Ph.D. Dissertation

Designing Progressive Visualization Systems for Exploring Large-scale Data

대용량 데이터 탐색을 위한 점진적 시각화 시스템 설계

February 2020

Department of Computer Science and Engineering
College of Engineering
Seoul National University

Jaemin Jo

Designing Progressive Visualization Systems for Exploring Large-scale Data

Advisor: Jinwook Seo

Submitting a Ph.D. Dissertation of
Computer Science and Engineering

November 2019

College of Engineering
Seoul National University

Jaemin Jo

Confirming the Ph.D. Dissertation written by
Jaemin Jo

January 2020

Chair	Myung-Soo Kim	(Seal)
Vice Chair	Jinwook Seo	(Seal)
Examiner	Bohyoung Kim	(Seal)
Examiner	Bongshin Lee	(Seal)
Examiner	Youngki Lee	(Seal)

Abstract

Jaemin Jo

Department of Computer Science and Engineering

College of Engineering | **Seoul National University**

Understanding data through interactive visualization, also known as visual analytics, is a common and necessary practice in modern data science. However, as data sizes have increased at unprecedented rates, the computation latency of visualization systems becomes a significant hurdle to visual analytics. The goal of this dissertation is to design a series of systems for progressive visual analytics (PVA)—a visual analytics paradigm that can provide intermediate results during computation and allow visual exploration of these results—to address the scalability hurdle.

To support the interactive exploration of data with billions of records, we first introduce SwiftTuna, an interactive visualization system with scalable visualization and computation components. Our performance benchmark demonstrates that it can handle data with four billion records, giving responsive feedback every few seconds without precomputation. Second, we present PANENE, a progressive algorithm for the Approximate k-Nearest Neighbor (AKNN) problem. PANENE brings useful machine learning methods into visual analytics, which has been challenging due to their long initial latency resulting from AKNN computation. In particular, we accelerate t-Distributed Stochastic Neighbor Embedding (t-SNE), a popular non-linear dimensionality reduction technique, which enables the responsive visualization of data with a few hundred columns. Each of these two con-

tributions aims to address the scalability issues stemming from a large number of rows or columns in data, respectively. Third, from the users' perspective, we focus on improving the trustworthiness of intermediate knowledge gained from uncertain results in PVA. We propose a novel PVA concept, Progressive Visual Analytics with Safeguards, and introduce PVA-Guards, safeguards people can leave on uncertain intermediate knowledge that needs to be verified. We also present a proof-of-concept system, ProReveal, designed and developed to integrate seven safeguards into progressive data exploration. Our user study demonstrates that people not only successfully created PVA-Guards on ProReveal but also voluntarily used PVA-Guards to manage the uncertainty of their knowledge. Finally, summarizing the three studies, we discuss design challenges for progressive systems as well as future research agendas for PVA.

Keywords: Information Visualization; Human-computer Interaction; Visual Analytics; Progressive Visual Analytics; Big Data; Large-scale Data; User Interaction; Dimensionality Reduction; User Study

Student Number: 2014-21782

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data sizes matter.

Tight collaboration between humans and computers through interactive visualization is a necessity in modern data science for understanding data. However, as data sizes have rapidly increased even for data analysis in casual circumstances, computation latency becomes a significant challenge to visual analytics. Despite the enormous advances in the speed of computing hardware, it is always possible to conceive a dataset that is too large to be queried in a reasonable time. In this dissertation, we present systems and algorithms for *Progressive Visual Analytics* (*PVA*)—visual analytics with an ability to accessing the intermediate results of computation—to enable scalable and reliable visual exploration of large-scale data.

1.1 Background and Motivation

Information visualization (InfoVis) leverages the human vision system, which has evolved to efficiently extract information from natural environments, to convey the characteristics of abstract data. With well-designed visualization, humans can understand and analyze complex data faster and more accu-

rately even without attention (i.e., preattentive processing [123]) than with conventional textual representation such as descriptive statistics [100]; for instance, Anscombe's quartet [11] exemplifies that visualization can unearth the underlying patterns of data that are invisible in descriptive statistics.

Although the earliest examples of InfoVis were *static*, mostly comprised of geographical maps drawn on papers, interactive visualization is now regarded as a core component in InfoVis. The most significant advantage of human interaction in InfoVis is that it increases the scalability of data exploration [97]; while a single static visualization can show only a limited scope of data, interaction allows people to view the multiple aspects of the data on a single screen. Analytical reasoning facilitated by interactive visual interfaces, formally called *Visual Analytics* [34], has now become an essential paradigm in understanding massive data in various fields such as astronomy, business, disaster and emergency management, security, and health [67].

However, over the past decades, we witnessed that the volume of data had increased at unprecedented rates, and the sheer size of data in modern data analysis has now become a major barrier to visual analytics. For example, a public enterprise dataset [37] contains over 4.3 billion logs about users' click events, and the GAIA dataset [28] has astrometric measurements of about 1.7 billion sources. Even on modern desktops, loading and processing such datasets require minutes to finish, making visual analytics no longer interactive. Using more powerful hardware can be a solution to this problem; however, the rate of improving the speed of processors is much slower than that of the capacity of collecting and storing data, which makes such a solution costly and impractical. From this perspective, conventional *monolithic* visualization systems—systems that respond after fully processing a

visualization query on the entire data—are not effective for the exploration of large-scale data common nowadays.

In contrast to monolithic visualization systems, *progressive* visualization systems allow humans to access intermediate results of a query even when the whole computation is not completed [45, 120]. Visual analytics through progressive computation, also known as *Progressive Visual Analytics* (*PVA*) [120], has demonstrated its effectiveness in exploring large-scale datasets such as event sequences [120], high-dimensional data [106], and social media data [14]. The main goal of PVA is to provide a sequence of continuous feedback to people, which approximates the precise result with increasing accuracy so that people can steer the exploration with less interruption and make their decisions early.

In this dissertation, based on the notion of PVA, we design and develop a series of progressive systems and algorithms as well as proposing a novel PVA concept altogether for scalable and reliable data exploration. We focus on three critical challenges in PVA applications in practice, namely: 1) **Vertical Scalability**, 2) **Horizontal Scalability**, and 3) **Knowledge Trustworthiness**. First, for interactive exploration of data with *a few billion rows* (**Vertical Scalability**), we introduce SwiftTuna that supports visual analytics on large-scale data with scalable visualization components and responsive interaction without precomputation. Second, for responsive visualization of data with *a few hundred columns* (**Horizontal Scalability**), we present PANENE, a progressive algorithm for the Approximate k-Nearest Neighbor (AKNN) problem that enables the progressive computation of t-Distributed Stochastic Neighbor Embedding (t-SNE) of high-dimensional data. Finally, for the *reliability* of the intermediate knowledge garnered from progressive data exploration (**Knowledge Trustworthiness**), we propose a novel PVA

concept, Progressive Visual Analytics with Safeguards, and develop a proof-of-concept system ProReveal.

1.2 Thesis Statement and Research Questions

Thesis Statement Well-designed progressive systems and algorithms can overcome the scalability and reliability challenges of visual analytics universal in modern data science, enabling the responsive and trustworthy exploration of large-scale multidimensional data.

To support the statement, we pose and address the following research questions in this dissertation:

RQ1. **Vertical Scalability**: how do we enable interactive visual exploration of large-scale data with scalability in both data processing and visualization?

RQ2. **Horizontal Scalability**: how do we responsively embed and visualize high-dimensional data on a 2D space without long initial computation delays?

RQ3. **Knowledge Trustworthiness**: how do we improve the trustworthiness of intermediate knowledge gained from progressive visual analytics?

1.3 Thesis Contributions

The contributions of this dissertation are as follows:

1. Design, development, and quantitative evaluation of **SwiftTuna**, an interactive system for exploring large-scale multidimensional data with a few billion rows.

2. Development of **PANENE** and its applications, such as Responsive t-SNE, along with system benchmark.

3. Definition and discussion on a novel PVA concept, **Progressive Visual Analytics with Safeguards**, and the development and user evaluation of a proof-of-concept system **ProReveal**.

1.3.1 Responsive and Incremental Visual Exploration of Large-scale Multidimensional Data

Addressing RQ1, we present SwiftTuna (Figure 1.1), an interactive system that streamlines the visual information seeking process on large-scale multidimensional data. For the interactive exploration of large-scale data, a preprocessing scheme (e.g., data cubes) has often been used to summarize the data and provide low-latency responses. However, such a scheme suffers from a prohibitively large amount of memory footprint as more dimensions are involved in querying and a strong prerequisite that specific data structures have to be built from the data before querying. SwiftTuna exploits an in-memory computing engine, Apache Spark, to achieve both scalability and performance without building precomputed data structures. We also present a novel interactive visualization technique, tailed charts, to facilitate large-scale multidimensional data exploration. To support responsive querying on large-scale data, SwiftTuna leverages an incremental processing approach, providing immediate low-fidelity responses (i.e., prompt responses) as well as delayed high-fidelity responses (i.e., incremental responses). The performance evaluation of SwiftTuna demonstrates that SwiftTuna allows data exploration of a real-world dataset with four billion records while preserving the latency between incremental responses within a few seconds.

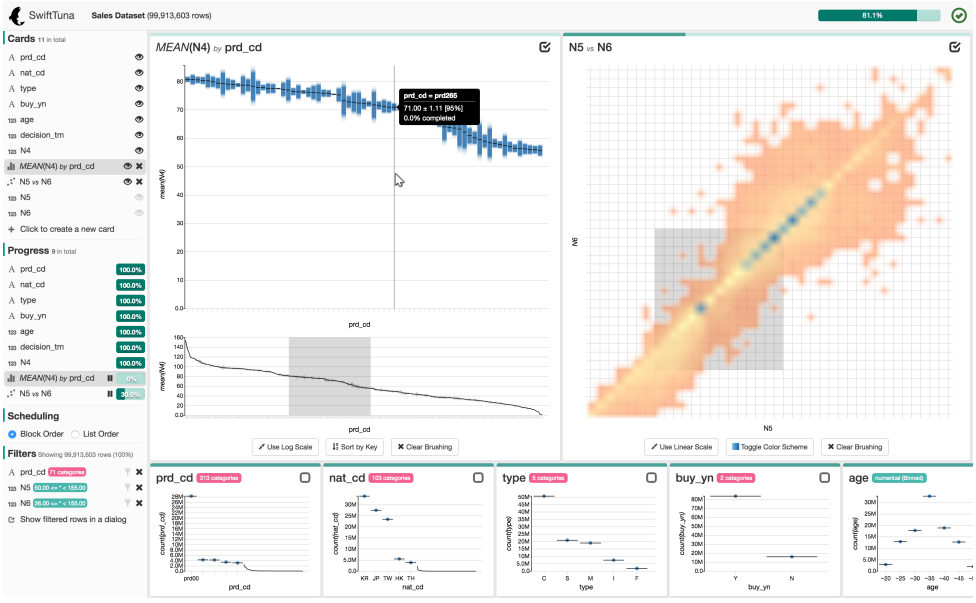


Figure 1.1: The SwiftTuna Interface

1.3.2 Progressive Computation of Approximate k-Nearest Neighbors and Responsive t-SNE

To approach RQ2, we present PANENE (Figure 1.2), a progressive algorithm for approximate nearest neighbor indexing and querying. Although the use of k-Nearest Neighbor (KNN) libraries is common in many data analysis methods, most KNN algorithms can only be queried when the whole dataset has been indexed, i.e., they are not online. Even the few online implementations are not progressive in the sense that the time to index incoming data is not bounded and cannot satisfy the latency requirements of progressive systems. This long latency has significantly limited the use of many machine learning methods, such as t-SNE, in visual analytics. PANENE is a novel algorithm for Progressive Approximate k-NEarest NEighbors, enabling fast KNN queries while continuously indexing new batches of data. Following the progressive computation paradigm, PANENE operations can be bounded

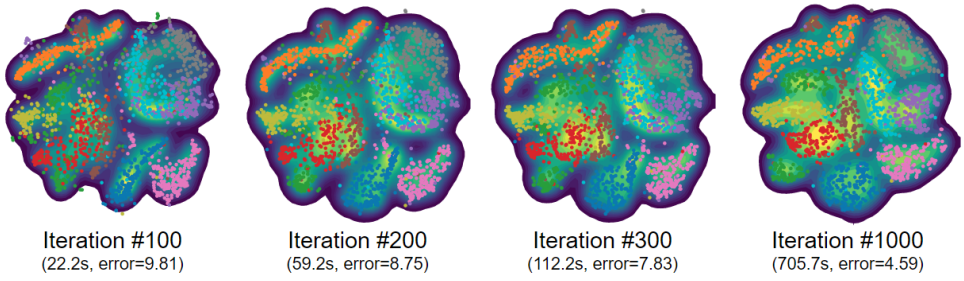


Figure 1.2: Responsive 2D Embedding of the MNIST dataset [71] using PANENE

in time, allowing analysts to access running results within an interactive latency. PANENE can also incrementally build and maintain a cache data structure, a KNN lookup table, to enable constant-time lookups for KNN queries. Finally, we present three progressive applications of PANENE, such as regression, density estimation, and Responsive t-SNE, opening up new opportunities to use complex algorithms in interactive systems.

1.3.3 Progressive Visual Analytics with Safeguards

To answer RQ3, we present a new visual exploration concept—Progressive Visual Analytics with Safeguards—–that helps people manage the uncertainty arising from progressive data exploration. Despite its potential benefits, intermediate knowledge from progressive analytics can be incorrect due to various machine and human factors, such as a sampling bias or misinterpretation of uncertainty. To alleviate this problem, we introduce PVA-Guards, safeguards people can leave on uncertain intermediate knowledge that needs to be verified, and derive seven PVA-Guards based on previous visualization task taxonomies. PVA-Guards provide a means of ensuring the correctness of the conclusion and understanding the reason when intermediate knowledge becomes invalid. We also present ProReveal (Figure 1.3), a proof-of-concept system designed and developed to integrate the seven safe-

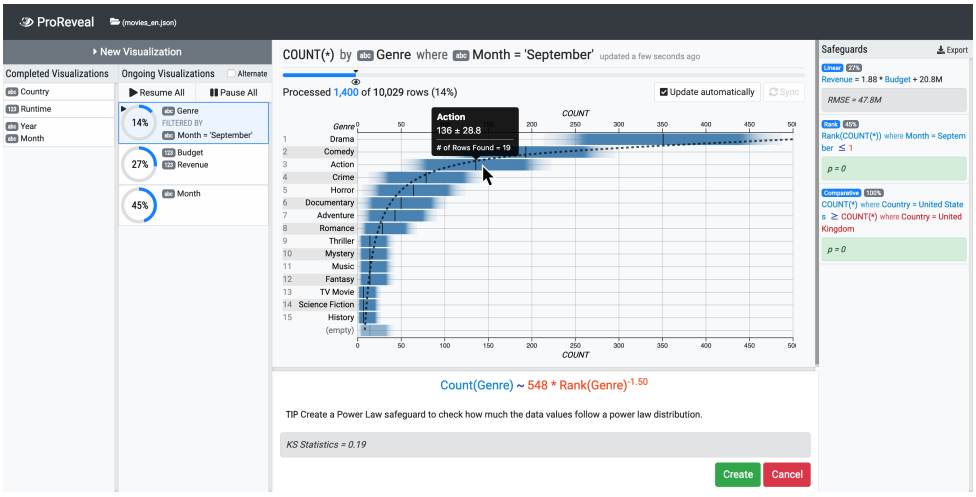


Figure 1.3: The ProReveal Interface

guards into progressive data exploration. Finally, we report a user study with 14 participants, which showed people voluntarily employed PVA-Guards to safeguard their findings and ProReveal's PVA-Guard view provides an overview of uncertain intermediate knowledge.

1.4 Structure of Dissertation

The rest of this dissertation is organized as follows: First, Chapter 2 presents a brief survey on the definition, systems, and models of PVA as well as previous work for scalable visualization systems apart from PVA. Chapter 3 illustrates the SwiftTuna system, describing its design considerations as well as implementation details. We also present a performance benchmark using a real-world dataset with four billion records. Chapter 4 presents the PANENE algorithm and its application, such as Responsive t-SNE. We report on the performance measurements of our algorithm in terms of latency and accuracy through a benchmark on two real-world datasets with 100 di-

mensions. Chapter 5 describes our novel PVA concept, Progressive Visual Analytics with Safeguards, and a proof-of-concept system, ProReveal. We also report on a user study with 14 participants to evaluate the usability of ProReveal and check if and how people interact with PVA-Guards. Chapter 6 discusses lessons learned from the whole course of the dissertation, opportunities for improving the systems, and future research agendas for PVA. Finally, Chapter 7 concludes the dissertation by summarizing the contributions and presents promising directions for future research.

Chapter 2

Related Work

In this chapter, we survey previous work from four thematic areas. First, we provide an overview of research contributions that have been made for Progressive Visual Analytics (PVA) since its first definition was given in 2014. Second, apart from PVA, we also review InfoVis systems and techniques that enable responsive interactions with large-scale data such as data cubes. Next, we review previous research on the k-Nearest Neighbor (KNN) problem and the t-Distributed Stochastic Neighbor Embedding (t-SNE) technique in the third and fourth sections respectively before we provide progressive computation approaches for them.

2.1 Progressive Visual Analytics

2.1.1 Definitions

What is PVA? Since Stolper et al. [120] first introduced PVA in 2014, there has been extensive discussion on the definition and requirements of PVA. Most early attempts have been made to especially distinguish it from similar computation paradigms, such as online computation and approximate query processing (AQP). Although there is still ongoing discussion on the

definition of PVA, in this section, we cover the three most relevant definitions from the most inclusive one to the most narrow one.

Stolper et al. defined PVA as "a paradigm that gives users access to semantically meaningful partial results during the execution of analytics, and allows exploration of these partial results in integrated, interactive visualizations" [120]. There are two important things to note in the definition. First, Stolper et al.'s definition does not restrict the interval between two partial results. This means any system that gives partial results can be progressive even though such a system would not be much useful if the interval is too long. Second, PVA is not only about speeding up computation; PVA systems should provide appropriate visualizations that allow people to explore the partial results interactively.

From a computation perspective, Fekete and Primet [45] defined progressive computation is computation that 1) returns a sequence of partial results on a growing subset of the original data 2) with a limited time interval q between the partial results 3) satisfying a property that the sequence of the partial results converges to the final result. Compared to Stolper et al.'s definition, Fekete and Primet defined progressiveness more strictly; the interval between partial results can be bounded a quantum q. Therefore, a system that gives intermediate results in the middle of computation (i.e., online [57]) is not progressive unless the interval between the results is guaranteed. However, in practice, it is very challenging to develop a system that guarantees to always respond in the given time limit. Still, PVA systems that can keep the latency within a reasonable bound for most cases would be useful.

The most recent definition of progressive computation discussed in the Dagstuhl seminar [44] is "computation bounded on time and data which reports intermediate outputs, namely a result, a measure of quality, and a mea-

sure of progress, within bounded latency, converging towards the true result, and controllable by a user during execution either by aborting, pausing, etc., or by tuning parameters (steering)." This definition explicitly mentions *user steerability* of computation; the user should be able to control the execution of progressive computation in any form, such as by pausing or resuming the computation.

These three definitions vary in terms of strictness, but they help us to understand the difference between progressive computation and other computation schemes. For example, online computation is akin to progressive computation [23] in that the user can grasp meaningful partial results before the whole computation is finished, but online computation does not guarantee the interval between partial responses. Similarly, iterative methods differ from progressive computation in that its algorithmic parameters cannot be changed by users interactively.

2.1.2 System Latency and Human Factors

Shorter latency is desirable for InfoVis systems [78]. However, in general, there is a trade-off between the latency and the throughput of a PVA system since delivering intermediate outcomes interrupts computation and thus imposes overhead. Then, how fast should a PVA system respond, striking a balance between the latency and the throughput? In the human-computer interaction (HCI) field, researchers have actively studied how the latency of a computing system affects user behavior and experience [78, 89, 99, 114], and these studies can provide a guideline on the time limit of a PVA system.

Liu and Heer [78] found that an extra delay of 500 ms decreased user activity, dataset coverage, and the rate at which users make observations. However, they injected the delay during user interactions, such as zooming

and panning, which is more intrusive than having the same amount of delay when visualization is initially drawn. Nielsen [99] distinguished three time limits for users' perception and attention: 0.1 second for feeling that the system is continuous (e.g., for animations), 1.0 second for maintaining the user's flow of thought, and 10 seconds for keeping the user's attention. Similar guidelines on latency have also been suggested by other researchers; for example, Miller [89] argued that response delays longer than 10 seconds would not permit users to think continuously and sustain problem-solving that contains a high degree of ambiguity.

Experiments in HCI have also shown that people can operate on visualization that is incrementally updated within a latency of a few seconds. For example, Fisher et al. [48] presented a prototype interface, *sampleAction*, which incrementally visualizes aggregates (i.e., a mean) with their confidence intervals. Through a user study with three data analysis experts, they found the experts were able to interpret confidence intervals for faster decision making, advocating the use of incremental query processing. To understand how progressive visualizations influence human behavior during exploratory data analysis, Zgraggen et al. [139] compared instantaneous, progressive, and blocking visualization using insight-based metrics such as the number of insights found per minute. They found that progressive visualization outperformed blocking visualization, even producing a comparable performance to ideal instantaneous visualization. Similar results were shown in Badam et al.'s user study [14], where a progressive interface, InsightsFeed, and its instantaneous version showed similar performance in terms of answer accuracy and user preferences.

To sum up, the acceptable time limit varies depending on users' situations, dataset sizes, and analysis types; however, a time limit of a few sec-

onds up to 10 seconds can be regarded as a reasonable and feasible time limit, aligned with the previous guidelines on latency and empirical study results.

2.1.3 Users, Tasks, and Models

Munzner [97] introduced a what-why-how framework for analyzing the usage of visualizations, and we can cast similar questions to PVA: who uses PVA, why they use PVA, and how PVA systems should be built.

Who uses PVA and Why do they use PVA? Micallef et al. [88] classified the users of PVA systems into three types: Observer, Searcher, and Explorer. Observer is a group of users who are interested in the output of a progressive process rather than interacting with the intermediate outcomes of the process. They want to monitor the progression of an algorithm, expecting the underlying process reaches to a predictable end. For example, a machine learning expert who wants to check if a training process proceeds well (e.g., with decreasing loss) can be an Observer. Their main tasks include ascertaining the quality or quantity of progressive outcomes and understanding the inner workings of a progressive algorithm. The second group of users, Searcher, are interested in answering a concrete question to large-scale data using PVA. Since Searcher often has a domain-specific time-critical question, it is important to show the uncertainty of answers and possible fluctuations or jumps in the uncertainty. The main tasks of Searcher also include comparing different executions of computation (e.g., executions with different parameters) to assess its robustness. The last group of users, Explorer, focuses on exploring data and building a comprehensive understanding. Among the three groups of users, Explorer is most interested in performing open-ended

exploration. The tasks of Explorer include gaining an overview of data or parameter spaces and investigating alternative scenarios.

There are also useful models that explain and characterize existing systems as well as providing guidelines on future systems. Mühlbacher et al. [94] characterized four strategies, such as data subsetting, complexity selection, divide and combine, and dependent subdivision, to increase user involvement in existing sequential algorithms. They also presented four types of user involvement according to two axes: the direction of information (feedback vs. control) and entity of interest (execution vs. result). Schulz et al. [111] defined partitioned data and visualization operators based on the established Data State Reference Model [32] to facilitate intermediate visualization updates.

Reviewing the results of previous studies on PVA, Angelini et al. [8] presented a comprehensive survey of existing progressive systems, especially describing the requirements and guidelines on PVA. Two of the major challenges of PVA they identified are (1) judging partial results and (2) handling fluctuating or even diverging progressions, which we want to address by introducing PVA-Guards as a means of managing the uncertainty from PVA. Angelini et al. [7] proposed ten fundamental quality indicators for showing the progression, stability, and certainty of intermediate results. ProReveal provides indicators for absolute progress (i.e., the amount of data processed so far), relative progress (i.e., the amount of data processed in the last iteration), and absolute certainty (e.g., a confidential interval for a bar). Finally, regarding interactions on progressive interfaces, Wu et al. [135] emphasized the importance of cumulatively rendering asynchronous results to improve the perceived speed and usability of interactive visualizations.

2.1.4 Techniques, Algorithms, and Systems

With the benefits of PVA, many researchers have endeavored to adopt progressive computation in domain-specific scenarios, expanding the border of PVA. The pioneering work of Stolper et al. [120] provided a progressive implementation of the SPAM algorithm [13] for extracting common patterns in event sequences. Enabling PVA on deep neural networks, Pezzotti et al. [105] presented DeepEyes, which supports detailed analysis while training a neural network. Sperrle et al. introduced the concept of Speculative Execution [118] that allows effective model optimization and refinement through progressive computation. Badam et al. [14] introduced InsightFeed for exploring Twitter data at scale. Vidal et al., [129] presented a progressive algorithm for Wasserstein barycenters of persistence diagrams.

