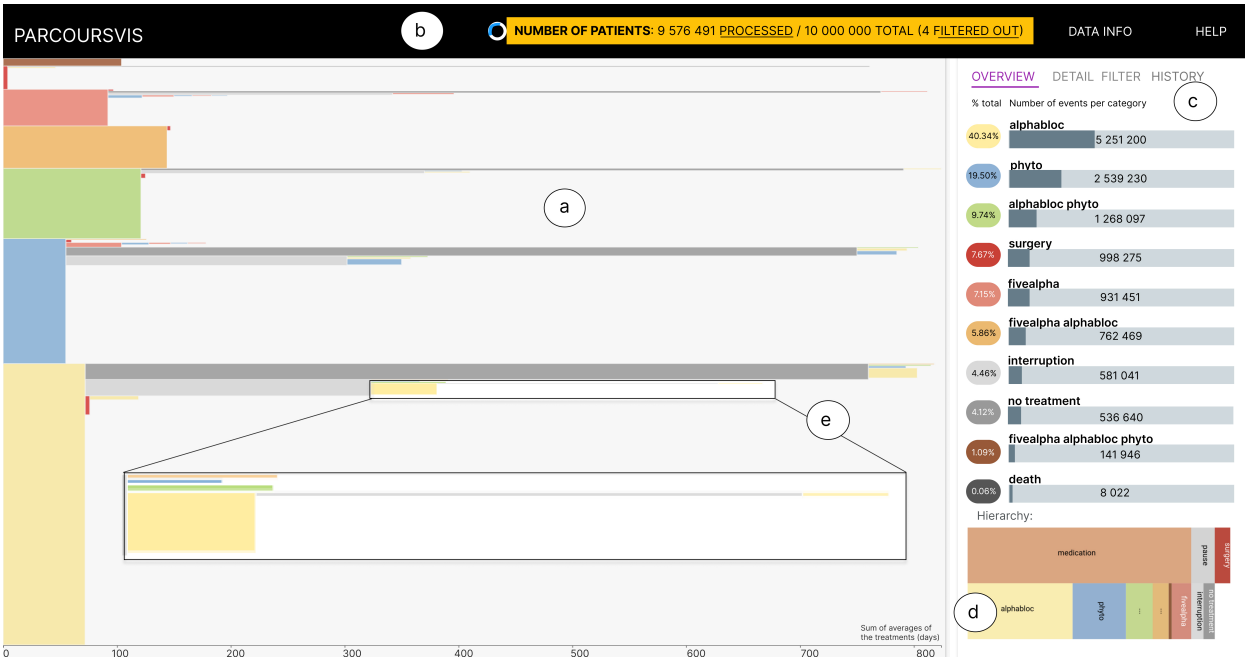

Fig. 1: The ParcoursVis system, designed to explore electronic health record sequences of patients at scale. (a) the main view, shows 10 million medical pathways for patients treated for non-cancerous prostate adenoma, visualized as an *Icicle tree*. (b) shows the number of filtered out and total patients. (c) shows, in the control panel, the legend and overall distribution of event types. (d) shows the hierarchy of event types used to summarize the tree. (e) zooms into a sub-tree, revealing more of the hierarchical structure. Each colored node represents a stage in a sequence of events starting from the root on the left, with time flowing to the right. For each node, the next event nodes are on the right, sorted from bottom to top by frequency. The node height is proportional to the frequency of the event and the width to the average duration. In this dataset, most patients started with the "alphabloc" treatment (bottom left), and kept taking it for life (few patients changed treatment since the node's right size is almost empty).

Abstract—We present ParcoursVis, a Progressive Visual Analytics tool designed to explore electronic health record sequences of patients at scale. Existing tools process and aggregate the whole dataset upfront before showing the visualization, taking a time proportional to the data size. Therefore, to remain interactive, existing tools are limited to data sizes that can be processed in under a few seconds to meet the latency constraints of human attention. To overcome this limitation and scale to larger sizes, ParcoursVis relies on a *progressive algorithm* that quickly shows an approximate initial result of the aggregation, visualized as an Icicle tree, and improves it iteratively, updating the visualization until the whole computation is done. With its architecture, ParcoursVis remains interactive while visualizing the sequences of tens of millions of patients, each described with thousands of events; three orders of magnitude more than similar systems. Managing large datasets allows for exploring rare medical conditions or unexpected patient pathways, contributing to improving treatments. We describe the algorithms we use and our evaluation concerning their scalability, convergence, and stability. We also report on a set of guidelines to support visualization designers in developing scalable progressive systems. ParcoursVis already allows practitioners to perform analyses on two real large medical datasets. Our prototype is open-source.

Index Terms—Scalability, Progressive Visual Analytics, Temporal Event Sequences, Electronic Health Records, Tree visualization, Icicle tree

1 Introduction

As datasets continue to increase in size and complexity, traditional visualization techniques struggle to maintain interactivity and responsiveness, thereby hindering effective exploratory data analysis. Addressing the scalability challenge necessitates progress in algorithmic implementations and the development of optimized data structures within visualization systems, but more importantly, a change in programming paradigm, using progressive visual analytics (PVA) to decouple data size and algorithm complexity from latency [13].

The scalability challenge is particularly acute in healthcare, where Electronic Health Records (EHRs) contain vast amounts of patient information. For instance, the French National Health Database [41] holds decades of health data on citizens, encompassing records of drug purchases, medical diagnoses, and hospital treatments. Such data have the potential to advance healthcare by enabling a better understanding of treatment practices and patient outcomes. As an

• *Ambre Assor, Mickael Sereno, and Jean-Daniel Fekete are with Université Paris-Saclay, CNRS, and Inria, E-mail: ambre.assor@inria.fr, mickael.sereno@inria.fr, jean-daniel.fekete@inria.fr*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

example, by reconstructing and visualizing temporal event sequences from drug purchase records, the examination of patients' treatment journeys over time is facilitated. This approach aligns with the goal of medical professionals to assess whether patients' treatments adhere to national guidelines and, if deviations occur, to understand the reasons. This is key to providing patients with the highest quality of care through evidence-based and safe recommendations. Yet, the volume of such datasets necessitates innovative visualization approaches capable of handling large-scale medical data while preserving analytical efficiency and usability. In this context, we developed ParcoursVis (Figure 1), a PVA tool created in collaboration with medical institutions to visualize patient care pathways derived from EHRs.

Existing work has laid a foundation for similar visualization tools aimed at healthcare analysis (e.g., [5, 28, 31]). For instance, EventFlow [29] aggregates event sequences into a *prefix tree* and visualizes them as an *Icicle tree*, as shown in Figure 1; a visualization that has proven effective in identifying optimal drug usage patterns. We, therefore, build on the EventFlow visualization technique for ParcoursVis, along with several improvements.

However, tools like EventFlow have generally been applied to relatively small datasets, sometimes encompassing fewer than 100 patients. According to our experiments, EventFlow is limited to about 20k patients. While such tools have offered valuable initial insights, they fall short in scalability, which has been reported as a main challenge for EHR visualization [49]. In practice, regional and national EHR databases present data at a far greater scale, often involving millions of patients. To our knowledge, no existing tool enables the management and interactive analysis of EHR datasets exceeding 100k event sequences [29, 49–51], which limits the scope of exploration. This constraint impedes the ability to gain a more comprehensive view of patient pathways, identify patterns and trends that may not be visible in smaller or sampled datasets, and discover rare patterns and outliers. With ParcoursVis, we allow visualizing temporal event sequences of more than 100M patients, potentially supporting the largest EHRs.

Large-scale interactive data processing poses significant challenges, particularly in terms of response times. To accommodate the need for a responsive system that supports health analysts in conducting explorations on large-scale EHR data, we rely on a Progressive Visual Analytics architecture [13, 42]. Our progressive approach provides a quick, approximate, yet useful preview of the data early on to maintain user attention within human latency limits and quick updates converging within seconds. Indeed, results from research in visualization show that user attention declines after 500 ms, and after five to ten seconds, users tend to abandon tasks or lose focus [25, 52]. After the initial preview, our algorithm continues processing the dataset, updating the visualization every second or so until it is fully processed.

Yet, the progressive updates of the aggregated tree can cause visual instability, as the nodes (i.e., rectangles on Figure 1a) are laid out at each update. The shifting of nodes can disrupt the viewer's mental map and create distracting visual flickers. Therefore, our approach may have an impact on the usability of the tool. To mitigate this issue, we introduce a *sorting with hysteresis* algorithm that limits this flicker during the progressive rendering.

In summary, this article presents our system and reports on its usability in terms of PVA, stability, and possibly flicker. Precisely, we:

1. Introduce our algorithm designed to process events progressively and evaluate its efficiency using metrics specific to PVA [38]. In particular, we report on **scalability** showing that our aggregation algorithm has linear complexity relative to the number of patients, that the processing of events per patient scales linearly with the number of CPU cores, and that it consistently meets the specified latency constraints [25, 52]. We also report on **accuracy** and show that ParcoursVis's progressive aggregation of sequences converges quickly for nodes close to the root of the tree, in line with analysts who explore the tree from the root to the leaves.
2. Introduce a sorting algorithm with hysteresis, i.e., maintaining node order for size differences below a threshold. We report on **stability**, demonstrating that sorting with hysteresis enhances both stability (i.e., avoiding flickering) and convergence of the progressive visualization.
3. We report on **usability** regarding the progressive aspect. Our professional participants have completed a variety of tasks seamlessly, without mentioning any delays in system's response or uncomfort with the progressive rendering of the Icicle tree.

According to the survey of Ulmer et al. [44], we describe the first progressive tree visualization algorithm. We outline its specific issues related to stability and uncertainty.

ParcoursVis improves the scalability of aggregated EHR event sequence visualization by at least three orders of magnitude compared to state-of-the-art systems. Its source code and evaluation scripts are available at https://gitlab.inria.fr/aviz/parcoursvis/.

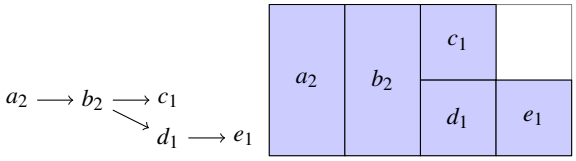

Fig. 2: ESS Aggregation of EHR Sequences as implemented by EventFlow and ParcoursVis. Patient *A* took the treatment sequence $\{a,b,c\}$ in that order, and patient *B* took $\{a,b,d,e\}$. The left side shows the *prefix tree* of these sequences, and the right side shows the Icicle tree visualization where node height encodes the frequency.

2 Related Work

Our work relates to temporal event sequences, scalability, PVA systems, and user uncertainty in the partial results. Our research goal is not to propose a new visualization technique but rather to explore the strategies to scale and adapt existing approaches that have already demonstrated their effectiveness and apply them to the largest EHR data available.

2.1 Temporal Event Sequences

Our tool, ParcoursVis, aims to support domain experts in visualizing and exploring patients' care pathways. Like EventFlow [29], it uses aggregation and filtering to visualize event sequences. According to the taxonomy in Wang et al.'s survey of EHR visualization [49], it falls into the "Event Sequence Simplification" (ESS) family of techniques (see Figure 2). This family is defined as "any technique used for reducing the visual complexity of event sequences in aggregated display overviews" [49]. Wang et al. distinguish seven families: ML, NLP, ESS, Geospatial Visualization, Clustering, Comparison, and Others. Among the 51 articles related to the visual analysis of EHR data, 11 belong to the ESS family. In other families, visualizing an overview of the event sequences is not the focus; they rather focus on one patient, a set of patients, or spatial areas sharing some characteristics.

