Tailorable Sampling for Progressive Visual Analytics

Marius Hogräfer and Hans-Jörg Schulz

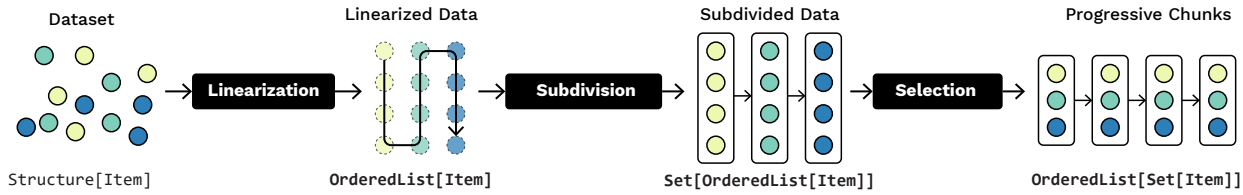

Fig. 1: Enabling tailorable PVA-sampling using a pipeline that structures the sampling process into three steps: linearization, subdivision, and selection. The steps are depicted here along the data they operate on: The linearization takes in the input data structure and transforms it into linear structure, which is then subdivided into bins in the subdivision step. The last step then produces the chunks forwarded into the PVA process by progressively selecting appropriate items from each bin. Different linearization, subdivision, and selection strategies can be combined to progressively sample a given dataset in various ways.

Abstract— Progressive visual analytics (PVA) allows analysts to maintain their flow during otherwise long-running computations by producing early, incomplete results that refine over time, for example, by running the computation over smaller partitions of the data. These partitions are created using sampling, whose goal it isto draw samples of the dataset such that the progressive visualization becomes as useful as possible as soon as possible. What makes the visualization useful depends on the analysis task and, accordingly, some task-specific sampling methods have been proposed for PVA to address this need. However, as analysts see more and more of their data during the progression, the analysis task at hand often changes, which means that analysts need to restart the computation to switch the sampling method, causing them to lose their analysis flow. This poses a clear limitation to the proposed benefits of PVA. Hence, we propose a pipeline for PVA-sampling that allows tailoring the data partitioning to analysis scenarios by switching out modules in a way that does not require restarting the analysis. To that end, we characterize the problem of PVA-sampling, formalize the pipeline in terms of data structures, discuss on-the-fly tailoring, and present additional examples demonstrating its usefulness.

Index Terms—Progressive Visual Analytics, Visual Analytics, Sampling

1 Introduction

Visual Analytics (VA) aims to combine the computational power of modern hardware with the reasoning skills of human analysts for data analysis through visualization, with analysts configuring the analytic computation based on observations of the resulting visualization. To be effective, VA requires that updates after an interaction appear within interactive response rates of around 1*s*, as analysts otherwise lose their analysis "flow" [15]. One challenge to interactivity in VA is the increasing size of datasets, which slows down computation times, thus making VA ineffective. To nevertheless bring the benefits of interactive visual analysis to large datasets, one approach is to split the dataset into smaller partitions and to then run the analysis on those smaller partitions, showing partial results to analysts. This so-called Progressive Visual Analytics (PVA) approach [37] puts analysts "back into the loop" of long-running computations, allowing them to regain the flow. Benefits of this approach have, among others, been highlighted by Zgraggen et al. [42] who show users of progressive systems to clearly outperform those using traditional "blocking" systems. Beyond early insights, additional benefits of PVA include the interactive parametrization of long-running computations by setting parameters on the fly, the ability to steer computations towards data subspaces of interest, early termination (stopping a long-running computation early on), and the ability to observe how otherwise intransparent "black box" computations evolve (see the detailed review by Angelini et al. [4]).

Nonetheless, for PVA to be beneficial, the partial results shown to users need to reflect the final result including any observable patterns therein. The goal for partitioning the data in PVA (which we refer to as PVA-sampling) is thus to *make the visualization as useful as possible as early as possible*. However, we identify two drawbacks in the current approach to partition the data, which limit the effectiveness of PVA.

First, the notion of usefulness in the sampling goal stated above highlights a relation to the analysis task: What data should be in a "useful" sample depends on what analysts are going to do with it. For example, we can consider the synthetic multivariate dataset depicted in Figure 2, containing two normally distributed numeric attributes (encoded as x and y positions) and a Boolean attribute (which is encoded as color). Depending on what task an analyst wants to perform on that data, there are different ways for how to make the sampling of that data most useful: For example, to gain an initial overview of the data, it makes sense to draw a uniform sample that helps depict the distribution of all three attributes. In the sample depicted in subfigure (1), the densely populated region in the center of the plot stands out. On the other hand, to analyze the local distribution of the Boolean attribute, it is more useful to sample the data along a regular grid, such that the density in each grid cell is even throughout the sample, which puts the focus on the Boolean attribute. In subfigure (2), we clearly notice the sharp circular border between the two attribute classes. Another task could involve training a classifier on the Boolean attribute. Then, a stratified sample is more useful, such that both attribute values are evenly represented in the training data, as depicted in subfigure (3). Lastly, analysts may also focus their analysis exclusively on one facet of the Boolean attribute, prioritizing these items in the sample as in subfigure (4). The challenge in all this is that the current state of the art sampling mechanism in PVA remains random sampling, but, as we saw above, there are many cases where this approach impedes the efficiency of PVA, as analysts would need to wait before the results computed over random samples become useful to them. While approaches have been proposed to provide scenario-specific sampling, these generally do not translate well beyond the scope they were designed for, so more tailorable solutions are needed.

• *Marius Hogräfer and Hans-Jörg Schulz are with Aarhus University. E-mail: {mhograefer,hjschulz}@cs.au.dk*

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

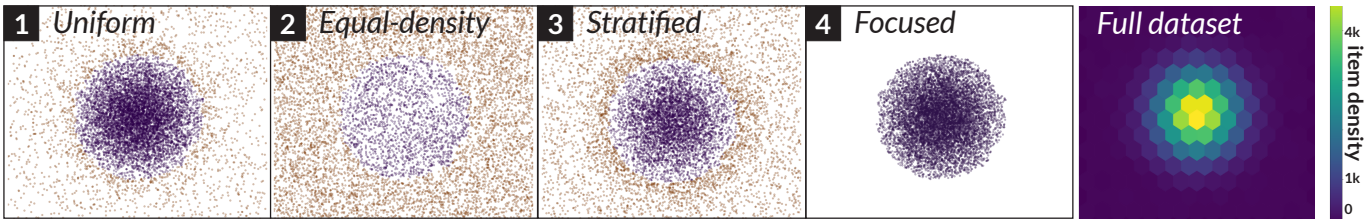

Fig. 2: Depicted is a synthetic dataset containing $100k$ items with two numeric attributes of normal distribution (encoded along the x and y axes) and a Boolean attribute. On the right, the distribution of the x and y attributes is depicted in a binned scatterplot. Four samples ($||sample|| = 10k$) are drawn from this dataset and depicted in subfigures (1) to (4), each of which is tailored to fit a specific analysis task. The Boolean attribute is encoded as the color. Each sample brings out different aspects of the dataset, showing the importance for tailored sampling in PVA.

In addition to the lack of tailorability and in contrast to sampling for regular VA tasks, analysts in PVA can dynamically change the course of their analysis mid-computation, accounting for insights gathered from the partial visualizations rather than restarting the analysis. This means that both the data type *and* the task may change. For instance, an analyst may begin by passively observing the progression over the multivariate dataset from our previous example to get a spatial overview of their dataset, and then move on to analyze items in the center, focusing on one value of the Boolean attribute. In non-progressive VA, making this switch is not an issue, as analysts can reconfigure and rerun the entire analysis whenever their task changes. However, in PVA, the analysis is an ongoing process and, thus, configurations need to take place on-the-fly. This is a challenge for the sampling, as the two tasks in the example above require completely different data: For supporting the overview task, samples should evenly represent the spatial distribution to bring out general patterns in the data, while for the second task, samples should focus on a specific region in the visualization. This shows that the traditional "fire-and-forget" approach to sampling, where the sampling is configured *once* before starting the analysis, no longer applies in PVA, exactly *because* analysts can now interact with that analysis. Yet, existing sampling techniques are generally tailored to specific tasks on specific data types. Thus, when task or data change like in the example above, analysts need to restart to change to a dedicated sampling technique, breaking exactly that flow that the progressive analysis was supposed to ensure in the first place. Thus, a new approach is needed that fits the dynamic demands of PVA.

To this end, we identify two main challenges for the effectiveness of PVA: (1) Current sampling algorithms cannot be tailored to the task, requiring dedicated sampling techniques whenever the standard method of random sampling falls short, and (2) the sampling cannot be adjusted while the computation is ongoing, whenever analysts change their task based on new insights, which reduces the effectiveness of PVA. In this paper, we address these challenges by introducing a new approach to PVA-sampling, which modularizes the sampling process and thereby allows tailoring it to the requirements of tasks and dataset in a way such that this tailoring can also take place without stopping the computation.

This paper expands on our 2022 EuroVA workshop paper [21], in which we first proposed the idea of a sampling pipeline for PVA. Concretely, this extension consists of the following main contributions:

- We added a characterization of PVA-sampling along its unique challenges that clearly distinguishes it from regular sampling.
- We added a formalization of the pipeline by defining its steps as transformations between data structures along the operator design pattern for visualization.
- We added dynamic tailorability as an additional requirement for PVA-sampling and show how the pipeline can be used to this end.
- We added a series of new examples on a real-world dataset, showing how the pipeline enables tailored sampling, but also how it can be used to recreate existing sampling approaches.

The remainder of this paper is structured as follows. First, we characterize PVA-sampling by outlining its unique requirements and distinguish it from related sampling approaches. Then, we introduce the modular sampling pipeline using a running example to demonstrate the effect that each step has on the final sample. Afterwards, we show how the pipeline allows for on-the-fly tailoring of the sampling and discuss additional benefits and limitations.