Progressive exploration of multidimensional data is another important research area. Pezzotti et al. [106] introduced approximate t-Distributed Stochastic Neighbor Embedding (A-tSNE), which significantly lowered the long latency of the t-SNE algorithm [84]. Choo et al. and Kim et al. [33, 68] presented the per-iteration visualization environment (PIVE) that allows users to interact with intermediate results of multidimensional dimensionality reduction and clustering methods. Turkay et al. [124] proposed DimXplorer that integrates the incremental PCA and mini-batch k-means clustering algorithms with data exploration.

Frameworks and techniques for efficient progressive computation and hypothesis building have also been suggested. Fekete and Primet implemented the Progressivis Toolkit [45] to provide a general and systematic platform for progressive implementations. Li and Ma [74] proposed P5, Portable Progressive Parallel Processing Pipelines that aimed at interactive data analysis and visualization on web browsers using GPU acceleration. Stat! [17] is

another interactive analytics environment for large-scale data that provides incremental and cumulative results based on SQL-like data queries. In Swift-Tuna and ProReveal, we focus on interactive data analysis on web browsers as well as seeking better scalability in both visualization and data processing by leveraging user interactions (e.g., Overview+Detail [132]) and a distributed computing engine, Apache Spark [138]. Glueck et al. [52] presented the Splash framework, which allows real-time on-demand navigation of large-scale data. For more efficient and robust sampling, Rahman et al. [108] proposed sampling-based incremental visualization algorithms that capture the salient feature of a visualization quickly. Finally, Zhao et al. [140] devised heuristics for eliciting hypotheses from visualization, and developed multiple hypothesis testing controls to manage false discoveries.

Moritz et al.'s research on optimistic visualization [90] is one of the studies that motivated the application of safeguards to PVA. In optimistic visualization, people explore data "optimistically," trusting early uncertain results from a visualization. Later, when the precise result is ready, people return to the visualization and check if there is a big difference between the earlier one and the precise one, which can be a signal of errors. In our work, however, we consider analysis scenarios where the precise result is not feasible in a single session due to long computation time. Therefore, rather than facilitating the comparison between the two results, we provide continuous feedback on the validity of intermediate knowledge that is statistically estimated, and notify people when intermediate knowledge becomes invalid. Furthermore, we allow people to formulate their intermediate knowledge in a structured form (i.e., PVA-Guards) rather than plain text, which is used to delegate the validation process to the system and enable people to continue their exploration.

2.1.5 Uncertainty Visualization

Intermediate results in PVA by nature are uncertain since the results are computed on a sample of data or they have not converged yet. Visualization for uncertainty has been extensively studied in the scientific visualization area [102] since scientific measurements often contain observational errors. It has been known that specific visual variables, such as fuzziness (or blurriness) and transparency, can be connected to uncertainty more intuitively than other variables [85]. Another intuitive visual variable is sketchiness, although it shows a higher perception error than blurriness and grayscale [25].

Related to PVA, Fisher et al. [48] visualized the confidence intervals of intermediate results to observe how analysts interact with incremental visualizations. They found that encoding the confidence intervals with traditional error bars was not easily understandable, demanding new ways to represent them for incremental visualizations. Based on the lesson learned from the sampleAction project, Fisher et al. [47] designed alternative visualizations for error bars, i.e., density strips and modified box-percentile plots. As other alternatives to error bars, Correll et al. [35] presented gradient plots and violin plots that outperformed error bars for inferential tasks. For visualizing the uncertainty of bivariate maps, Correll et al. [36] proposed Value-Suppressing Uncertainty Palletes (VSUPs), which represent uncertainty by combining the lightness and saturation channels. Another recent technique for showing uncertainty in scatterplots is Winglets [82], which encodes the uncertainty in the class membership of each data point by the orientation of length of two small curves (i.e., wings) near the point. In SwiftTuna and ProReveal, we employ gradient plots and heatmaps with VSUPs for their effectiveness.

2.2 Approaches for Scalable Visualization Systems

Even before Stolper et al. introduced PVA [120], there has been a body of work to achieve the scalability of interactive visualization systems. According to Godfrey et al. [53], these systems can be categorized into two paradigms: the preprocessing paradigm and the non-preprocessing paradigm.

The preprocessing paradigm includes systems that preprocess data to build a specific data structure in advance and uses this result for future queries. For example, online analytical processing (OLAP) cubes have been often used to summarize large data [31]. An OLAP cube is a multidimensional data structure that maps each categorical attribute in data to a dimension of the cube and precomputes statistical summaries, such as count or sum, for each combination of dimensions. Once built, an OLAP cube can process group-by queries very efficiently by looking up the corresponding cells instead of scanning the full data. However, the major limitation of OLAP cubes is its large memory footprint since the number of cells in the cubes grows exponentially as more dimensions are included in the cubes. Furthermore, data cubes must be built before querying, which is impractical for cold-start analytics—exploration without time-consuming precomputation [91].

To address the memory footprint issue, recent research has proposed space-efficient variants of data cubes. For example, imMens [79] introduced multivariate data tiles to support interactive linking between visualizations. These multivariate data tiles are small enough to be loaded and processed on a web browser, saving the cost of querying a remote server. Another example is Nanocubes [76], a hierarchical data structure for querying multidimensional spatiotemporal data. Nanocubes has been improved for various purposes: for more compact data structures using arrays [101], for model-

ing capabilities [131], for accelerating spatial visualization dashboards [136], and for adaptive data management [77]. However, still, it takes a few hours to initialize these cubes with a billion of data rows; for example, it took longer than one hour to initialize one of the state-of-the-art techniques [136] with 700 million data rows (about 100 GB).

On the other hand, non-preprocessing approaches responsively process data in a scalable manner upon a query, which is more useful for cold-start analytics. As shown in the CONTROL project [54, 56, 57], online aggregation builds and processes a sample at the query time, visualizes the estimates of a query with confidence intervals. The answers continually improve (e.g., the confidence intervals converge as more data points are sampled), allowing users to observe and control the results on the fly. In the database community, this computation scheme is called Approximate Query Processing (AQP). An example of AQP databases is BlinkDB [2], which generates multidimensional stratified samples to allow users to trade-off query accuracy for response time.

Another non-preprocessing approach to scalable visualization systems is to process large-scale data on a distributed computing cluster and visualize aggregated results. Apache Hadoop [115] is one of the most successful ecosystems to handle large-scale data. Apache Hadoop includes a parallel processing component, MapReduce [40]; however, MapReduce itself is impractical for low-latency querying since it spills intermediate computation results on disks, usually imposing a delay exceeding our latency threshold (i.e., 10 seconds). For example, Im et al. [59] presented a modified MapReduce-style algorithm in their VisReduce system for fast incremental data processing for visualization, arguing against the preprocessing scheme.

In-memory computing technology significantly sped up the performance of MapReduce while still keeping the flexibility of the non-preprocessing paradigm. We leverage an in-memory general processing engine, Apache Spark [138]. Apache Spark keeps data and intermediate results in the main memory to minimize disk I/O instead of writing them on disks. Since the main memory is cheaper nowadays, Apache Spark gains its popularity for in-memory processing of large-scale data. Built upon Apache Spark, Shark [43] allows querying large-scale data with SQL. Shark has been integrated into Apache Spark as SparkSQL [12]. Using SparkSQL, several web applications such as Databricks, Hue, and Apache Zeppelin integrated interactive data analytics with cluster computing. However, those applications only provide a limited number of mostly static visualization presets without supporting important interactions such as brushing and filtering. SwiftTuna incrementally processes data on a computing cluster and visualizes results with scalable visualizations and interactions that are designed to support large-scale data exploration.

2.3 The k-Nearest Neighbor (KNN) Problem

The early approaches to the k-nearest neighbor problem focused on finding the exact neighbors of a query point. First introduced in the seminal work by Bentley [20], k-d trees have been one of the most widely used methods for KNN queries. The original k-d tree iteratively splits the space with axis-aligned hyperplanes and builds a binary tree, allowing a logarithmic time complexity for KNN queries [50]. At each level in the tree, data is divided into two groups along the dimension in which the data has the highest vari-

ance. Then, the tree can be used to reject points in distant subspaces early on for a more efficient search.

While the original k-d trees are effective for searching in low-dimensional spaces, they suffer from significant performance degradation as the dimensionality of data increases; this problem is related to the so-called "curse of dimensionality" [87]. To overcome this limitation, recent research relaxed the requirements of KNN search by allowing it to return approximate neighbors with parameters for controlling the quality. These techniques, which are called approximate k-nearest neighbor (AKNN) search, are not guaranteed to return the exact k-nearest neighbors of a query point but good approximates of neighbors that are close to the query point in a short time. Due to their flexibility, AKNN techniques have become common in modern toolkits such as scikit-learn [103] and OpenCV [26]. We can categorize popular AKNN techniques in three families: space-partitioning trees, hash-based, and graph-based.

The simplest data structures for KNN are *space-partitioning trees*. They recursively divide a multidimensional space and build a tree structure that can be used to accelerate searching. Many k-d tree variants have been proposed to reduce the query time for KNN searches. Beis and Lowe [19] showed that limiting the number of visited nodes in a k-d tree could bring a large reduction in the query time with a small loss in accuracy. For KNN search in higher-dimensional spaces, Silpa-Anan and Hartley [116] presented the idea of multiple randomized k-d trees where data is recursively split along a dimension that is randomly chosen from a small set of candidate dimensions with the highest variance. Muja and Lowe [95] identified the two best algorithms for AKNN querying—randomized k-d trees and hierarchical k-means trees—and presented an algorithm that selects optimum parameters

for the algorithms in terms of speed and accuracy criteria. Going one step further, they successfully extended their work to perform distributed nearest neighbor search on a cluster of machines [96].

Another body of research adopted partitioning strategies with hyperplanes not aligned with axes. Examples include non-axis-aligned hyperplanes [119], random projection trees [39], trinary projection trees [61], ball trees [73], and several open-source implementations [21, 49, 103, 110].

Hash-based techniques use a set of *locality-sensitive hashing (LSH) functions* [5]. The core idea is that a pair of close points is more likely to fall into the same bucket after hashing than a pair of distant points. Therefore, hash-based techniques can efficiently search for neighbors by looking up the buckets that a query point falls into. The strength of hash-based techniques stems from the fact that they can provide a theoretical base on the search quality. Examples include LSH forest [18], multi-probe LSH [83], kernelized LSH [69], circular random variable-based matchers [3], and LSH for angular distance [6].

Graph-based techniques model multidimensional data points as a graph by mapping points to vertices and the neighborhood relationships to edges. Once the graph is built, AKNN search can be done by exploring the graph. From a KNN graph, Sebastian and Kimia [112] selected a few well-separated vertices (i.e., *seeds*) and iteratively moved the seeds to points that are closer to the query point until satisfactory neighbors were found. Hajebi et al. [55] provided theoretical guarantees for the accuracy and the computational complexity of such a greedy method. Wang et al. [130] introduced a new approach to constructing approximate KNN graphs by building exact neighborhood graphs for hierarchically divided data and combining the graphs. Recently, more sophisticated graph structures, such as navigable small-world graphs are used for KNN queries. In addition to short-range links in a tradi-

tional neighbor graph, navigable small-world graphs have long-range links that connect two distant points. Malkov et al. [87] showed that these long-range links can be used for logarithmic scaling of neighbor exploration. Later, Malkov and Yashunin [86] further improved the performance by introducing hierarchical structures to navigable small-world graphs. Yet, the construction of the graphs is costly and cannot easily be done online.

Throughout a few decades of KNN research, query time (i.e., time taken to perform a KNN search) has been the key measure for evaluating the performance of various techniques. Indeed, in most studies mentioned in this section, the authors assumed that data points had already been inserted in an index and measured the time taken to process queries. This is also the case with benchmarks in the public domain [10, 93]. However, such benchmarks are meaningful only when the data is kept constant. In more interactive scenarios, such as interactive analysis by human analysts, the data can be changed dynamically through user interaction, such as loading a new set of data or filtering out a subset of data. Thus, it is necessary to keep the whole process of KNN queries, including building and querying the index, interactive. In this chapter, inspired by Progressive Visual Analytics [120], we introduce a progressive k-d tree for approximate k-nearest neighbor search that can keep the latency for building, maintaining, and querying the index within specified time bounds. We chose to start with the multiple randomized k-d tree algorithm, which is simple yet one of the most efficient algorithms for AKNN queries [95].

2.4 t-Distributed Stochastic Neighbor Embedding

t-Distributed Stochastic Neighbor Embedding (t-SNE) [84] is a nonlinear dimensionality reduction algorithm that is widely used in data analysis. t-SNE maps a set of high-dimensional points, $\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N$, in a feature space to their low-dimensional representations (i.e., embedding), $\mathbf{y}_1, \mathbf{y}_2, ...\mathbf{y}_N$, minimizing the Kullback-Leibler (KL) divergence [70] between the joint probabilities of the original points and their low-dimensional representations. The points are often projected on a 2D space (i.e., $dim(\mathbf{y}_i) = 2$) and visualized through conventional scatterplots or density plots. t-SNE consists of two procedures: 1) computing the *k*-nearest neighbors of each point in the feature space (i.e., *neighbor computation*) and 2) gradient descent iterations for minimizing the KL divergence between the joints probabilities (i.e., *loss optimization*). The second loss optimization procedure is iterative; for example, intermediate embeddings of points can be visualized and improved during the iterations. However, the first procedure, neighbor computation, blocks the whole algorithm and introduces a long delay to the initial result. On a modern desktop, we found it took about an hour to compute the neighbors of 60,000 points with 784 dimensions in the MNIST dataset [71] in parallel.

Barnes-Hut Stochastic Neighbor Embedding (BH-SNE) [127] improved the complexity of the original t-SNE from $O(N^2)$ to $O(N \log N)$ by applying the Barnes-Hut approximation [16] to compute the contribution of points in the loss optimization procedure. Pezzotti et al., [106] presented Approximated-tSNE (A-tSNE), which visualizes the intermediate results of t-SNE and allows users to steer the t-SNE algorithm by adding, removing, and modifying the results. Recently, Pezzotti et al., [107] further improved t-SNE using a General-Purpose Graphics Processing Unit (GPGPU), introducing a

linear complexity loss optimization procedure. These techniques employed an approximate k-Nearest Neighbor (AKNN) algorithm, such as a forest of randomized KD-trees [95], to speed up neighbor computation, which is not progressive and still requires minutes to be initialized.

Chapter 3

SwiftTuna: Responsive and Incremental Visual Exploration of Large-scale Multidimensional Data

This chapter [1] illustrates the SwiftTuna system and describes its design considerations as well as implementation details to cover the first research question (***Vertical Scalability**: how do we enable interactive visual exploration of large-scale data with scalability in both data processing and visualization?*).

Although there have been great advances in visualization and database technologies, visual analysis of large-scale multidimensional data is still challenging. The foremost issue is the long latency of queries, which resulted from the sheer magnitude of the data. To tackle this issue, researchers in the InfoVis and database community have attempted to enable low-latency visual exploration of large-scale data. Through a survey of relevant studies and

[1] The preliminary version of Chapter 3 was published as a conference paper [62] in the proceedings of the 10th IEEE Pacific Visualization Symposium (PacificVis 2017).

visualization systems, we could identify the following four requirements for visual analytics systems for large-scale multidimensional data and define a design space for such systems Figure 3.1.

R1. Large-scale Data Processing. The system should be able to process large-scale data in a scalable manner. It is hard to define what large-scale or big data means in a concrete number [58]. In addition, even if there is a concrete definition, it may vary across domains or have to change as technology advances. However, to make the contribution of our work clear, we consider one billion entities as the minimum size of large-scale data. That number is the largest number of entries that have been used to evaluate interactive systems for large-scale data analytics in information visualization [76, 79].

R2. Responsive Interaction. It is known that a shorter interaction latency could promote insight generation [78]. However, querying on large-scale data often takes a few minutes or even a few hours, which is too long to keep users' attention [99]. To support fluent data exploration without losing users' attention on ongoing tasks, the system should respond to queries in less than 10 seconds.

R3. Interactive Multidimensional Exploration. Multi-dimensionality is another important aspect of large-scale data exploration because most real-world large-scale datasets have two or more attributes. The system should be able to illuminate various aspects of such datasets, enabling users to explore relationships among multiple dimensions through visualization. By allowing users to generate perceptually effective 1D or 2D projections of the multidimensional data, users would be able to gain meaningful insights covering multiple dimensions of the data. This re-

quirement also includes interaction on multiple dimensions; for example, users should be able to delve into a small set of interesting data by applying filters.

R4. Scalable Visualization. Not all traditional visualization techniques are applicable to large-scale data analysis. Most traditional visualizations could suffer from severe overplotting or cluttering problems when applied to large-scale data. Therefore, visualization designers have to consider the scalability of their visualizations more seriously to enable users to perceive and understand the visualizations of large-scale data.

We found that a holistic approach that satisfies all four requirements is rare. For example, imMens [79] supports real-time visual querying (R2) and scalable visualizations and interaction (R4) but lacks the scalability in data processing (R1) since it stores its data structure in the main memory on a single machine. Also, imMens only allows filtering on two dimensions (e.g., brushing on 2D binned plots) at once, which does not satisfy R3. VisReduce [59], which may be the most relevant to ours, focuses on fast and responsive information visualization (R2), but it is not validated in terms of our scalability requirement (R1). Also, it does not support interactive visualizations for multidimensional data exploration (R3).

In this chapter, we present SwiftTuna, a holistic approach to enable fluent visual exploration of large-scale multidimensional data. We explore the opportunity of incremental data processing for information visualization with a real-world scale in mind (R1). SwiftTuna does not resort to a prebuilt data structure (e.g., data cubes) but incrementally processes the raw data in a distributed manner, enabling users to instantly initiate visual analytics with their data and request queries that cover multiple dimensions (R3). Our

work is neither limited to processing large-scale data efficiently for visualization nor to designing visualizations and interaction for scalability. Rather, our work puts emphasis on designing and developing a holistic visual analytics system that streamlines the whole process of visual information seeking for large-scale multidimensional data.

SwiftTuna consists of three layers: the data processing layer, visualization layer, and querying layer. In the data processing layer on the server side, SwiftTuna leverages an in-memory cluster computing engine, Apache Spark [138], to achieve both high scalability and extendibility. For the visualization layer on the client side, we carefully design a user interface and visualizations that summarize multiple dimensions of the data in a scalable manner. The querying layer bridges the client and the server. To support responsive querying such as filtering on the data, SwiftTuna leverages an incremental processing approach [46] enables users to grasp immediate but low-fidelity responses (i.e., prompt responses) as well as delayed but high-fidelity responses from incremental processing (i.e., incremental responses).

We present the design of SwiftTuna in Section 3.1 and elaborate on responsive querying on SwiftTuna in Section 3.2. In Section 3.3, we evaluate our system with a real-world dataset that contains about four billion rows, and report the result. For continued research, we describe the implementation of SwiftTuna in Section 3.4.

3.1 The SwiftTuna Design

In this section, we describe the design considerations behind SwiftTuna and explain the design of our system.

3.1.1 Design Considerations

While addressing the four requirements in our design of SwiftTuna, we put emphasis on two specific goals: scalability and responsiveness. Scalability is not limited to a server-side architecture for large-scale data processing (R1), but encompasses multiple scalable visualizations and interactions (R3 and R4). We also strive to facilitate fluent data exploration by providing responsive feedback (R2). We have iteratively refined our system while performing design studies on real-world problems with large-scale data in manufacturing and online games following the nine-stage design study methodology framework [113]. As a result, we present the design considerations as follows:

DC1. Provide low-fidelity feedback promptly. A delayed response hinders users from observing the data and generalizing their findings [78] and causes them to lose their attention [99]. To enable fluid data exploration, we provide low-fidelity feedback promptly (i.e., prompt responses) based on a small sample from the data. The main purpose of prompt low-fidelity feedback is to allow users to visually confirm their queries early without looking at an empty screen waiting for a late final response.

DC2. Process results incrementally while estimating the final results. Inspired by progressive visual analytics [120], we visualize partial results of analytics and estimate the final results before a query is fully completed. This enables users to confirm or reject their hypotheses as early as possible during exploratory analysis and thus test more hypotheses with limited time and resources.

DC3. Enable flexible scheduling. To amplify the use of partial results, SwiftTuna provides flexible management of computing resources. For example, users can pause or stop queries in real time if they think partial results are enough for decision-making. This makes more resources on the server available for processing other queries. Also, users can prioritize a query of interest and examine the result on that query first.

DC4. Support multidimensional data exploration. Multidimensional exploration can lead users to find new insights across multiple attributes of the data. SwiftTuna organizes a small multiple in a single view that show various aspects of data in a series of 1D or 2D projections. Also, it supports essential interactions such as brushing, filtering, and details on demand that are known to be key tools for the information seeking process.

3.1.2 System Overview

SwiftTuna employs a client-server architecture. The client is a single-page web application where users can create queries, monitor the progress of the queries in real time, and interact with results to explore data (Figure 3.1). We will elaborate more on the interface and interaction of the client in the next section.

To support responsive feedback with large-scale data, the server provides a prompt response (DC1) and processes queries incrementally (DC2). When a query from the client arrives, the server first returns a prompt response that contains a low-fidelity result of the query, which is built from a small sample of the data (DC1). To incrementally process the data, the server separates the whole data into n blocks, B_0 to B_{n-1}. In turn, the server splits the incoming

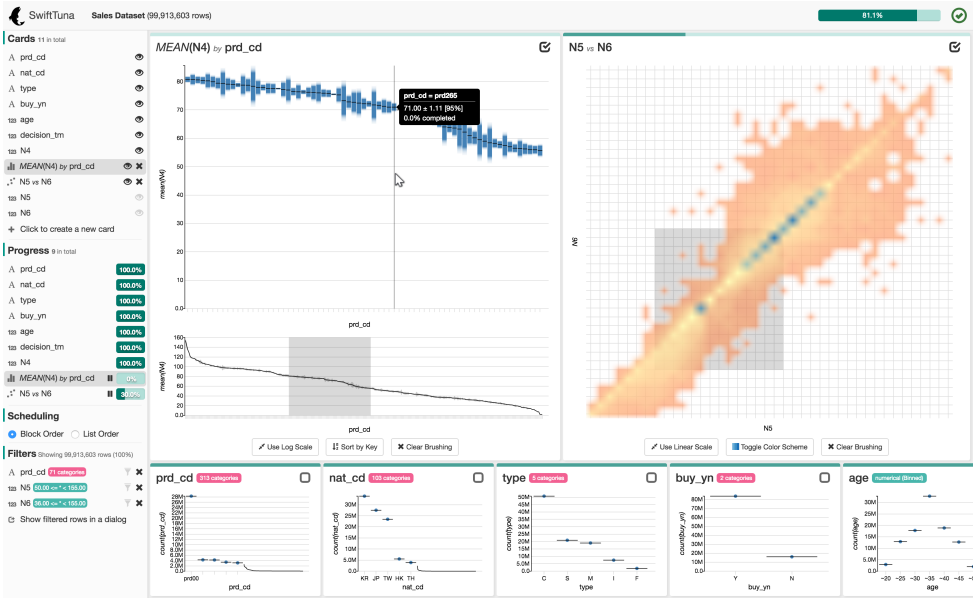


Figure 3.1: Interface of SwiftTuna. An analyst is exploring a multidimensional dataset with 100 million entities. Visualization cards (two expanded cards and six thumbnail cards) on the right side provide a univariate summary on a single dimension or visualize the relationship between two dimensions.

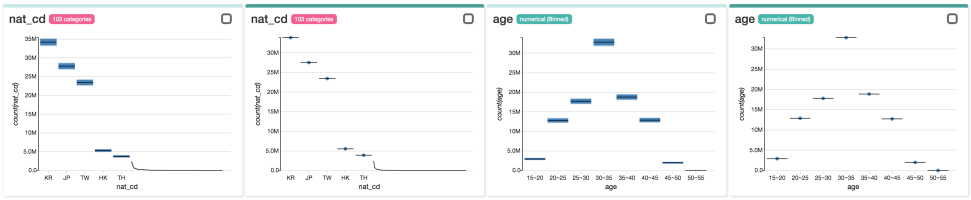
query into n jobs, with each job corresponding to each data block. These jobs are inserted into a job queue. The server takes the first job, which covers B_0, from the queue and runs the job on backend workers. Thereafter, the server polls the workers every 200 ms to check whether the job is done. Once completed, the server gathers the result from the workers and sends it back to the client. The remaining jobs in the queue (i.e., $n-1$ jobs that cover B_1 to B_{n-1}) are processed one by one in the same way.

From the client's point of view, a series of staggered responses arrives for a single query. The client accumulates and combines partial results. For example, suppose there is a query that calculates a frequency histogram of a categorical dimension, *nat_cd*, which represents the codes of nationalities. The first partial response only contains the frequency of countries in B_0.