Figure 1 shows the aggregated tree built from 10M sequences (patients) of events (e.g., treatments), visualized as an Icicle tree. As explained in Figure 2, the node height is proportional to the frequency. The seven nodes on the left of the visualization (Figure 1) represent these seven options at the beginning of the treatment. A patient can start by taking one or a mix of three drugs or directly undergo surgery. Each node has a width proportional to the duration of the treatment. The duration of treatment is often computed by aggregating several consecutive low-level events describing a patient repeatedly buying a prescribed drug at specific times, producing one continuous treatment event with a duration. ParcoursVis reuses the visualization technique of EventFlow because it was thoroughly tested and validated for analyzing medical temporal event sequences [5, 29, 31]. EventFlow, however, handles datasets with a limited number of sequences [5, 12, 29, 31]; about 20k sequences according to our experiments (available in the Supplemental Material). A common strategy for handling larger datasets is to select a random sampling of sequences [12]. Random sampling, however, loses rare and, therefore, important sequences that interest practitioners. Instead, we use PVA to be scalable and accurate.

In addition to visualization, EHR data is also analyzed using *sequence mining* augmented with visualization [27, 33, 42, 46]. The idea

consists of trying to automatically discover frequent and important patterns in the sequences to support the analyst in discovering unexpected information. This approach differs from ours since it only need to visualize the search results. Instead, our tool interactively visualizes all the treatment pathways following Shneiderman's mantra: overview first, zoom, filter, and details on demand [40].

2.2 Scalability

Existing systems that analyze event sequences, both in medical and non-medical domains, often fail to report their scalability [38] relative to the dataset sizes they handle. Wang et al.'s [49] survey does not report dataset sizes, although it reports scalability as the first future research challenge. In the medical domain, reported dataset sizes vary widely, with studies analyzing as few as 65 patients [3] to as many as 833,710 public health cases spanning over 10 years [21], passing by EventFlow [29] and its few thousands patients. Some works, such as DecisionFlow [18], mention handling over a million individual point events but provide no details on computational efficiency. In contrast, many other studies work with significantly smaller datasets, such as 5,800 patients [20], 2,336 patients [34], and 1,186 subjects [22]. The situation is similar in non-medical domains, where studies report dataset sizes in the range of thousands to millions of sequences, the largest dealing with 6,477 sequences of daily activities and 989,925 website visit sequences [46], yet offer no information on computational performance. We easily visualize this dataset with ParcourVis (available in Supplementary Material). Additionally, Liu et al. [26] mention that they use real-world datasets which can contain hundreds of thousands of unique sequences but do not provide algorithmic details for performance data.

Overall, for systems dealing with event sequence data, there is either a lack of information on performance relative to dataset size or a focus on small datasets, making it difficult to assess scalability.

2.3 Progressive Visual Analytics

PVA allows users to visualize the result of an algorithm while it is computed, in a progressive way. Instead of processing the whole data at once, waiting an unpredictable time, as standard algorithms do, PVA algorithms process data chunk by chunk or iteratively, or a mix of both [13] to control the latency and show intermediate results iteratively at a controlled pace. The main goal is to keep the overall system interactive and yield results within the typical user's attention span that lasts 1–10 seconds [30]. Practitioners already applied and studied the PVA paradigm on multiple types of visualizations, as described in the survey of Ulmer et al. [44].

In this work, we progressively aggregate event sequences of patients' care pathways into a prefix tree [49] that we visualize as an Icicle tree. Although PVA has been used for pattern mining in EHRs [42], to the best of our knowledge, we are the first to apply PVA to visualize EHR aggregated data at scale. We also introduce the first progressive tree visualization technique, according to the survey of Ulmer et al.

By presenting partial results quickly, PVA systems allow users to get quick feedback on the overview and queries. It also increases users' commitment and actions per minute [52]. In this article, we show that our PVA algorithm for computing the prefix tree is indeed scalable, according to the definition of Richer et al. [38]: the aggregation time scales linearly with the number of events, inversely proportional with the number of threads, and updates always meet the latency deadline.

2.4 Uncertainty and Trust

PVA systems start by showing partial results, implying uncertainty due to the progressive aspect of the computation. This *progressive uncertainty* [45] is reduced over time until the final result is computed, without any progressive uncertainty but potentially with other types of uncertainty, just like standard data and visualizations. The visualization shown early may or may not be close to the final result. It is, therefore, important for analysts to assess the quality of the progressive results to decide if they can trust the visualization or if they should wait longer.

Angelini et al. [2] explain that progressive visualizations undergo three stages: early partial results that are usually noisy and not trustworthy, mature partial results that are stabilizing and can be trusted but with uncertainty, and definitive partial results that are accurate and can take (wasted) time to finish. When our aggregation tree is computed progressively, all the nodes will undergo these three stages. Yet, the nodes closest to the root will become reliable and stable earlier than the deeper nodes since they are aggregating more sequences, and their stability is essentially related to the number of sequences they represent, assuming a well-behaved distribution of patients in nodes.

The uncertainty can be computed and visualized explicitly, through e.g., error bars, or implicitly through a proxy measure such as stability. If the visualization is not stable at some stage of the progression, it means that the result remains uncertain. The inverse is not always true: a progressive visualization that has remained stable for a certain time might still change before the end [9, 32].

Representing explicitly the uncertainty is possible but challenging. Understanding uncertainty in static visualization is not easy, even for statisticians [9]. For progressive visualization, many representations are possible, e.g., bar charts with error bars, violin plots, or gradient distribution [9, 15], but few have been studied and validated empirically. Patil et al. [32] have tried four designs for visualizing uncertainty on progressive bar charts, showing that two are effective. However, there is no study about the progressive visualization of an Icicle tree, to the best of our knowledge.

ParcoursVis takes a few seconds to update its tree fully, with an update every few seconds. It is not clear if users could make sense of any uncertainty visualization for many nodes in such a short update times. In our case, it might take as much time or more to make sense of an uncertainty visualization than to wait for the progressive visualization to complete. Instead, we rely on stability as a proxy for quality, assuming that subtrees will remain stable when the progressive uncertainty decreases. On the other side, when a subtree is not stabilized for a few seconds, the user will not be able to infer much from watching it. We study the convergence and stability of ParcoursVis's progressive Icicle tree visualization; instability becoming as a proxy for uncertainty [45].

3 ParcoursVis

We first describe ParcoursVis to set the overall context of our work before digging into its PVA-related algorithms. We also share knowledge about the PVA's impact on the software structure that we discovered when building ParcoursVis.

ParcoursVis is a web-based application inspired by EventFlow that processes data in a back end; users interact via a web browser front end. While ParcoursVis is specialized for exploring and analyzing EHR sequences, its core structure is more general and can handle the same kinds of event sequences and applications as EventFlow. Contrary to EventFlow, the rules for aggregating and merging events are written in C++ for performance and expressive power reasons. For real pathologies, many constants and aggregation rules need to be set according to state-of-the-art recommendations that cannot be expressed within EventFlow. Therefore, an engineer is needed to configure ParcoursVis to define new domain-specific rules when it is applied to a new medical question with specific kinds of events.

3.1 Example Use Case and Tasks

ParcoursVis is currently used in two main applications: non-cancerous prostate adenoma treatment analysis for the French social security and analysis of emergency patient pathways in all the public Parisian hospitals; we use examples from the former. Specific to our non-cancerous prostate adenoma use case, our application receives information from the Social Security reimbursement database (SNDS) about the drugs bought by patients (time, name, molecule), dedicated surgery, and deaths. There are three molecules possibly administered for treatment: alpha-blockers, specific herbal medicine, and 5-alpha, respectively labeled alphabloc, phyto, and fivealpha in Figure 1c. Those are the three concrete events our referring doctors selected to extract for their analysis. We call them the *low-level events*. ParcoursVis aggregates this information and synthesizes new events that we call *high-level events*. When the patient buys a box of pills with the same molecule every two weeks or so, all the successive low-level events are aggregated as one high-level event with a computed duration, e.g., "alphabloc" for 300

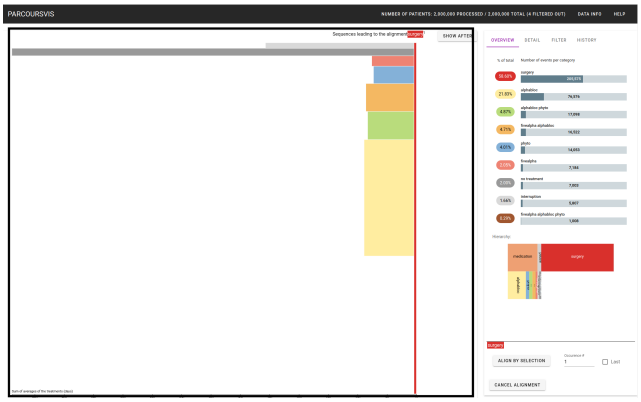

Fig. 3: The "Sequence" view visualization (squared) shows all events leading to a specified *prefix*, here the "surgery" event. About 60% of the patients who underwent surgery were treated before. A button allows showing all events that came after the prefix instead.

days. When the patient stops buying a box for more than six months, an "interruption" event is synthesized, and if the interruption is longer than one year, a "no treatment" event is synthesized, both with appropriate durations. Those thresholds are specified by the referring doctors according to state-of-the-art recommendations. Some of the low-level events can be combined, leading to specific aggregation rules: buying a box of alphabloc and a box of phyto within a certain time creates an "alphabloc phyto" treatment (line 3 of Figure 1c). This rule only applies to the three molecules, not to other events (surgery, death); it is domain specific. Our dataset also contains the age and information related to known chronic diseases of the patients (e.g., diabetes, hypertension) to explore possible correlations with treatment pathways.

ParcoursVis supports the following abstract tasks:

Simple Tasks that users may complete without assistance:

- Overview: visualize all the temporal event sequences (high-level event types).
- Zoom and Filter Events: visualize all the temporal sequences having a given prefix (children of a selected node) or filter out specific event types.
- Details on demand: visualize all the attributes and distributions related to a specific sequence (e.g., age distribution and duration distribution of the patients reaching a node).

Advanced Tasks that may require initial guidance:

- Filter by attribute: simplify the visualization by filtering the data based on the attributes and metadata of the sequence (e.g., its total length, age of the person it represents), and the resulting tree (e.g., size of the nodes).
- Configure view: setting an alignment prefix to visualize what had happened before and after a specific sequence.
- Abstracting: regrouping multiple types of events into a super-type, using a hierarchy (see Figure 1d).
- History and comparison: navigate through the history, i.e., previous filters, zooms, or alignments. Compare different trees corresponding to different history states by switching between them.

Analytical Tasks professionals want to answer:

- Application questions: identify trends, frequent, critical, or surprising pathways, and understand underlying causes, finding actionable strategies for improvement or resolution.
- Data questions: identify data errors, limitations, and determine more suitable data extraction methods.

3.2 User Interface

Our visualizations and user interfaces (GUI) are close to EventFlow, that has already been validated through many applications and case studies [5,29,31]. We focus on the specific features introduced by PVA.