2 Characterizing PVA-Sampling

PVA supplies analysts with early, partial visualizations that complete over time to bring the benefits of an interactive visual analysis to long-running computations. These partial visualizations can, for example, be produced by showing intermediate results from computations that iterate over the entire dataset, such as k-means clustering. The second approach for creating partial visualizations in PVA – which we address here – is to partition the data into chunks using sampling and to then run the full computation over these chunks. Sampling for PVA (which we refer to as *PVA-sampling* in this paper) is, however, quite different from regular sampling. This is because in PVA, sampling is an ongoing process that runs in parallel to the visual-interactive analysis, while regular, non-progressive sampling is an operation carried out once before the analysis. As a result, regular sampling is a binary selection that determines *if* an item appears in the sample and thus will become part of the analysis, whereas PVA-sampling is a ranking of data that determines *when* a data item becomes part of the analysis – effectively prioritizing some data items to be pushed through the analysis pipeline before others (see Figure 3). This seemingly simple conceptual difference yields three unique requirements for a useful PVA-sample:

The first requirement regards the frequency at which new samples are drawn. PVA-sampling needs to continuously produce samples throughout the analysis, such that new updates to the visualization arrive within human latency limits. Achieving these update rates is one of the main motivations for using PVA, as it allows analysts to maintain their analytic flow even during long-running computations [15]. This stands in contrast to regular sampling, where a single sample is drawn before the analysis, meaning that the human latency limits are not a consideration for the sampling. A side effect of this requirement is that PVA-sampling sometimes produces samples that are too small to be

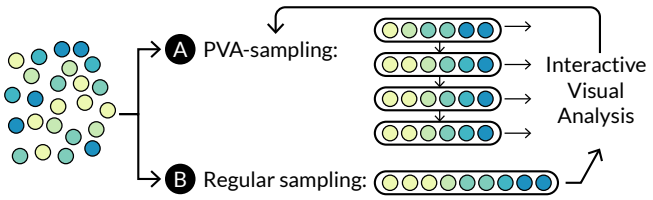

Fig. 3: Demonstrating the difference between PVA-sampling and regular sampling: Regular sampling (B) and the analysis of the data sample happen consecutively: The sample is drawn, and then analyzed. Thus, regular sampling defines for each item, whether it is part of the analysis or not. In contrast, PVA-sampling (A) is a continuous process taking place in parallel to – and potentially also influenced by – the analysis. It draws new samples (so-called chunks) from the full dataset up until the user stops this process or all data has been sampled. Thus, PVA-sampling defines for each item, when it becomes available to be shown and analyzed.

statistically representative. It is *because* the sampling is a process that these samples nevertheless become useful to analysts eventually, once enough data has been sampled.

The second requirement is that priorities are given to different parts of the data as to what to sample first and last. PVA-sampling defines for all items, *when* they are selected, and so all data is eventually part of the analysis – unless, of course, analysts terminate the computation beforehand. Regular sampling for VA, on the other hand, does not prioritize the data, but instead it defines for all items *if* they are selected as part of the sample. All items that are not part of the sample are, therefore, also not part of the analysis. The way in which PVA-sampling prioritizes certain data items depends on what makes the visualization useful as soon as possible (as outlined in the Introduction).

The third requirement for PVA-sampling is the flexibility of adjusting the sampling process while it runs, in order to sync it with the visual analytic process that is concurrently being carried out: as one process changes, so must the other. One side of this dependency is that sampling depends on the analysis, in that the task characterizes what items are useful. But it also means that the analysis is influenced by the sampling, as analysts gain new insights from the partial results in the samples, which again influences their task. For PVA-sampling to be flexible, this means that the process can be dynamically adjusted (i.e., without restarting) to tailor the data prioritization to changing tasks. This is not a requirement for regular sampling in VA: The sampling terminates before the analysis. Thus, in order to tailor it to the task, the sampling step must be rerun to create a more fitting sample.

3 Related Work

3.1 Sampling for non-progressive VA

In section 2, we distinguished PVA-sampling from regular sampling for PVA, in that regular sampling is an operation that concludes before the interactive visual analysis is run. Regular sampling for VA is commonly used to reduce the complexity of large datasets in cases where approximate results are as useful as seeing the full picture [23]. This can be desirable for many reasons, including reduction of computation time for complex analyses [23] and also clutter reduction in view space [14]. In contrast to PVA-sampling, only one "final" sample is used to conduct the analysis. Therefore, it is generally important that this sample is statistically representative of the dataset, so that the insights gathered from it also apply to the rest of the data. Olken and Rotem provided an early survey of sampling methods in 1990 [27] and many more sampling algorithms have been proposed since then. To the best of our knowledge, what they have in common, though, is that tailorability beyond a particular analysis scenario is generally not considered.

Another related method is active learning [36], where the goal is to find a "best" training set, i.e., a subset of the data for which a model performs best, even when compared to training it on all data. Rather than selecting "clear-cut" items far away from the decision boundary that clearly belong to a class, active learning aims to sample borderline items for which the prediction certainty is low. The goal is to sharpen the decision boundary around those edge cases, as those have a stronger impact on its performance. Active learning thus tries to make a model as accurate as possible as early as possible.

There are other examples of sampling processes, where the analysis and the sampling are iterative, in that the sampling is adjusted based on insights gained from the sample, and vice versa. One method similar – not only in name – to PVA-sampling is so-called progressive sampling, where an increasingly larger sample is drawn until a quality metric computed with that sample is reached or until a metric no longer improves. Usually, progressive sampling is used to reduce model training times by determining the smallest necessary training sample from a large dataset, beyond which the prediction accuracy no longer (noticeably) improves [32]. Starting from an appropriate initial sample size [18], the model is fully retrained on a *progressively* larger sample. Progressive sampling shares some similarities with PVA-sampling in that it aims to make a model as useful as possible as soon as possible.

In contrast to PVA-sampling, though, neither progressive sampling nor active learning are intended to eventually sample the entire dataset, and neither is intended to produce samples at interactive rates required for PVA.

Another approach using sampling as a process is streaming sampling, where the goal is to run an analysis over a potentially infinite data stream. One of the major challenges here is to maintain the "ground truth" data the sample is drawn from. For example, the well-established reservoir sampling method [39] produces a uniform sample over all data that has been processed so far while only requiring to keep a sample in memory. A more recent example is the approach by Losing et al. [25], who use clustering to summarize the data stream to a set of representatives. Similar to PVA-sampling, sampling of data streams is a continuous process, yet analysts may not end up seeing *all* data as only some new elements are being selected for the sample.

3.2 Sampling for PVA

Prior work has investigated some aspects of PVA-sampling as characterized in section 2. The standard sampling approach in PVA literature is arguably random sampling without replacement [5, 22, 24, 38, 42], as PVA is often proposed as an interactive method for the overview task. Tailored approaches have also been proposed. Most recently, Chen et al. presented sampling for progressive scatterplots [9] using three criteria to define a useful sample: preserve temporal coherence between successive samples, preserve the relative density and outliers, and achieve sufficient efficiency to retrieve samples within the latency constraints of PVA. Another approach is the work by Turkay et al., who introduced a method for adapting the size of the sample dynamically, such that the visualization is updated within a specified interval [38]. A common consideration for PVA-sampling is to reduce the error in the partial visualization. One example is the work by Rahman et al. who present an algorithm that prioritizes salient features in treemaps and line charts [33], or Sample+Seek [13] and BlinkDB [2] – two sampling approaches that reduce and bound errors and response times of certain query types on large datasets. Another example is the selective wander join method proposed by Procopio et al. [31] that addresses the challenges of sampling for queries containing data joins that also filter the data, achieving interactive sampling speeds in these cases.

The diversity of these techniques illustrates the benefit of (and need for) having tailored sampling algorithms, yet, it also shows that existing sampling techniques use dedicated, custom implementations as reusing parts of approaches to transfer them to other scenarios is generally not considered. Moreover, exchanging the sampling method mid-analysis is generally not discussed. This is why we propose a sampling pipeline for PVA, which modularizes the sampling process, allowing to tailor it to the requirements of an ongoing analysis, while increasing reusability of parts of the sampling process across scenarios.

4 A Pipeline for Tailorable PVA-Sampling

Here, we introduce our tailorable sampling approach for PVA using a pipeline. We first derive the steps of the pipeline from input and output structures and then discuss each step in detail, demonstrating their impact on the sampling with a running example.

4.1 Modularizing PVA-sampling along data structures

Pipelines are widely used in visualization to make complex processes modular and tailorable, for instance, the visualization pipeline [7], the data state reference model [10], or the Operator Pattern formulated by Heer and Agrawala [19]. Our pipeline draws from these ideas to enable "flexible and reconfigurable" output [19]. Concretely, we apply the Operator Pattern to PVA-sampling in that each step of the pipeline is a module "that performs a specific processing action, updating the contents of the [sample] in accordance with a data state model" [19]. Each step of the pipeline applies a transformation on the input dataset analysts are working with, transforming it to a specific output structure (i.e., the data state model in the above citation), finally leading to a series of data chunks. In relying on the operator pattern, we can make the complex PVA-sampling process tailorable. It also allows us to represent transformations of different complexity, as well as intermediary operations that operate within one step (i.e., data state). Tailoring the sampling then means modifying these transformations used along the

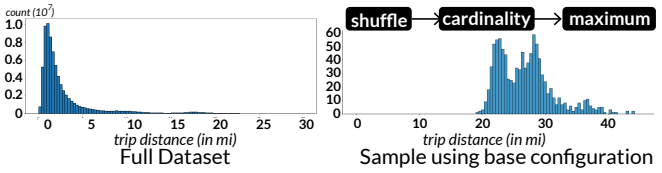

Fig. 4: Left: Distribution of the trip distance attribute in the full dataset used in the running example, showing a clear spike in short distance taxi rides. Right: The distribution of the same attribute in a sample produced with the base pipeline ($||sample|| = 10k$), showing a noticeable shift in the distribution towards longer trip distances.

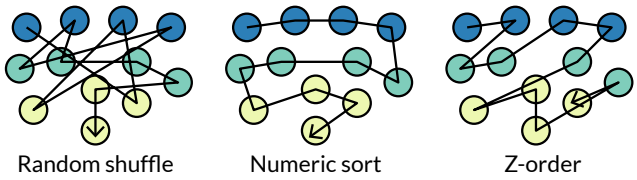

Fig. 5: Three examples for linearizing the same dataset: Random shuffling, sorting by a numeric attribute, and sorting spatially in z-order.

pipeline, and because transformations conform to the same input and output structure, we increase reusability between scenarios.