When the second result arrives, which covers B_1, the client accumulates the frequency by comparing country names. Then, the client updates progress bars and the corresponding visualization.

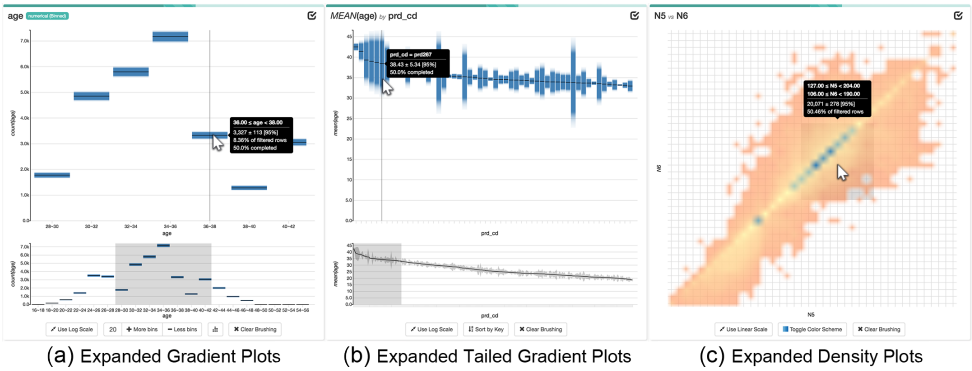
SwiftTuna currently supports four visualization-related queries, Frequency Histogram, Binned Histogram, Pivot Dot Plot, and Density Plot, as well as two data-related queries, Count and Load Raw Data. We chose the four visualizations following the concept of binned plots [79], which are known to convey global patterns and outliers well despite the size of the data. A visualization is shown in a visualization card (Figure 3.4) that serves as a basic unit of analysis. Frequency histograms (Figure 3.2a) and binned histograms (Figure 3.2c) provide a univariate summary on a categorical or numerical dimension, respectively. On the other hand, pivot dot plots (Figure 3.3b) and density plots (Figure 3.3c) are appropriate for visualizing a relationship between two dimensions. Pivot dot plots aggregate a numerical dimension with designated aggregation function (MIN, MEAN, MAX, or SUM), grouping rows by a categorical dimension. Density plots visualize the relationship of two numerical dimensions.

Since SwiftTuna separates the data into blocks and processes each block one by one, only partial results are available in the middle of the process. To allow users to quickly access the results, we estimate the final results from the partial results based on known statistical procedures (DC2). For example, since the partial results are from a sample of the population, we estimate population statistics using sample statistics (e.g., using the sample mean and the sample standard deviation to estimate the population mean). Exceptionally, we decided not to estimate the final results of pivot dot plots that use MIN or MAX aggregation because those statistics are quite sensitive to outliers and thus cannot be estimated robustly.



(a) Tailed Gradient Plots (b) Tailed Dot Plots (c) Gradient Plots (d) Dot Plots

Figure 3.2: Tailed charts and dot plots. (a) Tailed gradient plots prioritize prominent categories (e.g., the most frequent five categories) by visualizing them in half of the visual space, while the rest of the categories are outlined in another half of the space with a line (i.e., a tail). Gradients show the 95% confidence intervals of estimated values. (b) When all data are processed, the gradients eventually converge, and tailed dot plots replace tailed gradient plots. (c) Gradient plots. (d) Dot plots.



(a) Expanded Gradient Plots (b) Expanded Tailed Gradient Plots (c) Expanded Density Plots

Figure 3.3: Expanded visualization cards. (a) An expanded gradient plot (expanded from Figure 3.4c). (b) An expanded tailed gradient plot. (c) An expanded density plot. Density plots do not provide the context view for brushing, but users can directly brush on the focus view.

3.1.3 Scalable Visualization Components

To enable visual analytics on large-scale data, it is necessary to visualize results in a scalable way (R4). We present a novel visualization, tailed charts (Figure 3.2a and Figure 3.2b), as well as improving previous visualization methods [79] with effective interaction techniques such as Overview+Detail [29]. In this section, we describe scalable visualization components of SwiftTuna in detail for each query type (Table 3.1).

Query Type	Partial Results	Complete Results (Thumbnail)	Complete Results (Expanded)
Frequency Histogram 1 categorical	Tailed Gradient Plots (Figure 3.2a)	Tailed Dot Plots (Figure 3.2b)	Dot Plots (Figure 3.2d)
Pivot Dot Plot 1 numerical + 1 categorical Aggregation function			
Binned Histogram 1 numerical	Gradient Plots (Figure 3.2c)	Dot Plots (Figure 3.2d)	
Density Plot 2 numerical	Density Plots (Figure 3.3c)		

Table 3.1: Visualization Methods According to Query Types.

Binned Histograms. A binned histogram shows a univariate summary of a numerical dimension (Figure 3.2c and Figure 3.2d). SwiftTuna creates 40 bins by default, and calculates the number of rows that fall into each bin. We heuristically chose to use 40 bins, striking a balance between performance and the flexibility in resizing bins without querying the server. The 40 bins are aggregated into eight bins and visualized through a dot plot (Figure 3.2d).

Frequency Histograms and Pivot Dot Plots. A frequency histogram shows the distribution of categories in a categorical dimension, while a pivot dot plot visualizes aggregate values of a numerical dimension (e.g., mean) over a categorical dimension. Both visualizations are different from binned histograms in that the values on the x-axis are categorical and thus cannot be aggregated. Therefore, the x-axis often becomes crowded as the number of categories increases, which is a frequent situation in large-scale data analysis. As a remedy, we design a novel visualization, tailed charts (Figure 3.2a and Figure 3.2b). Tailed charts prioritize prominent categories (e.g., the most frequent five categories) by visualizing them with salient visual elements in half of visual space, while the rest of the categories are outlined in another half of the space with a line (i.e., a tail). We designed two variants of tailed charts: tailed gradient plots (Figure 3.2a) and tailed dot plots (Figure 3.2b), to summarize a large number of categories on the x-axis. We believe tailed charts allow users to identify the most prominent values effectively as well as understand the overall distribution of all data.

Density Plots. A density plot shows the relationship between two numerical dimensions (Figure 3.3c). We create 1,600 bins (40 bins for each axis) and count the number of rows for each bin. We ensure the minimum opacity of a bin to 0.5 if the bin has at least one row, as in a previous work [66]. The cen-

ter points of bins are color-coded by a univariate or bivariate color scheme. Bilinear interpolation is used to color-code pixels between the center points.

Visualizing Uncertainty. Since SwiftTuna processes queries incrementally and estimates the final results through statistical procedures, it is essential to maintain graphical integrity by revealing the errors of estimated values. One possible approach is to overlay traditional whisker-based error bars on visualizations. However, recent studies revealed that the traditional error bars suffer from perceptual issues [35, 48]. Thus, we adopt an alternative, the gradient plots [35], as a primary visual component for encoding errors, which are proven to be perceptually more robust.

When queries are initially issued, gradient plots (Figure 3.2c) and tailed gradient plots (Figure 3.2a) visualize low-fidelity results, showing 95% confidence intervals. As in a previous work [35], a 95% confidence interval is filled fully opaquely, and the opacity decreases following the inverse cumulative probability function. As more blocks are processed and the amount of error decreases, the gradients shrink vertically. When all data are processed, the gradients eventually converge to look like thin horizontal bars. Finally, they disappear, and the gradient plot is replaced with a dot plot (Figure 3.2d) or a tailed dot plot (Figure 3.2b) according to the number of values on the x-axis. We choose to use dots instead of bars because they are as effective as bars and more consistent with gradient plots in that both dot plots and gradient plots use position encoding. Although we do not estimate the results for MIN and MAX aggregation, we opt to show a one-sided gradient with a fixed height, indicating the results are incomplete.

To visualize errors of tails in the tailed charts, we opt to connect and fill 95% confidence intervals of categories without using gradients. Since the

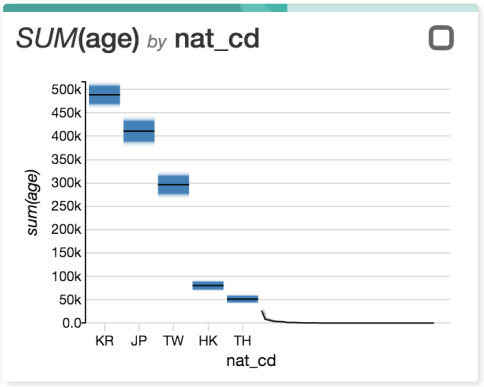


Figure 3.4: A visualization card showing the sum of a numerical dimension (*age*) over a categorical dimension (*nat_cd*).

width of each gradient often becomes less than one pixel, the filled area is clearer to read than gradients.

3.1.4 Visualization Cards

A visualization card is a small zoomable widget that serves as a basic unit of analysis (Figure 3.4). The main purpose of the visualization card is to help users manage queries in a visual form and explore results interactively (DC4). A visualization card corresponds to one query. The title of a card describes its query in a short string format. Below the title, the result of the query is visualized. On the top edge of the card, a progress bar shows the current progress of the query. Users can hover the mouse cursor over the progress bar to check the number of rows that have already been processed, are being processed, and will be processed.

As shown in Figure 3.4 and Figure 3.2, a visualization card is initially in a thumbnail mode, showing the result and progress of its query in a compact view. Users can expand a card to see details and interact with it in a larger view by clicking on the checkbox on the top-right corner (Figure 3.3a). When

a card is expanded (i.e., an expanded mode), the visualization is split into two sub views, a focus view and a context view, and a control panel appears below the context view.

All bins (for binned histograms) or categories (for frequency histograms or pivot dot plots) are shown in the context view. Users can brush on the context view to examine the details of the brushed area in the focus view. If at least 20 pixels in width cannot be allocated to each category because of the large number of categories or a short screen width, we use a line and connect the confidence intervals over categories instead of dots and gradients, as in tails of tailed charts (Figure 3.3b). For the density plots, we do not provide the context view because it is certain that the number of bins on the x-axis is not too many (i.e., 40 bins for each axis). Instead, users can create a 2D brush directly on the focus view.

Users can also fine-tune the visualization of a card in the control panel. The possible options vary depending on query types such as options to switch between a linear scale and a log scale on the y-axis or a color scheme (for all cards), change the number of bins (up to 40, for binned histograms), sort the x-axis alphabetically or by value (for frequency histograms and pivot dot plots), and clear a brush (for all cards). Users can regard a numerical but discrete dimension as either categorical or numerical dimension. For example, suppose there is a numerical dimension, age, given in integer. If age remains numerical, a binned histogram visualizes the distribution of age. Otherwise, if age is considered as a categorical dimension, a frequency histogram shows the occurrence of an individual age value. For density plots, users can switch a color-coding scheme from a univariate one to a bivariate one to spot low-density regions in a salient color that can be a clue to outliers. Note that all

interactions in a control panel are handled by the client without querying the server, preventing users from waiting longer.

3.1.5 User Interface and Interaction

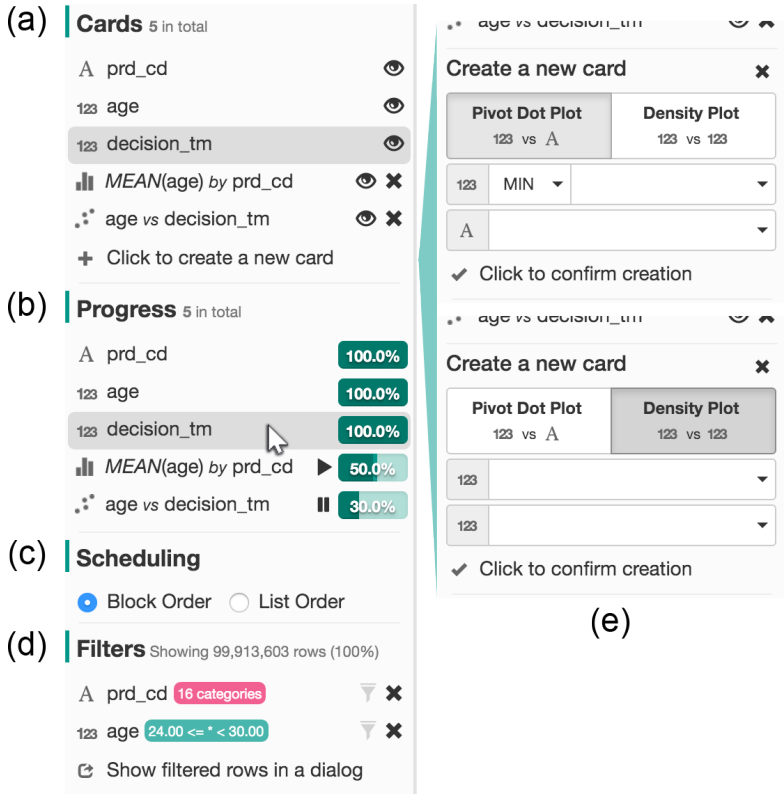


Figure 3.5: Card list panel. (a) The card list shows the list of visualization cards. (b) The progress list illustrates the progress of each card with a progress bar. Users can stop or resume processing the query of a card by clicking on stop and play icons, respectively. (c) Users can prioritize queries with two options: block order and list order. We elaborate on scheduling in Section 3.2. (d) Each time users brush on a visualization, a filter that represents the brushed area is added to the filter list. Users can click on a funnel icon to activate the filter. (e) Users can create a new card for two dimensions (e.g., for a density plots between two numerical dimensions) by clicking on a plus icon at the bottom of the card list.

As shown in Figure 3.1, SwiftTuna's interface mainly consists of two components, the card list panel on the left side, and the main workspace on the

right side. We decided to create initial visualization cards (a visualization card corresponds to a single query) for every dimension in data as a starting point for users' exploratory analysis. Users can create, remove, or prioritize the visualization cards in the card list panel while they can interact further with them in the main workspace. The card list panel supports multidimensional data exploration (DC4). Users can manage visualization cards in a list form, schedule queries, and apply filters (DC3).

Card List. At the top of the card list panel, all visualization cards are listed with icons that abstract queries (Figure 3.5a). Users can hide a visualization card that is out of interest by toggling an eye icon next to its name. Hidden cards are excluded from querying, reducing the workload of the server. To create cards for conjunctive visualizations such as pivot dot plots and density plots, users can click on a plus icon at the bottom of the card list and specify two dimensions in an appeared widget (Figure 3.5e).

Progress List. Users can interactively manage the processing order of visualization cards in the progress list (DC3, Figure 3.5b). The progress list shows a priority list of cards and users can reorder it through drag-and-drop interaction. When they decide that the query of a visualization card is processed enough (e.g., if the confidence intervals are narrow enough), they can stop processing the query by clicking on a pause icon. These features allow users to flexibly schedule the job queue of the server.

Filter List. Users can explore a subset of data by applying filters. A filter has a condition defined by either a range (e.g., $[20, 30]$ for a numerical dimension age) or a list of categories (e.g., $\{KR, JP\}$ for a categorical dimension *nat_cd*). They can create a filter by brushing on a visualization card. Then, the filter is added to the filter list, displaying its condition (Figure 3.5d). When users activate a filter by clicking on the funnel icon next to its con-

dition, all cards visualize only rows that satisfy the condition of the filter. SwiftTuna supports conjunctive filtering (i.e., combination of multiple filters). Since SwiftTuna does not build precomputed data structures for a determined set of dimensions as in data cubes, users can apply multiple filters on every dimension in data simultaneously. For more detailed analysis, users can inspect the filtered raw data in a paged list by opening a data viewer (the bottommost button in Figure 3.5d).

Main Workspace. The main workspace allows users to examine expanded visualization cards in detail while keeping other cards accessible through horizontal scrolling (DC4, Figure 3.1). The main workspace presents visualization cards that are not hidden in the card list panel (Figure 3.5a). Initially, all cards are in a thumbnail mode, positioned in a grid-like view with four cards in a row. When users expand some of cards, the expanded cards are horizontally arranged in the upper part of the main workspace, and remaining cards shrink and move to the bottom part of the workspace.

3.2 Responsive Querying

Inspired by Fisher's workflow for incremental visualization system [46], we adopt the incremental querying approach to achieve responsive querying for visualization-related queries for each type of visualizations (i.e., frequency histograms, binned histograms, pivot dot plots, and density plots).

3.2.1 Querying Pipeline

When visualization cards need to be updated (e.g., users activated a filter or added a new card), the client requests queries with each query corresponding to each card. For example, if users apply a filter on data, all visible vi-

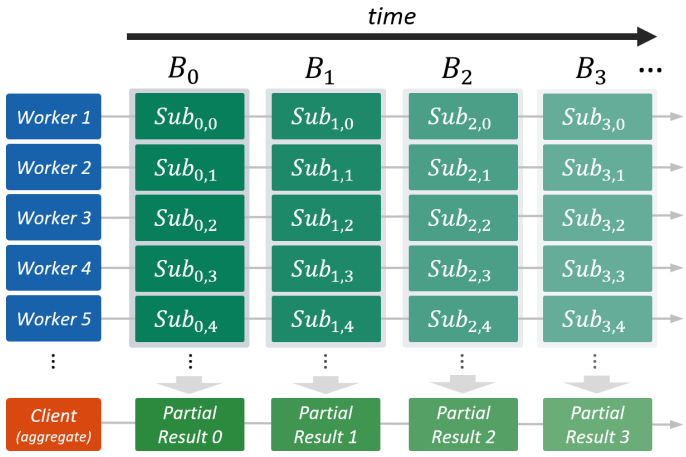


Figure 3.6: Massive parallel processing in SwiftTuna. SwiftTuna separates raw data to n blocks (i.e., B_0 to B_{n-1}), and processes each block in a parallel and distributed manner.

sualization cards should be updated; therefore, the client requests as many queries as the visible cards.

To understand the querying pipeline of SwiftTuna, suppose that an analyst issued two queries, Q1 for a frequency histogram of a categorical dimension *nat_cd* and Q2 for a binned histogram of a numerical dimension age. When the queries arrive at the server, each query is split into n jobs where n is a tunable number of blocks. To alleviate a possible bias in raw data, a processing index (from 0 to $n-1$) is randomly assigned to each block without duplication, and blocks are processed in this random order. We denote a job as $J(Q_i, B_j)$ for a query Q_i and a block B_j where i is the index of a query and B_j is the j-th $(0 \leq j < n)$ block in the random processing order. In the above scenario, if the number of blocks, or n, is 10, 20 jobs are created from the queries: $J(Q_i, B_j)$ where $i = 1, 2$ and $0 \leq j < 10$.

Then, the created jobs are inserted into a job queue in a certain order. SwiftTuna supports two scheduling modes: block order and list order. The default block order prioritizes blocks over queries. In this order, the server

executes jobs with smaller processing indices first. For example, after finishing the first job, $J(Q_1, B_0)$, the server runs $J(Q_2, B_0)$ rather than $J(Q_1, B_1)$. The block order is useful when obtaining early results of all queries. In contrast, the list order prioritizes queries over blocks, meaning that the server completes all jobs related to the first query and moves to the next query. For example, in the list order, $J(Q_1, B_1)$ is executed after $J(Q_1, B_0)$. Users can switch the scheduling mode and reorder the priority of queries in the card list panel (Figure 3.5c).

When users apply filters, two additional procedures occur. First, the server represents the filters using SQL syntax and appends it to all jobs as an argument. This allows workers to filter out the data. Second, a counting job, which counts the number of rows that satisfy the filters, is prepended to the job queue. The result of the counting job is sent to the client and used to display the number of filtered rows.

Basically, the server takes a job, $J(Q_i, B_j)$, from the job queue and runs it on the cluster. Here, since we have a single job and multiple workers, we need to split the job again into tasks to leverage the parallel processing of Apache Spark. The server separates block B_j into a designated number of sub blocks (e.g., the number of workers in the cluster), and creates tasks with each task corresponding to each sub block. Then, the workers run the tasks in parallel (Figure 3.6). The server gathers the results from the workers and sends them back to the client as an incremental response for the query Q_i. Note that at this time, the client has incremental responses for the query Q_i on the blocks B_0 to B_j. If the query is not completed (i.e., $j < n - 1$), the client visualizes the responses with error indicators (e.g., gradient plots) and shows incomplete progress bars.

3.2.2 Prompt Responses

As the importance of a quick response has been advocated in a previous study [48], SwiftTuna provides prompt responses to enable users to visually confirm queries in a very early stage of analysis. Prompt responses for queries are built from a small sample of the entire data stored on the server without passing the queries over to workers. This feature allows the server to react to the queries almost instantly, helping users to focus their attention on analysis. When two or more queries are requested, the server packs all prompt responses for the queries and transfers the packed result back at once, reducing the network overhead.

When the client receives a prompt response for a query, it shows initial visualizations based on the prompt response. After a few seconds, when the first incremental response for the query arrives at the client, the client discards the prompt response and replaces it with the first incremental response. This is because when the query uses binning (e.g., binned histograms), it is impossible to merge the two responses (i.e., the prompt response and the first incremental response) due to the different sizes and ranges of the bins.

3.2.3 Incremental Processing

Although the first incremental response replaces a prompt response, the client accumulates the remaining incremental responses and estimates the final results. The detailed procedures vary depending on the query type. In this section, we explain the accumulative estimation per query type.

For count queries such as frequency histograms, the accumulation procedure is straightforward. Given two incremental responses, we compare the frequency of each category and add up two frequencies if the same category exists in both responses. For the estimation, since we already know the

number of all rows in the data, we approximate the final results and its 95% confidence intervals [80].

Binned histograms and density plots are similar to frequency histograms in that they count the number of rows. However, to accumulate incremental responses for those queries, the sizes and ranges of the bins used in the responses have to be chosen before processing. We could use a fixed range for each numerical dimension, but we found that using a fixed range often yielded unsatisfactory binning results especially when a number of rows in data were filtered out. As a remedy, SwiftTuna adds an additional job, a range job, before processing those queries, which calculates the minimum and maximum values of a numerical dimension only with remaining rows after filtering. Then, the size and ranges of bins are determined by uniformly dividing the interval between the minimum and maximum values of the dimension. The estimation step is the same as that of the frequency histogram query since they are all count estimates.

Accumulating responses for pivot dot plots is a bit different. Pivot dot plots aggregate a numerical dimension over a categorical dimension. Currently, four aggregation functions are supported: MIN, MAX, MEAN, and SUM. Merging two incremental responses for MIN and MAX functions is straightforward. For a category in both responses, we compare two MIN (or MAX) values in the responses and choose the smaller (or larger) one according to the aggregation function. As mentioned before, we decided not to estimate the final MIN and MAX values because they are quite sensitive to outliers.

For MEAN and SUM functions, the server provides three values for each category in incremental responses: the frequency of the category, the sum of a specific numerical dimension, and the squared sum of the dimension.

When a SUM function is chosen, the second value (i.e., sum) is directly visualized. Otherwise, for a MEAN function, we calculate the mean by dividing the second value (i.e., sum) by the first value (i.e., the frequency of a category). The squared sum is used to calculate variance and 95% confidence intervals for mean and sum estimates.

3.3 Evaluation: Performance Benchmark

To evaluate our system in terms of scalability and responsiveness, we conducted performance benchmarks using a real-world dataset.

3.3.1 Study Design

Since we could not find a distributed and incremental visualization system that is publicly available, we evaluated the feasibility and performance of our system with a real-world large-scale dataset. We supposed a practical scenario where an analyst remotely queries the data with a laptop in an office.

Cluster. A cluster was hosted on a cloud computing environment, Amazon Elastic Compute Cloud (EC2). We created 16 spot instances of the r3.8xlarge tier, which was optimized for memory-intensive application, in the ap-northeast-1 region (Tokyo). Each instance was equipped with Intel E5-2670 v2 (32 vCPUs), 244 GB of main memory, and two 320 GB SSD storages. To organize a computing cluster, we used Hadoop 2.4.0 and Apache Spark 1.6.0 in a standalone mode. Among 16 instances, one served as a master and the remaining 15 instances acted as workers. All instances ran on Amazon Linux AMI 2013.03.

Dataset. We used Criteo's Terabyte Click Logs dataset [37], which contained sampled click feedback of online advertisings (ads) for 24 days. The dataset was 1.03TB in a tab-separated format and had 4.3B entries with 40 di-

mensions. The 40 dimensions consisted of a binary dimension, 13 numerical dimensions, and 26 categorical dimensions. We excluded the binary dimension from benchmarks because it had only two different values. Since the dataset was all anonymized, we named the numerical dimensions as N0 to N12, and the categorical dimensions as C0 to C25. We filled all missing values with an integer 0 (for numerical dimensions) and a string "empty" (for categorical dimensions).

We considered the number of blocks (i.e., the separated pieces of data for incremental processing), n, as an independent factor and tested our system under two different values of n: 240 and 2,400 blocks. Since the raw data was provided in 24 similar-sized chunks (one chunk for one day), we divided each chunk into 10 or 100 blocks. All blocks were represented in a columnar format (i.e., Parquet [37]) and stored in the distributed file system of the cluster. During the benchmarks, the blocks were loaded in the main memory of the cluster, enabling in-memory calculation. For prompt responses, we randomly sampled 0.001% of data (about 10MB, 0.001% of 1.03TB), and kept the sample on the master of the cluster.