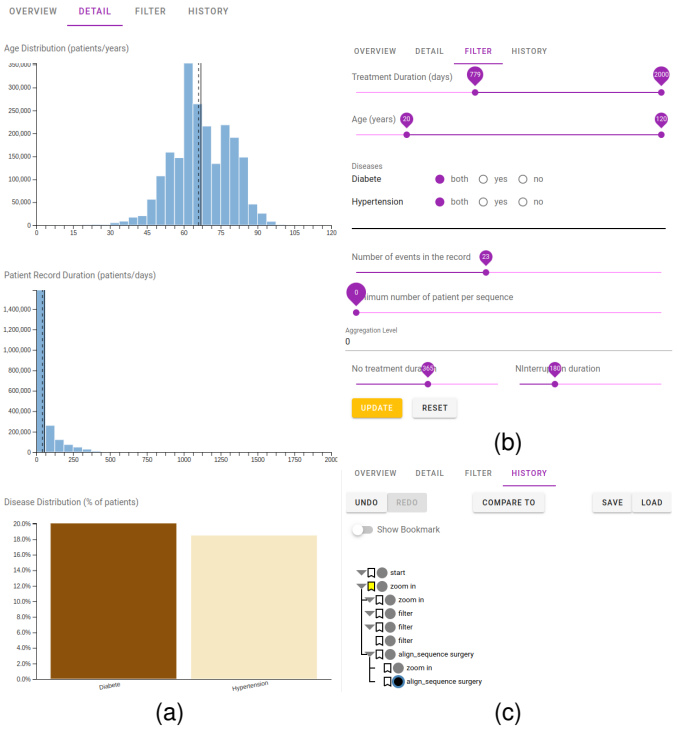

Fig. 4: (a): The "Detail" tab. (b): The "Filtering" tab. (c): The "History" tab. See Figure 1c for the Overview tab.

3.2.1 Main View and Prefix View

The "Main" view (Figure 1a) shows the aggregated tree of all patients' care pathways. The width of a node represents the mean duration of the high-level event it is associated with; its height represents the number of sequences (frequency). We sort the children of each node according to their frequency with some caveats described in Section 3.4.4 to improve the stability of successive progressive updates. The color of a node describes its high-level event type. Clicking on a node shows details on demand in the "Detail" tab of the control panel (Figure 4a).

Users can change the view and focus on a specific event or sequence of events. The "Prefix" view (Figure 3) shows the sequence tree that leads to or follows a given prefix, specified interactively.

Each of these views is updated progressively for scalability. We want to guarantee interactive latency while avoiding PVA from interfering with the user's exploration process.

3.2.2 Control Panel

Besides the "Main" or "Prefix" views, the control panel shows detail on demand and a dialog box to set a prefix on which to align the view.

Overview The "Overview" tab (Figure 1c) shows (1) the number of high-level events aggregated and (2) the hierarchy associated with the data using an Icicle tree. Hovering a node shows the number of aggregated events it contains. Those numbers are updated continuously when the progressive algorithm is running.

The hierarchy groups multiple high-level events into a higher category, e.g., all drugs under the category "medication". Figure 1d visualizes the hierarchy and its distribution in the partial aggregated tree. For applications with many event types, using higher hierarchy levels simplifies the visualization. Users can modify the level of hierarchy at any time. It is instantaneous, as it only impacts the prefix tree visualization without altering how the progressive algorithm aggregates the low-level events.

Detail The "Detail" tab (Figure 4a) shows the data distributions of a selected node. Currently, we show the age and duration distributions using histograms and the comorbidities using a bar chart. Those graphs

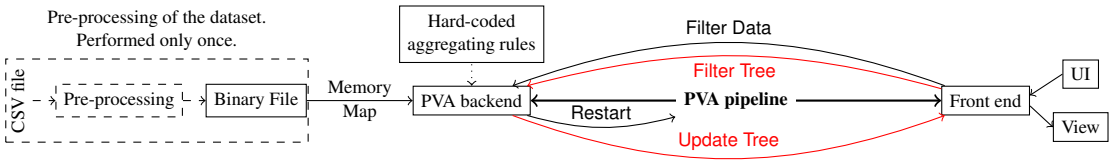

Fig. 5: Architecture of ParcoursVis, organized in a pre-processing stage, a back end, and a front end. See the state of the tree in red.

are updated iteratively when the progressive algorithm runs.

Filtering The "Filtering" tab (Figure 4b) allows to select a subset of patients and to filter the view according to parameters and node attributes. When users adjust the aggregating parameters (e.g., the thresholds defining the interruption events) or select a subset of patients by specifying query parameters, the progressive algorithm restarts from the beginning, considering only sequences matching the query and updating the rules that aggregate low-level events into high-level events. In contrast, filtering the view only manipulates nodes from the already computed tree (e.g., hide small nodes or merge nodes into higher hierarchical levels); it does not process the data again and results in instantaneous updates. To filter the view, users can specify a hierarchical level, truncate the tree to a maximum depth (e.g., show only the first three depth levels), and select a minimum number of patients per sequence to hide small nodes, simplifying the tree.

History and Comparison The "History" tab (Figure 4c) is inspired by the history comparison feature of Pister et al. [36] that relies on the Trrack [10] history management library. It shows the history of the user's actions as a tree; the user can jump to any past state of the exploration by selecting a history node.

The user can compare the current state with another one using the "compare to" button. ParcoursVis initializes a second context for the second state. The user can visualize the aggregated trees from both contexts in a back-and-forth manner using the "Next" and "Previous" buttons (not shown on the screenshots). The node associated with the current state in the "History" tab, has a blue halo, while the compared node has a green halo. Selecting this second context initially restarts the progressive aggregation using the parameters associated with its state. As the context being compared may not have finished processing all patients when switching back and forth between the compared context, we do not restart the aggregation from the beginning but resume it from where it was left previously. This avoids unnecessary waiting from the user to get back to where the context was previously and avoids breaking the user's mental model related to the context because the progressive algorithm may never get back to the exact previous view.

Overall, the PVA aspect of ParcoursVis required us to consider the usability issues of every feature that either (1) forces the aggregation algorithm to restart, (2) conversely, avoids restarting the algorithm when the front end can handle it alone, and (3) affects the user experience in an unexpected way (e.g., as with the "History" tab).

3.3 Implementation

Figure 5 describes the architecture of ParcoursVis. Our front-end application is made of about 3000 lines of JavaScript that uses `D3.js` [7] and `Vue.js` [47]. This allows us to show visualizations on a regular web browser accessible to anyone and, in particular, to referring doctors without any installation. However, web technologies are not designed for heavy computation. We, therefore, rely on an optimized back-end library written in C++ (3000 lines) and used as a Python module that progressively aggregates the data sent to the front end. The back-end Python server code is 127 lines long. We also provide test suites for JavaScript and C++ that double the running code size. The code is open-source at https://gitlab.inria.fr/aviz/parcoursvis/.

While our main view shows aggregations and visualizations similar to EventFlow, our engineering design differs in many ways from it for scalability and stability purposes.

id	age	type	date	drug	comorbidities
0	71	fivealpha	2015-05-29	dutastéride	Diabetes
0	71	fivealpha	2015-06-05	dutastéride	Diabetes
1	50	alphabloc	2003-09-26	alfuzosine	Hypertension
1	50	alphabloc	2003-10-03	alfuzosine	Hypertension
1	50	alphabloc	2003-10-10	alfuzosine	Hypertension
...					

Table 1: CSV file showing records using our format, typical of event sequence systems. Patient #0 is 71 and bought the drug "Dutastéride" containing the 5-alpha molecule in May and June 2015. He suffers from diabetes.

3.4 Optimization of ESS Visualization Programs

All ESS systems we know are implemented using a structure similar to the simplified code in Listing 1. They take a file as input, usually in CSV format as shown in Table 1 and produce a prefix tree that is then rendered on screen. Their run time is essentially linear in the number of events, but existing implementations use data structures that are expensive both in memory and execution time, limiting the dataset size to achieve the computation under the 1–10s latency barrier. Improving the scalability can be done with three strategies: 1) optimizing the data structures and algorithms, 2) parallelizing the program ⇉, and 3) turning it into a progressive program. Only applying strategies 1–2, the function "aggregate" (line 15 of Listing 1) will continue to take a time proportional to the number of events. Above some number, the time will exceed the acceptable latency and degrade the interaction.

ParcoursVis applies the three strategies, as shown in Listing 2. It optimizes the loading time, the data structures, parallelizes the aggregation ⇉, and performs the computation in a progressive manner to decouple processing time from latency.

3.4.1 Optimizing Data Preprocessing

First, as shown in Figure 5, ParcoursVis relies on pre-parsed data stored in binary format, avoiding the expensive step of parsing CSV data at each program start while ensuring all events per patient are sorted chronologically and that patients are shuffled to optimize the aggregation.

Running the preprocessing step takes three minutes for 50 million events stored in CSV format (as in Table 1). It is performed once for a given dataset and generates the binary files that are memory-mapped when the ParcoursVis's back end starts.

Whereas the non-progressive program loads the CSV files in a time linear to the file size (Listing 1 line 2), the progressive back end loads the binary files in negligible time using *memory mapping*, as shown in Listing 2 lines 3–4.

3.4.2 Optimizing Aggregation ⇉

Our aggregation algorithm (called `parallel_aggregate` in Listing 2) is similar to `aggregate` in Listing 1 but rewritten for speed. It keeps a complexity linear with the total number of events but is much faster and guarantees a controlled latency. It is parallelized ⇉ and, to become progressive, it works by chunk.

The first optimization relies on the memory layout, which organizes the low-level event sequences in consecutive memory positions so the aggregation algorithm can access them sequentially. Modern hardware strongly optimizes this access pattern by prefetching memory caches along with sequential access. The aggregation process takes a table of low-level events indexed per patient and outputs a prefix tree meant to be rendered. Note that the run time is essentially linear in

```
def ESS(filename):
    events = pd.read_csv(filename, index_col=None)
    events.sort_values(by=["id", "date"], inplace=True)
    patients = events["id"].unique()
    tree = aggregate(patients, events)
    render(tree)

@dataclass class Node:
    type: str
    count: int = 0
    children: dict = {}
    agedist: Counter = Counter()
    durationdist: list[int] = []

def aggregate(patients, events):
    tree = Node(type="root")
    for patient in patients:
        patient_events = events[events["id"]==patient]
        aggregate_patient(patient_events, tree)
    return tree

def aggregate_patient(lowlevel, tree):
    current_node = tree  # start from the root
    context = Context()  # we need to maintain a context for aggregating events
    for event in lowlevel+ENDEVENT:  # Add a fake empty event at the end
        for highlevel in context.low_level_to_high_level(event):
            # low_level_to_high_level returns 0 or more events
            # highlevel looks like: {type: "alphabloc", age: 42, duration: 300}
            current_node = merge_node(current_node, highlevel)
```

Listing 1 Overview of a non-progressive ESS visualization.

the number of events, but existing implementations use data structures that are expensive both in memory and execution time, limiting the dataset size allowed to remain interactive. In EventFlow, transforming low-level events into high-level ones is expensive due to the rich set of possible transformations supported, as shown in the Supplementary Material.We write the function `low_level_to_high_level` directly in C++ for two reasons: expressive power and optimization. This function is specific to a treatment or use-case (e.g., non-cancerous prostate adenoma, used as an example here) so we hard-code the rules, e.g., the definition of what is an "interruption" synthetic event in a specific treatment. We thus make sure the rules follow state-of-the-art recommendations. We allow the analysts to change some parameters within meaningful limits during the exploration. The rules are more complex than what EventFlow can express, yet rather simple to express using a regular programming language like C++. ParcoursVis's implementation is 190 lines long. Adapting it to another scenario, described in Section 5, took a few days and had a similar length.