From a high-level perspective, PVA-sampling generally transforms the input dataset into ordered chunks. These chunks are then forwarded to the PVA process, i.e., sampling is generally positioned ahead of the analytical processing step in the data state reference model [24]. We can describe that input dataset as an arbitrary structure defined over a set of items $Structure[Item]$. We purposefully do not prescribe a particular data type like table or graph here to keep the sampling pipeline independent from them, and we also keep the structure of *Item* arbitrary for the same purpose. On the other end of the process, the chunks produced by the sampling are a list of subsets of the input dataset $OrderedList[Set[Item]]$. These chunks – that is, each $Set[Item]$ – are disjoint and for every item in the dataset, the sampling assigns a position in exactly one chunk. A PVA-sampling method is a function that transforms data from this input to that output structure, and the sampling pipeline P must therefore conform to the following high-level structure:

$$P : Structure[Item] \rightarrow OrderedList[Set[Item]]$$

In order to make the complex, monolithic transformation P tailorable, we modularize it into three steps: *linearization*, *subdivision*, and *selection*. In the first step, the data is put into linear order, which is then subdivided into bins, from which the chunks are then assembled in the last step, selecting the most appropriate item(s) from each bin. The data structures and steps are summarized in Fig. 1.

Running Example: We introduce each step of the pipeline using a running example, demonstrating the effect of different operators at a particular step on the final sample. Concretely, we sample the 2018 Yellow Taxi trip dataset[1] of taxi rides in New York City. This dataset contains around 112 Million items, each representing a taxi ride along numeric (e.g., trip distance), categorical (e.g., pickup and dropoff zone codes), and temporal (e.g., pickup and dropoff time) attributes. For illustrative purposes, we enriched this dataset with geospatial attributes for pickup and dropoff locations, by generating random locations in the polygons belonging to each zone code. Its size makes this dataset a clear candidate for a progressive analysis, and along its diverse attributes there are many interesting patterns to explore, allowing us to illustrate tailorable sampling. On that data, we use the following "base" pipeline as a running example: *random* linearization → *cardinality* subdivision → *maximum* selection. The linearization strategy puts the data in random order, the subdivision splits it up into bins of equal size, and the maximum strategy selects taxi rides with the highest values along the trip distance attribute. his pipeline yields an overview of long taxi rides in the data already from the first chunk before filling in data on shorter taxi rides coming in later chunks. Thus, this pipeline supports analysts unfamiliar with the dataset, who are interested in quickly developing an idea for what characterizes such long taxi rides (in contrast to prioritizing "the longest"), for example, in terms of their spatial distribution. The ground truth distribution (for practicality, we use the first $11M$ lines, i.e., 10%, of the data) as well as the distribution yielded by the base pipeline are depicted in Figure 4. In the examples, we show how to further tailor the sampling by making adjustments to this base setup at every step. Specifically, we demonstrate the impact on the distribution inside the *first* sample of $10k$ items using a particular pipeline and report the runtime for each operator.

To this end, we implemented the pipeline as a proof-of-concept in Python 3.9, using numpy (version 1.22.2)[2] and pymorton (version 1.0.5)[3]. Based on this implementation, for each strategy we also report its processing times computed on standard laptop hardware (intel core i7-8550U with $1.6 - 3.4GHz$ on $16GB$ of RAM) to indicate their impact on the analysis. The code of our implementation, the data preprocessing we applied, as well as notebooks for reproducing the figures included in this section can be found on Github[4].

4.2 Linearization: Putting the data in order

In the first step, we linearize the data into the standardized structure $OrderedList[Item]$. This is necessary, as the data type influences the way it can be processed (i.e., a graph dataset needs to be treated differently than tabular data). Thus, by harmonizing the data into the linear list structure at the first step, we allow the rest of the pipeline to be largely independent of the input data type, while also increasing the reusability and flexibility of downstream operators.

How to appropriately linearize the dataset depends on the analysis scenario. The harmonization aspect is mostly influenced by the data structure. Because of the simplicity of linear lists, there often exist (multiple) linearization algorithms for a particular dataset, which means that as long as we can find a fitting linearization algorithm for our data, our pipeline supports it. Hierarchical data, for example, can be linearized using traversal strategies like depth-first search, geospatial data can be linearized using space-filling curves like the Hilbert [12] or z-order curve [44], and graphs can be linearized along their shortest path using a traveling salesman heuristic [28].

Once in linearized form, we can further reorder the items based on its attributes. Depicted in Figure 5 are three examples of strategies for sorting data based on their numeric values. The first strategy shown there is *random shuffling*, which puts the data in random order. Shuffling is widely applicable, as it does not use any item-driven metrics to sort the data, but instead makes as little assumptions about the data as possible. This also makes shuffling a good option whenever analysts' interest in the data is unclear, or whenever analysts want to overcome unwanted sorting biases in the data. The second strategy depicted is *numeric sorting*, which as the name implies, orders the data by a numeric attribute of interest. Sorting means that similar items appear after one another in linearized order, which makes it easier to tailor the sampling to patterns found along that attribute, for example by sampling for similar values or by prioritizing outliers. For spatial data, we can use space-filling curves to put it in order, for example in *z-order* as depicted in the third example. Analyzing data based on spatial proximity is a common requirement in the analysis of spatial and geographic data, so space-filling curves allow tailoring the sampling accordingly.

Example: Depicted in Figure 8 are the effects of these three linearization strategies on the base pipeline. In the first configuration, we want to make as little assumptions about the underlying data in the linearization which is useful for analysts unfamiliar with a dataset, thus we

[1] https://data.cityofnewyork.us/Traasportation/2018-Yellow-Taxi-Trip-Data/t29m-gskq

[2] https://numpy.org/

[3] https://pypi.org/project/pymorton/

[4] https://vis-au.github.io/prosample

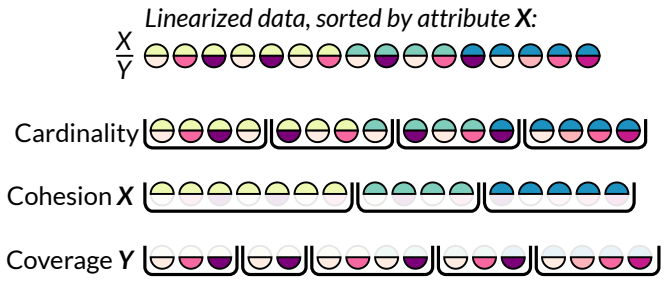

Fig. 6: Examples for subdivision strategies over the same, two-dimensional linearized data: cardinality (splitting into equal-sized bins), cohesion (splitting at greatest differences in successive values, along the same attribute as used for the linearization), and coverage (splitting every time min and max values are found, along a different attribute than as used for the linearization).

use *random* shuffling in the linearization step. Comparing the resulting sample to the ground truth by the trip distance distribution, we notice a clear skew towards larger values. This makes sense, as every item has the same probability for appearing in a subdivision bin, and therefore, when selecting maximum values from each bin, the sample will contain mostly larger values. To focus the analysis on the spatial distribution of the trip distance attribute, we can use the *z-order* linearization resulting in items to appear after one another in the linearized data if their ride pickup locations are located close to each other. Thus, the subdivision bins created from this linearization are spatially ordered, and again, we see that the depicted sample distribution is skewed towards larger values, suggesting that longer rides are distributed in geospace instead of being concentrated in particular areas. In our third configuration we *sort* the data by the attribute of interest, which means that the bins created by the subdivision contain items of similar values. The effect on the distribution is that the sample largely resembles the ground truth distribution of trip distance, meaning that the sample is *not* skewed towards larger values. This configuration is useful when analysts want to prioritize other parts of the dataset while ensuring a representative distribution of the trip distance attribute.

4.3 Subdivision: Splitting the data into bins

To construct useful chunks from this now sorted, but otherwise difficult-to-query data, we next define a scenario-specific structure on top of this list by splitting it up into bins. We express these bins as $Set[OrderedList[Item]]$, where each list $OrderedList[Item]$ is a section of the linearized data, keeping the order defined by the linearization step. While this order may be relaxed by the subsequent selection step, prescribing it here allows for some tailored selection strategies that utilize it (e.g., selecting the *first* item in each set). Rather than already producing the chunks at this stage, this intermediate step instead constructs a "search structure" of the data that we then query for the most relevant data at a certain point of the analysis in the next step. Essentially, we take a "divide-and-conquer" approach to the search problem for useful items, shifting from a global scope to the scope of smaller bins.

There are many ways in which one may subdivide the linearized data. The two extreme approaches are to put all data into one bin or every item into its own bin. The former can be used to model a sequential read, taking the first n items from the bucket at every chunk, while the latter allows to directly query the data, for instance to find the top-k largest values. For all cases in between, we need to define a criterion for comparing consecutive items that subdivides the data in a desired manner. In Figure 6, we outline three strategies. One simple criterion is to divide the data by *cardinality*, subdividing it in regular intervals so that every bin contains the same number of items. This strategy makes little assumptions about the underlying data, making it a good fit for cases where analysts are unfamiliar with a dataset or where they want to explore it. For cases where the user interest is more clearly defined, we can, for example, increase the *cohesion* within bins by subdividing the linearized data whenever we measure a large difference between successive items. This provides us with a subdivision where each bin contains a set of similar items, which in turn allows us to tailor the sampling based on that similarity criterion.

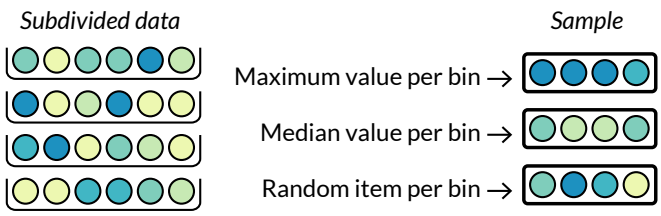

Fig. 7: Examples of selection strategies used on on the same subdivided data: Maximum value, median value, and selecting a random element.

Yet, measuring the similarity may not always be possible, desirable, or useful and so another strategy is to increase the *coverage* over an attribute of interest. This means that, rather than making the items similar to each other per bin, we create bins that have similar statistic properties. This allows controlling for the probability of selecting certain values from a bin. In the example in Figure 6, we create bins for successive pairs of min and max values along an attribute in the data.

Example: Exchanging the subdivision strategy has a noticeable effect on the distribution in the samples we draw, as depicted in Figure 9. Dividing the data by *cardinality* (here $||bins|| = 1k$), subdivides the data into bins that evenly subdivide the linearized (i.e., randomly shuffled) data. Thus, a "dense" region in the input where the linearization strategy finds many similar values are spread out over many bins, while "sparse" regions are packed into a few. The linearization used in the base pipeline, however, randomly distributes the data, so each bin potentially contains the entire value range for all attributes. Since we are selecting the maximum value from each bin, the distribution depicted in the figure is skewed towards larger values for this case.

Looking at the *cohesion* strategy in the second plot, we can see how the distribution differs from the first. This strategy further skews the sample towards larger values, because we can utilize two characteristics of our scenario: the trip distance attribute contains relatively few large values, and these values are randomly distributed because of the shuffling linearization. As a result, large values are likely to lie close to smaller values in the linearized data, and therefore, this subdivision creates new bins for the largest values in the data. This allows analysts to prioritize extreme values.