Client. The client was a 15-inch 2015 Mid Mac Book Pro (OS X 10.11.3; 2.8 GHz Intel Core i7 CPU with 16 GB of main memory). The client was located in Seoul, Korea, and connected to the Internet through Wi-Fi. The client ran a web browser (Google Chrome 49.0.2623.87) and connected to the server. We injected additional codes into the client to make it automatically request queries, and record timestamps each time a response was returned.

Query. We chose four numerical dimensions (N6, N5, N3, and N11) and four categorical dimensions (C25, C7, C3, and C4) that had various ranges and cardinalities. Using those dimensions, we tested 14 queries that consisted of four binned histograms (Q1 – Q4), two density plots (Q5 and Q6),

four frequency histograms (Q7 – Q10), and four pivot dot plots (Q11 – Q14) (Tables 2 and 3).

Measurement. We mainly focused on measuring the perceptual latency of our system rather than using system performance metrics. We measured 1) the range or cardinality of related dimensions, 2) the latency of prompt responses (the time from when users requested a query to when the first feedback on the query was shown), and 3) the mean interval between two successive incremental responses. Since we supposed an analyst accesses a large dataset remotely through Wi-Fi, additional delays could be included in the measurement, such as network latency resulting from the wireless connection, or the distance between Seoul and Tokyo, which we believe to better reproduce real-use cases. We repeated each query at least 40 times (100 times for prompt responses and measurements with 2,400 blocks, 40 times for measurements with 240 blocks). For all time measurements, we calculated 5% trimmed mean, which discarded 5% of measures from the highest and lowest, respectively.

Range Approximation. For the binned histograms (Q1 – Q4) and density plots (Q5 and Q6), the ranges of bins (40 bins for the binned histograms and 1,600 bins for the density plots) should be determined before actually counting the number of rows for each bin. As explained in Section 3.2.3, we prepended an additional job, range job, into a job queue that calculates the minimum and maximum values of a given dimension for the entire data. However, in our preliminary benchmark, we found that such additional jobs took a few minutes, hurting the responsiveness of the system. Therefore, we decided to calculate the range of a dimension only using the first block, not using the entire data. Since the ranges of bins were approximated only using the first block, some outlying values that were out of the approximated

ranges can be found while processing the remaining blocks. We made such values fall into one of the two extreme bins (i.e., the first bin that had the minimum value, and the last bin that had the maximum value) and counted the number of such rows as shown in Table 3.2 (Out of Range).

3.3.2 Results and Discussion

The results for binned histograms and density plots are shown in Table 3.2. For the measurements that were affected by the number of blocks (n), we presented two numbers in a single cell: the number out of parentheses was measured when n was 2,400, while the number in parentheses was measured when n was 240.

On average, users were able to receive a prompt response within 0.5 seconds regardless of the range of a queried dimension and the type of the query. When data were split into 2,400 blocks (i.e., $n = 2{,}400$), a block covered approximately 1.75 million rows. About two seconds were required to either calculate the range of a dimension on the first block (to determine the range of bins) or build a binned histogram for one block. This means that users could grasp the first incremental response on 1.75 million rows in four seconds after receiving prompt responses, and the next incremental responses followed every two seconds.

With respect to the granularity of a block, a smaller-size block ($n = 2{,}400$) yielded faster responses. However, a bigger-size block was preferred in terms of throughput. For example, when n was 2,400, 1.78 seconds were taken to sweep 1.75 million rows in a block and create a binned histogram for N6 (Q1). However, it took only two times longer (3.52 seconds) to process ten times more rows (17.5 million rows) when n was 240. This implies there was

	Type	Dimension	Range	Prompt Responses (s)	Range Approximation (s)[a]	Out of Range	Incremental Responses (s)[a]
Q1	Binned	N6	0 - 5.2K	0.20 ± 0.028	1.89 ± 0.46 (2.54 ± 0.37)	401 (21)	1.78 ± 0.90 (3.52 ± 0.87)
Q2	Binned	N5	0 - 65K	0.19 ± 0.025	2.03 ± 0.53 (2.60 ± 0.41)	4,270 (1,073)	1.88 ± 0.97 (3.17 ± 0.75)
Q3	Binned	N3	0 - 746K	0.20 ± 0.027	1.84 ± 0.38 (2.53 ± 0.37)	74 (72)	1.77 ± 0.74 (3.44 ± 1.05)
Q4	Binned	N11	0 - 35M	0.20 ± 0.025	1.81 ± 0.35 (2.55 ± 0.40)	10,755 (5,077)	1.91 ± 0.84 (3.54 ± 1.58)
Q5	Density	N6, N5		0.30 ± 0.036	1.91 ± 0.42 (2.57 ± 0.42)	4,625 (1,089)	1.84 ± 0.88 (3.78 ± 0.77)
Q6	Density	N3, N11		0.31 ± 0.040	1.87 ± 0.37 (2.57 ± 0.40)	10,892 (5,149)	1.88 ± 0.61 (3.46 ± 1.05)

Table 3.2: Benchmarks for Binned Histograms and Density Plots. [a]Data are measurements with 2,400 blocks (and with 240 blocks). **Type**: the type of requested visualizations, binned histograms (Binned) or density plots (Density). **Range**: true range of a dimension. **Range Approximation**: mean time taken to approximate the range of a dimension using only the first block. **Out of Range**: number of values of a dimension that were out of the approximated range while processing remaining blocks after the first one. **Incremental Responses**: mean interval between two successive incremental responses.

	Type	Dimension	Cardinality	Prompt Responses (s)	Incremental Responses(s)[a]
Q7	Frequency Histogram	C25	36	0.19 ± 0.028	1.62 ± 0.15 (2.66 ± 0.30)
Q8	Frequency Histogram	C7	1.4K	0.27 ± 0.039	2.79 ± 1.00 (3.50 ± 0.65)
Q9	Frequency Histogram	C3	7.4K	0.27 ± 0.044	2.76 ± 0.85 (3.94 ± 1.21)
Q10	Frequency Histogram	C4	20K	0.39 ± 0.058	2.85 ± 0.78 (3.93 ± 1.31)
Q11	Pivot Dot Plot	MEAN(N11) by C25		0.20 ± 0.024	1.82 ± 1.06 (3.17 ± 0.31)
Q12	Pivot Dot Plot	MIN(N11) by C25		0.20 ± 0.023	1.89 ± 1.09 (3.59 ± 1.35)
Q13	Pivot Dot Plot	MEAN(N11) by C7		0.38 ± 0.051	2.96 ± 1.30 (4.19 ± 1.01)
Q14	Pivot Dot Plot	MEAN(N11) by C3		0.52 ± 0.085	2.53 ± 1.21 (3.88 ± 0.93)

Table 3.3: Benchmarks for Frequency Histograms and Pivot Dot Plots. [a]Data are measurements with 2,400 blocks (and with 240 blocks). **Incremental Responses**: mean interval between two successive incremental responses.

an overhead that cannot be efficiently parallelized, such as network latency or the time taken for communication between nodes.

We approximated the minimum and maximum values of a dimension only using the first block, not processing the entire data, to rapidly determine the ranges of bins. Only a few thousand values were out of the approximated range; however, it is quite a small number compared to the size of the entire data (4.2 billions). This means that the dimension range estimation only using the first block is sufficiently effective in the exploration of large-scale data. It would be also useful if the system highlighted those out-of-range values. For example, binned histograms and density plots can be improved to show those values with visual cues on boundaries of visualization. We leave this improvement to future work.

In Table 3.3, we presented the results of frequency histograms and pivot dot plots. Overall, compared with binned histograms and density plots, slightly longer intervals were required for each incremental response. As the cardinality of a categorical dimension increased, it took longer to create a frequency histogram of the dimension (in the order from Q7 to Q10). This can result from either the additional number of keys that should be shuffled between workers, or the longer network transmission time due to a longer list of frequencies. The mean interval between incremental responses was not significantly affected by whether an aggregation function was applied (Q7 vs. Q11) or the type of an aggregation function (Q11 vs. Q12).

Through our benchmarks, we found practical evidence that SwiftTuna could support fluent and incremental data exploration of large-scale multi-dimensional data. To allow users to begin data exploration immediately, we may precompute univariate histograms (i.e., a binned histogram or a frequency histogram for every dimension) only for the initial screen.

3.4 Implementation

The client was written in JavaScript with open-source libraries such as D3.js [24], Angular.js, and Bootstrap. The client connected to the server via WebSocket to receive incremental responses and progresses in real time. For the server, we created a computing cluster on Amazon Elastic Compute Cloud (EC2). The number and type of instances and the versions of installed software were described in Section 3.3. One instance served as a master and performed two important roles: supervising workers in the cluster as a master and running a web server to which the client connected. The web server was written in Python 2.7 upon a micro web framework, Flask. When users requested a query to the web server, the web server chose an appropriate program (i.e., a driver program) according to the query type, and ran the program on the cluster. We implemented various driver programs, with each corresponding to each query type (e.g., one for binned histograms and another for pivot dot plots). The driver programs were implemented in Scala 2.10.4 using Spark-SQL API.

3.5 Summary

This chapter proposes an interactive data exploration system, SwiftTuna, which enables fluent information seeking process on large-scale multidimensional data. SwiftTuna enables large-scale processing by exploiting an in-memory computing engine, Apache Spark, which allowed fast and scalable data processing as demonstrated in our benchmarks. To provide the responsive feedback for interaction, SwiftTuna processes queries incrementally while providing prompt responses using a small sample, delivering immediate and continual feedback. To achieve scalable visualization, interaction techniques

(e.g., Focus+Context) were integrated into visualizations that are appropriate for large-scale data (e.g., density plots). The design of two variants of tailed charts (i.e., tailed dot plots and tailed gradient plots) can improve crowded x-axes in frequency histograms. Since SwiftTuna does not resort to a preprocessing scheme (e.g., data cubes), it lends itself to multidimensional exploration through filters on multiple dimensions.

Chapter 4

PANENE: A Progressive Algorithm for Indexing and Querying Approximate k-Nearest Neighbors

This chapter [1] presents a novel progressive algorithm, PANENE, for Progressive Approximate k-NEarest NEighbors, answering the second research question (***Horizontal Scalability**: how do we responsively embed and visualize high-dimensional data on a 2D space without long initial computation delays?*).

Progressive data analysis has recently gained in popularity due to its ability to deliver ongoing results before the whole computation is completed [45, 120]. However, despite the advantages, it is not always simple or even possible to convert a sequential algorithm directly to a progressive one. Such a hurdle hinders the applicability of progressive computation to a wider range of data analyses. In this chapter, we address one important problem: pro-

[1] The preliminary version of Chapter 4 was published as a journal article [65] in IEEE Transactions on Visualization and Computer Graphics (TVCG) followed by a workshop paper [64].

gressively finding *k*-nearest neighbors of a given point in a multidimensional space, i.e., the *k*-nearest neighbor problem. In contrast to sequential or online algorithms that have been proposed, PANENE is *progressive*; it guarantees to finish its operations in a given number of cycles and thus does not block the whole system when loading or processing data continuously. It allows us to bring useful machine learning methods, such as t-SNE [84], into visual analytics, which has been limited due to their long computation time.

The *k*-nearest neighbor (KNN) problem is an optimization problem of finding the k closest points to a query point in a multidimensional metric space. Formally, given N points $P = \{p_1, \cdots, p_N\}, p_i \in \mathbb{R}^D$, and a query point $q \in \mathbb{R}^D$, a KNN search finds the k-nearest points of q in P. Formally, this operation can be stated as follows:

$$KNN_k(q) \mapsto \{i_1, i_2, \cdots i_k\}, \text{ where } i_j \in [1, N]$$

$KNN_k(q)$ is a set of indices that satisfy the following condition:

$$\forall i \in KNN_k(q)\ \forall j \in [1, N] - KNN_k(q), \|q, p_i\| \leq \|q, p_j\|,$$

where $\|q, p\|$ is the distance between q and p. The KNN problem is a building block of many data mining and visualization methods, such as clustering [122], classification [103], embedding [106], and non-parametric density estimation [51]. Thus, designing an efficient progressive algorithm for the KNN problem is an important step towards extending the applicability of progressive computation.

One straightforward approach is to calculate the distances from a query point q to every point in the dataset and take the k closest. However, this method is inefficient, since it has to iterate over all points and thus has time complexity of $\theta(N)$. A more efficient approach is to use a search data struc-

ture or an indexing method such as a k-d tree, which usually reduces the time complexity to a logarithmic scale.

In recent years, there have been important advances in data structures and algorithms to speed up KNN queries. These advances have mostly focused on optimizing the query time, considering that the indexing was done once for all data points and thus the indexing time was less important than the query time [10]. However, for progressive systems, both times are important because data can be loaded progressively, the KNN queries can be done progressively, and therefore the index should be updated progressively as well.

A popular approach to improve the query time is to compute approximate k-nearest neighbors instead of exact ones. Approximate k-nearest neighbor search (AKNN) techniques are more efficient than exact KNN, but all of them also require building an index. For example, the most efficient method to date, the hierarchical navigable small-world graph (HNSW) [86], needs all data points to be loaded upfront and a special graph structure to be built before querying. From the visual analytics point of view, such a precomputation leads to long loading time, hampering the interactivity of the entire system. Only few AKNN techniques, such as FLANN [96], support *online* updates; they allow inserting new points even after an index is built. However, this is not sufficient for visual analytics because the insertion time is not bounded. Indeed, we observed that FLANN pauses longer than 10 seconds to update its index with a few hundred thousand points, exceeding the time limit to keep the user's attention [99].

In this chapter, we present a *progressive* algorithm for indexing and querying approximate k-nearest neighbors. Our algorithm computes k-nearest neigh-

bors iteratively with each iteration finishing in a given number of operations (i.e., progressively). The contributions of this chapter are

- a progressive k-d tree data structure that can sustain a controlled latency while it is created, updated, and queried;
- an algorithm for building and updating a lookup data structure that we call a KNN lookup table, which enables constant-time lookups for KNN queries; and
- progressive applications of PANENE, including regression, density estimation, and responsive t-SNE [84].

This chapter is organized as follows: we first review previous approaches to the KNN problem and discuss new challenges and requirements in interactive visualization systems. In Section 4.1, we show how we improve the sequential k-d tree algorithm to become first online and then progressive, which is the first contribution of PANENE. In Section 4.2, we elaborate on the second contribution of PANENE: KNN lookup tables, meant to speed up repeated KNN queries. In the following two sections, we evaluate the performance of PANENE through benchmarks (Section 4.3) and present applications in interactive analysis (Section 4.4). Finally, we discuss the limitations of this work and future work.

4.1 Approximate *k*-Nearest Neighbor

In this section, we first describe a sequential algorithm using randomized k-d trees for approximate k-nearest neighbor (AKNN) search, and then improve it, to become first online and then progressive. Among many algorithms mentioned in the related work section, we chose to improve a k-d tree because 1) it is known to be efficient and yet easy to implement [95] and

2) an online version of the algorithm is available as open-source [49], so we could directly compare our progressive version to the online version.

4.1.1 A Sequential Algorithm

A k-d tree is a binary tree built by recursively partitioning a multidimensional space using axis-aligned hyperplanes [20] and used thereafter to search, guaranteeing $O(\log N)$ search time and $O(N \log N)$ build time. At the root node, the algorithm chooses a *cutting* dimension that has the largest variance and assigns points to child nodes: The points whose values on the cutting dimension are less than the median are assigned to the left child, and the remaining points are assigned to the right child. This procedure repeats until only one point remains in a node. In the randomized k-d tree forest, we randomly choose a cutting dimension among the top n dimensions with the largest variance, typically, $n = 5$ (see Algorithm 1). This allows building multiple randomized trees for the same data and representing neighborhood in high-dimensional spaces more effectively.

Algorithm 1 has three strong limitations: First, it requires all the points in L to be already loaded in main memory before it can build the randomized k-d trees. Such a constraint forces analysts to wait until all data is read from disk or database before performing any analysis. Second, once the data structures are built, the algorithm does not allow any modification, such as inserting new points or removing points. Finally, the running time of the algorithm solely depends on the size of input (i.e., l), and thus its latency cannot be controlled.

Algorithm 1 A sequential algorithm for recursively building a randomized k-d tree of the given l points in L

```
 1: procedure BuildSequential(L)
Input: L is a list of l points of D dimensions.
Output: The root node of a randomized k-d tree
 2:     if L has only one point then
 3:         node ← a new leaf node
 4:         node.point ← L[0]
 5:         return node
 6:     end if
 7:
 8:     node ← a new internal node
 9:     calculate the variance of each dimension in L
10:     node.cutdim ← a random dimension with large variance
11:     node.cutval ← median([p[node.cutdim] for p in L])
12:
13:     left ← [p for p in L if p[node.cutdim] ≤ node.cutval]
14:     right ← [p for p in L if p[node.cutdim] > node.cutval]
15:
16:     node.left ← BuildSequential(left)
17:     node.right ← BuildSequential(right)
18:     return node
19: end procedure
```

4.1.2 An Online Algorithm

In contrast to a sequential algorithm, an online algorithm allows adding new points to the trees even after they are built. This benefits interactive systems in that analysts do not have to wait until all data is loaded. Rather, the data is split into batches, loaded onto the system incrementally, and can be used for further online algorithms. Analysts can access the running result between the batches, obtaining improved approximations of the final results.

However, we only found one implementation of online AKNN: the FLANN library [96]. It can build an initial k-d tree of the points in the first batch using Algorithm 1, and other points can be added into the tree thereafter.

The insertion procedure of the FLANN library is very akin to that of a binary tree: Starting from the root node, each point moves to either the left or the right child of an internal node by comparing its value at the dimension chosen to split the values at that node—called the *cutdim*—against the median value computed initially for that node—the *cutval*— until it reaches a leaf node. Then, the leaf node becomes an internal node, and the two points (i.e., the point being inserted and the point of the leaf node) become the children of the former leaf node. Algorithm 2 describes the insertion procedure in more details.

Algorithm 2 An algorithm for inserting a new point p into a randomized k-d tree with a root node *node*

```
 1: procedure InsertPoint(node, p)
Input: node is the root of a k-d tree
Input: p is a new D-dimensional point
Output: p is inserted as one of the leaf nodes in the tree
 2:     if node is a leaf node then
 3:         mark node as an internal node.
 4:         calculate the absolute difference between p and node.point at each dimension
 5:         choose a cutdim dimension with the largest difference
 6:         cutval ← (p[cutdim] + node.point[cutdim])/2
 7:         if p[cutdim] ≤ cutval then
 8:             node.left ← a new leaf containing p
 9:             node.right ← a new leaf containing node.point
10:         else
11:             node.left ← a new leaf containing node.point
12:             node.right ← a new leaf containing p
13:         end if
14:         return
15:     end if
16:
17:     if p[node.cutdim] ≤ node.cutval then
18:         InsertPoint(node.left, p)
19:     else
20:         InsertPoint(node.right, p)
21:     end if
22: end procedure
```

As more points are inserted into a k-d tree, it can become unbalanced, lengthening the query time. In the FLANN library, the distribution of the points in the first batch heavily affects the overall performance, since they are used to build the "skeleton" of the tree. At worst, if all the updates after the first batch are skewed to one side of the k-d tree, all the remaining points are inserted in a linked list, and the search time becomes linear with the number of points. This implies the need to rebalance the tree when it is too unbalanced. In real cases, the imbalance is never that extreme but can vary substantially if the data added has a different distribution than the initial tree. The imbalance leads to a slower query time with little degradation of the result quality. On the other side, when updating a tree for a large dataset, assuming the data is stationary, the distribution of incoming data will at some point converge to the distribution of the whole dataset, and the tree will remain balanced even after new points are inserted.

FLANN's implementation of k-d trees uses a simple strategy for rebalancing the trees: It reconstructs all trees each time the dataset doubles in size from the initial dataset (i.e., the first batch). Therefore, the k-d trees can become unbalanced as new data is loaded but eventually will be reconstructed. When loading a large dataset progressively, even if the incoming distribution matches the current k-d tree structure, FLANN will always reconstruct its k-d trees when the dataset doubles in size. To sum up, the FLANN implementation suffers from three problems:

1. the k-d tree may become unbalanced when data is added, leading to longer KNN searches;
2. the k-d tree is always reconstructed when the dataset doubles in size, leading to long interruptions in the KNN search at unpredictable moments; and

3. the k-d tree is always re-created when the dataset doubles in size, even when it remains balanced.

4.1.3 A Progressive Algorithm

To overcome the limitations of online k-d trees, we made three main changes to the FLANN algorithm:

1. we maintain a quality measure for each k-d tree,
2. we construct a fresh and balanced k-d tree with all the points when the measure reaches a bad quality threshold, and
3. the construction is done in a task parallel/interleaved with the query task, thus spreading the load and avoiding brutal changes in query times. When a new k-d tree is built, we drop the most unbalanced one and replace it with that new one.

To estimate the quality of a k-d tree, we use the following method: Assume a k-d tree of size N is balanced; its depth is $\lceil \log_2 N \rceil$. The points are stored as leaves, so accessing a point will require $\log_2 N$ operations. When a k-d tree becomes unbalanced, its depth will vary, and the query time for a point p will be proportional to the depth of p. On average, the query time for accessing p is $\phi_p \times \mathrm{depth}(p)$, where ϕ_p is the probability of searching p and $\mathrm{depth}(p)$ is the depth of p in the tree. On a balanced k-d tree, the cost of querying for an arbitrary point is $\log_2 N$, whereas for a specific k-d tree T, the cost $c(T)$ is

$$c(T) = \sum_{p \in T} \phi_p \times \mathrm{depth}(p) \tag{4.1}$$

The *loss* of a tree is thus the difference between the actual cost and the lower bound, i.e., the number of additional operations we should perform to search a point on average. To decide when we should trigger the computation

of a fresh tree, this loss should be compared to the cost of rebuilding the whole tree: $N \log_2 N$. We compute the loss during each query (i.e., $c(T) - \log_2 N$) and accumulate the loss throughout all trees. Once the accumulated loss exceeds a threshold or a specific proportion of the rebuilding cost (i.e., $\alpha \times N \log_2 N$, where α is a reconstruction weight), we start the reconstruction.

In practice, we do not compute Equation 4.1 for every update but maintain the cost incrementally. For each point p, we maintain the depth of the point, $\mathrm{depth}(p)$, and the number of times the point is searched, $\mathrm{freq}(p)$. Let's define $\sum \mathrm{freq} = \sum_{p \in P} \mathrm{freq}(p)$; then ϕ_p can be calculated by $\phi_p = \frac{\mathrm{freq}(p)}{\sum \mathrm{freq}}$. Suppose that we insert a new point q into the tree with the InsertPoint procedure and that q reaches an existing point p at a leaf node. As the leaf node becomes an internal node and p becomes its child, the depth and frequency of p increase by one. The updated cost C' is computed from the current cost C as follows:

$$C' = \frac{\sum \mathrm{freq} \times C + \mathrm{freq}(p) + \mathrm{depth}(p) + 1}{\sum \mathrm{freq} + 1}$$

After the update, we increment $\mathrm{freq}(p)$ and $\mathrm{depth}(p)$ by one.

When we need to reconstruct a k-d tree (i.e., the accumulated loss exceeds the threshold), we distribute the reconstruction load across multiple iterations by building the tree *incrementally*. To this end, we implement a non-recursive version of Algorithm 1 using a build queue, allowing the whole procedure to be interleaved between iterations. The progressive reconstruction algorithm (Algorithm 3) is similar to Algorithm 1, except that recursive calls are replaced with insertion into the queue.

To achieve progressiveness, the algorithm should work only for a given number of operations and stop, allowing the system to access the ongoing results. Each time the progressive algorithm runs, it is given a certain quan-

Algorithm 3 A progressive algorithm for building a new k-d tree

```
 1: procedure InitializeBuild(L)
Input: L is a list of l points of D dimensions.
Output: returns the root node of the new tree
 2:     queue ← a new work queue
 3:     root ← a new node
 4:     queue.push((root, L))
 5:     return root
 6: end procedure
 7:
 8: procedure ProcessBuildQueue(ops)
Input: ops is the number of operations for reconstruction
Output: returns true if reconstruction is done
 9:     count ← 0
10:
11:     while count < ops and queue is not empty do
12:         node, L ← queue.pop()
13:         count ← count + 1
14:
15:         if L has only one point then
16:             mark node as a leaf node.
17:             node.point ← L[0]
18:             continue
19:         end if
20:
21:         calculate the variance of each dimension in L
22:         node.cutdim ← a random dimension with large variance
23:         node.cutval ← median([p[node.cutdim] for p in L])
24:
25:         left ← [p for p in L if p[node.cutdim] ≤ node.cutval]
26:         right ← [p for p in L if p[node.cutdim] > node.cutval]
27:
28:         node.left ← a new internal node
29:         node.right ← a new internal node
30:
31:         queue.push((node.left, left))
32:         queue.push((node.right, right))
33:     end while
34:
35:     return true if queue is empty
36: end procedure
```

tum of time, specified as a maximum number of operations that the algorithm is allowed to perform before returning ongoing results and releasing the control. The algorithm assigns a fraction of the operations to insertion tasks and the rest to reconstruction tasks. An insertion task reads one data point and inserts it into the k-d trees as described in Algorithm 2. If reconstruction is needed after insertion, the algorithm builds a new k-d tree incrementally by calling the function InitializeBuild first and the function ProcessBuildQueue in the following iterations, as described in Algorithm 3. When the new k-d tree is built, the algorithm replaces the most unbalanced tree with the new one.