⁛ Second, for the progressive aspect, we rely on the patients being shuffled. Shuffling is important in PVA for faster convergence and to overcome order bias in the data [39]. Keeping patients in chronological order would initially reveal pathways that followed older medical protocols and delay the visualization of more recent ones, biasing the early overview snapshots.

The main loop performs the tree aggregation along with the progressive calculations of the distributions. As shown in Listing 2 line 6, the back end partially aggregates the tree with a certain number of sequences (a *chunk*) and sends it to the front end (function `render` in Listing 2 line 8) and, when the function returns (the user interface received the data), it continues the aggregation for the next chunk.

To determine the `chunk_size` parameter for a given iteration (kept constant in the pseudo-code of Listing 2), we measure the average speed (patients per second) of the last six iterations and multiply it by the desired latency (typically 1 second). We start with the pessimistic chunk size of 100,000 patients and converge to a chunk size adapted to the network latency and performance of the back end.

We send the resulting tree to the rendering client in JSON format after two transformations: the duration lists are transformed into histograms, and small nodes below a specified threshold of patients (10–50) are trimmed. In our applications, the final tree has around 2k nodes; it can be transferred in a few milliseconds to the front end through a web socket.

⇉ Finally, Line 11 in Listing 2 shows the parallelization of the algorithm. It is "embarrassingly parallel" except for merging the resulting trees in line 18, which could slow down the process. Section 4 evaluates the scalability of our aggregation algorithm.

```
def ESS(directoryname):
    chunk_size = 100,000  # tuned according to the processing speed
    events = mmap_events(directoryname, "events")
    patients = mmap_patients(directoryname, "patients")
    full_tree = Node(type="root")
    for patients_chunk in split(patients, chunck_size):
        tree = parallelized_aggregate(chunk, events)
        full_tree.merge(tree)
        render(root_tree)

def parallelized_aggregate(patients_chunk, events):
    patients_per_thread = splitIn(patients, number_of_threads)
    trees = [None] * number_of_threads  # will receive the threads trees
    parallel for thread in range(number_of_threads)
        tree = aggregate(patients_per_thread[thread], events)
        trees[thread] = tree
    for tree in trees[1:]:  # merge all the trees in the first one
        trees[0].merge(tree)
    return trees[0]
```

Listing 2 Progressive Parallelized ESS Visualization. The type `Node` and function `aggregate` are the same as in Listing 1.

```
def HysteresisSort(oldNode, newNode, inertia=20.0/1080):
  '''Sort nodes with a hysteresis condition
  oldNode: The node at the previous iteration
  newNode: The node at the current iteration. newNode.children should keep the
        same order as oldNode.children
  inertia: The hysteresis inertia
  return newNode with children sorted in descending order with an inertia '''
  if oldNode is None: # No previous iteration -> quick sort
    return sorted(newNode.children, key=lambda node: node.count,
                  reverse=True)

  # Apply a bubble sort with hysteresis to fulfill Equation 1.
  hasPermuted = True
  while hasPermuted:
    hasPermuted     = False
    minValue        = newNode.children[0].count
    lastMinValueIDx = 0
    for i in range(1, len(newNode.children)):
      # If Equation 1 is not fulfilled -> Reorder the nodes
      if minValue < newNode.children[i].count and \
         (newNode.children[i].count - minValue)/newNode.count > inertia:
        for j in range(i, lastMinValueIDx, -1):
          (newNode.children[j-1], newNode.children[j]) = \
            (newNode.children[j],   newNode.children[j-1])
        lastMinValueIDx += 1 # lastMinValue moved
        hasPermuted = True
      elif minValue >= newNode.children[i].count:
        # using '>=' minimizes the permutations
        minValue = newNode.children[i].count
        lastMinValueIDx = i

  return newNode
```

Listing 3 The *Hysteresis Sort* algorithm in Python.

3.4.3 Progressive Rendering ⁛

Rendering is done on the front end in two passes: first, it preprocesses the tree according to user-specified parameters, such as the maximum tree depth and event filters, generating a rendering tree. Then, it lays out the Icicle tree in SVG format that eventually appears on the screen. The first step rewrites the aggregated tree into a new one when only simple transforms are involved, saving back-end heavy work. Typically, to filter out node types or apply a hierarchy transform (see Section 3.2.2 for details). This allows for a smoother user experience since a tree is usually composed of a few thousand nodes that are fast to process compared to the dataset that can contain millions of events.

To prevent the progressive algorithm from flooding the rendering, the front end notifies the back end to continue the aggregation process once it has fully displayed an updated tree. When the user changes any of the parameters used for aggregation, the aggregation algorithm restarts from the beginning, iteratively updating the rendering.

3.4.4 Stabilizing Rendering ⁛

The visualization of the ParcoursVis's aggregated tree is an Icicle tree; it is laid out horizontally, node width encoding average (or median) node duration and node height encoding frequency, sorted to help compare siblings and show the frequency distribution per node. The stability of the aggregated visualization between iterations mainly depends on two node parameters: the *width* and the *order* of siblings. Changing the node width across updates can slightly shift subtrees, but changing sibling order can strongly change the overall layout, producing instability interpreted as uncertainty by users. We want to avoid

Type	Super-type	# of events
Alphabloc	Treatment	1,098,585
Phyto	Treatment	530,402
Fivealpha	Treatment	192,127
Alphabloc Phyto	Treatment	265,312
Fivealpha Alphabloc	Treatment	161,166
Fivealpha Alphabloc Phyto	Treatment	26,928
Interruption	Interruption	123,465
No Treatment	Interruption	111,188
Surgery	Surgery	208,907
Death	Death	1,738

Table 2: The ten event types used in our example use case.

this instability if it is only an artifact of the progressive algorithm.

With our progressive aggregation algorithm, nodes with close frequencies could switch positions between iterations, e.g., two nodes containing almost 20% of the paths each could swap back and forth across iterations due to slight variations in the distribution of patients. We designed a *Hysteresis Sort* algorithm (Listing 3) to prevent these changes when the frequency difference is irrelevant visually. It is specifically designed for the progressive visualization of trees that are relying on sorting the siblings by size.

Our *Hysteresis Sort* sorts children perfectly at the first iteration. For each new iteration, when the size of two adjacent nodes is almost the same, we consider the two sizes as equivalent and maintain their previous order, as shown in Listing 3. Almost the same means the frequencies differ by a small number ε; we call it the *inertia*. Our algorithm ensures that the nodes are almost sorted with an error bounded with the inertia, according to the following equation:

$$\forall\{i,j\}, i<j \implies freq(child_j) - freq(child_i) \leq \varepsilon \quad (1)$$

We sort children in descending order, bottom to top, as we want new children created between two progressive iterations to be inserted at the end of the list of siblings (on top). This is because, statistically, rare nodes appear later than frequent nodes in the progression. Our algorithm is a variation of bubble sort. Since Equation 1 does not fulfill the triangular inequality (if $a-b \leq \varepsilon$ and $b-c \leq \varepsilon$, we cannot be sure that $a-c \leq \varepsilon$), we are forced to verify the whole list again when children are moved, using the variable `hasPermuted` on line 27 of Listing 3. The worst-case complexity of the algorithm is $O(n^2)$, but since the children are mostly sorted, its actual complexity is linear, and the number of children is usually small.

We choose a default inertia small enough to be unnoticeable visually, except at the first level, where it represents 20 pixels in height on a 1080p monitor. Below this value, two nodes are considered equivalent in size. Users can change inertia interactively if desired.

4 Evaluation

One goal of ParcoursVis is performance and scalability to explore large-scale pathways visualizing billions of events aggregated while remaining interactive. Our evaluations report scalability, stability, completion time, and speed metrics. Usually, speed is completion time divided by the number of items considered, but with PVA, speed, and completion time are decoupled as users can make early decisions if a partial result is "good enough". We, therefore, report on the time taken for the results to become good enough to make an accurate decision regarding basic tasks. In addition to our git repository, we provide all data analyses on an osf.io repository.

4.1 Data

ParcoursVis reads a dataset of low-level events, translates it into high-level ones, and builds its aggregated representation: the prefix tree. Our evaluation relies on our original synthetic dataset of $2M$ patients treated for non-cancerous prostate adenoma. It uses five low-level event types—Alphabloc, Phyto, Fivealpha, Surgery, and Death—and translates them into ten high-level event types grouped into four higher-level categories, shown in Table 2.

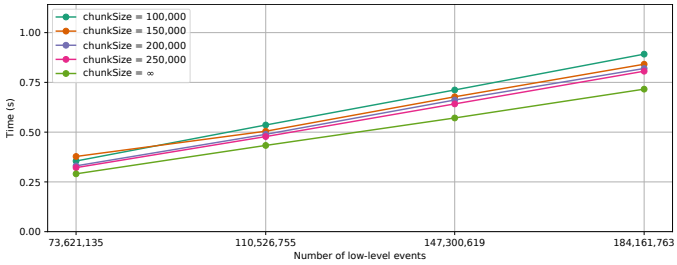

Fig. 6: Total computation time per size compared with five chunk sizes using 6 threads. *chunkSize* $= \infty$ means non-progressive.

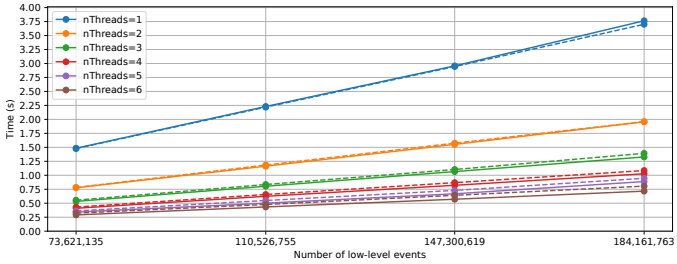

Fig. 7: Total computation time per size compared by # of threads in progressive (dotted) and non-progressive (plain) settings.

To measure the scalability of ParcoursVis, we generated four synthetic data with *size* $\in \{4M, 6M, 8M, 10M\}$ patients using the strategy described in the supplementary material; their characteristics are listed below. The aggregated tree characteristics of the original 2M patients dataset are in the first line.

Patients	Low-Level	High-Level	Pathways	Nodes
2M	*50,640,284*	*2,719,818*	*948*	*1,122*
4M	73,621,135	5,441,380	926	1,109
6M	110,526,755	8,164,831	948	1,134
8M	147,300,619	10,884,766	966	1,155
10M	184,161,763	13,607,620	972	1,161

Before measuring the stability of ParcoursVis, we analyzed the time to aggregate the dataset with our progressive algorithm (i.e., chunk by chunk) compared to treating the whole dataset at once, not progressively. This *competitive analysis* is standard for online algorithms [6]. The reported completion times do not consider the rendering of the tree nor the possible network delay of a back-end/front-end architecture.

4.2 Aggregation

We expect our algorithm (Listing 2) to be linear in the number of events and threads. We report the computation times it takes to aggregate them both progressively and non-progressively. Instead of aggregating patients during a quantum of time, we aggregate with a fixed number of sequences for better control and reproducibility [14]. We vary *chunkSize* $\in \{100K, 150K, 200K, 250K\}$ sequences. As we expect our algorithm to be linear in the number of low-level events, we use that metric as the size of the databases in our evaluation. We ran our evaluations on an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz that has 6 cores and 12 logical threads. We disabled simultaneous multithreading (SMT) to enforce that each thread is associated with a physical core. We measured the time using a monotonic wall-clock.