Likewise, the distribution yielded by the *coverage* strategy in the third plot is also skewed towards larger values. Bins here are created by matching pairs of successive upper and lower values (0.05 and 0.95 quantile), which means that the rare, large values are also likely to appear in many bins, and therefore are likely to appear in the sample. As a result, the two samples yielded by the cohesion and coverage strategies in this configuration are rather similar.

4.4 Selection: Placing items into chunks

In the third and last step, we then construct chunks by selecting the most relevant items from the subdivided data using a scenario-specific prioritization strategy. Following the high-level input and output structures of progressive sampling, this step outputs the chunk structure $OrderedList[Set[Item]]$. In the selection step, the goal is to construct a useful sample of the entire dataset by selecting the most appropriate items per bin from the subdivided data.

Given what data analysts are interested in, they can choose an appropriate selection strategy. Three examples are depicted in Figure 7. Analysts interested in extreme values, for example, may choose the *maximum* strategy, which selects the largest value of an attribute of interest. Analysts who want to get an overview over the dataset may select the *median* element from each bin to get a representative element, or select items randomly to increase the spread of the data. For cases where the user interest is not clear, yet, analysts can use the *random* strategy, which does not consider the values of the data but

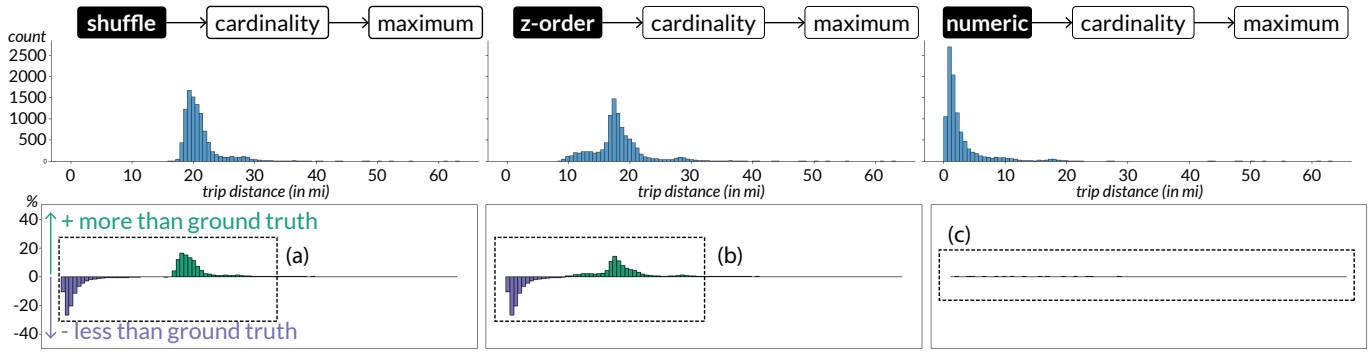

Fig. 8: Impact of the linearization strategy on the sampling ($||sample|| = 10k$). Depicted in the first row is the distribution along the trip distance attribute for *random shuffling*, *sorting with a z-order curve* along the pickup location attribute, and *sorting numerically* by the trip distance. The second row shows the relative difference in % compared to the distribution in the full dataset. We notice how the pipelines using shuffling and z-order linearizations produce samples that are clearly skewed towards larger values (a, b), while the numeric linearization maintains the original distribution (c). Timings on $11M$ items: shuffle $202.56s$, numeric sort $192.65s$, z-order $583.5s$.

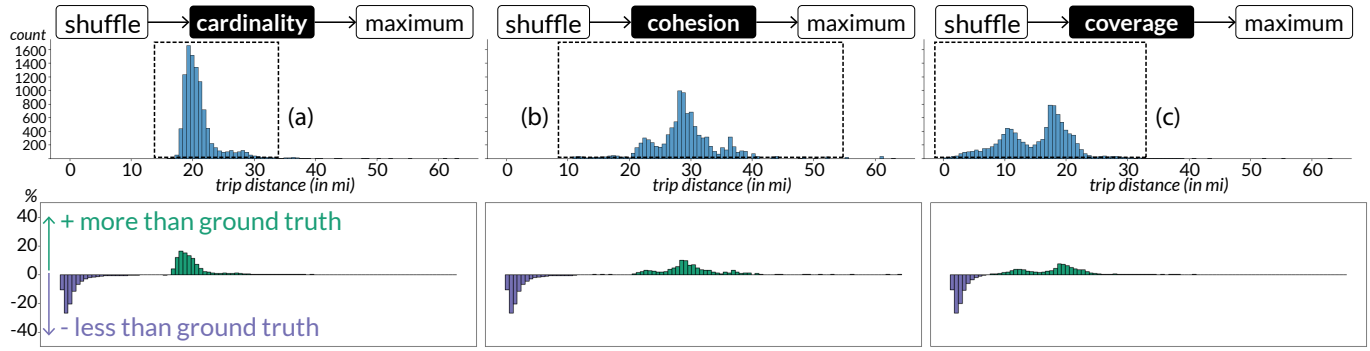

Fig. 9: Impact of the subdivision strategy on the sampling ($||sample|| = 10k$). Depicted in the first row is the distribution along the trip distance attribute for subdivision by *cardinality* ($||bins|| = 1,000$), by *cohesion* (splitting at the top $1,000$ biggest differences), and by *coverage*, with the latter two considering the trip distance attribute. The second row shows the relative difference in % compared to the distribution in the full dataset. We can clearly see all three samples being skewed towards larger values, with interesting differences in which values become most frequent: In the cardinality case, most values are found around $20mi$ (a), cohesion around $30mi$ (b), and coverage produces peaks at 10 and $20mi$ (c). Timings on $11M$ items: cardinality $0.028s$, cohesion $2.514s$, coverage $29.017s$.

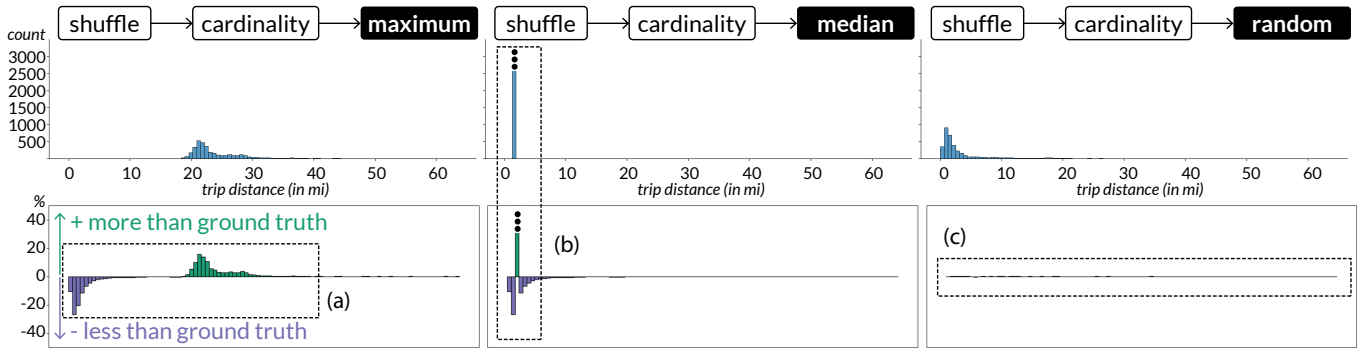

Fig. 10: Impact of the selection strategy on the sampling ($||sample|| = 10k$). Depicted in the first row is the distribution along the trip distance attribute for selecting the *maximum value*, the *median value*, and selecting *randomly*. The second row shows the relative difference in % compared to the distribution in the full dataset. We can clearly see how the selection step affects the sample: The maximum selection skews the sample towards larger values (a), while the median strategy selects mostly small values (b), since most taxi rides in the data are short distance. The sample produced with random selection mostly stays true to original distribution (c). Timings on $11M$ items: maximum $0.411s$, median $0.32s$, random 2.172

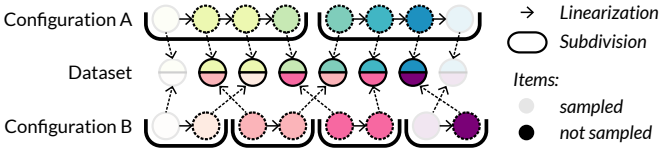

Fig. 11: By storing pointers to the original input data, the pipeline configuration can be tailored at runtime using precomputed linearization and subdivision structures.

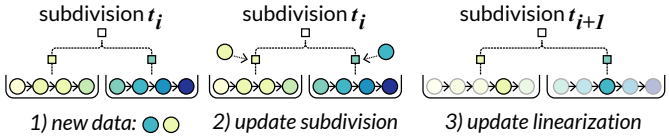

Fig. 12: An incremental variant of the pipeline concept, showing how using dynamic data structures for representing linearized and subdivided data allows including new data to the pipeline, thus allowing to incrementally update the sampling on-the-fly without accessing the entire dataset.

picks elements randomly from each bin.

How many items to select per chunk also depends on the analysis scenario. One heuristic is to select as many items as possible, while still ensuring that the progressive computation produces results within interactive response times [16], which can even be dynamically adjusted to account for fluctuations in recent computation steps [38]. Another option is to always select a fixed number of items per bin, thus guaranteeing a fixed chunk size in the computation.

Example: When comparing the effects of changing the selection strategy on the base pipeline, we notice clear differences in the distribution of the trip distance attribute in the plots in Figure 10. Overall, the selection strategy arguably provides the most "direct" way of controlling the output distribution in our example: Selecting the *maximum* value skews the distribution of the trip distance attribute in the sample towards larger values, selecting the *median* prioritizes average items (since a vast majority of taxi rides in New York City are relatively short distance, this strategy yields mostly short trips), and picking *randomly* means that we approximate the global distribution of the attribute. Thus, depending on the task, we can find clear use for all three: When looking for extreme values, analysts should choose a min/max strategy, while analysts interested in getting a representative sample may either select representative points from each bin using the median strategy, or get a representative distribution of the overall dataset instead by selecting randomly.

5 Tailoring PVA-Sampling on-the-fly

PVA is an inherently dynamic process, in that the analysis task may change while the computation is ongoing, the input dataset may grow over time, or analysts may want to prioritize different subspaces of interest in the data as their analysis progresses. Rather than having to restart the analysis to adjust the sampling (and thereby breaking the flow of the analysis [15]), the sampling should be tailorable dynamically. In this section, we demonstrate how the modular architecture of the pipeline allows us to do just that, showing how it accounts for changes in task, dynamic input data, and changes in scope.