Algorithm 4 describes two procedures for initializing and updating progressive k-d trees, respectively. Both procedures take the *ops* parameter, which is the number of operations allowed for each iteration. The algorithm can freely use this number of operations to perform either insertion or reconstruction tasks. It can be either specified by the user or adaptively tuned by the system to limit the latency. The update procedure takes an additional parameter τ that determines the fraction of insertion tasks over reconstruction tasks. For example, when $\tau = 0.5$, half of the operations are used (i.e., $ops/2$) to insert new points and the other half to reconstruct a tree. A progressive k-d tree with a larger value of τ will prioritize indexing new points, giving a lower priority to maintaining the trees balanced. Note that τ is only used during reconstruction; if reconstruction has not been started due to small accumulated loss, or because the trees remain balanced, regardless of the value of τ, all *ops* operations will be assigned to the insertion task.

Algorithm 4 uses an abstract data structure, *a data source*, as an input stream. A data source is a virtual list that represents N points of D dimensions. However, the data source does not need to have all points loaded at

Algorithm 4 An algorithm for initializing and building progressive k-d trees

1: **procedure** INITIALIZEPROGRESSIVETREES(*dataSource*, *ops*)
Input: *dataSource* is an abstract input stream
Input: *ops* is the number of points for initialization
Output: return k-d trees initialized with *ops* points from *dataSource*
2: *initPoints* $\leftarrow$ *dataSource*.$loadNewPoints(ops)$
3: *trees* $\leftarrow$ a new k-d tree forest built with *initPoints*
4: *updating* $\leftarrow false$
5: *loss* $\leftarrow 0$
6: *N* $\leftarrow$ *ops*
7: **return** *trees*
8: **end procedure**
9:
10: **procedure** UPDATEPROGRESSIVETREES(*dataSource*, *ops*, α, τ)
Input: *dataSource* is an abstract input stream
Input: *ops* is the number of operations allowed for an iteration
Input: α is a reconstruction weight
Input: τ is the fraction for insertion tasks
Output: a tuple of updated k-d trees and a list of newly inserted points
11: **if** all points in *dataSource* have been inserted **then**
12: **return** (*trees*, *[]*)
13: **end if**
14: **if** *updating* **then**
15: *insertionOps* $\leftarrow \tau \times$ *ops*
16: *updateOps* $\leftarrow (1 - \tau) \times$ *ops*
17: **else**
18: *insertionOps* $\leftarrow$ *ops*
19: *updateOps* $\leftarrow 0$
20: **end if**
21: *newPoints* $\leftarrow$ *dataSource*.$loadNewPoints(insertionOps)$
22: **for** *p* in *newPoints* **do**
23: *N* $\leftarrow$ *N* $+ 1$
24: **for** *tree* in *trees* **do**
25: INSERTPOINT(*p*, *tree*)
26: incrementally update the imbalance cost of *tree*
27: **end for**
28: **end for**
29: **if** not *updating* and *loss* $> \alpha \times N \log_2 N$ **then** ▷ *loss* is accumulated by a search procedure
30: *updating* $\leftarrow$ *true*
31: *loss* $\leftarrow 0$
32: *newTree*$\leftarrow$INITIALIZEBUILD(*dataSource*)
33: **end if**
34: **if** *updateOps* > 0 **then**
35: *done* $\leftarrow$ PROCESSBUILDQUEUE($updateOps$)
36: **if** *done* **then**
37: replace the most unbalanced tree in *trees* with $newTree$
38: **end if**
39: **end if**
40: **return** (*trees*, *newPoints*)
41: **end procedure**

the beginning; it can load some of them when needed. The $loadNewPoints$ function (e.g., line 21 in Algorithm 4) loads a given number of points on demand, avoiding uncontrolled latency resulting from a full initial loading. In addition, the data source abstracts the implementation of the loading procedure from the progressive computation, allowing users to choose the best method for that purpose.

Finally, note that even though a tree is balanced almost perfectly, the loss (i.e., $c(T) - \log N$) is a small positive number, eventually leading to the construction of a new tree. To prevent this, we can optionally accumulate the loss only when the loss exceeds a certain value.

4.1.4 Filtered AKNN Search

Visual analytics should allow users to explore data by applying filters dynamically. To avoid rebuilding whole k-d trees when the user filters points, our progressive search algorithm can restrict its search to a selection of points. This selection is implemented through a very fast bit vector library [30]: The list of filtered points is converted into a compressed bitmap and passed to the search function that gathers the k neighbors, making the search function ignore the points in the bitmap. This filtered search is slightly slower than a non-filtered search, depending on the number of points filtered, but always faster than rebuilding whole k-d trees before performing the query.

The original FLANN algorithm has a provision for removing points from the k-d trees, marking them as deleted. Deleted points, just like our filtered points, are ignored by the search method. They are also filtered out from the reconstruction of balanced trees. Our implementation offers a more flexible mechanism at a low cost.

4.2 *k*-Nearest Neighbor Lookup Table

One frequent application of nearest neighbor search is finding the neighbors for every single point in the data, which is known as *all nearest neighbor search* [126]. Similarly, we can think of a search problem of finding the approximate *k*-nearest neighbors for every single data point in data, which we will call *all approximate k-nearest neighbor* (*AAKNN*) *problem*. The AAKNN problem becomes common in modern analytic methods to replace a complete distance matrix with a useful approximation that remains manageable in space and time. For example, the Approximate t-Distributed Stochastic Neighbor Embedding algorithm (A-tSNE) [106] uses AAKNN search to efficiently compute the distances between the k-nearest points. Therefore, providing an efficient data structure and algorithm for the AAKNN problem will allow progressive systems to support such general and sophisticated methods.

A KNN lookup table is a 2D table with N rows and k columns where N is the number of points in data P and k is the number of neighbors that we want to compute. Formally, a KNN lookup table T is defined as follows:

$$T[i] \mapsto KNN_k(P[i]), \text{ where } i \in [1, N]$$

As a sequential approach, we can precompute and store in a $N \times k$ table the k-nearest neighbors for each point in P to enable constant-time lookup. Specifically, using any AKNN method, we fill each row of the $N \times k$ table so that the i-th row in the table contains the approximate *k*-nearest neighbors of the i-th point in the data; it can contain either its index, its distance to the i-th point, or both. This simple method suffers from similar limitations to

those of sequential k-d trees such as Algorithm 1: It requires all points to be accessible when building the table, and the build time is not bounded.

To build the table progressively, we will use a forest of progressive k-d trees internally, built and maintained progressively as we described in the previous sections. We need an additional progressive algorithm to construct and maintain the table. Our algorithm is iterative, with each iteration having the following three phases:

1. (Indexing) Load new points from a data source and insert them into the internal progressive k-d trees.
2. (Appending) Compute the neighbors of the new points and append the result to T.
3. (Update) Update the old neighbors in T with the new points.

In the indexing phase, a set of new points is loaded from a data source P and inserted into the progressive k-d trees. As we discussed in the previous section, we do not load a fixed number of points for each iteration, but the number is determined in Algorithm 4 based on three factors: the number of allowed operations for the current iteration (ops), the fraction of time used for insertion tasks vs. rebuilding tasks (τ), and whether a new k-d tree is being built or not. Let $Batch_1$ be the list of new points loaded in the first indexing phase. In the following appending phase, we compute the k-nearest neighbors of the points in $Batch_1$ and append the result to the table T as new rows. To this end, for each point p in $Batch_1$, a single KNN query is performed on the k-d trees updated in the indexing phase, which takes $O(logN)$ time. Then, the neighbors of the i-th point are stored in $T[i]$, allowing constant-time lookup for future queries on the point. At this moment, since all neighbors in T are up to date, we skip the update phase and continue to the second iteration.

During the indexing phase in the second iteration, we load a set of new points, $Batch_2$, and insert them into the trees. As in the first iteration, we also append $|Batch_2|$ new rows with neighbors. So far, we have filled in $|Batch_1| + |Batch_2|$ rows into T with their approximate k-nearest neighbors. The problem is that the neighbors for the points in the first batch $Batch_1$, which were computed during the first iteration, can be outdated because there can be closer neighbors in $Batch_2$. Therefore, we need to find what we call *dirty points* in T and update their neighbors.

The update phase *repairs* the table by recomputing the neighbors of dirty points. One straightforward approach would be, each time a new point p is added to T, we iterate over all points in T and check whether a point needs updating by comparing the distance to its k-th neighbor (i.e., the farthest neighbor) and the distance to p. Then, if the point is *dirty*, we drop its k-th neighbor and insert p into its neighbor set as a new neighbor. In this method, we need an $O(N)$ loop for dirtiness check each time we insert a point, which results in $O(N^2)$ complexity in total.

Our implementation improves the time complexity of the update procedure by approximating the search process. Our assumption is that, for a new point p, its neighbors from older generations in $KNN_k(p)$ are likely to have p as a new neighbor. This means we can first inspect the dirtiness of the neighbors of the new point, $KNN_k(p)$, to narrow down the search space. As an extreme case, if the neighborhoods between points are completely mutual (i.e., $q \in KNN_k(p) \Rightarrow p \in KNN_k(q)$), we can greatly reduce the search space by only considering the points in $KNN_k(p)$.

Based on this optimistic assumption, we first search for dirty points in $KNN_k(p)$. We define those dirty points as a set $S_1(p)$. This can be written as

follows:

$$S_1(p) = \{q \mid q \in KNN_k(p) \wedge p \in KNN_k(q)\}$$

Note that in real cases, the neighborhood between two points is asymmetric, i.e., $q \in KNN_k(p) \wedge p \notin KNN_k(q)$. Therefore, there can be dirty points that have p as a new neighbor but are not in the *k*-nearest neighbors of p, and we miss them in $S_1(p)$. To find these missing dirty points, we expand our search space one step further by applying our assumption again on $S_1(p)$. We define $S_2(p)$ as follows:

$$S_2(p) = \{q \mid q \in S_1(p) \wedge p \in KNN_k(q)\} - S_1(p)$$

From a global point of view, calculating $S_2(p)$ *propagates* the dirtiness to the neighbors of the points in $S_1(p)$. We define a set $DP_2(p)$ that accumulates all dirty points that we have found so far:

$$DP_2(p) = S_1(p) \cup S_2(p)$$

Generally, we define $S_i(p)$ and $DP_i(p)$ as follows:

$$S_{i+1}(p) = \{q \mid q \in S_i(p) \wedge p \in KNN_k(q)\} - DP_i(p)$$
$$DP_{i+1}(p) = DP_i(p) \cup S_{i+1}(p)$$

We continue to propagate dirtiness until no new dirty points are found, i.e., $S_{i+1}(p)$ becomes empty. Our algorithm is *approximate*; it does not guarantee to find all dirty points. However, if the input points follow our assumption, our algorithm will find most of the dirty points by narrowing the search space to the vicinity of the query point p, not checking all the input points. The accuracy of a KNN lookup table is therefore determined by the approxi-

mation in both k-d trees and dirtiness propagation. In Section 4.3.2, we report on the overall accuracy of KNN lookup tables on real data.

At worst, our algorithm checks all the points, just as the naïve approach does. In other words, the number of points searched in the third phase is not bounded, which can result in a long delay before completing the update phase. Note that our algorithm can be seen as a graph traversal if we regard points as vertices and the neighborhood between points as edges. Therefore, taking a similar approach to breadth-first search (BFS), we can limit the number of checked points in the update phase by introducing an update queue. Specifically, for a point in $S_i(p)$, we first insert it into the update queue. For each iteration, we take a fixed number of points from the queue and propagate a dirtiness check. We also use a bitmap as a set to prevent a point from being inserted into the queue more than once. This allows the update phase to finish after checking a fixed number of points, preserving the progressiveness of the algorithm.

Similar to the progressive k-d tree algorithm that used a parameter τ to choose the fraction of insertion and reconstruction tasks, the KNN lookup tables have an additional parameter, named a queue update fraction, λ, to balance the number of operations assigned to internal k-d trees (i.e., the indexing phase) and queue updates (i.e., the update phase). Given the two parameters (τ and λ) and the number of allowed operations (ops), the algorithm assigns the operations as follows: First, $(1-\lambda)ops$ operations are assigned to the indexing phase for updating internal progressive k-d trees. The algorithm calls the function UpdateProgressiveTrees in Algorithm 4 with $(1-\lambda)ops$ and τ as arguments. The function returns the number of inserted points, which is the number of points that are inserted in the appending phase. Finally, in the update phase, at most, λops points are taken from the

Algorithm 5 An algorithm for building a progressive KNN lookup table

```
1: procedure InitializeProgressiveTable(dataSource, k)
Input: dataSource is an abstract input stream
Input: k is the number of neighbors we want to compute
Output: initialize an empty KNN lookup table
2:     InitializeProgressiveTrees(dataSource, 0)
3:     table ← an empty table with 0 rows and k columns
4: end procedure
5:
6: procedure BuildProgressiveTable(dataSource, ops, k, τ, λ)
Input: dataSource is an abstract input stream
Input: ops is the number of operations allowed for an iteration
Input: k is the number of neighbors we want to compute
Input: τ is a fraction for insertion tasks for internal k-d trees
Input: λ is a queue update fraction
Output: a KNN lookup table
7:     queue ← an empty queue
8:     queued ← an empty bitmap dictionary
9:
10:    function UpdatePoint(trees, i)          ▷ Update the neighbors of a single point i
11:        queued[i] ← false
12:        neighbors ← trees.getKNeighbors(dataSource[i], k)
13:        table[i] ← neighbors
14:        for an integer j such that 0 ≤ j < k do
15:            if not queued[neighbors[j]] then
16:                queue.push(neighbors[j])
17:                queued[neighbors[j]] ← true
18:            end if
19:        end for
20:    end function
21:
22:    treeOps ← (1 − λ) × ops
23:    tableOps ← λ × ops
24:    trees, newPoints ← UpdateProgressiveTrees(dataSource, treeOps,τ)
25:
26:    for point in newPoints do
27:        UpdatePoint(trees, point.index)
28:    end for
29:
30:    count ← 0
31:    while queue is not empty and count < tableOps do
32:        index ← queue.pop()
33:        if not queued[index] then
34:            queued[index] ← true
35:            UpdatePoint(trees, index)                    ▷ Propagate changes
36:        end if
37:        count ← count + 1
38:    end while
39: end procedure
```

update queue and updated if dirty. Algorithm 5 shows the complete algorithm for progressive KNN lookup tables.

4.3 Benchmark

Online algorithms are usually evaluated using competitive analysis [23]: their performance are compared against an equivalent offline algorithm and the ratio is reported. This ratio only makes sense when the algorithm needs to complete to its end, but one premise of progressive data analysis is that some decision can be made before the algorithm ends. Competitive analysis does not account for these important early termination cases. On the other extreme, Eichmann et al. [41] have proposed benchmarks to compare the effectiveness of databases to fulfill the requirements of data analysis sessions with or without progressive support. This benchmark requires a full system used on a realistic task, which is not our purpose. In our work, we remain at the algorithm level and compare our implementation with the only online implementation available, provided by FLANN.

We conducted two benchmarks to evaluate our progressive k-d trees and KNN lookup table. The first benchmark compares the performance of online k-d trees and our progressive k-d trees and the second measures the build and query time of a KNN lookup table.

4.3.1 Online and Progressive *k-d* Trees

In this benchmark, we compare the performance of online and progressive k-d trees. We used one real dataset (the GloVe [104] dataset) and one synthetic but more structured dataset (a *Blob* dataset). Both datasets had 1 million of 100-dimensional points. The GloVe dataset had embedding vectors of English words, while the synthetic Blob dataset had points sampled from

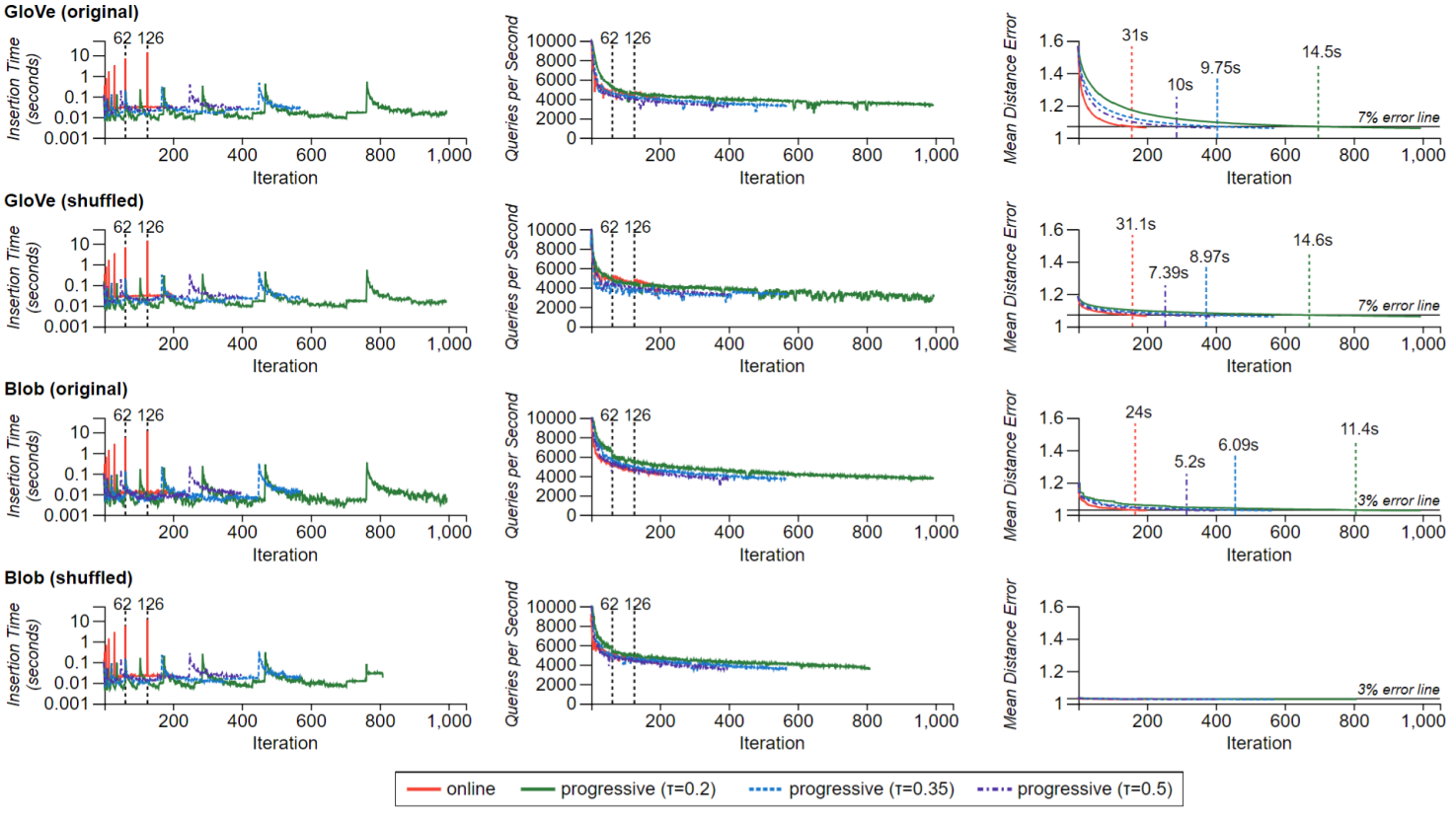


Figure 4.1: Benchmark results of online and progressive k-d trees according to datasets (*GloVe* [104] and *Blob*) and ordering conditions (*original* and *shuffled*). FLANN's online k-d tree (the red line) rebuilt the k-d trees each time the data size doubles, producing insertion times longer than 10 seconds (e.g., leftmost charts at the 62nd and 126th iterations).

100 isotropic Gaussian blobs, 10,000 points for each. For reproducibility, the Blob dataset was generated through scikit-learn's blob generator [103]. The points in the Blob dataset appear in an increasing order of clusters; for example, the first 10,000 points were from the first Gaussian blob, the next 10,000 points were from the second Gaussian blob, and so on. As query points, we used another 1,000 embedding vectors for the GloVe dataset and random 100-dimensional vectors for the Blob dataset. To see the effect of the inserting order of points, we used two conditions: the order was 1) kept as in the original dataset (*original*) and 2) shuffled (*shuffled*), potentially producing balanced k-d trees earlier.

For each iteration, we gave 5,000 operations to both the online and the progressive k-d tree (i.e., $ops = 5,000$). The online version used up all operations to add new points (i.e., 5,000 points were inserted to k-d trees during one iteration). For the progressive k-d tree, we used three different values for τ: 0.2, 0.35, and 0.5. A progressive tree with a higher value of τ prioritizes insertion tasks, assigning fewer operations to maintaining the balance of the tree. Note that the value of τ only affects the algorithms when reconstruction occurs. For example, if the accumulated loss due to imbalance has not reached a certain threshold, all operations will be assigned to insertion tasks for the progressive k-d trees. We set the value of α (i.e., the reconstruction weight) to 100. The benchmark was conducted on a single machine, which was equipped with Intel Core i7-7700K CPU (4.2GHz) and 16GB of main memory. We used eight threads to process KNN queries in parallel. All algorithms used four randomized k-d trees and searched 2,048 nodes.

We queried 20 neighbors (i.e., $k = 20$) for each point in the test data and measured insertion time, queries per second (QPS), and mean distance error for the neighbors found. Insertion time is the time taken to insert points into

trees in a batch (e.g., 5,000 points in the case of online trees). Queries per second (QPS) is the mean number of queries processed in a second, indicating the balance of search trees.

The quality of sequential AKNN algorithms have usually been measured using *search precision* [96], defined as the fraction of exact neighbors returned from an approximate algorithm. However, we found search precision underestimates the quality of online and progressive algorithms; for example, if only 10% of the data is indexed and the exact neighbors are uniformly distributed in data, the search precision of a progressive algorithm cannot surpass 10%, since 90% of the exact neighbors are not in the index. Therefore, we measured a relative error, *mean distance error* (MDE); for each query point, we first compute the ratio between the distances to its exact k-th nearest neighbor and to its approximate k-th nearest neighbor and then calculate the mean of the ratios for all query points. An MDE of one means that the exact neighbors were found, and an MDE of two means on average the algorithm found neighbors that are two times farther than the exact ones.

Figure 4.1 shows the changes in measures per iterations according to datasets and ordering conditions. Since the online trees used all 5,000 operations to insert new points, the corresponding red line ends at the 200th iteration (1,000,000 / 5,000 = 200). The online algorithm builds new trees each time the number of inserted points doubles. This behavior yields spikes in the insertion time (the red lines on the leftmost charts in Figure 4.1). The spikes clearly revealed the limitation of the online trees: At the 126th iteration, the online trees produced a peak latency in insertion time that was longer than 10 seconds. In contrast, regardless of datasets and ordering conditions, the progressive trees kept the insertion time under one second: in progressive k-d trees, abrupt changes in insertion time were removed. Since

we controlled the number of operations in one iteration to achieve progressiveness, not execution time, insertion time varied depending on the numbers of insertion and reconstruction tasks.

Regarding QPS, the online trees had a performance gain after tree reconstruction, since the trees became well balanced (i.e., at the 62nd and 126th iterations in the middle column of Figure 4.1). The progressive trees showed lower performance, but the gap could be narrowed by adjusting the value of α (i.e., the reconstruction weight). Progressive trees with a smaller value of τ yield better QPS, but the differences were small.

As more points were inserted to k-d trees, the mean distance error (MDE) of answers decreased, finally converging to an MDE of 7% for the GloVe dataset and 3% for the Blob dataset. We measured the time from the beginning to the moment when the MDE converges and marked it in the rightmost charts in Figure 4.1. The online tree took the smallest number of iterations to reach the final MDE. The reason may be that it used all its operations to insert new points, so exact neighbors were more likely to be in the trees and searched. Indeed, progressive k-d trees with a larger value of τ assigned more operations to index new points, producing faster convergence. However, due to the longer insertion time, the online trees took the longest time to reach the final MDE, which suggests the effectiveness of our progressive k-d trees.