4.2.1 Results

It takes from 3.75s with one thread (2.67M patients per second) to 0.75s with six threads (13M patients per second) to process our biggest synthetic dataset of 10M patients in a non-progressive environment (Figure 6). The progressive environment yields similar results but with an overhead that depends on the *chunkSize* parameter. Figure 6 shows that the computation time decreases when the chunk size increases. Figure 7 also shows that the overhead increases with the number of threads. For the mono-thread configuration, the overhead is negligible. We explain this overhead by the number of tree merges the algorithm does

(see Listing 2 line 18), which depends on the average and the number of threads as follows: $merges = (nbThreads - 1) \times nbPatients/chunkSize$.

For $chunkSize = 100K$, $nbPatients = 10M$, and $nbThreads = 6$, we merge the trees 500 times, each composed of $\approx 1,100$ nodes.

For the best-case scenario of $chunkSize = 250K$ and $nbThreads = 6$, each progressive iteration takes about 19.8ms (median) to complete (12.6M patients per second).

Figure 7 shows that, for all numbers of threads, the computation time of our algorithm remains linear in the number of low-level events for both progressive and non-progressive environments. Plotting the total computation time per size compared to the number of threads, we find that our algorithm running time is inversely proportional to the number of threads. Looking at the computational speed per size compared to mono-thread computations, we find that the speed increases linearly with the number of cores. A linear least-squares regression gives a slope ≈ 0.83 and $R^2 > 0.99$. A one-way ANOVA on the linear regression predictions gives a *p-value* > 0.99; the slope coefficients for all the chunk sizes are the same.

We conclude that the runtime of our progressive and non-progressive algorithms is inversely linear to the number of cores and linear with the number of low-level events. Tree merge time is negligible.

4.3 Stability

We now focus on the stability and convergence of ParcoursVis over the progressive iterations. We compare our *Hysteresis Sort*, described in Section 3.4.4, with the regular sort (called *RSort* in the remaining) that EventFlow uses, i.e., sorting nodes based on frequency. We configured *Hysteresis Sort* with *inertia*$= 20/1080 \approx 0.02$ by default, which corresponds to about 20 pixels for the root level of our visualization on a standard 1080p monitor, and below one pixel for the deeper tree levels.

We considered two external factors affecting the ordering of nodes. First, we avoid what Monroe et al. [29] call *confetti* visualization by only showing nodes representing more than a certain *MinSize* number of patients. We evaluate ParcoursVis along the *MinSize* variable: we output a node and its children only if $size(node) \geq MinSize$ with $MinSize \in \{0, 25, 50\}$.

The second factor impacting the stability of the tree is the convergence of the intermediate progressive results, which is sensitive to the number of sequences each iteration processes; larger chunks will lead to faster convergence. The number of intermediate steps depends on the CPU speed, machine load, and the time quantum that are hard to control. We, therefore, vary the $chunkSize \in \{100K, 150K, 200K, 250K\}$ to control the number of patients processed per iteration to evaluate when nodes stabilize while ensuring reproducibility for the evaluation.

4.3.1 Metrics of Stability

For *Hysteresis Sort* and *RSort*, we report two metrics for stability. We also compute a binary metric *Bad_Statistics*, true if the frequency of a node at $iteration = i$ differs from more than 5% to the final result. We use a frequency of 0 for not-yet-registered nodes. We report the lowest possible $iteration = i$ for both metrics.

Stability by Depth A depth is considered stable at iteration i if, below that depth, there is no change of order in the trees generated from $iteration = i$ to the final result.

Stability Per Node A node is considered stable if its parent is stable, if this node and its siblings have the same rank in their parent's list from $iteration = i$ to the final result. The root is always considered stable. If a node is not created at a given iteration, we consider that it is placed at the end (last rank) of the children list.

We emphasize that since we compare nodes from $iteration = i$ to the final result, our report on stability is similar to convergence.

4.3.2 Results

Our evaluation shows that all nodes have a similar frequency from the first iteration to the last, regardless of *MinSize* and *chunkSize*, meaning that the frequency distribution of nodes converges fast for datasets like ours. As we shuffled the datasets, this result is expected for high values of *chunkSize*. We then do not report on the variable *Bad_Statistics* further.

We then look at how fast *Hysteresis Sort* and *RSort* stabilize in depth and relative position (Figure 8 and Figure 9). We normalized the iteration ID from 0 to 1 to compare all evaluations with different *chunkSize*. The distribution of the number of stabilized nodes and depths compared to the iteration ID is not normal. Thus, we cannot rely on parametric statistical tests. Following recommendations of [1, 4, 11], we instead rely on estimation techniques with effect-size and confidence intervals (CIs). We bootstrap all the evaluation results per sorting strategy and per depth, resulting in $size(MinSize) \times size(chunkSize)$ different values per bootstrap. Depth starts from 0, which corresponds to the children of the root, as the root is, in our case, an abstract object that we do not visualize in ParcoursVis.

Figure 8 shows that the prefix tree is stable after the progressive algorithm processed 80% (at the right edge of the 95% CI) of the dataset at the maximum tested $depth = 4$ using the *Hysteresis Sort*, compared to approximately 90% using *RSort*. Our results (Figure 9) show strong evidence (statistically significant differences) that *Hysteresis Sort* and *RSort* start to diverge at *depth=1*, with a large effect size for $depth \in \{1, 2\}$. Because deeper depths depend on shallower ones, the results of *RSort* for $depth = 1$ strongly impact the results of deeper depths. This seems less the case for *Hysteresis Sort* (Figure 8).

We then looked at the stability per node categorized by their depths. Figure 9 shows strong evidence that *Hysteresis Sort* outperforms *RSort* at stabilizing nodes sooner for all $depth \geq 1$. In addition, we looked at the stability of nodes categorized by their frequencies. We computed, per node, the difference in stability of *Hysteresis Sort* compared to *RSort* for the evaluation, set with $MinSize = 0$ and $chunkSize = 100K$, which are the minimum values we tested. We see that *Hysteresis Sort* and *RSort* behave the same for most nodes, which we expected as most nodes do not have siblings with similar frequencies (see $iteration = 0$). Second, the sorting strategy mostly impacts nodes with low frequency (lower than 200, i.e., representing less than 1% of our population). Third, while *Hysteresis Sort* strongly improves the stability of most of the remaining nodes, it can degrade the stability of some small nodes for a few iterations. Overall, we conclude on the effectiveness of *Hysteresis Sort* compared to *RSort*.

Finally, we wanted to see if the variables *MinSize* and *chunkSize* have an effect on the stability. Figure 10 suggests that *MinSize* strongly impacts the stability of the positions of the nodes by removing small changes. Because the dataset is shuffled, large nodes tend to stabilize sooner than smaller ones, explaining the results. However, this figure shows that *MinSize* has a stronger impact on the *Hysteresis Sort* than on the *RSort*. We do not find any effect of the *chunkSize* variable for the stability of nodes.

5 Applications

ParcoursVis is applied to two distinct use cases, enabling participants to visualize real large datasets with an unprecedented level of scalability. We provide usability insights, focusing on its progressive aspect when used by experts. Participants were able to perform tasks comparable to those in other ESS systems with no perceived latency.

5.1 Non-cancerous prostate adenoma in France

We provided ParcoursVis to referring doctors to explore the non-cancerous prostate adenoma dataset. We received positive feedback. One referring doctor reported that his wife thought he had a new video game because he was spending hours exploring the data interactively. They were excited to be able to explore such a large dataset to eventually find unexpected cases. One of the referring doctors have collected

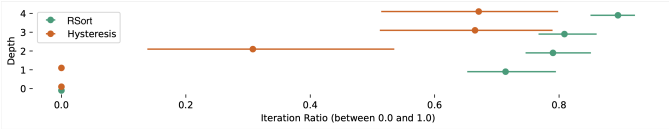

Fig. 8: Stabilization speed of *Hysteresis Sort* and *RSort* by depth. Iteration ratio (0 to 1) indicates dataset processing progress.

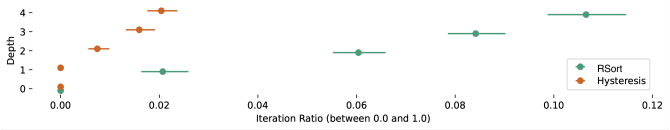

Fig. 9: How fast the *Hysteresis Sort* and *RSort* stabilize nodes by depth.

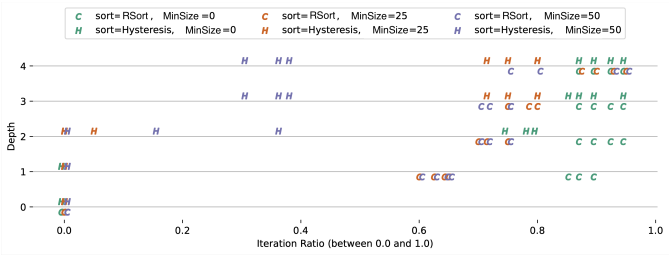

Fig. 10: The stability of the tree by depth and *MinSize* values. Each category has 4 *chunkSize* values. We jitter the points on the x-axis to avoid points overlap for different *MinSize* values. The figure highlights clusters of categories: the higher MinSize, the sooner the tree is stabilized. *MinSize* has a stronger effect on *Hysteresis Sort* than on *RSort*.

many insights and surprising findings that need inquiries now. For example, they thought that surgery would cure the patients when, in reality, many patients (18%) need to resume treatment and sometimes undergo another surgery.

5.2 Emergency Pathways in Parisian Hospitals

We also used ParcourVis for visualizing patient pathways in the 16 public Emergency Departments (EDs) in Paris. For over 15 years, EDs have faced a steady increase in patient volume, leading to frequent overcrowding and prolonged waiting times. Analyzing patient care pathways offers the potential to identify specific stages where the process becomes delayed, resulting in extended waiting times or medical accidents. Our project deals with 1.5M patient visits over three years. We wrapped the ParcoursVis front end into a widget for use within the secure Jupyter Lab environment provided by Paris Public Hospitals IT, allowing for easy access by our collaborators.

We interviewed, in an ongoing usability study, two professionals interested in analyzing patients' pathways: an emergency physician highly motivated to find more effective organizational approaches and a computer scientist aiming at achieving high-fidelity simulation of patients' flow in EDs. None of our participants mentioned any latency issues, not even performance issues. They were able to focus on their professional questions.

We also used ParcoursVis to answer several questions during the project's meetings, allowing us to quickly characterize and select important pathways to optimize, such as the most frequent ones, the most time-consuming ones, and the most "undesirable" ones.

6 Discussion

6.1 Performance Analysis

The competitive analysis (Section 4.2) validates the linear complexity of our aggregation algorithm, both in the number of threads and low-level events. We aggregate 10 million synthetic patients ($\approx 180M$ low-level events) on our dedicated computer in less than one second. ParcoursVis is meant to be used on a server shared by multiple users; aggregating sequences in the wild takes longer. Moreover, our evaluation relies on short sequences on which we applied a small number of filters. Longer sequences and more filters cause linear slowdowns. PVA solves these issues with almost no overhead.

We filter small nodes to avoid confetti visualizations, but users can configure it. In our evaluation using $MinSize = 50$ (i.e., 0.0025% of the two million patients dataset), the view aggregating the dataset stabilizes at around half the computation time for the deepest nodes using *Hysteresis Sort*, 25% sooner than *RSort* (Figure 10). The *MinSize* variable thus can strongly increase the speed at which users can work on their data if they are not interested in small nodes.