5.1 Dynamically tailoring to changing tasks

In contrast to non-progressive VA, the task a user performs on the data in PVA may change mid-analysis as new insights arise from the partial results. Analysts in the *progressive explorer* role in particular, who use PVA to "gain a comprehensive understanding of the data and process" [26] may repeatedly switch their task. For example, an analyst may at first want to gain a general overview of the data, to then move on to more specific tasks like analyzing outliers. This switch clearly influences how the pipeline should be configured, as different parts of the data become useful: To gain an overview on the taxi data, a pipeline for random uniform sampling (e.g., *shuffle* → *cardinality* → *random*) is appropriate, whereas when targeting outliers, the sampling should prioritize rare items (e.g., *z-order* → *cohesion* → *maximum*). In fact, dynamically changing the task effectively may require exchanging all parts of the pipeline at any point in time. The challenge here is that – as with the long-running computation on the data that is made interactive by the sampling in the first place – the complexity of the data increases the complexity of carrying out the pipeline steps.

Both steps, the linearization and subdivision, access the entire dataset and, depending on the complexity of the chosen strategy, exchanging (recomputing) them can cause noticeable delays in the analysis. One way to nevertheless enable dynamic exchange is by precomputing combinations of linearization and subdivision operations, thus paying the computation cost ahead of the analysis. To account for changes in the analysis scenario at runtime, we can then simply use either of the precomputed data structures as input to the selection step (see Figure 11). For example, by storing pointers to the input dataset, we can keep track of which data has been sampled so far, even when exchanging the subdivision. For this to be viable, however, analysts need to consider how much precomputation is "worth it". When gaining a first overview of a new, unfamiliar dataset, this pre-processing may in fact *not* be worth the wait, as in these cases one pipeline may suffice. Yet for analysts who often analyze the same dataset multiple times, having a catalog of linearizations and subdivisions to tailor their sampling can be particularly useful.

In contrast, the selection step can be evaluated on a per-chunk basis. Rather than naïvely running it exhaustively over the entire dataset to preemptively set a chunk for each item, we can instead construct the next chunk on-demand, significantly reducing its complexity. This way, we essentially use the selection step as a query over the subdivided data, allowing for efficient on-demand retrieval. It also means that we can dynamically exchange the selection operation at runtime by simply changing what query we use here.

5.2 Dynamically tailoring to incremental input data

Next, we look at how the pipeline can accommodate changes in the input dataset. Up until now, we took an "upstream", global perspective on PVA-sampling, where sampling is positioned at the beginning of the PVA process with access to the full dataset, and where all other operators in the PVA process wait for the chunks produced by it. This perspective is also rather common in the literature [24,37]. However, as noted by Schulz et al. [34], it falls short in capturing dynamics in PVA. This is because different operators along the PVA process have different requirements to the input data, with some operators like clustering requiring a broader data coverage than a progressive scatterplot. Thus, using only one sampling ahead of the process is often insufficient. The solution proposed by Schulz et al. is a buffer/sequencer model for incremental visualization, where each operator gets to manage their own priority queue. To integrate dynamic input data with the pipeline, we essentially need to make the sampling itself incremental, such that whenever the input dataset of the pipeline changes, the sampling can reflect that as well. In other words, we need to incrementally compute the linearization, subdivision, and selection steps.

To make the first two steps of the pipeline incremental, we can simply maintain incremental data structures for linearized and subdivided data, into which we can insert new data and from which we can remove processed data. For example, for an incremental linearization of tabular data, we can use a sorted index structure like a binary search tree over $List[Item]$. Whenever the input data changes, we can efficiently remove or insert those items using that index, and in effect make the linearization incremental. Similarly, we can adopt incremental subdivisions by using data structures like incremental segment trees [41]. A tree structure is compatible with $Set[OrderedList[Item]]$, in that leaf $Items$ are grouped by inner nodes of the tree, and this structure can be also efficiently updated incrementally. For complex subdivision operations like 1-dimensional clustering, dedicated incremental algorithms can be utilized [35]. Lastly, as discussed in the previous section, the selection step already runs on-demand per chunk, rather than exhaustively over

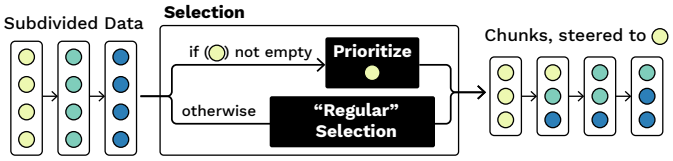

Fig. 13: Integrating computational steering with the sampling pipeline: Computational steering can be expressed as an extension of the selection step. This selection prioritizes a subspace of interest by retrieving that data first and then applies the "regular" selection strategy on the remaining data.

the entire dataset. Thus it is inherently fit for incremental updates.

5.3 Dynamically tailoring to changing scope

A big advantage when progressively analyzing data is that analysts can steer the computation towards data subspaces that currently interest them, while that computation is still ongoing. In other words, the scope of the analysis scenario can change dynamically (from the entire dataset to a subspace of interest). Steering generally means to prioritize data inside a user-selected region of interest in the sampling while the analysis is running, retrieving other data later. As a result, the visualization of that subspace is "completed" earlier, allowing for more certain decisions on the data in this particular subspace, making steering a powerful mechanism for rapidly exploring emerging patterns in the visualization. An example of this is a *progressive searcher* [26] zooming into a specific area of interest on a density plot to see details about the region that interests them: The computation can then focus on data that lies inside the zoomed-in region, rather than spending resources on data that lies outside of it. Below, we discuss how steering can be integrated into the pipeline (see also Figure 13).

Sampling during steering differs from regular sampling in multiple ways. One difference is that chunks during steering are no longer samples of the entire dataset, but they are instead skewed towards the subspace of interest. When steering the progression, the chunks analysts see may exclusively contain data from that subspace, while after the steering, i.e., once that subspace is exhausted, items from that subspace will not appear in chunks at all. Another difference is that steering is situation-dependent, in that interesting subspaces arise *while* the computation is running, and analysts may change what subspace they steer towards multiple times during the same progression (see the progressive observer role [26]). As a result, steering can rarely be configured before the sampling starts, unlike the conditions for representative sampling.

In the context of PVA, it is therefore necessary to be able to model steering as part of the pipeline in order to leverage the full potential of PVA. We can do so by considering the data structure that steering operates on. Effectively, steering reorders the items in the sampling, such that items from the subspace of interest appear in early chunks, while the remainder of the data is sampled afterwards. In terms of the data structure in the pipeline, steering is, thus, a transformation with the structure $OrderedList[Set[Item]] \rightarrow OrderedList[Set[Item]]$. This means that we can model steering as part of the selection.

One way to achieve this integration is to focus the selection step on a single bin or at least a subset of bins that contain interesting data. A requirement here is, however, that the data is linearized and subdivided by a suitable similarity criterion, such that bins group data that are similar, making interesting items appear in similar bins. In our running example from section 4, for instance, an analyst may be interested in taxi rides that take place around midnight. If the data is subdivided into hourly intervals, we can steer the progression by selecting items only from the respective bins from that interval.

In cases where the similarity criterion does not match the user interest, we can integrate steering by adjusting the selection strategy used for each bin. For example, to prioritize taxi rides around a certain time of day over all bins, we can select items within that time interval from bins that contain such items, and otherwise select items that are as close as possible to that interval. Thus, we need to adjust the selection strategy *per bin* based on its data. To achieve this steering, we can maintain simple descriptive metrics for each bin (such as min/max/mean per attribute) and then switch the selection strategy for each.

An even more dynamic option to integrate steering is to combine the previous two approaches, adjusting the number of items selected per bin based on whether it contains items or not (as outlined in Figure 13): Then, the selection greedily selects all items from the subspace of interest from all bins, until the chunk is "full" or the subspace is exhausted. Again, this can be achieved in a rather straightforward manner by maintaining descriptive metrics for all bins, this time basing the number of selected items on them.

Thus, depending on how accurate the steering should be and how much effort is viable, the sampling pipeline supports these needs. As a result, any tailored sampling pipeline following the input and output structures outlined in Sec. 4 can benefit from steering. Second, we can likewise integrate any steering mechanism with the pipeline as long as it supports the structure of the selection step. This means that, regardless of how exactly a subspace is prioritized (e.g., based on a one-to-one mapping [11], iterative rebinning [40], derived from decision trees [20]), it can be combined with the sampling pipeline through the query filter it defines over the remaining data.

6 Utilizing the Pipeline for Tailored Sampling

Having formalized task-tailorable PVA-sampling into a modular pipeline, we next provide examples of how this pipeline can be used to benefit analysts, again considering the taxi dataset for reference.

6.1 Recreating existing samplings

By formalizing PVA-sampling along a pipeline, our goal is not to replace nor outperform scenario-specific samplings, but we want to supplement them. In the previous section, we showed how analysts can configure custom samplings with the pipeline. Here, we want to exemplify another advantage of the pipeline, which is that we can also use it to recreate existing sampling approaches in terms of linearization, subdivision, and selection strategies. The idea is that, whenever the qualities of a particular sampling algorithm are required, they can not only be expressed in the pipeline structure, but also further be tailored and adjusted, *because* they are in the pipeline structure. To demonstrate this, we model well-known sampling algorithms with our pipeline, showing their output in Figure 14.

- **random sampling**: random shuffling linearization → cardinality-based subdivision → random selection.
- **stratified sampling**: numeric sort-by-attribute linearization → interval-based subdivision → median selection.
- **sampling for balanced spatial autocorrelation**: z-order linearization → cardinality-based subdivision → balancing autocorrelation selection.

This highlights the expressiveness of our sampling pipeline, in that we can create both simple approaches like random sampling, but also rather specialized approaches as in the spatial autocorrelation example. This sampling controls the distribution of four categories (called LL, HL, LH, and HH), which express whether a local value is greater than neighboring values (yielding H* or L* categories) and the global mean value (yielding *H or *L categories) of a spatial variable. This is inspired by the approach of Zhou et al. [45], who demonstrate that sampling using spatial autocorrelation allows for effective exploration of large (and therefore often cluttered) geospatial datasets.

6.2 Recomposing sampling pipelines

Modular design reduces implementation efforts by increasing reusability of partial solutions, in that we can compose the operators from existing sampling methods to create a new approach. For example, we can recompose operators from other pipelines in this section into a new sampling approach. In the example depicted in Figure 15, which is inspired by z-order sampling proposed by Zheng et al. [43], we use the z-order linearization from the autocorrelation sampling, a cohesion-based subdivision, and the median selection from stratified sampling.