In the Blob dataset under the *shuffled* condition, the progressive trees generally took the fewest iterations to index the whole dataset and the shortest time to reach to the final MDE. The reason may be that the dataset had randomly sampled points in a random order, so the trees spent the fewest operations in rebalancing and the approximation in k-d trees was effective.

The benchmark showed that the value of τ can be tuned to achieve the desired behavior of progressive k-d trees. Using a smaller value of τ leads to shorter insertion time and larger QPS, improving the scenarios where high throughput is important. However, the downsides are that 1) more iterations are required to index input points and 2) it takes longer to reach a certain level of accuracy. Nonetheless, regardless of the value of τ, we found that our progressive trees outperforms online trees for progressive systems in that the insertion time can be maintained below a specific bound with comparable QPS and better accuracy.

4.3.2 *k*-Nearest Neighbor Lookup Tables

The second benchmark evaluates the performance of KNN lookup tables. As in the first benchmark, we used the GloVe and Blob datasets and the two ordering conditions (i.e., original and shuffled). For progressive k-d trees, we used the same settings as those used in the first benchmark except $ops = 4,000$ for shorter insertion time. Since KNN lookup tables were designed for the all approximate k-nearest neighbor problem, we sampled 1,000 points from the training data as test data instead of using an extra set of 1,000 points. We set the value of τ for internal progressive k-d trees to 0.5. To see the effect of dirtiness tests, we used three different values for λ (i.e., the queue update fraction): 0.3, 0.4, and 0.5. Other constants were identical to those of the first benchmark, such as $k = 20$ and $\alpha = 100$. For the internal indexer, we used four progressive trees. KNN queries for updating lookup tables traversed at most 2,048 nodes in each tree. As in the first benchmark, we measured insertion time, queries per second (QPS), and mean distance error (MDE).

Figure 4.2 shows the result of the second benchmark. Overall, the insertion time of progressive KNN lookup tables increased compared to that of a

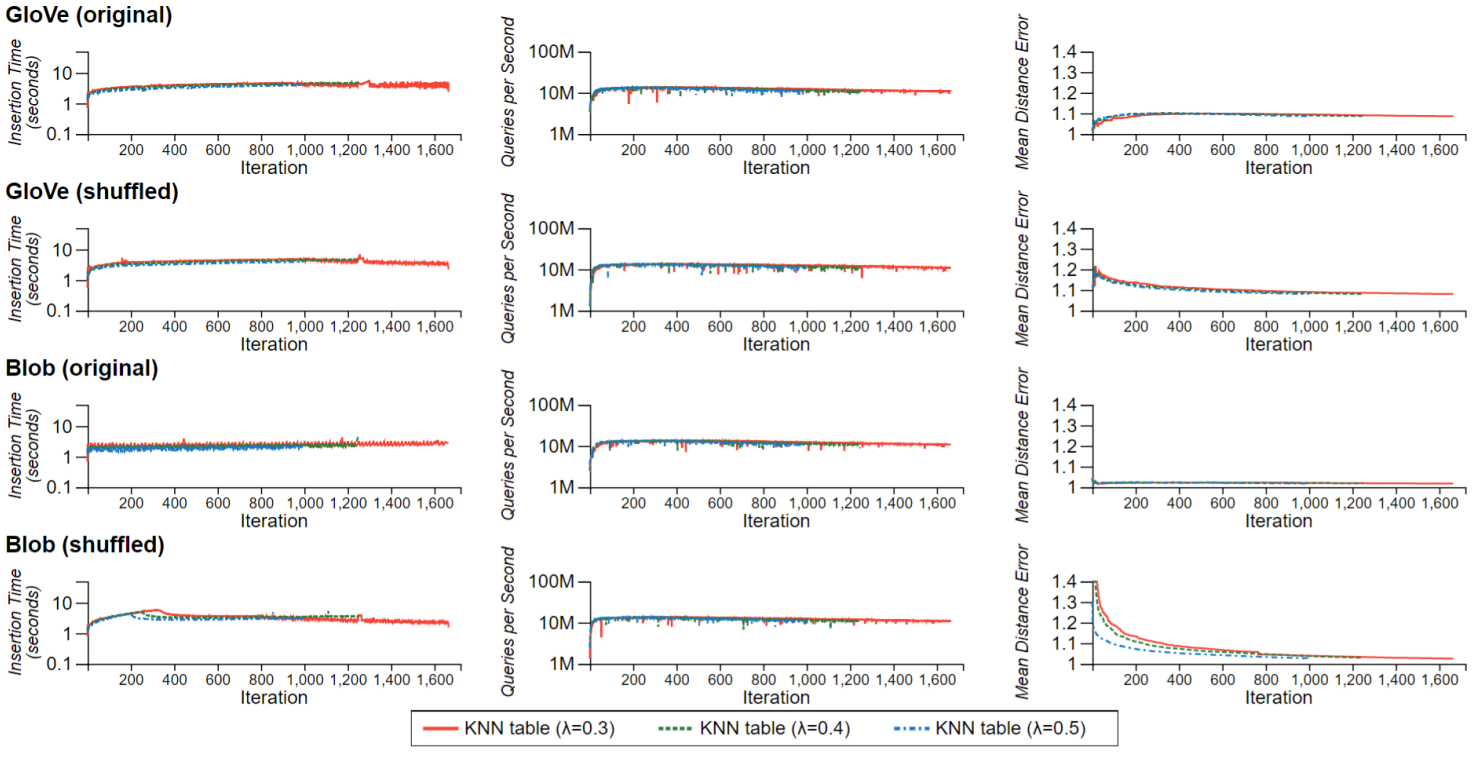


Figure 4.2: Benchmark results of *k*-nearest neighbor lookup tables according to datasets (*GloVe* [104] and *Blob*) and ordering conditions (*original* and *shuffled*).

progressive tree. We profiled the time to complete each phase in KNN lookup tables and found that the longer insertion times resulted from computing the k-nearest neighbors of every unseen point (i.e., points in a batch). However, since the KNN lookup tables can answer a KNN query in constant time (i.e., one table lookup operation), QPS of the KNN lookup tables was a few orders of magnitude higher than that of progressive trees. In a sense, a KNN lookup table increases build time but significantly reduces the query time with a complexity of $O(1)$ instead of $O(\log_2 N)$.

The effect of the queue update fraction (λ) was clearly seen in the benchmark. Using a smaller value of λ resulted in shorter insertion time at the cost of high mean distance error, since it gave a low priority to maintaining the table up to date (i.e., dirtiness tests). Note that QPS of the tables was not affected by the value of λ, since a query was merely a lookup operation that can be done in constant time. This provides flexibility in changing the behavior of KNN tables. For example, one can adjust λ to strike a balance between insertion time and accuracy depending on applications.

4.4 Applications

The long computation time of sequential KNN methods has limited the potential use of data mining algorithms in interactive visualization systems. In this section, we improve three popular sequential algorithms to become progressive: KNN regression, KNN density estimation, and the t-SNE algorithm [84], adding them to the toolbox of interactive analysis tools.

4.4.1 Progressive Regression and Density Estimation

One common use of k-nearest neighbors is interpolating an unknown target value using training data, which is called *KNN regression*. Suppose that

training data X consists of N instances, $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N$, and each instance has a target value, y_i. The target value of an unseen instance $\mathbf{x}_{new}$ can be predicted based on the target values of its neighbors:

$$y_{new} = \sum_{p \in KNN_K(x_{new})} w_p y_p \tag{4.2}$$

where w_p is the weight for the neighbor p. The fractions can be either a constant (i.e., $1/k$) or inversely proportional to the distance to x_{new}. The training data X can be indexed progressively using our k-d trees, which gives us an estimate of $KNN_k(\mathbf{x}_{new})$. Finally, we can compute and improve y_{new} using Equation 4.2 in a progressive manner.

Another possible application of PANENE is progressive KNN density estimation. KNN density estimation is similar to KNN regression except that it predicts the density of training data on a specific point instead of a target value. The goal of KNN density estimation is to provide density information on 2D input points to help users understand the distribution of the input points. Again, suppose that training data X consists of N instances, $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N$. Using a Gaussian kernel with a bandwidth h, the density of X on a specific point $\mathbf{p}$ is given by $\rho(p)$:

$$\rho(p) \propto \sum_{x \in X} e^{-\frac{(p-x)^2}{2h^2}} \qquad \hat{\rho}_K(p) \propto \sum_{x \in KNN_K(p)} e^{-\frac{(p-x)^2}{2h^2}} \tag{4.3}$$

which has a complexity of $O(N)$ since it iterates over all the points. The main idea of KNN density estimation is that, as the target point $\mathbf{p}$ becomes farther away from an instance $\mathbf{x}$, the Gaussian kernel will give it a smaller value converging to zero, and its impact on $\rho_K(\mathbf{p})$ becomes negligible. This gives us an opportunity for approximating the density using only the neighbors of q. The approximated density $\hat{\rho}_K(\mathbf{p})$ can be computed by Equation 4.3 (right).

Given 2D input points, we first choose sample points on a grid of r rows and c columns. Using a progressive k-d tree, we estimate and improve the density of input points on each sample point. Then, density isolines are computed using the marching square algorithm [81] and visualized through a contour plot. Note that the number of neighbors K should be chosen with care, since the estimated density can be saturated with small K as the data size grows. In the Appendix, we included an example of progressive KNN density estimation using three different values of K.

4.4.2 Responsive t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) [84] is a nonlinear dimensionality reduction algorithm that is widely used in data analysis. Given a set of high-dimensional points, $\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N$, in a feature space, t-SNE maps the points to low-dimensional points (i.e., embedding), $\mathbf{y}_1, \mathbf{y}_2, ...\mathbf{y}_N$, that describe the similarities between the original points in the embedding space. Usually, the high-dimensional points are projected on a 2D space and visualized through conventional scatterplots or density plots, which can help the user to understand the distribution and structure of the original points. The original t-SNE algorithm runs with a complexity of $O(N^2)$.

To improve the computation time of the original t-SNE algorithm, the Barnes-Hut Stochastic Neighbor Embedding (BH-SNE) [127] reduces the complexity to $O(N \log N)$ by applying the Barnes-Hut approximation [16] to compute the contribution of points. BH-SNE is faster than the original t-SNE algorithm, but not enough to guarantee that the computation latency will remain under a few seconds [99].

From a high-level point of view, the BH-SNE algorithm consists of two parts: 1) computing the k-nearest neighbors of each point to measure the dis-

tance between the points and 2) a gradient descent iteration that minimizes a loss function between the distributions of points in the original space and the embedding space. The long initial delay of BH-SNE mainly stems from the neighbor computation. Any KNN methods covered in Section 2.3 can be adopted to speed up the neighbor computation. However, those methods are still *blocking*, which eventually causes longer precomputation times as the data size grows.

To improve the responsiveness of BH-SNE, we created a variant that computes both the nearest neighbors and the embedding progressively; we call it *responsive t-SNE*. As a proof-of-concept prototype, we have implemented responsive t-SNE by integrating PANENE with the BH-SNE algorithm. Responsive t-SNE spreads the load of neighborhood computation to later iterations, alleviating the initial overhead coming from the blocking KNN methods. Specifically, we used PANENE's KNN lookup table to progressively index and compute the k-nearest neighbors of each point. In contrast to the previous BH-SNE algorithm where neighbor computation must precede the gradient descent loop (i.e., loss minimization), we move the neighbor computation inside the training loop: The training loop of our algorithm alternates between 1) updating the KNN lookup table (i.e., indexing new points) and 2) updating the projection to minimize loss. As training proceeds, the projection is improved in terms of both quantity (i.e., the projection includes more points) and quality (i.e., the projection minimizes the error between the original points and the embedded points).

To compare our responsive t-SNE algorithm with the BH-SNE algorithm, we ran both algorithms on the MNIST dataset [71] as an exploratory benchmark. The MNIST dataset consists of 60,000 vectors, each vector representing 784 (28×28) pixels of a handwritten digit scanned. We used an open-source

implementation of BH-SNE [22] as a baseline. In contrast to the BH-SNE algorithm, where the projection (i.e., $\mathbf{y}_i$) is randomly initialized for all points, we used random initialization only for the points in the first batch. For each point in later batches, we set its initial position to the centroid of its k neighbors, which is a better starting point. We set the perplexity of t-SNE to 10, which led the algorithm to compute 30 neighbors for each point, and the threshold of the Barnes-Hut algorithm (i.e., θ) to 0.5.

Figure 4.3 shows the time taken to initialize the two algorithms and the resulting embeddings over the iterations. Each point in the scatterplots represents a single vector, with its color encoding the corresponding digit (i.e., 0 to 9). The BH-SNE algorithm took 44.7 minutes to initially compute the nearest neighbors, while our algorithm produced an initial estimate in a few seconds. To assess the quality of embeddings, we measured the Kullback-Leibler divergence (i.e., loss) between the distributions of points in the original and embedding spaces as in the original paper [84]. The embeddings were improved over iterations, giving the final loss of 3.96 for BH-SNE and 4.59 for responsive t-SNE.

Since responsive t-SNE computes an embedding incrementally, the quality of the resulting embedding can be affected by the order of input points. For example, we can create an extremely skewed dataset by sorting the MNIST data by the digit each instance represents (e.g., from 0 to 9). To alleviate the bias resulting from the skewed data, we introduce a technique called *periodic exaggeration*, which is inspired by a technique from a previous study [84]. Periodic exaggeration regularly multiplies a constant factor to the conditional probabilities between points, which allows the clusters in the data to form separated clusters in the embedding. For instance, the responsive t-SNE algorithm increases the conditional probabilities by a factor at the beginning

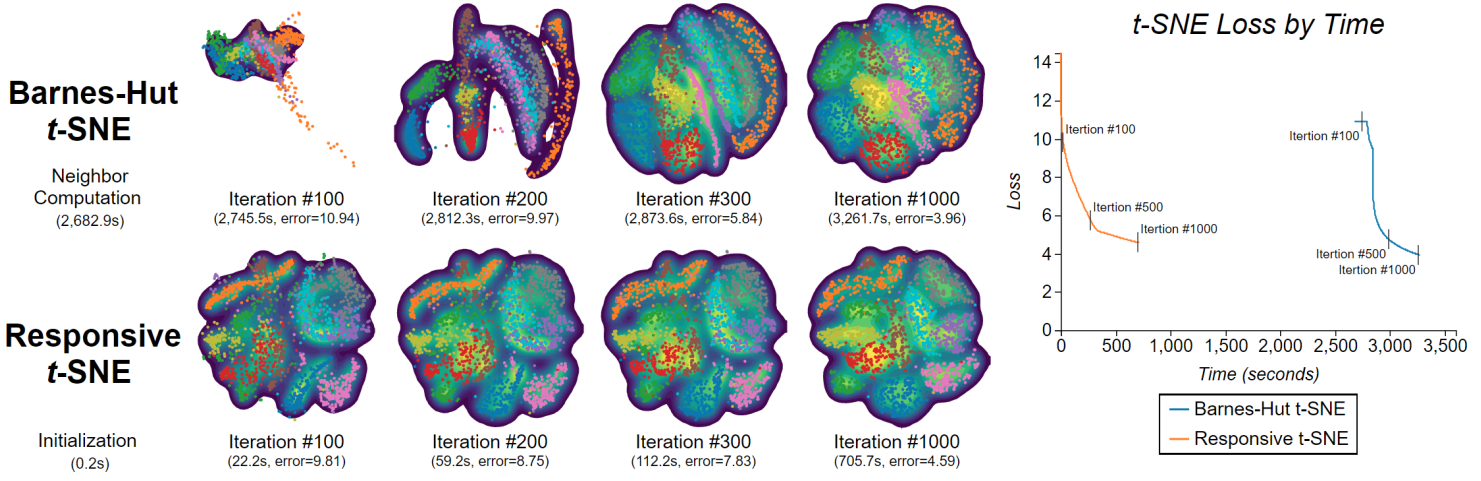


Figure 4.3: Embedding of the MNIST dataset using Barnes-Hut t-SNE and Responsive t-SNE. The Barnes-Hut t-SNE algorithm took about 45 minutes to precompute the nearest neighbors of data points. Our responsive t-SNE produced the initial result in a few seconds by computing the nearest neighbors progressively and running the optimization loop of the t-SNE algorithm in an alternate manner. Each circle in the scatterplots represents a handwritten digit (28×28 pixels) with its category (0 to 9) color-encoded.

of the 100 first iterations, and restores them to 1 after 30 iterations. Close-by points become tightly grouped during the exaggeration phase, allowing points to move more easily to nearby groups, thus avoiding the algorithm to be trapped in local minima. The tension between points lessens after exaggeration, making the points distributed according to their original distances.

In the Appendix, we attached a figure that shows the effect of periodic exaggeration and how the embedding changes over time when the order of the input points is skewed. Exaggeration was applied for 30 iterations at the beginning of every 100 iterations. During exaggeration (i.e., the leftmost three columns of the figure in the Appendix), one can observe that points with the same color move to be densely packed. In our prototype, we fixed the duration and period of exaggeration, but in an interactive analysis, we can involve users by allowing them to set those parameters and choose the moment to start the exaggeration.

One common task in exploratory visual analytics is to understand the overall distribution of multidimensional data. To support this task, we can construct a visualization pipeline by combining responsive t-SNE and progressive density estimation. The multidimensional data is first projected on a 2D plane progressively through responsive t-SNE. Then, we measure the density of the embedding for each sample point on a grid using the progressive density estimation algorithm. Finally, we can draw a contour plot to show the density, giving an overview of the multidimensional data within a controlled latency.

4.5 Implementation

We implemented the core of PANENE in C++ as well as a Python binding of PANENE called PyNENE, for a wider range of applications, such as its integration in the ProgressiVis toolkit [45]. PANENE relies on the OpenMP library [38] to process multiple queries in parallel, and on the Roaring Bitmaps library [30] to support efficient filtering. PANENE, the parameters for benchmarks, and applications presented in this chapter are available at `github.com/e-/PANENE` under the BSD 2-clause "Simplified" License.

4.6 Discussion

We controlled the running time of progressive algorithms by providing the number of allowed operations (ops) as a parameter of the algorithms. When we chose ops, our primary goal was maintaining the latency of our algorithms within the attention-preserving time limit (i.e., about 10 seconds) [99], which gave us a few thousands of operations for one iteration in our settings. However, in practice, this number should be determined and dynamically adjusted depending on various factors, such as the type of workload (e.g., disk access or CPU computation), the computing power of machines, and its impact on the performance of algorithms. Therefore, we need an extra step that maps the number of operations to the actual execution time so as to maintain the progressiveness of the system, which is related to time. One example is a time predictor, presented by Fekete and Primet [45], which dynamically infers the number of operations per second of algorithms by monitoring their execution.

In progressive k-d trees, we assumed that an insertion task (i.e., adding a point to trees) and a reconstruction task (i.e., splitting a node in a back-

ground tree reconstruction) were atomic, and that running either of these tasks takes the same time. However, in an extreme case where the data size is large, it can be impossible to run even one operation in the given time quantum. For example, splitting nodes into two groups on lines 25 in Algorithm 3 takes a time proportional to N, which will eventually block the system when N is large enough. One possible remedy is to make the split operation itself progressive. For now, in our benchmark settings, it took approximately 0.15 seconds on average to split a node with 10 millions of 100-dimensional points.

Finally, we assumed that our algorithms run on a single thread. However, the only constraint we have with the model we rely on [45] is that modules are run sequentially. Inside a module, algorithms can use as many threads as needed. Our KNN search uses multiple threads up to the number of k-d trees for searching, but allocating the threads remains a decision of the programmer to balance other needs. Allowing the tree reconstruction to be done in a separate thread could avoid slowing down the indexing operation, but at the cost of locking mechanisms; more work needs to be done to measure the best thread allocation strategies.

4.7 Summary

Although k-nearest neighbor computation is common in data mining algorithms, the long computation time has hindered its application for visual analytics. This chapter presents PANENE, a combination of progressive k-d trees and KNN lookup tables, for progressive approximate k-nearest neighbor search. We made three major changes to the previous online k-d trees: maintaining a quality measure to determine when to reconstruct trees, trig-

gering reconstruction when needed, and introducing a build queue to spread the reconstruction load. Through benchmarks, we found our progressive algorithms could alleviate the abrupt changes in latency that degrade the interactivity of visualization systems. Finally, we presented three applications of PANENE: progressive regression, progressive density estimation, and responsive t-SNE.

Chapter 5

ProReveal: Progressive Visual Analytics with Safeguards

This chapter [1] describes our novel PVA concept, Progressive Visual Analytics with Safeguards, and a proof-of-concept system, ProReveal, answering the third research question (***Knowledge Trustworthiness**: how do we improve the trustworthiness of intermediate knowledge gained from progressive visual analytics?*).

We propose *Progressive Visual Analytics with Safeguards*, a novel visual exploration concept that helps people manage the uncertainty arising from progressive data exploration. Progressive visual analytics (PVA) allows people to access the partial results of visualization queries in the middle of computation, helping them make data-driven decisions faster even with large-scale data. However, such *intermediate* knowledge can be incorrect due to various machine and human factors. For example, many PVA systems build and use samples of raw data to estimate results, which leaves a discrepancy between the precise results and the results based on the samples. Another

[1] The preliminary version of Chapter 5 is under minor revision for publication in IEEE Transactions on Visualization and Computer Graphics (TVCG).

reason can be a human factor such as misinterpreting the uncertainty of intermediate knowledge and making a hasty decision. To address this issue, we introduce *PVA-Guards*, which can be used to validate intermediate knowledge during or after exploration and ensure its correctness.

We are especially interested in exploration scenarios where it is infeasible to obtain precise results during a single session due to a long computation time. In this case, people can rely only on partial and uncertain results to steer their exploration and draw conclusions. As an illustrative example, suppose an analyst, Zoey, is progressively exploring a large sales dataset of an online bookstore. She is interested in gathering useful information about the country that published the most books. Therefore, she first creates a bar chart of the number of books published in each country. The system shows the early estimates of publication counts, and she finds the USA has the highest publication count. However, as the bar chart gets updated (with more data processed), she notices that the difference in the publication counts for the USA and China is small.

Zoey needs to make a decision based on her uncertain intermediate finding—choosing either the USA or China as having the highest publication count—to look into the country next. One common strategy to handle this uncertainty is to wait longer until the visualization looks more certain (e.g., wait for narrower confidence intervals or larger height difference between the two bars). However, few systems guarantee how long she needs to wait to achieve a certain level of trustworthiness, which can decrease the benefits of PVA, that is, allowing early decision making. Using another strategy, Zoey can instead take the risk of proceeding with the uncertainty; she chooses one of the two countries and proceeds to her next visualization. In this case, she needs

to consider the worst case of such an optimistic strategy: her choice turning out to be wrong.

Our concept, Progressive Visual Analytics with Safeguards, provides a means of managing such uncertainty resulting from progressive visual analytics by allowing people to leave PVA-Guards on uncertain intermediate knowledge. A PVA-Guard is a hypothetical representation of intermediate knowledge that people garnered during progressive data exploration. In the example of Zoey's exploration, her intermediate knowledge is that the publication count for the USA is greater than that of China, which can be represented as a `Comparative` PVA-Guard, such as $PubCount(USA) > PubCount(China)$, on a bar chart. Once a PVA-Guard is placed, the system statistically estimates the validity of her intermediate knowledge and gives continuous feedback on its validity. In case the PVA-Guard becomes invalid, the system notifies her so that she can manage the incorrect intermediate knowledge. Therefore, with the PVA-Guard on, Zoey can continue her exploration at a desired pace and confidence.

The main contributions of this chapter are as follows:

- We define PVA-Guards, present concrete examples, and discuss design considerations to realize our new progressive visual analytics concept (Section 5.1);
- We design and implement a proof-of-concept PVA system, ProReveal (Figure 5.1), integrating seven types of PVA-Guards: `Value`, `Rank`, `Range`, `Comparative`, `Power Law`, `Normal`, and `Linear` (Section 5.2);
- We report a qualitative user study with 14 participants conducted to investigate how people use and interact with PVA-Guards for their progressive exploration (Section 5.3).

5.1 Progressive Visual Analytics with Safeguards

While PVA allows people to access intermediate results in the middle of computation, the trustworthiness of the results remains equivocal. Such uncertainty can make analysts wait for more certain results or lead them to a hasty decision. We approach this problem by allowing analysts to explicitly leave a PVA-Guard on uncertain intermediate knowledge that needs to be verified. These PVA-Guards not only provide means for testing the correctness of the conclusion drawn from exploration even when analysts are unavailable, but also can be used to trace the provenance of intermediate knowledge when some of them turned out to be incorrect. We envision that PVA-Guards can provide the following benefits:

- *Speed*: People can continue to explore data even when ongoing results are not certain enough, leaving the task of validating intermediate knowledge to the system.
- *Correctness*: The set of PVA-Guards can be used to validate the correctness of intermediate knowledge gathered from exploration later or in the middle of analysis.
- *Trace*: When some intermediate knowledge turned out to be wrong, PVA-Guards can serve as traces of exploration that enable people to alter or re-run the analysis.