6.2 Stability

We evaluated the stability of aggregated trees for one use case with an inertia of 20/1080. For that configuration, we show *Hysteresis Sort* is better than *RSort*. We believe this generalizes to other datasets, although the stabilization times may vary depending on the distribution of events. Our *Hysteresis Sort* algorithm relies on a modified version of the bubble sort algorithm with a theoretical worst-case quadratic complexity. In practice, our sort time is linear because we use a limited number of types to be able to assign them to color effectively. The scalability of ParcoursVis and ESS, in general, remains limited by the number of event types they can handle [49].

6.3 Evaluation Criteria for Progressive Visualization

Progressive systems are relatively new and we need more evaluation criteria to assess their usability [37].

We did not consider comparing the usability of our system in its current state to a non-progressive system given the latency we measured that makes non-progressive systems completely unusable (measures are in the Supplementary Material). During our expert interviews, participants were unaware that the tool used progressive algorithms and did not notice it; we consider this lack of mention a usability success for progressive systems and algorithms. It is a new evaluation criteria.

As explained in Section 2.4, we rely on stability as a proxy for convergence. For update durations limited to tens of seconds, we believe instability is an acceptable proxy for uncertainty, and our expert users did not express any confusion or trust issues about it.

ParcoursVis could also include visualization elements that indicate the threshold for our hysteresis node ordering, coupled with direct user interaction. With this more advanced feature, conducting studies on trust vs. stability in the visualization could be relevant.

6.4 Improving the Scalability of EHR Visualizations

Enhancing the efficiency of data structures and algorithms is crucial for scalability. However, relying solely on this approach overlooks the potential for greater results. As demonstrated in this article, partitioning data into chunks and effectively managing aggregated cumulative data are crucial for optimizing performance at scale. Indeed, incorporating progressive computation in the main loop further enhances scalability and decouples interactivity from data size . In the latest version of ParcoursVis, we also perform progressive computations on-demand, such as the distributions of a large number of attributes associated with patients when a node's details are demanded; this avoids slowing down the main aggregation loop while supporting a rich set of attributes.

Furthermore, effective uncertainty visualization depends on the convergence rate. When the convergence is longer than 30 seconds or so, uncertainty can be visualized on its own since users have time to read it and decide if what they see is good enough while the progression continues. For faster convergence, stability can become a proxy for quality with the caveat of avoiding spurious instability due to progressive algorithm artifacts, as described in Section 4.3.

7 Conclusion

We presented ParcoursVis, a PVA tool we designed to explore patients' care pathways at scale. We described its use, including the history navigation and comparison that are novel for HRS systems. We described its scalable aggregation algorithm and its stabilizing sorting algorithm. We showed quantitative evaluations showing the aggregation algorithm scales linearly with the number of events and inversely with the number of threads. We also showed that our Hysteresis sort improved the stability of the Icicle tree visualization. ParcoursVis can be used effectively to explore patients' treatments and conditions aggregated at the scale of the largest countries to improve public health based on data.

For a dataset scaled to China's population, non-progressive aggregation on our machine would take 27s. In contrast, our PVA architecture

ensures updates in under 1s, with most tree nodes stabilizing within 10s, enabling exploration of datasets at the scale of countries.

Each new study requires new features, and we are planning to extend ParcoursVis to make it more generic and applicable to new use cases without programming. We also plan to support datasets updated continuously. That would require certain adjustments to the system, specifically running regular pre-processings of the updated database, e.g., every 15 minutes. ParcoursVis should be able to restart immediately when a new dataset is available, but it should avoid interrupting ongoing explorations. By providing our system in open-source, we want to push this research field to a wider audience relying on PVA to provide scalability and improving public health.

Acknowledgments

The authors wish to thank Catherine Plaisant for her valuable input, which mostly comes from her experience with the EventFlow project. This work was supported in part by a grant from the Health Data-Hub, and from the URGE AP-HP/Inria project.

References

[1] V. Amrhein, S. Greenland, and B. McShane. Scientists rise up against statistical significance. *Nature*, 567(7748):305–307, 2019. doi: 10.1038/d41586-019-00857-9 8

[2] M. Angelini, G. Santucci, H. Schumann, and H.-J. Schulz. A Review and Characterization of Progressive Visual Analytics. *Informatics*, 5(3):31:1–31:27, 2018. doi: 10.3390/informatics5030031 3

[3] J. Bernard, D. Sessler, T. May, T. Schlomm, D. Pehrke, and J. Kohlhammer. A visual-interactive system for prostate cancer cohort analysis. *IEEE Computer Graphics and Applications*, 35(3):44–55, 2015. 3

[4] L. Besançon and P. Dragicevic. The Continued Prevalence of Dichotomous Inferences at CHI. In *Proc. CHI*, pp. 14:1–14:11. ACM, New York, May 2019. doi: 10.1145/3290607.3310432 8

[5] M. V. Bjarnadóttir, S. Malik, E. Onukwugha, T. Gooden, and C. Plaisant. Understanding Adherence and Prescription Patterns Using Large-Scale Claims Data. *PharmacoEconomics*, 34:169–179, 2016. doi: 10.1007/s40273-015-0333-4 2, 4

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998. 7

[7] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185 5

[8] J. Brickell and V. Shmatikov. The Cost of Privacy: Destruction of Data-Mining Utility in Anonymized Data Publishing. In *Proc. KDD*, 9 pages, pp. 70—78. ACM, New York, 2008. doi: 10.1145/1401890.1401904 2

[9] M. Correll and M. Gleicher. Error Bars Considered Harmful: Exploring Alternate Encodings for Mean and Error. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2142–2151, 2014. doi: 10.1109/TVCG.2014.2346298 3

[10] Z. Cutler, K. Gadhave, and A. Lex. Trrack: A Library for Provenance-Tracking in Web-Based Visualizations. In *Proc. VIS*, pp. 116–120. IEEE, Los Alamitos, 2020. doi: 10.1109/VIS47514.2020.00030 5

[11] P. Dragicevic. Fair Statistical Communication in HCI. In *Modern Statistical Methods for HCI*, chap. 13, pp. 291–330. Springer Intern., Cham, 2016. doi: 10.1007/978-3-319-26633-6_13 8

[12] F. Du, B. Shneiderman, C. Plaisant, S. Malik, and A. Perer. Coping with Volume and Variety in Temporal Event Sequences: Strategies for Sharpening Analytic Focus. *IEEE Trans. Vis. Comput. Graph.*, 23(6):1636–1649, 2017. doi: 10.1109/TVCG.2016.2539960 2

[13] J.-D. Fekete, D. Fisher, and M. Sedlmair. *Progressive Data Analysis: Roadmap and Research Agenda*. Eurographics, Nov. 2024. doi: 10.2312/pda.20242707 1, 2, 3

[14] J.-D. Fekete and J. Freire. Exploring Reproducibility in Visualization. *IEEE Computer Graphics and Applications*, 40(5):108–119, 2020. doi: 10.1109/MCG.2020.3006412 7

[15] D. Fisher, S. M. Drucker, and A. C. König. Exploratory visualization involving incremental, approximate database queries and uncertainty. *IEEE Computer Graphics and Applications*, 32(4):55–62, 2012. doi: 10.1109/MCG.2012.48 3

[16] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-Preserving Data Publishing: A Survey of Recent Developments. *ACM Comput. Surv.*, 42(4), article no. 14, 53 pages, June 2010. doi: 10.1145/1749603.1749605 2

[17] A. Goncalves, P. Ray, B. Soper, J. Stevens, L. Coyle, and A. P. Sales. Generation and evaluation of synthetic patient data. *BMC Medical Research Methodology*, 20(108), 2020. doi: 10.1186/s12874-020-00977-1 1

[18] D. Gotz and H. Stavropoulos. Decisionflow: Visual analytics for high-dimensional temporal event sequence data. *IEEE Trans. Vis. Comput. Graph.*, 20(12):1783–1792, 2014. doi: 10.1109/TVCG.2014.2346682 3

[19] J. Guan, R. Li, S. Yu, and X. Zhang. A method for generating synthetic electronic medical record text. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 18(1):173–182, 2021. doi: 10.1109/TCBB.2019.2948985 1

[20] S. Guo, K. Xu, R. Zhao, D. Gotz, H. Zha, and N. Cao. Eventthread: Visual summarization and stage analysis of event sequence data. *IEEE Trans. Vis. Comput. Graph.*, 24(1):56–65, 2018. doi: 10.1109/TVCG.2017.2745320 3

[21] S. Jiang, S. Fang, S. Bloomquist, J. Keiper, M. J. Palakal, Y. Xia, and S. J. Grannis. Healthcare data visualization: Geospatial and temporal integration. In *VISIGRAPP (2: IVAPP)*, pp. 214–221, 2016. 3

[22] P. Klemm, K. Lawonn, S. Glaßer, U. Niemann, K. Hegenscheid, H. Völzke, and B. Preim. 3d regression heat map analysis of population study data. *IEEE Trans. Vis. Comput. Graph.*, 22(1):81–90, 2015. 3

[23] T. Kokosi and K. Harron. Synthetic data in medical research. *BMJ Medicine*, 1(1), 2022. doi: 10.1136/bmjmed-2022-000167 1

[24] N. Li, T. Li, and S. Venkatasubramanian. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *Proc. ICDE*, pp. 106–115. IEEE, Los Alamitos, 2007. doi: 10.1109/ICDE.2007.367856 2

[25] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014. doi: 10.1109/TVCG.2014.2346452 2

[26] Z. Liu, Y. Wang, M. Dontcheva, M. Hoffman, S. Walker, and A. Wilson. Patterns and sequences: Interactive exploration of clickstreams to understand common visitor paths. *IEEE Trans. Vis. Comput. Graph.*, 23(1):321–330, 2016. 3

[27] Z. Liu, Y. Wang, M. Dontcheva, M. Hoffman, S. Walker, and A. Wilson. Patterns and sequences: Interactive exploration of clickstreams to understand common visitor paths. *IEEE Trans. Vis. Comput. Graph.*, 23(1):321–330, 2017. doi: 10.1109/TVCG.2016.2598797 2

[28] T. E. Meyer, M. Monroe, C. Plaisant, R. Lan, K. Wongsuphasawat, T. S. Coster, S. Gold, J. Millstein, and B. Shneiderman. Visualizing Patterns of Drug Prescriptions with EventFlow: A Pilot Study of Asthma Medications in the Military Health System. In *Proc. VAHC*, pp. 55–58. ACM, New York, 2013. 2

[29] M. Monroe, R. Lan, H. Lee, C. Plaisant, and B. Shneiderman. Temporal Event Sequence Simplification. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2227–2236, 2013. doi: 10.1109/TVCG.2013.200 2, 3, 4, 8

[30] J. Nielsen. Usability Heuristics. In *Usability Engineering*, chap. 5, pp. 115–163. Morgan Kaufmann, San Diego, 1993. doi: 10.1016/B978-0-08-052029-2.50008-5 3

[31] M. Ozkaynak, O. Dziadkowiec, R. Mistry, T. Callahan, Z. He, S. Deakyne, and E. Tham. Characterizing workflow for pediatric asthma patients in emergency departments using electronic health records. *Journal of Biomedical Informatics*, 57:386–398, 2015. doi: 10.1016/j.jbi.2015.08.018 2, 4