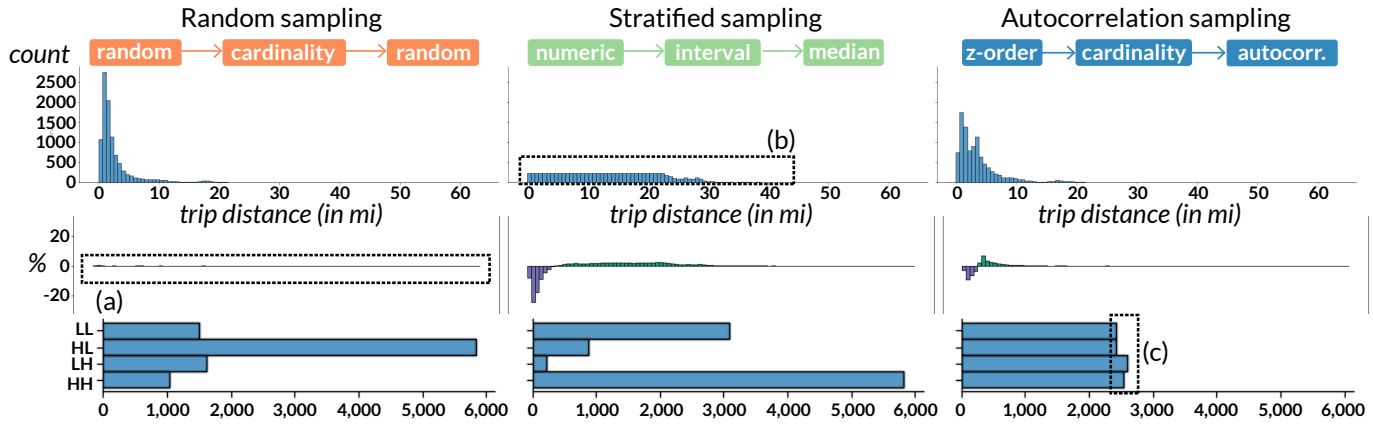

Fig. 14: Examples of existing sampling methods recreated using the pipeline: Random sampling, stratified sampling, and spatial autocorrelation sampling. (a) Random sampling maintains the distribution along the trip distance attribute, while stratified sampling purposefully samples the value range evenly (b). Autocorrelation sampling on the other hand ensures that the autocorrelation categories are equally distributed in the sample (c).

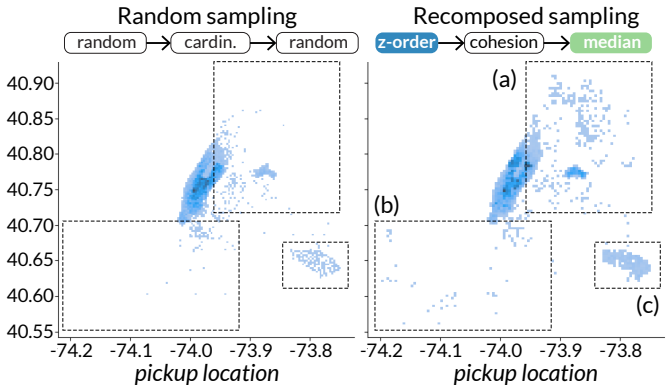

Fig. 15: Example of tailored sampling created by recomposing the operators used in Figure 14 and Figure 16 for a distinct sampling on the taxi dataset ($||sample|| = 10k$). The tailored sampling depicted on the right noticeably preserves sparse regions (a), outliers (b), as well as local structures (c) compared to the random sample on the left.

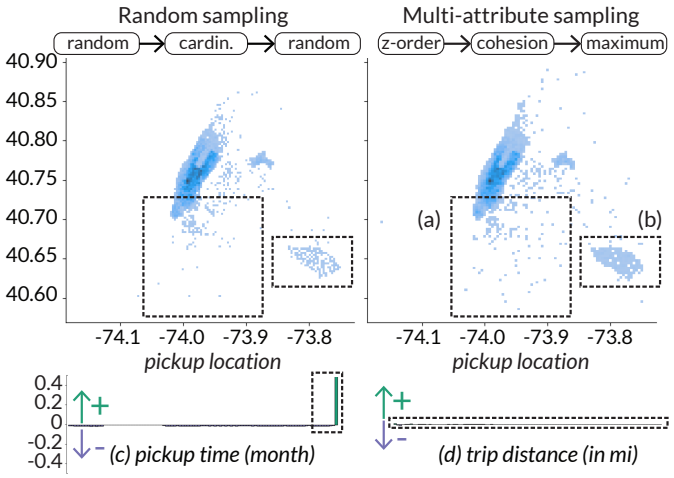

Fig. 16: Example of sampling tailored to multiple attributes on the taxi dataset ($||sample|| = 10k$). The pickup location in the tailored sample better maintains both outliers (a) and dense regions (b) than a random sample, albeit not as clearly as in the sampling tailored *only* to the spatial distribution in Figure 15. The histograms show that the sampling is also clearly skewed to pickup times late in the year (c), while the distribution of the trip distance matches the overall distribution in the dataset (d).

Linearization and subdivision both tailor the sampling towards the spatial location of the data points, in that the linearization places points in successive order if they are close to each other in the pickup location attribute, and the subdivision splits up this data whenever there is a large distance between values. This results in bins that contain items from distinct regions in view space, and their cardinality depends on the spatial density of that region: densely populated areas are contained in large bins, while sparse regions are contained in small bins. In turn, when selecting elements from all bins, this increases the visibility of outlier points in the sample, while maintaining sparse regions as well as dense patterns in the sample, which is clearly visible in Figure 15.

This demonstrates the modularity benefits of the pipeline, which allows reusing existing modules for tailored sampling rather than requiring a completely new implementation.

6.3 Tailoring the sampling towards multiple attributes

The modularity also allows us to independently tailor each step of the pipeline to account for a different aspect of the analysis scenario, covering complex analysis scenarios. As an example, we here configure a sampling tailored for analysts exploring the relationship between the spatial distribution of long taxi rides in December of 2018. Accordingly, we configure a pipeline tailored towards three attributes at the same time: The linearization sorts the data along a z-order curve over the pickup location, the subdivision increases coverage over the trip distance attribute, and the selection picks the maximum value along the pickup date attribute. The linearization ensures that items within the same bin are also located close to each other in geospace, which helps to preserve outliers and dense regions in the sample when selecting items from all bins. The subdivision then ensures that all bins contain both long and short trips, which ensures that this distribution is maintained in the sample. The selection then skews the sample towards trips that are latest in the year. The effect is visible in Figure 16.

This example demonstrates the flexibility of the pipeline in two ways: First, we can consider multiple data *attributes* in the sampling. This means that the sampling can be tailored to address more complex analysis scenarios, controlling the sample distributions for more than one attribute. The second benefit is that we can consider multiple data *types* in the sampling. This is noteworthy, as many existing sampling algorithms are geared towards one particular data type (sampling for spatial data, temporal sampling, sampling multivariate data, ...).

7 Discussion

Here, we consider aspects of integrating tailorable sampling into analyses: integrating it into real-world database systems, integrating it as part of the analysis itself, and integrating it from a usability perspective.

7.1 Implementing tailored sampling into DBMS

We produced the examples in this paper using a proof-of-concept implementation written in Python, demonstrating our pipeline's utility. A logical next step is to consider if and how our pipeline can be integrated with existing database systems in order to drive real-world analyses. To this end, we here explore a promising use case of doing so, namely ProgressiveDB [6], a middleware that enables progressive queries on existing SQL-based databases like MySQL or Postgres. ProgressiveDB works by rewriting incoming SQL queries into a set of smaller queries that can be completed within human latency limits, which is achieved internally by relying on the partitioning support of the underlying database, or by defining a dedicated sampling column and index. Currently, ProgressiveDB only supports random uniform sampling. Below, we discuss how our pipeline can in principle be integrated into ProgressiveDB requiring minimal additional modifications. To integrate the three steps of the pipeline with ProgressiveDB, we can take advantage of features of the underlying SQL database as well as features provided by ProgressiveDB.

First, linearizations can be precomputed on the dataset as indexes. Modern SQL databases support a variety of data types for creating complex indexes, supporting numeric, temporal, textual, and spatial attributes, thus, inherently offering a variety of linearization strategies for tailoring the sampling. Nevertheless, specialized linearizations such as sorting by z-order may only be supported by some databases (e.g., Apache Impala). Moreover, building indexes over large datasets takes time (e.g., 360*s* to define an index on the trip distance attribute in the taxi dataset), requiring analysts to trade-off flexibility and utility.

Next, for the subdivision step, we can take advantage of the internal partitioning feature of modern database management systems, which assigns rows to partitions based on column values. Partitioning divides large tables into separate files. It is originally designed to optimize storage of datasets across file systems. Yet, ProgressiveDB utilizes it to reduce the response time per query to human latency limits, consolidating the responses of multiple queries into one approximate result. Thus, similar to the subdivision step, we can tailor the sampling by adjusting the partitioning mechanism. Like the linearization step, partitioning can, however, be time consuming on large datasets.

Lastly, we can implement the selection step by utilizing the steering feature of ProgressiveDB, which defines so-called progressive views. Progressive views perform the query on the partitions, aggregating their results, similar to how the selection strategies pick the most relevant elements from each bin in the subdivision, and they can be dynamically defined during the analysis. In ProgressiveDB, we can express different selection strategies via standard SQL operators like `SELECT`, `WHERE` and `ORDER BY` clauses. In addition, as discussed in subsection 5.3, this setup allows us to still maintain the ability to steer by dynamically switching to another view containing the steering filters. Another added benefit is that ProgressiveDB automatically supplies uncertainty information for each completed chunk (called "confidence").

Overall, such an integration would utilize the features provided by ProgressiveDB, therefore we expect it to perform comparable to ProgressiveDB in terms of retrieval and pre-processing times. However, while this implementation would benefit from the optimizations and scalability of modern database systems, the performance bottlenecks of this implementation clearly lie in the pre-computation necessary for building indexes and partitioning the data. These are well-known limitations when working with massive tabular databases and not unique to tailorable sampling. A common practice to alleviate them is to run these lengthy pre-processes over night when only few or no users are connected to the database at all, in order to benefit from the added performance during the day, when most traffic is expected. Nevertheless, the requirement of additional pre-computation is a clear limitation of our pipeline, which we further discuss in subsection 8.1.