5.1.1 Definition

A PVA-Guard (hereafter, a Guard) is a hypothetical representation of intermediate knowledge that an analyst gathers during progressive data explo-

ration. It can be formally defined as follows:

$$\langle \textit{PVA-Guard} \rangle := \langle variable \rangle \ \langle operator \rangle \ \langle operand \rangle \tag{5.1}$$

where

$$\langle operand \rangle := \ empty \ | \ \langle variable \rangle \ | \ \langle constant \rangle.$$

For example, a Guard where $\langle variable \rangle = PubCount(USA)$, $\langle operator \rangle = >$, and $\langle operand \rangle = PubCount(China)$ indicates the USA's publication count is greater than China's.

In the definition, the first $\langle variable \rangle$ part refers to the subject of intermediate knowledge that the Guard tests. It can be a single value, such as the value of a cell of a heatmap, the rank of a bar in a bar chart, or even the distribution of values in a histogram. In the middle of progressive exploration, $\langle variable \rangle$ is uncertain and estimated through a statistical procedure if applicable.

The $\langle operator \rangle$ part indicates the type of intermediate knowledge and an operation a Guard performs. For example, common comparison operators (e.g., $\leq$) are useful when we want to compare a variable to a constant or another variable. Other operator can be $\sim$ (i.e., follows) and $\propto$ (i.e., is proportional to) that state a variable (i.e., a distribution of values) follows a certain distribution or is proportional to another variable, respectively.

The last $\langle operand \rangle$ part refers to the object of intermediate knowledge. The type of $\langle operand \rangle$ is determined by the operator. For example, for comparison operators, $\langle operand \rangle$ can be either a constant or another variable. $\langle operand \rangle$ can also be a specific distribution whose parameters are known; for example, when the operator is set to $\sim$, $\langle operand \rangle$ can be a normal distribution such as $\mathcal{N}(20, 10^2)$. Finally, $\langle operand \rangle$ can be unspecified when the

operator does not require any operand such as an existence operator that checks whether a variable exists.

A *validity measure* of a Guard provides an estimate about how certain the intermediate knowledge is (Table 5.1). It can be a boolean value (i.e., *true* or *false*), indicating whether the Guard holds or not, but is not limited to, especially when the intermediate knowledge itself does not have an dichotomous answer (e.g., knowledge that the distribution of values follows a specific distribution). In some cases, a validity measure can be computed during exploration, which would be useful because it can help people judge the trustworthiness of the intermediate knowledge and steer exploration. For example, we can provide statistical significance or a p value on the difference of $PubCount(USA)$ and $PubCount(China)$ through a Student's t-test using sample statistics.

When computing such statistical significance during progressive computation, we can consider the given dataset as a finite population. In this case, processed rows through progressive computation can be regarded as a sample drawn from the finite population without replacement. Because the population is finite, we can obtain a definitive answer on the statistical significance when the entire dataset is processed; for example, a p value will converge to either 0 or 1 in the end. On the other hand, we can view the dataset as a sample drawn from a hidden infinite population (i.e., the world). In this case, even after the whole dataset is processed, the result remains uncertain. We take the former perspective; we estimate the final result using the statistical procedures for a finite population of size N (i.e., the number of rows in data). Note that by setting N to ∞, we can compute p values and confidence intervals from the latter perspective. Details on the procedures can be found in supplementary materials.

Name	Domain	Examples in ProReveal
Probability	$[0, 1]$	p value from a t-test
Quality	$[0, 1]$	Kolmogorov-Smirnov statistic
Error	$[0, \infty)$	Root mean square error (RMSE)
Boolean value	$\{true, false\}$	Estimates on MIN and MAX

Table 5.1: Validity measures of PVA-Guards

For both cases, however, p values obtained in the middle of computation do not guarantee to faithfully indicate the precise result, so interpretation of these values should be done with care and often requires prior knowledge, considering the importance of the decision.

5.1.2 Examples

To elicit candidates for meaningful Guards, we started from identifying what knowledge people can gain from a single visualization. We inspected a low-level task taxonomy that consists of the tasks that people perform on a visualization and possible outcomes from the tasks. Amar et al. [4] identified ten low-level tasks of analytic activity in information visualization: Retrieve Value, Filter, Compute Derived Values, Find Extremum, Sort, Determine Range, Characterize Distribution, Find Anomalies, Cluster, and Correlate. In this section, we demonstrate how the knowledge gained by performing each of these tasks can be represented as a Guard. We categorized these 10 tasks into four sets depending on 1) whether intermediate knowledge from the task can be stated and validated as a Guard and 2) the type of a possible validity measure for the Guard.

Tasks as Guards with Statistical Significance. The first set of tasks allows people to represent intermediate knowledge as a Guard, and there is also a statistical test for the validity of the knowledge which gives a statistical sig-

nificance, such as a p value. For example, the **Retrieve Value** and **Compute Derived Value** tasks are to identify an attribute of a single data item and a derived value (e.g., a mean) of a set of data items, respectively. The intermediate knowledge from these tasks can be an estimate of the target value, and Guards for this value include hypotheses such as the value is greater or less than a threshold, or is in a specific range. For all the cases, p values can be given as validity measures through a t-test, considering the processed data items as a sample of the entire dataset, although those measures should be interpreted with care. Similarly, the results of the **Find Extremum** and **Sort** tasks are a data item and its rank, respectively, which can be described as a Guard stating that the rank of the item is equal to a number, or higher or lower than a threshold.

Tasks as Guards with Validity Measures. The intermediate knowledge of tasks in the second set can be described as a Guard with an interpretable statistic as a validity measure. For example, the intermediate knowledge for the **Characterize Distribution** task can be a specific distribution that data values are expected to follow. In a Guard form, $\langle variable \rangle$ is the distribution of values, $\langle operator \rangle$ is $\sim$, and $\langle operand \rangle$ is the distribution one identified, such as $\mathcal{N}(\mu, \sigma^2)$. To measure how similar the actual distribution of values is to the expected distribution, one can use the Kolmogorov-Smirnov statistic [117] as a validity measure, which is defined as the maximum difference between the cumulative probability functions of two distributions. For the **Correlate** task, the process of determining the relationship between two attributes, both $\langle variable \rangle$ and $\langle operand \rangle$ are the values of two different attributes. $\langle operator \rangle$ will vary depending on the relationship one found; for example, if a linear relationship is of interest, one can use a $\propto$ operator

and the error from linear regression between two attributes (e.g., root mean square error, RMSE) as a validity measure.

Tasks as Guards without Validity Measures. The third set of tasks consists of the tasks from which quantifying the validity of intermediate knowledge is infeasible in the middle of computation. For example, the **Determine Range** task is an activity of finding the span (i.e., minimum and maximum) of a given attribute. The intermediate knowledge can be expressed by describing an acceptable range in a similar way to the Retrieve Value task. However, in this case, the validity measure would be a boolean value (*true* or *false*), since it is challenging to reliably estimate the minimum and maximum values of an attribute due to the sensitivity of these values. Similarly, the **Filter** task is designed to find the data items that satisfy given conditions. Possible intermediate knowledge from the task is whether such data items exist, that is, $\langle operator \rangle$ will be $exist$ with $\langle operand \rangle$ of $empty$, but it is also hard to predict the existence robustly before processing the entire dataset.

Tasks as Ill-defined Guards. The last set of tasks includes high-level tasks such as **Find Anomalies** and **Cluster**. One can create a Guard for these tasks such as $Num\ of\ Clusters\ in\ Heatmap = 3$, but it is hard to validate such a Guard even after the entire dataset is processed because these tasks require a choice of complex algorithms and parameters. Guards for these tasks would be useful since they can capture higher-level knowledge, but we leave designing and validating such Guards as future work.

5.1.3 Design Considerations

In this section, we discuss design considerations in realizing progressive visual analytics with PVA-Guards on interactive visualization systems.

Input: Explicit vs. Implicit. Guards need to capture intermediate knowledge people want to verify, and thus we need to consider the explicitness of interactions that people use to present their intermediate knowledge to a system. One end of the explicitness continuum is a fully explicit input where people clearly write down a Guard, for example, using a programming language. In this case, people may articulate their knowledge most accurately, but such an approach can be cumbersome and interrupt exploration. On the opposite end of the continuum, we can imagine a fictional system that automatically identifies intermediate knowledge without any user intervention (e.g., eye tracking technology can be used to capture the user intention [60]). This non-intrusive approach, however, can be error-prone, leading people to spend more time fixing the incorrectly captured Guards.

Other interaction methods can be placed in the middle of the continuum. One example is semantic interactions [42] where user interactions are associated with an intention, such as moving two document icons closer for presenting similarity between the documents. In ProReveal, we designed a user interface to allow users to explicitly present their knowledge obtained from a visualization to the system.

Validation: Online vs. Offline. Another tension exists regarding when to validate Guards, since verifying the Guards themselves consumes computational resources. Seeking the greatest accuracy, one can validate Guards during progressive data exploration (i.e., online) at the cost of sacrificing some computational resources that could be spent on exploring data. In contrast, one can validate the Guards after the exploration (i.e., offline), which can be done even when one is offline (e.g., after work). In this case, since Guards that need to be validated are already known, offline computation can speed up the validation process, for example, by testing related Guards together.

A hybrid approach is also feasible to balance accuracy and efficiency, for example, by allowing people to choose which Guards to be validated online or offline based on their importance. In ProReveal, we validate Guards online and show their validity measures in a PVA-Guard list (Section 5.2.3).

Violation: Passive vs. Active. When a Guard turns out to be wrong (e.g., the value of a variable is out of an expected range or the error of linear regression is too large), a system needs to report such a violation to analysts. Simply, the system can take a passive approach, for example, by alerting the analysts and waiting for actions. This is similar to common monitoring systems such as intrusion detection systems (IDS) for network security [1]. Analysts can prescribe how to react to such violations in advance, which allows more active and complex operations. More advanced systems would be able to suggest new values or parameters for the broken Guard and automatically re-run the analysis even when analysts are offline. In all the cases, the Guards can indicate the provenance of intermediate knowledge and can be used to manage the violation. In ProReveal, we employ a passive approach; we present the validity measures of PVA-Guards and let analysts manage violations.

5.2 ProReveal

To provide a clear example of our concept in practice, we designed and implemented a proof-of-concept system, ProReveal. ProReveal integrates PVA-Guards into progressive data exploration, allowing people to seamlessly articulate their findings as Guards in the middle of exploration. With ProReveal, we realize seven important Guards: `Value`, `Rank`, `Range`, `Comparative`, `Power Law`, `Normal`, and `Linear`. In this section, we elaborate on our design

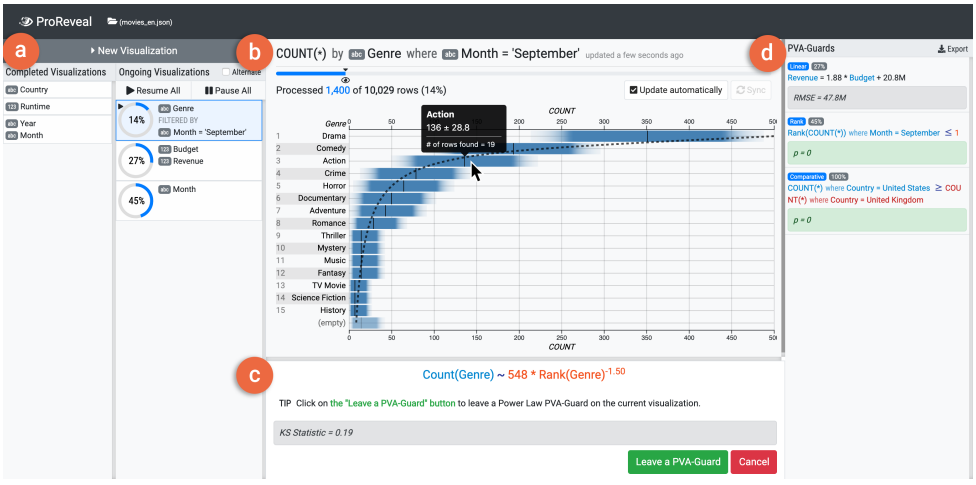


Figure 5.1: The ProReveal Interface. a) Two visualization lists, one for ongoing visualizations and the other for completed visualizations, provide feedback on progress and the ability to control the execution. b) In the main view, A gradient plot is showing the estimated counts of movies of each genre. c) In the PVA-Guard panel, an analyst is creating a Power Law PVA-Guard to leave intermediate knowledge that the distribution of values follows a power law distribution. d) The PVA-Guard view gives an overview of the created PVA-Guards and shows the estimates of their validity.

rationale and the challenges we confronted while integrating PVA-Guards into a progressive data exploration system.

5.2.1 Design Rationale

DR1: Seek Simplicity but Include Essentials. Serving as a proof-of-concept system for demonstrating the seven Guards, ProReveal is designed to provide an initial platform for observing how people use and interact with Guards in progressive data exploration. To this end, we sought simplicity while including the features essential in progressive visual analytics to our design. For example, ProReveal implements important requirements for PVA, such as uncertainty visualization, feedback on progress, execution control, prioritization, and providing quality measures. Nonetheless, we keep the remain-

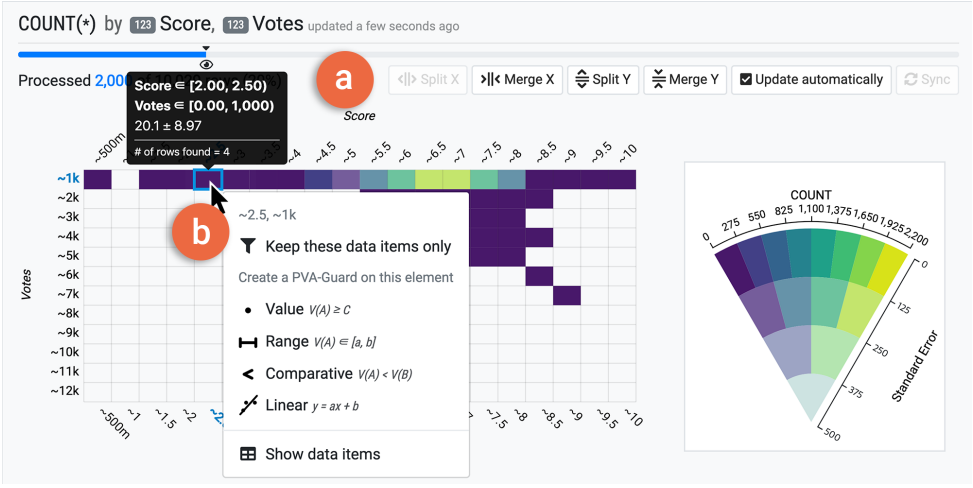


Figure 5.2: The main view with a heatmap. a) A toolbox on the main view provides additional features of a visualization, such as changing the number of bins or postponing automatic updates of a visualization. b) The PVA-Guard menu, appearing when people click on a bar or a cell, enables three special operations on the visual element: 1) creating a new visualization only with the data items under the element, 2) leaving a PVA-Guard on the element, and 3) showing the underlying data items.

ing parts of ProReveal as simple as possible. For example, we decided to provide only two types of visualization (i.e., a gradient plot and a heatmap with Value-Suppressing Uncertainty Palettes [36]) and limit the types of visualization queries.

DR2: Allow Explicit Presentation of PVA-Guards. On a continuum of the explicitness of Guard presentation, we designed our system to support explicit presentation of Guards; people explicitly demonstrate directly on the visualization what intermediate knowledge they want to keep. In ProReveal, people can click on a visual element (i.e., a bar in a bar chart or a cell in a heatmap) and select the type of Guard that they want to leave on the element (Figure 5.2b). This can allow people to create Guards in a familiar and accurate manner, preventing potential errors that can occur when a more implicit method is employed.

DR3: Respect Intermediate Results. Theoretically, people can make Guards that are inconsistent even with the current intermediate results. For example, suppose a bar whose value is 50 with a standard error of 10, estimated by randomly sampling 10% of data. One can make a Guard that confirms the value of the bar is greater than 100, but it is unlikely to happen. In ProReveal, the use of such overly optimistic Guard is discouraged by permitting only Guards that respect the current intermediate results. For instance, in a `Value` Guard that compares the value of a bar or a cell with a given constant, the operator (i.e., $\geq$ or $\leq$) is automatically set by comparing the current value and the constant. Another example can be a `Range` Guard where the center of the target range is always set to the current value of a bar (i.e., only the width of the range can be controlled).

DR4: Keep PVA-Guards Visible. The collection of Guards can serve as an overview of progressive data exploration and the provenance of knowledge generated. Moreover, an unexpected value for their validity measures can be a signal for further inspection or altering the direction of exploration. To gain these benefits, we keep Guards and their validity measures always visible on the interface (i.e., in a PVA-Guard view, Figure 5.1d). We also compute and update the validity measure of a Guard online each time its source visualization (i.e., one on which the Guard is created) is updated to facilitate such steering.

5.2.2 Progressive Visualization

Visualization in PVA systems is required to be scalable and effectively encode uncertainty [62]. We chose to use two visualization techniques from literature: gradient plots [35] for univariate visualization and heatmaps with the Value-Suppressing Uncertainty Palettes (VSUP) [36] for bivariate visualiza-

tion. Both visualization techniques can provide perceptually effective ways to encode uncertainty; gradient plots show the confidence intervals of values as gradients and VSUPs represent uncertainty through the lightness and saturation channels.

In ProReveal, people are allowed to select up to two fields that they want to include in a new visualization. The type of a new visualization is determined depending on the types of the selected fields: a gradient plot is used for a single categorical field (C), a single quantitative field (Q), and a pair consisting of a categorical field and a quantitative field (CQ), while a heatmap is used for two fields of the same type (CC and QQ). Hereafter, we use the letters C and Q before the name of fields and visualizations to denote the types of the fields and the types of the selected fields for the visualization (e.g., a C gradient plot for a gradient plot with a categorical field), respectively.

Gradient Plots. A gradient plot encodes the confidence interval of a value using opacity; 95% two-tailed t-confidence intervals are shown fully opaque, while outside of the intervals, the opacity decays with respect to the cumulative probability of an underlying t distribution [35] (Figure 5.1). We call a gradient in a gradient plot a *bar*, and the center of a gradient (i.e., an estimated value) the *value* of a bar.

In ProReveal, we display gradient plots horizontally for better scalability on the number of displayable categories. C gradient plots show the number of data items each category of a C field has. We sort the categories in descending order by their counts, so that the category with the most data items is shown on the top. When C and Q fields are chosen, the Q field is aggregated over the C field through an aggregation function (i.e., MEAN, SUM, MIN, or MAX). Then, the result is visualized in a CQ gradient plot where the cate-

gories are sorted by the value of the aggregated *Q* field in descending order, consistent with *C* gradient plots.

Q gradient plots bin a *Q* field and visualize the number of data items in each bin, which can be seen as a histogram with uncertain counts. In contrast to *C* gradient plots, we do not sort the bins by their counts to maintain the order of bins. The bin size is determined by an initial sample; we first compute the range of a fixed number of sample values (i.e., 200) and divide the range into a specific number of bins (i.e., 40). We adjust the number of bins, if needed, to ensure the edges of the bins are nice numbers. When a value that lies outside of the range is found during computation, the range is stretched and new bins are created to encompass the value (as was done in a previous study [62]).

Heatmaps. Heatmaps show the number of data items for each combination of categories (for *CC* heatmaps) or bins (for *QQ* heatmaps) as a matrix (Figure 5.2). The color of a cell represents the estimated value (i.e., count) and its standard error using a VSUP [36]. A VSUP encodes a value using the hue of a color (i.e., through the viridis colormap [92]) and the uncertainty of the value using the luminance and saturation of the color. *CC* heatmaps place the categories of each *C* field on the horizontal and vertical axes respectively, and similar to *C* and *CQ* gradient plots, the categories are sorted by the number of data items in each category. Instead of categories, *QQ* heatmaps create bins for each *Q* field and show the count of data items in each 2D bin. For both *Q* fields, we use the same procedure as *Q* gradient plots to determine the number and size of bins.

5.2.3 The ProReveal Interface

ProReveal employs a web-based user interface that enables progressive visual analytics with PVA-Guards (Figures 5.1 and 5.2). ProReveal supports features that are essential in progressive data exploration, such as feedback on progress, execution control, and prioritization (DR1). In this section, we briefly describe the ProReveal interface and interactions. For more detail, please refer to the videos in the supplementary materials. A self-contained demonstration of ProReveal is also available on the Web (see Section 5.2.5). The ProReveal interface consists of three parts: visualization lists, a main view, and a PVA-Guard list.

Visualization Lists. In ProReveal, all visualizations are listed in one of the two juxtaposed lists: one for completed visualizations on the left and the other for ongoing visualizations on the right (Figure 5.1a). When an ongoing visualization is completed, it moves to the completed visualization list. Both lists display a summary of each visualization by presenting the fields used in the visualization, including those for filters. On the ongoing visualization list, a progress ring shows the progress, which becomes hidden when completed. People can pause, resume, and remove a visualization through the buttons that appear when they hover the mouse cursor on a summary (i.e., execution control).

Multiple visualizations can exist in the ongoing list, but only one visualization can be computed at any moment. The visualization being computed is marked with a play icon on its progress ring. By default, the topmost visualization on the ongoing list is processed first to the end. But, one can reorder the visualization on the ongoing list by drag-and-drop interaction. Alterna-

tively, one can check an "Alternate" button to alternate computation between visualizations.

Visualization Creator. To create a new visualization, one can open a visualization creator by clicking on the "New Visualization" button above the visualization lists. The visualization creator shows the list of fields in a dataset labelled by type ("abc" for categorical and "123" for quantitative). One can choose up to two fields, and when one chooses one *C* field and one *Q* field (i.e., for a *CQ* gradient plot), a list of available aggregate functions (i.e., MEAN, SUM, MIN, and MAX) appears below the field list. Based on the types of the chosen fields, ProReveal automatically creates a new visualization (DR1) and adds it to the top of the ongoing visualization list.

Main View. The main view (Figures 5.1b and 5.2) shows the currently selected visualization. By default, a visualization is immediately updated each time a new partial result is available. However, one can postpone the automatic updates by unchecking the "Update automatically" button and manually refresh the visualization through the "Sync" button (Figure 5.2a). Due to a long-tail distribution of values, visualizations for large-scale data often suffer from excessive white space. For *C* fields, we limit the number of categories visible to a fixed number (i.e., 50), and people can see all the categories if they need. For *Q* fields, we allow people to change the granularity of bins on the fly, but the minimum bin size is determined based on a small sample (as mentioned in Section 5.2.2), and one cannot split the bins smaller than that size. When one hovers the mouse cursor over a visual element (i.e., a bar or a cell) in the main view, a tooltip pops up, describing the corresponding categories or intervals of the element, the estimated value (i.e., an aggregate value for *CQ* gradient plots or a count otherwise) and its standard error, and the number of data items found for the corresponding categories or intervals.

	Visualization				
	Gradient Plots			Heatmaps	
Field Types	*C*	*Q*	*CQ*	*CC*	*QQ*
Value	O	O	O	O	O
Rank	O		O		
Range	O	O	O	O	O
Comparative	O	O	O	O	O
Power Law	O	O			
Normal		O			
Linear					O

Table 5.2: Types of applicable PVA-Guards for each field combination

PVA-Guard Menu. The PVA-Guard menu (Figure 5.2b), which can be opened by clicking on a visual element of interest (hereafter, target), shows three types of operations one can perform on the target: 1) creating a new visualization only with the underlying data items of the target (i.e., filtering), 2) leaving a Guard on the target, and 3) showing the underlying data items of the target in a pop-up table. When the filter operation is chosen, the visualization creator appears on the visualization, but in this case, a new visualization is computed only for the data items of the target, not for all data items in the dataset. The types of available Guards vary depending on the visualization and the field types selected (Table 5.2), and only applicable Guards are shown in the PVA-Guard menu.

PVA-Guard Panel. When a Guard type is chosen on the PVA-Guard menu, the PVA-Guard panel appears (Figure 5.1c) below the main view, previewing the Guard and its validity measure. $\langle variable \rangle$ of the Guard is set to the target element on which the PVA-Guard menu was invoked. One may want to set $\langle constant \rangle$ of the Guard; we design the interactions for setting $\langle constant \rangle$ according to the Guard type (see Section 5.2.4). $\langle operator \rangle$ of

the Guard is automatically chosen based on the Guard type and $\langle constant \rangle$ (DR3). Finally, one can click on the "Leave a PVA-Guard" button in the panel to activate the Guard.

PVA-Guard List. The PVA-Guard list provides an overview of created PVA-Guards and their validity measures (DR4, Figure 5.1d). We decided not to provide a history of a validity measure, since it is not meaningful and can give the illusion that the measure is converging, especially for p values. We color-encoded p values to facilitate interpretation; a p value is shown in green when it is lower than a confidence level (i.e., $\alpha = 0.05$), red when it is higher than 0.5 (i.e., worse than a random guess), and yellow otherwise. When people hover the cursor over a Guard, the linked visualization (i.e., the visualization on which the Guard is created) is highlighted in yellow in the visualization lists, and when people clicked a Guard, the linked visualization is shown in the main view. Finally, people can export and download Guards in a JSON format for future use.