[32] A. Patil, G. Richer, C. Jermaine, D. Moritz, and J.-D. Fekete. Studying Early Decision Making with Progressive Bar Charts. *IEEE Trans. Vis. Comput. Graph.*, 29(1):407–417, 2023. doi: 10.1109/TVCG.2022.3209426 3

[33] A. Perer and F. Wang. Frequence: interactive mining and visualization of temporal frequent event sequences. In T. Kuflik, O. Stock, J. Y. Chai, and A. Krüger, eds., *19th International Conference on Intelligent User Interfaces, IUI 2014, Haifa, Israel, February 24-27, 2014*, pp. 153–162. ACM, 2014. doi: 10.1145/2557500.2557508 2

[34] A. Perer and F. Wang. Frequence: Interactive mining and visualization of temporal frequent event sequences. In *Proceedings of the 19th international conference on Intelligent User Interfaces*, pp. 153–162, 2014. 3

[35] H. Ping, J. Stoyanovich, and B. Howe. DataSynthesizer: Privacy-preserving synthetic datasets. In *Proc. SSDBM*, 5 pages, pp. 1:42–5:42. ACM, New York, 2017. doi: 10.1145/3085504.3091117 1

[36] A. Pister, C. Prieur, and J. Fekete. Combinet: Visual query and comparison of bipartite multivariate dynamic social networks. *Comput. Graph. Forum*, 42(1):290–304, 2023. doi: 10.1111/CGF.14731 5

[37] G. Richer, J.-D. Fekete, and M. Sedlmair. Evaluation for progressive data analysis. In J.-D. Fekete, D. Fisher, and M. Sedlmair, eds., *Pro-*

gressive Data Analysis: Roadmap and Research Agenda, pp. 149–170. Eurographics, Nov. 2024. doi: 10.2312/pda.20242707 9

[38] G. Richer, A. Pister, M. Abdelaal, J.-D. Fekete, M. Sedlmair, and D. Weiskopf. Scalability in Visualization. *IEEE Trans. Vis. Comput. Graph.*, pp. 1–15, 2023. to appear. doi: 10.1109/TVCG.2022.3231230 2, 3

[39] F. Rusu, C. Binnig, and C. Weaver. Data management for progressive data analysis. In J.-D. Fekete, D. Fisher, and M. Sedlmair, eds., *Progressive Data Analysis: Roadmap and Research Agenda*, pp. 33–48. Eurographics, Nov. 2024. doi: 10.2312/pda.20242707 6

[40] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pp. 336–343, 1996. doi: 10.1109/VL.1996.545307 3

[41] Système National des Données de Santé (SNDS). https://www.snds.gouv.fr/. Accessed: 2023-02-01. 1

[42] C. D. Stolper, A. Perer, and D. Gotz. Progressive Visual Analytics: User-Driven Visual Exploration of In-Progress Analytics. *IEEE Trans. Vis. Comput. Graph.*, 20(12):1653–1662, 2014. doi: 10.1109/TVCG.2014.2346574 2, 3

[43] The uci machine learning repository. https://archive.ics.uci.edu. Accessed: 2025-02-23. 3

[44] A. Ulmer, M. Angelini, J.-D. Fekete, J. Kohlhammer, and T. May. A Survey on Progressive Visualization. *IEEE Trans. Vis. Comput. Graph.*, 30(9):6447–6467, 2024. doi: 10.1109/TVCG.2023.3346641 2, 3

[45] A. Vilanova, M. Angelini, S. K. Badam, and J.-D. Fekete. Uncertainty and quality for progressive data analysis. In J.-D. Fekete, D. Fisher, and M. Sedlmair, eds., *Progressive Data Analysis: Roadmap and Research Agenda*, pp. 92–107. Eurographics, Nov. 2024. doi: 10.2312/pda.20242707 3

[46] K. Vrotsou and A. Nordman. Exploratory visual sequence mining based on pattern-growth. *IEEE Trans. Vis. Comput. Graph.*, 25(8):2597–2610, 2019. doi: 10.1109/TVCG.2018.2848247 2, 3

[47] Vue: The Progressive JavaScript Framework. https://vuejs.org/. Accessed: 2023-03-21. 5

[48] J. Walonoski, S. Klaus, E. Granger, D. Hall, A. Gregorowicz, G. Neyarapally, A. Watson, and J. Eastman. Synthea™ novel coronavirus (COVID-19) model and synthetic data set. *Intelligence-Based Medicine*, 1-2(100007), 2020. doi: 10.1016/j.ibmed.2020.100007 1

[49] Q.-R. Wang and R. S. Laramee. EHR STAR: The State-Of-the-Art in Interactive EHR Visualization. *Comput. Graph. Forum*, 41(1):69–105, 2022. doi: 10.1111/cgf.14424 2, 3, 9

[50] K. Wongsuphasawat and D. Gotz. Exploring Flow, Factors, and Outcomes of Temporal Event Sequences with the Outflow Visualization. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2659–2668, 2012. doi: 10.1109/TVCG.2012.225 2

[51] K. Wongsuphasawat, J. A. Guerra Gómez, C. Plaisant, T. D. Wang, M. Taieb-Maimon, and B. Shneiderman. LifeFlow: Visualizing an Overview of Event Sequences. In *Proc. CHI*, pp. 1747—-1756. ACM, New York, 2011. doi: 10.1145/1978942.1979196 2

[52] E. Zgraggen, A. Galakatos, A. Crotty, J.-D. Fekete, and T. Kraska. How Progressive Visualizations Affect Exploratory Analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(8):1977–1987, 2017. doi: 10.1109/TVCG.2016.2607714 2, 3

Supplementary Material

Comparison with EventFlow

We tried the EventFlow program with a subset of our dataset containing 10 k, 20 k, and 50 k synthetic patients. In comparison, ParcoursVis can be used with tens of millions of patients without noticeable latency; the online demo https://parcoursvis.lisn.upsaclay.fr/ uses 10 M patients in a web setting, and we use ParcoursVis with 20 M patients. ParcoursVis supports datasets 100–1,000 times larger. To provide a baseline to better understand the scalability of ParcoursVis, we compare it to EventFlow version v2.3.4 for three important tasks and features on the three dataset sizes. We run the java virtual machine with 40 Gb of memory (`java -Xmx40G -jar EventFlow_v2.3.4.jar`).

Operation	10 k	20 k	50 k
Load	**4 s**	12 s	40 s
Visualize	20 s	85 s	410 s
Hide Event	**4 s**	16 s	96 s
Show Event	20 s	77 s	500 s
Merge Event	25 s	80 s	454 s
Memory	13 G	26 G	28 G

- Loading time: EventFlow can download a CSV file in a reasonable time for 10 k and 20 k patients, not more.
- Visualization time: Once loaded, the datasets take quite a long time to appear on screen; the aggregation and layout times are longer than the 10 s limit.
- Hiding and showing back an event are not symmetrical. Hiding takes a relatively short time under 20 k patients. Showing back the event takes a much longer time, always longer than the 10 s limit.
- Interaction time: hovering the Icicle tree is always instantaneous in EventFlow, but above 20k patients, any action that changes the prefix tree takes from 16 seconds to 10 minutes.
- Features: EventFlow lets users interactively specify several parameters and aggregation rules. We measured merging/aggregating consecutive "alphabloc" events. ParcoursVis performs this aggregation in its main loop. It always takes more than 10 s with EventFlow.

We also report the memory usage when running the Java virtual machine. Our measures show that EventFlow cannot be considered interactive above 10 k patients, and even then, many operations take longer than 10 s to complete. In contrast, ParcoursVis can perform all its operations with a latency of under a couple of seconds, even for 20 M patients.

Synthetic Data Generation

All the figures in our article rely on synthetic data. As health records are sensitive by nature, data regulations forbid sharing them outside private and secured platforms, and only to authorized persons. If we want to share our project openly and allow the reproduction of our evaluations, we need to create a dataset statistically realistic while protecting the patients' anonymity.

We use our aggregated tree as a model to produce synthetic patients since it holds a simplified but accurate statistical profile of all our patients. We can indeed replay the probabilities stored in the tree to generate new patients from which, if we aggregate those synthetic patients again, we should get another statistical similar prefix tree.

The tree contains nodes (Listing 4), each node N_i representing a pathway P_i from the root to N_i:

- count: The number of patients who started their treatment with prefix P_i, reaching N_i.
- children: The list of events following P_i. For each event E_j, we export its frequency, i.e., probability of $P(E_j|P_i)$.
- age_distribution: The age distribution of the patients reaching N_i.
- The duration distribution of the last event (treatment) reaching N_i.
- The comorbidity distribution for the patients reaching N_i.

From this tree, we can generate a new patient in three phases: (1) generating a random sequence of high-level events from the tree, (2) adding realistic attributes to the high-level events, and (3) generating a plausible low-level event sequence from the high-level events. While it is common to rely on statistics and machine learning models [17, 19, 23, 35, 48] to generate synthetic dataset, to the best of our knowledge, we are the first ones to rely on a prefix tree to generate synthetic datasets.

Random sequence: We start at the root node that becomes the current node. Iteratively, we chose a next event from the current node with a probability weighted by the event frequencies (e.g., 40% of chance to pick "alphabloc" in the root node of Figure 1a). This next event is appended to our random sequence. We iterate until the next event selected is "end of treatment."At the end of this phase, we have a list of high-level events, and we also keep the list of the current nodes.

Adding Attributes: The nodes created at line 24 of Listing 2 by `getHighLevelEventSequence` contain a name, an age, a duration, and possible comorbidities. We generate the same attributes to our events at this stage. We generate the attributes backward, starting from the last event, associated with the last node that we kept previously. Using the age distribution of the last node, we generate a random age that we set to the last

```
1   struct Node {
2     /* Data needed for ParcoursVis*/
3     unsigned    count; // Number of sequences passing through this node
4     EventType   name; // event name, such as phyto or surgery
5     list<Node*> children; // children nodes
6     Histogram duration_distribution; // duration distribution
7
8     /* Data related to our medical use case. Not useful in our evaluation */
9     Histogram             age_distribution; // age distribution
10    map<Disease, unsigned> disease_count; // distribution of comorbidities
11  };
```

Listing 4 Structure of a tree Node

created event. Same for the duration, and from the frequency of comorbidities, we generate a comorbidity pattern. Then, we proceed backward to the event list and tree nodes to create our events, updating the age according to the generated duration, propagating the comorbidity pattern (we do not change it yet), and picking a random duration again. This process is repeated until we reach the first event. Our high-level sequence is then complete.

Generating Low-Level Events: We first pick a plausible starting date at random for our low-level sequence. From the generated high-level sequence, we split events with duration into a sequence of atomic events (e.g., drug purchases) at regular intervals (15 days for each box of pills). Each low-level event is then given a date. We also remove the synthetic "interruption" and "no treatment" events, updating the date of the next concrete event accordingly. This phase generates a plausible sequence of events for one synthetic patient, similar to the CSV file Table 1.

Our strategy uses the statistical profile of each node; when we generate a large number of synthetic patients and aggregate them, the final tree is very close statistically to the original one. However, our approach does not preserve higher-level statistics, such as correlations or dependencies between different node statistics. Also, we do not model the evolution of comorbidities; for each patient, we randomly select a set according to the last event and keep it during the whole sequence. More work is needed to take into account the appearance and progression of the comorbidities of aging patients.

In the end, our synthetic data allows to showcase our system, but we do not claim to model the patients accurately enough for other purposes.