7.2 Tailored sampling as data analysis method

An interesting observation from applying the pipeline concept to PVA is that it enables analysts to tailor the sampling to explore the data, whereas previously, sampling served to prepare such analyses. For instance, one takeaway in the example in subsection 4.2 was that longer rides appeared to be spatially spread out throughout the dataset, or in subsection 4.3, it was observed that the sample was "skewed" towards larger values. These insights were gained from adjusting the sampling, indicating that our pipeline provides analysts with a novel means of exploring massive datasets. Comparable to steering approaches like Sherpa [11], which allow focusing the computation on a data subspace of interest, tailorable sampling gives analysts another means of controlling the data flow towards desired data, thus, making the partial visualization more expressive of the desired information. In contrast to non-progressive VA, tailorable sampling shifts the data exploration step ahead in the analysis pipeline: Where configuring the sampling previously took place *before* the analysis, in PVA the sampling *is part of* it, in the sense that chunk-wise provisioning of the raw data is already influenced by the analysis to be performed. This characteristic of PVA only became apparent after applying tailorable sampling in practice, indicating the need for more empirical work studying how it influences analysts' workflows in practice, for instance.

7.3 Usability aspects of tailored sampling

The added flexibility of tailorable sampling inherently introduces challenges about how to integrate it with analysis workflows to the analysts' benefit, rather than impeding their work with added complexity. One way of tying the pipeline configuration to the analysis scenario is via the visualization technique used to display the data, as it gives an indication as to what aspects in the data analysts are interested in, i.e., parts of the data that the sampling should bring out. For example, analysts using a map to display the data are likely investigating spatial properties of the data, which the pipeline can support via a *z-order* linearization. Single-attribute visualizations like bar charts and histograms can be supported by tailoring the sampling towards that attribute, e.g., *sorting* the data by attribute and dividing it into bins that increase *cohesion*. Trellis plots that show subsets of the data by splitting it along an attribute in multiple, side-by-side plots could be supported by increasing *coverage* over that attribute in the subdivision. Multi-attribute visualizations like scatterplots can benefit from ensuring a uniform sampling, e.g., via a *random* selection operation. Then, tailoring the sampling can take place whenever analysts change their visualization; either automatically by switching to pre-defined operators per visualization technique, or by providing analysts with a selection of adjustments to choose from. Furthermore, to inform changes to the pipeline, a preview of the effect that different options have on the visualization can be provided (see subsection 8.3). Considerations like these arise from the unique features of PVA and there is obviously more to explore in this regard; in particular the usability and user experience aspects of such an approach requires dedicated empirical studies. Our pipeline lays the groundwork for more work in this direction.

8 Implications and Limitations

While the pipeline provides analysts with new flexibility during their sense-making process, it also requires additional effort and, therefore, impacts the efficiency of the analysis, i.e., the difference between the cost of using it compared to its potential gains. We discuss this impact below along three phases: before, during, and after the sampling.

8.1 Impact before sampling

Regardless of whether analysts use our pipeline or not, configuring the sampling has an impact on the efficiency before starting the analysis. This is because analysts, either way, need to reflect on how to bring out interesting patterns in the data that are relevant to their task, to then choose an appropriate sampling that achieves this. In contrast to the non-progressive case, our pipeline allows analysts to fine-tune this decision later on, reducing the time spent before seeing the data and, thus, improving the efficiency. However, the pipeline also requires that analysts first familiarize themselves with the pipeline steps and their strategies, which in turn reduces the efficiency for pipeline-novices. To reduce these initial configuration cost, it is beneficial to provide pre-configured pipelines that fit many common analysis scenarios such as the random sampling pipeline that supports overview tasks. This way, analysts do not need to start "from scratch" each time and only

need to make small adjustments to their needs. Yet, conversely, even when providing a catalogue of pipelines to choose from, analysts need to decide whether a particular sampling actually fits their analysis scenario. While these costs are higher in the beginning, when analysts first familiarize themselves with the pipeline concept, they remain even for expert users. While in subsection 7.3 we provide first considerations based on the visualization technique to alleviate the configuration (see subsection 7.2), there is still a clear need for guidance based on other aspects of the analysis scenario (e.g., dataset, task, analysis role [26]).

Another important consideration for the efficiency before using the pipeline is whether it makes sense to invest time and effort into enabling tailorability, rather than using an "out-of-the-box" sampling that cannot be tailored. This is because the pipeline also imposes a temporal cost on the analysis. In particular providing the flexibility of multiple linearization and subdivisions requires time- and storage-space-consuming pre-processing. This delay in the analysis is a clear limitation of the pipeline concept: For quick one-off analyses, it may not be worth to ensure tailorable sampling. Moreover, for scenarios for which the qualities of an existing PVA-sampling (see subsection 3.2) suffice, it can make sense to rely on these "off-the-shelf", optimized solutions rather than recreating them with the pipeline steps. Yet, PVA is often geared towards expert users – the progressive explorer described by Micallef et al. [26] – that continuously analyze a massive dataset, investigating patterns, testing hypotheses on-the-fly, dynamically switching tasks, i.e., scenarios where investing pre-processing time upfront makes sense as it increases the scope of the analysis. It is exactly these complex analysis scenarios that our pipeline promises to be most beneficial in and, as we discussed in subsection 7.1, the impact of lengthy pre-computations can be reduced by modern database systems. At the same time, there is a clear need for guidelines and support systems to suggest or otherwise inform analysts' decision on which operators to pre-compute based on their analysis scenario (see subsection 7.3).

8.2 Impact during the sampling

In addition to costs for the initial configuration, utilizing the pipeline to its full potential — dynamically tailoring the sampling during the analysis — requires further effort. For example, analysts need to constantly consider, whether the pipeline currently used for sampling still fits their analysis scenario, and then make the necessary adjustments; costs that were simply not necessary when using one pre-configured sampling throughout. An important consideration to this end is automating the sampling configuration as indicated in the previous section, where a change in the visualization technique triggers changes to the pipeline. Yet, as with guidance for the initial configuration, there are many additional factors of the analysis scenario to consider as input parameters driving useful guidance for "in-situ" automated configurations. Another question concerns the degree of guidance [8], that is, whether to inform analysts that there may be better options, suggesting changes, or to automatically make changes to the pipeline, not informing analysts at all, allowing them to focus on the data.

A central assumption in PVA is that the quality of the partial visualization (e.g., in terms of progress, stability, and certainty [3]) increases over time. Its certainty in particular — the amount of error by which the actual, final result could still deviate from the current result [3] — influences what tasks can be effectively performed: While early (very uncertain) visualizations support passive tasks like observing the data, later (very certain) visualization support active insight generation and verification [4]. It is therefore crucial for analysts to be aware of the certainty when working with partial visualization, i.e., during decision-making on incomplete results. Error estimation methods, which supply the metrics for the certainty of results, are often bound to the variance characteristics of the sampling method [2, 17, 33]. In other words, the certainty across the partial visualization will increase over time, with the certainty in some subspaces increasing faster than others, depending on the sampling that is used. However, when switching sampling methods on-the-fly, the rate at which the certainty increases for a subspace changes. This is obviously an important consideration for tailorable sampling, and analysts must be (made) aware of it while tailoring the pipeline, constantly calibrating their sense-making process to the current sampling method to avoid biases [30].

Fig. 17: Three ways of integrating a tailorable focus sampling with a static sampling providing the context of the data into a scatterplot.

Another point to note here is that many error estimation methods assume a continuous, fixed sampling throughout the analysis, which clearly raises the question of how to estimate the error at all when the sampling can be switched out at all times. This question is not unique to tailorable sampling as it, for example, also relates to systems that allow steering the progression, that is, to temporarily focus the sampling on subspaces of interest. Dedicated research is necessary for measuring and reducing the error of error estimation techniques in PVA and to handle fluctuations in the estimates after tailoring the sampling.

8.3 Impact after sampling

After a tailored sample is drawn, we need to avoid that analysts draw false conclusions from the incomplete results. The related literature has evaluated useful approaches for directly encoding the result uncertainty into the visualization [29, 30]. Beyond these technique-specific solutions, another way to remedy the effects of tailored and, therefore, potentially skewed samples is to combine the chunks of tailorable sampling with a "baseline sampling" that remains constant throughout the analysis. A similar idea is used in BlinkDB [1], where both a uniform sample and a set of stratified samples are maintained. Here, combining multiple samples provides "tighter approximation errors" and "significantly reduces [...] the subset error". Similarly, to avoid analysts misinterpreting the certainty of results after tailoring the sampling, we can combine multiple PVA samplings generated by different pipelines. For example, a "focus" sampling that is dynamically tailored by the analyst to decrease the uncertainty for data of interest can be combined with a "context" sampling that remains static throughout the analysis. This ensures that globally, certainty constantly increases across the dataset, yet it also reduces the impact of tailoring.

There are different ways of encoding these two samplings in the visualization, and we outline some preliminary ideas here (depicted in Figure 17). The first way is to not encode their differences, that is, not differentiating them in the visualization by simply merging the two samples into one. This approach introduces the least complexity into the visualization, but means that analysts need to constantly monitor uncertainty themselves. To make the impact of the tailored sampling more apparent in the visual encoding, we can also encode each sample differently. In a scatterplot, for example, we can assign different colors, allowing analysts to assess their differences, e.g., in terms of their distribution or their stability between chunks. A third approach is to directly encode their difference into the visualization. This allows analysts to focus on data produced by the tailored focus sample, while being able to assess to what degree parts of the visualization are prioritized or neglected by it. An example of this can be found in the interface of ProSteer [20], where a direct encoding of sampling differences was used to assess the utility of a steering algorithm.

Overall, tailored sampling also impacts the uncertainty of progressive visualization, and analysts need to be made aware in order to calibrate their analysis tasks to the result completeness. Empirical work is necessary to evaluate the discussed mitigation strategies.

9 Conclusion

In this paper, we introduced the notion of tailorable PVA-sampling, which differs from "regular" sampling in VA, in that it is a continuous process rather than a computation step, leading to a unique set of requirements. Tailorable PVA-sampling allows to fit the sampling mechanism to the task, to make the progressive visualization as useful as possible as early as possible. We achieved this by providing a pipeline consisting of three consecutive modules (linearization, current needs of the analyst. We demonstrated the flexibility of this pipeline

in a series of examples, both taking a step-by-step perspective, where we demonstrate how exchanging each module affects the output, but also by taking a holistic view, showing how the modularity allows recreating existing sampling methods, reusing operators, and tailoring to complex user interests at once. We then showed how the pipeline can be dynamically exchanged at runtime, to account for highly dynamic user interests common to PVA, changes in the input data, and the scope of the analysis. Our approach allows, for the first time, to tailor PVA-sampling to the needs of the analyst without requiring dedicated reimplementations, while also allowing for adjusting the sampling on-the-fly without restarting the analysis. It lays the groundwork for future research in this direction, including library support and empirical evaluations, as well as automated approaches to reduce user effort.

Acknowledgments

We thank Jakob Burkhardt for his work on the implementation in early stages of the project, as well as Helwig Hauser and Marc Streit for our fruitful discussions on the topic. We also thank our anonymous reviewers for their insightful feedback on the paper. We gratefully acknowledge funding of this research by the Innovation Fund Denmark (IFD) through the Grand Solution project *Hospital@Night*.

References

[1] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: interactive queries on very large data. *Proc. of VLDB*, 5(12):1902–1905, aug 2012. doi: 10.14778/2367502.2367533 11

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proc. of EuroSys*, pp. 29–42. ACM, 2013. doi: 10.1145/2465351.2465355 3, 11

[3] M. Angelini, T. May, G. Santucci, and H.-J. Schulz. On Quality Indicators for Progressive Visual Analytics. In *Proc. of EuroVA*, pp. 25–29. Eurographics Association, 2019. doi: 10.2312/eurova.20191120 11

[4] 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 1, 11

[5] S. K. Badam, N. Elmqvist, and J.-D. Fekete. Steering the Craft: UI Elements and Visualizations for Supporting Progressive Visual Analytics. *Computer Graphics Forum*, 36(3):491–502, 2017. doi: 10.1111/cgf.13205 3

[6] L. Berg, T. Ziegler, C. Binnig, and U. Röhm. ProgressiveDB: Progressive Data Analytics as a Middleware. *Proc. VLDB Endow.*, 12(12):1814–1817, 2019. doi: 10.14778/3352063.3352073 10

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision To Think*. Academic Press, 1999. 3

[8] D. Ceneda, T. Gschwandtner, T. May, S. Miksch, H.-J. Schulz, M. Streit, and C. Tominski. Characterizing Guidance in Visual Analytics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):111–120, 2017. doi: 10.1109/tvcg.2016.2598468 11

[9] X. Chen, J. Zhang, C.-W. Fu, J.-D. Fekete, and Y. Wang. Pyramid-based Scatterplots Sampling for Progressive and Streaming Data Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):593–603, 2022. doi: 10.1109/TVCG.2021.3114880 3

[10] E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proc. of Symposium on Information Visualization*, pp. 69–75. IEEE, 2000. doi: 10.1109/INFVIS.2000.885092 3

[11] Z. Cui, J. Kancherla, H. C. Bravo, and N. Elmqvist. Sherpa: Leveraging User Attention for Computational Steering in Visual Analytics. In *Proc. of VDS*, pp. 48–57. IEEE, 2019. doi: 10.1109/VDS48975.2019.8973384 8, 10

[12] I. Demir, C. Dick, and R. Westermann. Multi-Charts for Comparative 3D Ensemble Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2694–2703, 2014. doi: 10.1109/TVCG.2014.2346448 4

[13] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proc. of SI*, pp. 679–694. ACM, 2016. doi: 10.1145/2882903.2915249 3

[14] G. Ellis, E. Bertini, and A. Dix. The Sampling Lens: Making Sense of Saturated Visualisations. In *Extended Abstract Proc. of CHI*, pp. 1351–1354. ACM, 2005. doi: 10.1145/1056808.1056914 3

[15] N. Elmqvist, A. V. Moere, H.-C. Jetter, D. Cernea, H. Reiterer, and T. J. Jankun-Kelly. Fluid interaction for information visualization. *Information Visualization*, 10(4):327–340, 2011. doi: 10.1177/1473871611413180 1, 2, 7

[16] J. Fekete and R. Primet. Progressive Analytics: A Computation Paradigm for Exploratory Data Analysis. *arXiv.org preprint*, 1607.05162, 2016. doi: 10.48550/arXiv.1607.05162 7

[17] D. Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *Proc. of LDAV*, pp. 73–80. IEEE, 2011. doi: 10.1109/LDAV.2011.6092320 11

[18] B. Gu, B. Liu, F. Hu, and H. Liu. Efficiently Determining the Starting Sample Size for Progressive Sampling. In *Proc. of ECML*, pp. 192–202. Springer, 2001. doi: 10.1007/3-540-44795-4_17 3

[19] J. Heer and M. Agrawala. Software Design Patterns for Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, 2006. doi: 10.1109/TVCG.2006.178 3

[20] M. Hogräfer, M. Angelini, G. Santucci, and H.-J. Schulz. Steering-by-Example for Progressive Visual Analytics. *ACM Trans. Intell. Syst. Technol.*, 13(6):96:1–96:26, 2022. doi: 10.1145/3531229 8, 11

[21] M. Hogräfer, J. Burkhardt, and H.-J. Schulz. A Pipeline for Tailored Sampling for Progressive Visual Analytics. In *Proc. of EuroVis Workshop on Visual Analytics (EuroVA)*, pp. 49–53. Eurographics, 2022. doi: 10.2312/eurova.20221079 2

[22] J. Jo, S. L'Yi, B. Lee, and J. Seo. ProReveal: Progressive Visual Analytics With Safeguards. *IEEE Transactions on Visualization and Computer Graphics*, 27(7):3109–3122, 2021. doi: 10.1109/TVCG.2019.2962404 3

[23] B. C. Kwon, J. Verma, P. J. Haas, and C. Demiralp. Sampling for Scalable Visual Analytics. *IEEE Computer Graphics and Applications*, 37(1):100–108, 2017. doi: 10.1109/MCG.2017.6 3

[24] J. K. Li and K. Ma. P5: Portable Progressive Parallel Processing Pipelines for Interactive Data Analysis and Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1151–1160, 2020. doi: 10.1109/TVCG.2019.2934537 3, 4, 7

[25] V. Losing, B. Hammer, and H. Wersing. KNN Classifier with Self Adjusting Memory for Heterogeneous Concept Drift. In *Proc. of ICDM*, pp. 291–300. IEEE, 2016. doi: 10.1109/ICDM.2016.0040 3

[26] L. Micallef, H.-J. Schulz, M. Angelini, M. Aupetit, R. Chang, J. Kohlhammer, A. Perer, and G. Santucci. The Human User in Progressive Visual Analytics. In *Proc. of EuroVis Short Papers*, pp. 19–23. The Eurographics Association, 2019. doi: 10.2312/evs.20191164 7, 8, 11

[27] F. Olken and D. Rotem. Random sampling from database files: A survey. In *Proc. of SSDBM*, pp. 92–111. Springer, 1990. doi: 10.1007/3-540-52342-1_23 3

[28] M. Onus, A. Richa, and C. Scheideler. Linearization: Locally Self-Stabilizing Sorting in Graphs. In *Proc. of the ALENEX Workshop*, pp. 99–108. SIAM, 2007. doi: 10.1137/1.9781611972870.10 4

[29] A. Patil, G. Richer, C. Jermaine, D. Moritz, and J.-D. Fekete. Studying Early Decision Making with Progressive Bar Charts. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):407–417, 2023. doi: 10.1109/TVCG.2022.3209426 11

[30] M. Procopio, A. Mosca, C. Scheidegger, E. Wu, and R. Chang. Impact of Cognitive Biases on Progressive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(9):3093–3112, 2022. doi: 10.1109/TVCG.2021.3051013 11

[31] M. Procopio, C. Scheidegger, E. Wu, and R. Chang. Selective Wander Join: Fast Progressive Visualizations for Data Joins. *Informatics*, 6(1):1–21, 2019. doi: 10.3390/informatics6010014 3

[32] F. Provost, D. Jensen, and T. Oates. Efficient Progressive Sampling. In *Proc. of SIGKDD*, pp. 23–32. ACM, 1999. doi: 10.1145/312129.312188 3

[33] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfield. I've Seen "Enough": Incrementally Improving Visualizations to Support Rapid Decision Making. *Proc. of VLDB Endowment*, 10(11):1262–1273, 2017. doi: 10.14778/3137628.3137637 3, 11

[34] H. Schulz, M. Angelini, G. Santucci, and H. Schumann. An Enhanced Visualization Process Model for Incremental Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(7):1830–1842, 2016. doi: 10.1109/TVCG.2015.2462356 7

[35] D. Sculley. Web-Scale k-Means Clustering. In *Proc. of WWW*, pp. 1177–1178. ACM, 2010. doi: 10.1145/1772690.1772862 7

[36] B. Settles. Active Learning Literature Survey. Technical Report 1649, University of Wisconsin – Madison, Department of Computer Sciences,

2009. 3

[37] C. D. Stolper, A. Perer, and D. Gotz. Progressive Visual Analytics: User-Driven Visual Exploration of In-Progress Analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, 2014. doi: 10.1109/TVCG.2014.2346574 1, 7

[38] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser. Designing Progressive and Interactive Analytics Processes for High-Dimensional Data Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):131–140, 2017. doi: 10.1109/TVCG.2016.2598470 3, 7

[39] J. S. Vitter. Random Sampling with a Reservoir. *Transactions on Mathematical Software*, 11(1):37–57, Mar. 1985. doi: 10.1145/3147.3165 3

[40] M. Williams and T. Munzner. Steerable, Progressive Multidimensional Scaling. In *Proc. of VIS*, pp. 57–64. IEEE, 2004. doi: 10.1109/INFVIS.2004.60 8

[41] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 12:262–283, 2003. doi: 10.1007/s00778-003-0107-z 7

[42] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. How Progressive Visualizations Affect Exploratory Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 23(8):1977–1987, 2017. doi: 10.1109/TVCG.2016.2607714 1, 3

[43] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li. Quality and efficiency for kernel density estimates in large data. In *Proc. of SIGMOD*, pp. 433–444. ACM, 2013. doi: 10.1145/2463676.2465319 8

[44] Y. Zheng, Y. Ou, A. Lex, and J. M. Phillips. Visualization of Big Spatial Data using Coresets for Kernel Density Estimates. In *Proc. of VDS*, pp. 23–30, 2017. doi: 10.1109/VDS.2017.8573446 4

[45] Z. Zhou, C. Shi, X. Shen, L. Cai, H. Wang, Y. Liu, Y. Zhao, and W. Chen. Context-aware Sampling of Large Networks via Graph Representation Learning. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1709–1719, 2021. doi: 10.1109/TVCG.2020.3030440 8