Anonymization Generating synthetic patients seems to ensure patients' anonymity, but generating a large number of patients may disclose statistical information from the aggregated tree that is too revealing, as an aggregated tree from millions of synthetic patients can recreate small nodes.

To be able to share synthetic data, according to national regulations, we should process the aggregated tree. The regulations are different for different countries. As fully anonymizing the aggregated tree is out of the scope of this article, we instead give readers interested in that topic a few references that discuss the potential threats of attackers, and the solutions to minimize them with the cost of lesser diversity in the aggregated data [8, 16, 24].

Additional Figures

Here, we convey additional Figures related to Section 4.

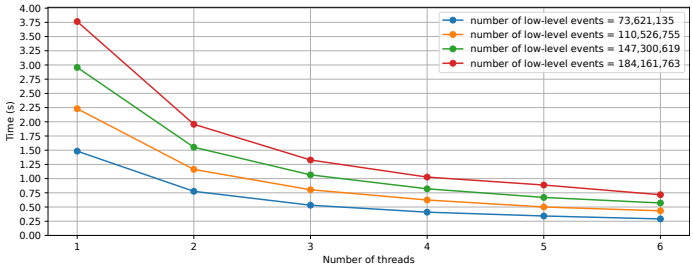

Fig. 11: Total computation time per size of the datasets compared to the number of threads; non progressive processing. The results follow the $\frac{1}{x}$ law; see also Figure 12.

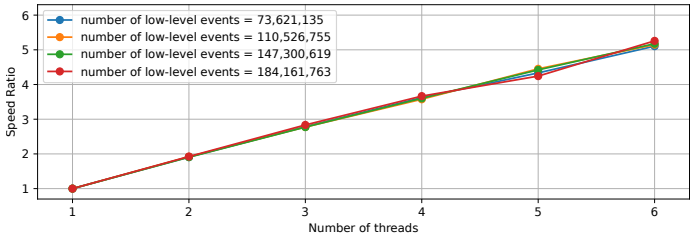

Fig. 12: Computational speed per size of the dataset compared to mono-threaded computations; non progressive environment. The speed increases linearly with the number of CPU cores.

The two following figures are complement to Figure 8 and Figure 9.

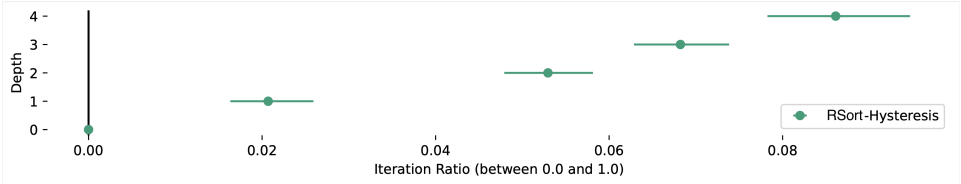

Fig. 13: The 95% CIs, computed by Bootstrapping (BCA), of how fast the *Hysteresis Sort* stabilizes nodes compared to *RSort* by depth.

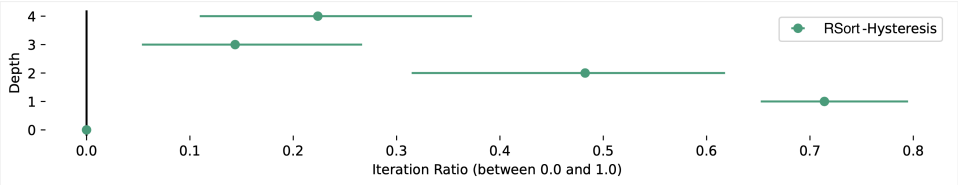

Fig. 14: The 95% CIs, computed by Bootstrapping (BCA), of how fast the *Hysteresis Sort* stabilizes the tree compared to *RSort* by depth.

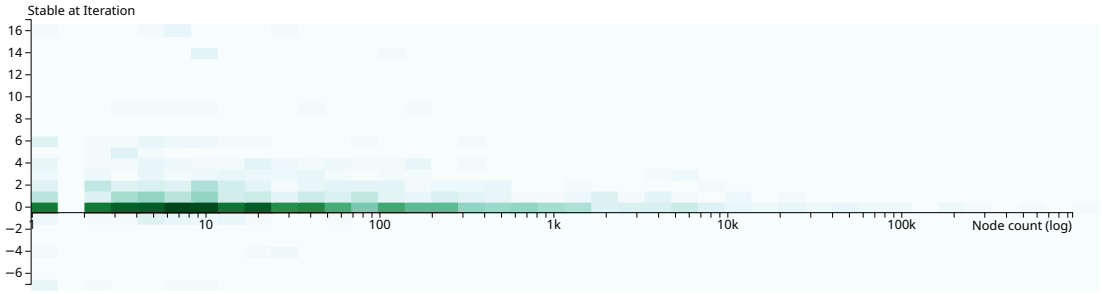

Fig. 15: A heatmap showing how soon (in number of iterations) the *Hysteresis Sort* stabilizes nodes (categorized per their log-scaled frequency) compared to the *RSort*. Positive values along the y-axis means that the *Hysteresis Sort* stabilizes nodes sooner than the *RSort*. The heatmap is computed for *threshold* $= 0$ and *chunkSize* $= 100K$, and considers only nodes whose depths are equal or lower than 4, counting from 0. We use `D3.js` and its BuGn color scale.

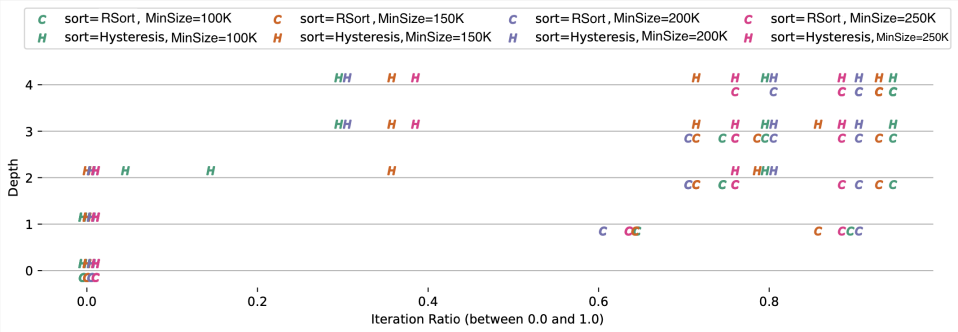

Fig. 16: The stability of the tree by depth and *chunkSize* values. We applied a small offset on the x-axis to distinguish overlapping points of two different *chunkSize* values. Each category has 3 *MinSize* values. The figure does not highlight clusters based on *chunkSize*, suggesting that this variable has no effect on how fast the tree stabilizes itself.

Other Application Domains

ParcoursVis has been designed to visualize health records but could be used for other kinds of event sequence data. We adapted it to visualize the `MSNBC.com` Anonymous Web Data, the largest public dataset mentioned in the related work [46]; it was retrieved from the UCI Machine Learning Repository [43]. It contains 989,925 sequences of website visits, with 18 types of web pages, i.e., event types, and the average number of visits per user is (average sequence length) 5.7.

Converting it to ParcoursVis's format and adapting ParcoursVis took about one hour, mostly to write a script to transform the file format into ParcoursVis's CSV format and specify the colors for the events. The result is shown at Figure 17 and ParcoursVis is usable without noticeable latency. A deeper adaptation to this application domain would require changing the vocabulary shown by ParcoursVis (it still refers to the visitors as "patients" and to the visits as "treatments" in the figure), dealing with time (the dataset does not contain time information), and adapting the details and filter panels to use relevant attributes. The core visualization would remain the same.

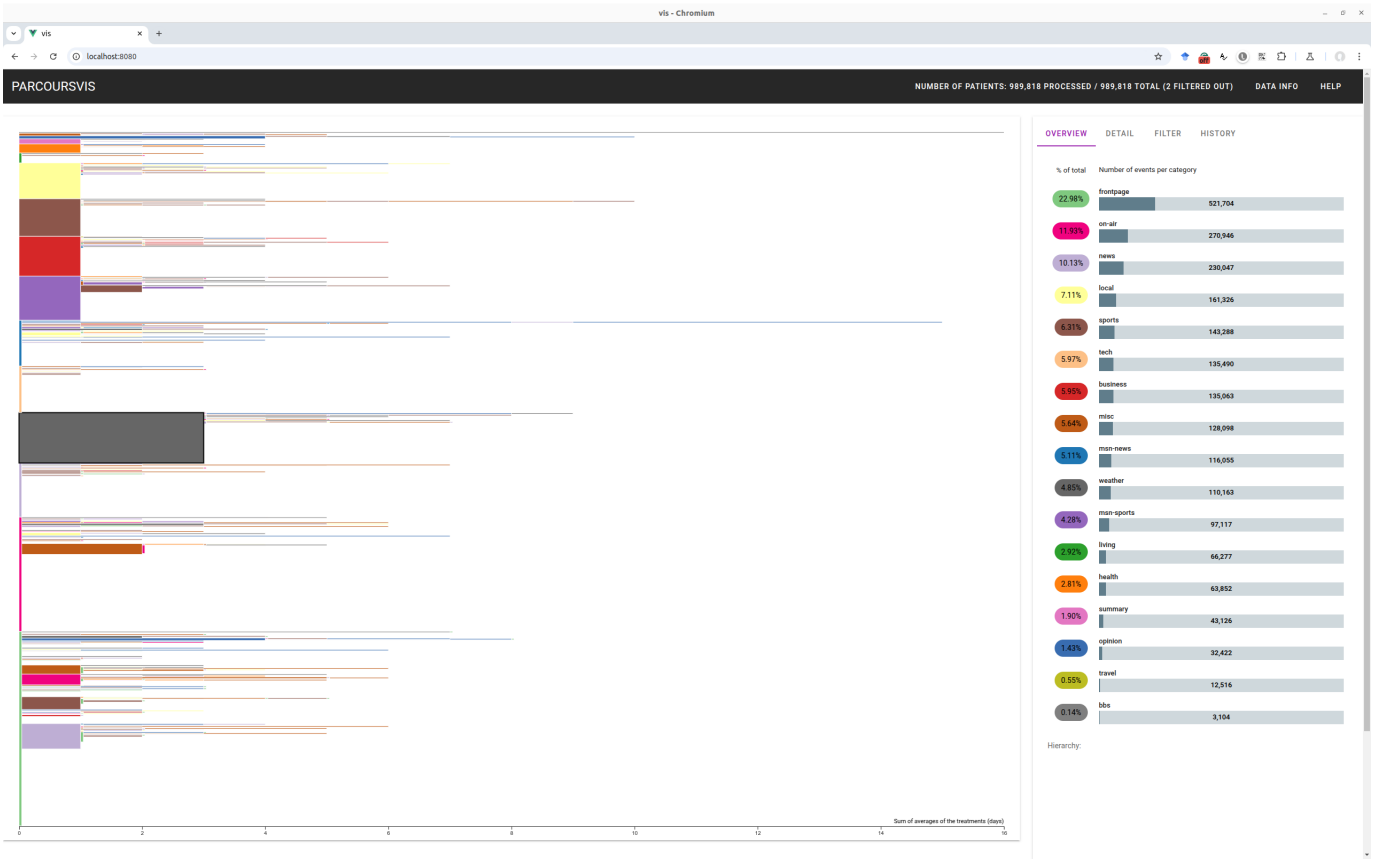

Fig. 17: The MSNBC.com Anonymous Web Site dataset containing about 1 million web page visits visualized with ParcoursVis.