5.2.4 PVA-Guards

As shown in Section 5.1.2, we identified what intermediate knowledge people can derive from our target visualizations (i.e., gradient plots and heatmaps) based on previous literature [4, 97] (Table 5.3). Based on our identification, we designed and implemented seven Guards in ProReveal: `Value`, `Rank`, `Range`, `Comparative`, `Power Law`, `Normal`, and `Linear`.

Value

One common task in a bar chart and a histogram is to read the value of a specific category or interval. This task has different names in the literature, such as Retrieve Value [4] or Identify Attribute [97]. In ProReveal, such tasks

Name	Example ($\langle variable \rangle\ \langle operator \rangle\ \langle constant \rangle$)	Validity Measure	Relevant Tasks	
			Amar et al.[4]	Munzner [97]
`Value`	$Price(Apple) \leq \$2$	*p* or *Boolean value*	Retrieve Value & Compute Derived Value	Identify Attribute
`Rank`	$Rank(Price(Apple)) \leq 10$	*p*	Find Extremum & Sort	Identify Extremum
`Range`	$Price(Apple) \in [\$1, \$3]$	*p*	Retrieve Value & Compute Derived Value	Summarize Attribute
`Comparative`	$Price(Melon) \geq Price(Apple)$	*p*	-	Compare Attribute
`Power Law` `Normal`	$Prices\,of\,Fruit \sim a \times Rank^{-k}$ $Prices\,of\,Fruit \sim \mathcal{N}(\mu, \sigma^2)$	*Quality*	Characterize Distribution	Summarize Trend
`Linear`	$Prices\,of\,Fruit \propto Sizes\,of\,Fruit$	*Error*	Correlate	Identify Correlation

Table 5.3: Examples of PVA-Guards, validity measures, and their corresponding tasks

correspond to reading the value of a bar on a gradient plot or a cell on a heatmap. A `Value` Guard allows people to present intermediate knowledge obtained while performing those tasks: the value of a bar or a cell is less or greater than a specific constant (e.g., $Price(Apple) \leq \$2$).

- $\langle variable \rangle$:= value of a bar | value of a cell (e.g., $Price(Apple)$)
- $\langle operator \rangle$:= $\geq$ | $\leq$
- $\langle constant \rangle$:= a number (e.g., $\$2$)

Interaction. When the `Value` Guard is selected on the PVA-Guard menu, an orange constant bar appears on the visualization space (for a gradient plot) or the legend (for a heatmap), showing the value of $\langle constant \rangle$. People can drag the constant bar left or right to change the value. By default, $\langle constant \rangle$ is set to the current value of $\langle variable \rangle$, and $\langle operator \rangle$ is set to $\leq$. $\langle operator \rangle$ is automatically chosen by comparing the value of $\langle variable \rangle$ and $\langle constant \rangle$ when the Guard is created (DR3).

Rank

A `Rank` Guard indicates that the rank of a category is higher or lower than a threshold (e.g., $Rank(Price(Apple)) \leq 10$). The `Rank` Guard is related to tasks of Find Extremum and Sort [4]. The `Rank` Guard is available in a *C* or *CQ* gradient plot where the target rank can be directly specified as a horizontal line on the visualization.

- $\langle variable \rangle$:= rank of a bar (e.g., $Rank(Price(Apple))$)
- $\langle operator \rangle$:= $>$ | $\leq$
- $\langle constant \rangle$:= a rank (e.g., 10)

Interaction. An orange horizontal bar (i.e., the rank bar) appears on the gradient plot, showing the rank of $\langle constant \rangle$. People can drag the rank bar up and down to set the desired rank. By default, $\langle constant \rangle$ is set to the current rank of $\langle variable \rangle$, and $\langle operator \rangle$ is set to $\leq$. $\langle operator \rangle$ is automatically chosen by comparing the rank of $\langle variable \rangle$ and $\langle constant \rangle$ when the Guard is created (DR3).

Range

A `Range` Guard is a two-sided version of a `Value` Guard. The `Range` Guard indicates that the value of a bar or a cell is in a certain range (e.g., $Price(Apple) \in [\$1, \$3]$). With DR3, we only allow symmetric ranges whose center is at $\langle variable \rangle$ when the Guard is created, preventing people from choosing an arbitrary range. Therefore, the `Range` Guard can be seen as representing acceptable bounds that $\langle variable \rangle$ lies on at the end.

- $\langle variable \rangle$:= value of a bar | value of a cell (e.g., $Price(Apple)$)
- $\langle operator \rangle$:= $\in$
- $\langle constant \rangle$:= a range whose center is at $\langle variable \rangle$ (e.g., $[\$1, \$3]$)

Interaction. When the `Range` Guard is chosen, a gray brush appears on the visualization space (for a gradient plot) or the legend (for a heatmap), showing the range of $\langle constant \rangle$. The shape of the brush depends on the visualization type (i.e., rectangular-shape for a gradient plot and pie-shape for the heatmap legend). People can drag the left or right edge of the brush to change the range. The center of the range stays at the current value of $\langle variable \rangle$; as $\langle variable \rangle$ is progressively updated, the center of the range moves, maintaining its width. By default, $\langle constant \rangle$ is set to the 98% confidence interval of $\langle variable \rangle$.

Comparative

A `Comparative` Guard allows people to safeguard comparison between two values, a common task performed on visualization [97]. The `Comparative` Guard takes and compares two variables of the same type (e.g., values of two bars or two cells), e.g., $Price(Melon) \geq Price(Apple)$.

- $\langle variable1 \rangle$:= value of a bar | value of a cell (e.g., $Price(Melon)$)
- $\langle operator \rangle$:= $\geq$ | $\leq$
- $\langle variable2 \rangle$:= value of a bar | value of a cell (e.g., $Price(Apple)$)

Interaction. $\langle variable1 \rangle$ is set to the element (i.e., a bar or a cell) on which the PVA-Guard menu was invoked with click, while people can right-click on a bar or a cell to set $\langle variable2 \rangle$. ProReveal automatically chooses $\langle operator \rangle$ by comparing the value of $\langle variable1 \rangle$ and $\langle variable2 \rangle$ when the Guard is created (DR3).

Power Law and Normal

Identifying the distribution of values is another important task [4, 97]. `Power Law` and `Normal` Guards (hereafter, distributive Guards) present intermediate knowledge that the distribution of values follows a power law or normal distribution, respectively, e.g., $Prices\ of\ Fruit \sim \mathcal{N}(\mu, \sigma^2)$. The `Power Law` Guard is available for both *C* and *Q* gradient plots, but the `Normal` Guard is only available for *Q* gradient plots, since a normal distribution requires quantitative values for its domain. Following DR3, the parameters (e.g., μ and σ for a normal distribution) of the distributive Guards are automatically fit to the data at the moment of creation. Moreover, we chose to update the parameters to fit the most recent data, even after the Guard is created, since in

exploratory analysis people often want to identify the shape of distribution rather than check if the distribution has specific parameters.

- $\langle variable \rangle$:= values of bars (e.g., $Prices\ of\ Fruit$)
- $\langle operator \rangle$:= $\sim$
- $\langle constant \rangle$:= $PowerLaw(k, \alpha) \mid \mathcal{N}(\mu, \sigma^2)$

Interaction. For consistency, people have to open the PVA-Guard menu first by clicking a bar to create the distributive Guards, even though those Guards are not for a single bar but for the whole gradient plot. Then, a dotted curve appears on the gradient plot (Figure 5.1b), showing the current distribution (i.e., $\langle constant \rangle$). As the gradient plot is updated progressively, the parameters are automatically changed to fit the most recent result.

Linear

Designed for tasks of identifying the correlation between two quantitative fields [4, 97], a `Linear` Guard indicates that two Q fields are linearly correlated, e.g., $Prices\ of\ Fruit \propto Sizes\ of\ Fruit$. The `Linear` Guard is only available for QQ heatmaps, and the values of the second Q field is linearly modeled by the first one.

- $\langle variable1 \rangle$:= a Q field (e.g., $Prices\ of\ Fruit$)
- $\langle operator \rangle$:= $\propto$
- $\langle variable2 \rangle$:= a Q field (e.g., $Sizes\ of\ Fruit$)

Interaction. Similar to distributive Guards, people open the PVA-Guard menu first by clicking any cell on a heatmap to create a `Linear` Guard. Then, a dotted line appears on the heatmap, showing the fitting line. As the heatmap

is updated progressively, the parameters of linear fitting (i.e., the slope and the intercept) are automatically changed to fit the most recent result.

5.2.5 Estimation and Implementation

For progressive computation, ProReveal builds uniform samples of a dataset without replacement and processes the samples one by one to yield progressive results. For each sample, we compute the count (for both *C* and *Q* fields), sum, and squared sum (for *Q* fields) of data values and accumulate the numbers over samples. For visualizations that use COUNT, MEAN, or SUM aggregation functions, we statistically estimate the target value and its standard error using the accumulated statistics. For MIN and MAX aggregation functions, which are more sensitive to outliers, we only show the MIN or MAX value we found so far. For scalable computation, ProReveal processes large-scale data on a distributed computing engine, Apache Spark [138], with a similar architecture to a previous study [62]. Please refer to the videos in the supplementary materials to check out our system running on about 1.7 billion entries of the GAIA dataset [28, 75].

ProReveal employs four types of validity measures (Table 5.1): p values for `Value`, `Rank`, `Range`, and `Comparative`, quality (e.g., Kolmogorov-Smirnov statistic) for `Power Law` and `Normal`, error (e.g., root mean square error) for `Linear`, and truth values for Guards on visualizations with MIN and MAX aggregation functions. The validity measures are computed using the sample statistics; therefore, if a visualization is paused, the Guards left on it are not updated. For the progressive computation of uncertainty and validity measures, we assumed that the number of rows in the dataset is known. For more detail, please refer to our supplementary materials.

The metadata of fields, such as type, are conjectured using the fixed number (i.e., 200) of data items, as mentioned in Section 5.2.2. However, people can specify the metadata in a separate file, such as the range or desired bin size of a *Q* field. The ProReveal interface is implemented using TypeScript [125], D3.js [24], and Angular [9]. The source codes of the interface and backend are available at `https://github.com/proreveal`. An interactive demo is also available at `https://proreveal.github.io/ProReveal`.

5.3 Evaluation

We conducted a user study to understand if and how people use PVA-Guards and to evaluate the usability of ProReveal.

5.3.1 Study Design

Participants. We recruited 14 participants (3 females and 11 males) from a university, ranging in ages from 20 to 31 years. We screened the participants through a questionnaire to ensure that 1) they were familiar with using and interpreting common visualizations (e.g., bar charts) and 2) took at least one statistics class with understanding of statistical hypothesis testing and confidence intervals. They received about US$20 for their participation.

Tasks and Datasets. We designed two tasks to evaluate different aspects of data exploration with Guards. From Task 1, we wanted to assess the usability of our interface and interactions for creating Guards. We provided the participants with a structured form of interaction sequences (i.e., exploration recipes). The participants were asked to follow five exploration recipes and answer the question at the end of each recipe as accurately as possible in three minutes.

Consisting of three steps and one question, an exploration recipe describes interactions occurring in analysis where a participant creates a visualization (VIS1 step), leaves a Guard on a finding from the visualization (Guard step), creates another visualization by applying a filter from the first one (VIS2 step), and answers a specific data-driven question. In the VIS1 step, participants were asked to create a visualization by choosing one or two fields in the visualization creator. Then, in the Guard step, they were instructed to make a Guard on the visualization; the parameters for the Guard, such as type, variable, and constant, were described in the recipe, if needed. Next, the participants were asked to create another visualization (VIS2) by selecting a specific category or interval on the first visualization. Finally, they were asked to identify the category or interval with the most data items by answering questions with five choices. If the visualization was not certain enough, the participants needed to wait for a clearer answer.

We designed five templates for the exploration recipes (Table 5.4) that cover five types of Guards (`Value`, `Rank`, `Range`, `Comparative`, and `Linear`) and five field combinations (*C*, *Q*, *CQ*, *CC*, and *QQ*). To control the difficulty, all templates use the same number of fields in total (i.e., three in VIS1 and VIS2). Based on the templates, we created 10 exploration recipes, R1–R5 with a weather dataset for a tutorial and R6-R10 with a birdstrike dataset for Task 1.

With Task 2, we wanted to investigate how participants use ProReveal in progressive data exploration and how they employ Guards when they are not explicitly instructed to use them. We asked participants to explore the given data to find meaningful and trustworthy insights that they want to share with colleagues, considering themselves data scientists, which is similar to the approach used in the evaluation of previous data exploration sys-

Name	VIS1	Guard	VIS2
R1, R6	*C*	Rank	*QQ*
R2, R7	*Q*	Range	*CC*
R3, R8	*CQ*	Comparative	*C*
R4, R9	*CC*	Value	*Q*
R5, R10	*QQ*	Linear	*C*

Table 5.4: Five templates of exploration recipes

tems [63, 134]. To limit the potential effect of datasets, we used two different datasets; seven participants explored a movie dataset and the other seven explored a Korean SAT dataset. After the 10-minute exploration, we asked them to briefly explain the visualizations they created to check if they created the visualizations as they intended.

Both the movie and SAT datasets had six *C* fields and six *Q* fields.We used four datasets: a weather dataset (2,922 rows and 8 fields) [128] for tutorial videos, quizzes after each video, and exercise recipes; for Task 1, a birdstrike dataset (9,987 rows and 15 fields) [128]; for Task 2, a movie dataset (10,029 rows and 12 fields) [121] and a Korean SAT dataset (14,098 rows and 12 fields). For all the datasets, we randomly shuffled the order of rows to limit the effect of potential bias.

Latency Condition. A body of research exists on how the latency of interactive systems can affect user behavior [98, 99]. To control the latency of ProReveal, we simulated the latency of each response by drawing a random number from a normal distribution with a mean of 3,000 ms and a standard deviation of 1,000 ms. The mean and standard deviation of latency were based on a benchmark of a scalable visualization system [62] and were longer than those used in previous studies (e.g., 600 ms and 1,200 ms [139], and 500 ms and 2,500 ms on average [135]). The first response of a visualization was pro-

vided faster (i.e., in 300 ms), which was also feasible in practice [62]. Each response covered 1% of data, so it took five minutes on average (i.e., 3,000 ms $\times$ 100 responses) to finish a visualization if it was the only one being computed in the system. In the experiment, progressive computation was done on a web browser, instead of a backend, to control the latency and avoid unexpected delay due to computation on distributed nodes.

Apparatus. Participants were seated in front of a desktop with two 24-inch monitors at a resolution of 1920 $\times$ 1080. The ProReveal interface was shown on the left monitor (hereafter, Interface), while a web app for the user study was presented on the right monitor (hereafter, Presenter). Presenter managed experimental sessions, such as playing tutorial videos, presenting exploration recipes one by one, and receiving answers from the participants. Presenter remotely controlled Interface; the participants could begin exploring the given data on Interface with the data loaded when they were ready. Both Interface and Presenter logged all important interactions for analysis.

Procedure. After signing a consent form, participants participated in a tutorial session during which they watched four videos played on Presenter: one introductory video about progressive visual analytics, two videos about ProReveal and uncertain visualizations, and the last one about Guards. After watching each video, they took quick quizzes and practiced what they learned from the video on Interface. The entire tutorial took approximately 35 minutes. Then, the participants tried five exercise recipes (i.e., R1-R5), displayed one by one on Presenter. Since it took about five minutes for one visualization to finish, they were not able to see a complete result during the session. After finishing the exercise recipes, participants carried out Task 1 (i.e., R6-R10) without any support from the experimenter. The order of the recipes in Task 1 was randomized.

After an optional three-minute break, the participants were introduced to Task 2. The interface was configured to initially show one univariate visualization for each field in a dataset to provide a starting point for exploration. After the 10-minute exploration, participants reviewed their visualizations one by one with the experimenter. The experimenter transcribed 1) why they created each visualization and 2) what they found from the visualization. After completing both tasks, participants responded to an SUS (System Usability Scale) questionnaire [27] and described their experience with ProReveal. The entire session took about 80 minutes.

5.3.2 Results

We report the results of our user study in three parts: the accuracy and time in Task 1, the number of insights found in Task 2, and qualitative feedback from the participants.

Task 1. Out of 70 (5 recipes $\times$ 14 participants), participants chose correct answers except for only one case where the participant hastily chose one of two competing cells, overlooking their uncertainty. We also checked whether the participants correctly created Guards as in the recipes. Due to the limitation of drag and drop interactions in accuracy, we asked the participants to set a constant as closely as possible to a specific value or a certain range for `Value` and `Rank` Guards, and we permitted 5% margin for the constants of those Guards. We found the participants correctly created Guards in all 70 recipes. Participants spent on average 5.71 ($\sigma = 2.61$) seconds on creating the first visualization, 22.56 ($\sigma = 11.37$) seconds on creating a Guard, 16.75 ($\sigma = 5.67$) seconds on filtering and making the second visualization, and 85.43 ($\sigma = 30.75$) seconds on answering the question of recipes.

Seven participants voluntarily created additional Guards (26 out of 70 recipes) to confirm their answers even though they were not asked to do so. They mostly (in 20 recipes) created `Comparative` Guards to choose one between the top two, since we asked them to choose the answer with the most data items. Other Guards additionally used in Task 1 were `Rank` (8 recipes), `Value` (1), and `Power Law` (1).

Task 2. On average, the participants created 7.64 ($\sigma = 2.21$) visualizations except the initial univariate visualizations given by default. From those visualizations, they found 4.77 ($\sigma = 1.79$) insights by leaving 3.29 ($\sigma = 2.40$) Guards on the visualizations. They used the `Linear` Guard the most (22 times), followed by `Rank` (11 times), `Comparative` (9), `Range` (2), and `Normal` (2). `Value` and `Power Law` Guards were not used in Task 2. We did not find any significant difference between the datasets (i.e., movie and SAT) on the number of visualizations created, insights reported, and Guards created ($ps > .05$, ns).

Subjective Feedback and Interview. ProReveal received an average SUS score of 78.39 ($\sigma = 10.99$), which lies in between "Good" and "Excellent" adjective ratings [15]. Through an interview, we surveyed major strategies our participants developed to decide a correct answer. Observing a clear gap between confidence intervals was the most frequently used one (8 participants), and other responses were waiting until a specific amount of data was processed (7), using Guards (6), waiting as long as possible (2), and checking the stability of visualization over time (2).

In the interview, two participants also suggested new types of PVA-Guards that would be helpful for their exploration. P5 suggested it would be useful if he could leave a Guard on multiple visual elements at once (e.g., three bars), which calls for a new type of variable, that is, a *group* variable. With

`Rank` Guards, such a group variable can be used to safeguard the knowledge that a group of categories are on the top (i.e., in the top 3). In addition, P7 suggested a Guard that tests the significance of linear regression would be helpful in his daily analysis, thus complementing `Linear` Guards.

5.4 Discussion

Our study results suggest that participants could understand the concept of Guards and incorporate Guards in their data exploration through the ProReveal interface. Participants could follow all visualization recipes and choose correct answers except for only one case after about 30-minute training, and seven participants voluntarily used Guards in Task 1. In addition, as the average SUS score suggests, they positively rated the ProReveal interface.

In this section, based on our results and observations as well as the participants' feedback, we reflect on how participants used Guards in progressive data exploration. We then discuss the limitations of our lab study and future research directions.

Benefits of PVA-Guards and ProReveal. In Task 1, six participants reported that using Guards was their major strategy to deal with uncertainty. P8 stated, "*The major benefit of Guards was the feedback on my intermediate findings from progressive visualization.*" In addition, P11 noted, "*Guards helped me to build trust on my hypotheses and proceed to subsequent analysis,*" which advocates the benefit of PVA-Guards.

The PVA-Guard list of ProReveal served as an overview of uncertain intermediate knowledge, which seems to be helpful in recalling the context of safeguarded knowledge. For example, P4 said "*I could be aware of the overall progress, because* [*the PVA-Guard list*] *persistently presents my Guards. I left*

Guards on interesting but uncertain knowledge, so that they are visible on the screen, and I continued to explore data." In addition, P11 stated, "*I can resume analysis on the visualizations (on which the Guards are left), because I captured them as Guards, which, I believe, can speed up the analysis,*" indicating she used Guards to recall the context of previous findings and resume the analysis.

Diversity in Uncertainty Interpretation. Our study revealed that participants employed different strategies to interpret the uncertainty of a visualization. Eight (out of 14) participants checked whether there was a clear gap in confidence intervals between two competing bars, but a "clear" gap is still arbitrary and subject to bias. Waiting for a specific amount of data being processed (seven out of 14) was the second most frequently used strategy, but it does not guarantee right decisions and can also misguide decisions. The amount of data participants thought enough to make decisions varied (e.g., 20%, 25%, or 33%). Considering the potential threats resulting from such heterogeneity in uncertainty interpretation, we believe PVA-Guards can be a systematic means for preventing people from making incorrect, hasty decisions and validating their correctness even after wrong decisions are made.

Unexpected Use of the PVA-Guard Panel. We observed that participants sometimes used the PVA-Guard panel (Figure 5.1d) to bolster confidence on their hypotheses by just checking uncertainty measures (e.g., p values) from a preview of a Guard without actually creating it. Participants opened the PVA-Guard panel (by choosing a Guard type on the PVA-Guard menu) 4.93 times ($\sigma = 3.05$) on average in Task 2, but only 3.29 ($\sigma = 2.40$) Guards were actually created. This means that the participants closed the panel without creating a Guard in approximately 33% of cases. P3 said, "*I used the panel just to check if the popularity field is linearly related to the score field, since I wanted to*

know how popularity is calculated," which indicates that he appropriated the PVA-Guard panel to understand the data.

Concerns on the Use of p Values. We used p values as validity measures, but this should be done with care in future designs. The benefits of p values would be their interpretability and familiarity, but we found that they can give an illusion of certainty. P2 stated, "*I was surprised that p values change more than I expected. I am not sure I can absolutely trust p values.*" Similarly, P6 noted, "*At first, I tried to choose an answer as fast as possible, but after I saw fluctuation in p values, I became more cautious.*" Although the interface would be more complicated, providing control over multiple hypothesis testing can alleviate this problem [140].

Limitations and Future Work. In our study, participants explored data that they have not seen before, although we used datasets that they were familiar with (i.e., movie and SAT). This might be a reason why participants preferred Guards with relative values (i.e., `Linear`, `Rank`, and `Comparative`) in Task 2 to Guards with absolute values (i.e., `Value` and `Range`) that require deeper understanding of the data domain.

We aimed to evaluate the usability of ProReveal in a lab study. Therefore, our latency conditions such as the latency of a visualization (i.e., 3 seconds on average) and the time taken to complete a visualization (i.e., 5 minutes on average) were shorter than ones common in practice. We are interested in deploying our system and investigating how Guards can be used to validate the conclusions from the exploration of real-world data where visualizations take a few hours or a day to finish. In this case, a session can span a few days, so it would be important to allow people to recall and continue the previous analysis where PVA-Guards can be possibly used. Furthermore, it would be also interesting to explore a different combination of design choices from

ones we made. For example, we may want to employ more active approaches (e.g., alert an analyst) for the exploration of large-scale data since one can be offline when a violation happens due to the length of the session.

Guards can be logically combined to express higher-level knowledge. However, since the ten low-level tasks [4] mainly focus on analytic activities on multidimensional tabular datasets, our Guards are not complete enough to represent general knowledge gained from data exploration. However, if we have a solid task taxonomy for a certain type of datasets, for example, a task taxonomy for graph visualization [72], we can apply a similar approach to ours to build a new set of Guards. Finally, we designed the interactions for Guards on two visualization types (i.e., gradient plots and heatmaps). We chose the two visualizations because they can visualize univariate or bivariate results with a proven capability of showing uncertainty. In the future, it would be interesting to extend Guards to a wider range of visualizations such as progressive parallel coordinates [109].

5.5 Summary

Despite the benefits of progressive visual analytics (PVA), managing the trustworthiness of intermediate outcomes has been regarded as a core concern when applying PVA to a wider range of scenarios. To tackle this problem, we present a novel concept of Progressive Visual Analytics with Safeguards and a proof-of-concept system, ProReveal, which incorporates seven types of PVA-Guards. ProReveal allows people to present their uncertain intermediate knowledge as PVA-Guards on visualizations during progressive data exploration. Then, ProReveal validates PVA-Guards online, providing their validity measures as an estimate of uncertainty. The results of our

user study were promising; we found that people can utilize this concept to gain trust in the intermediate knowledge and steer their exploration. We also found a potential benefit of PVA-Guards—a means of achieving consistency in progressive data exploration to alleviate the heterogeneity in uncertainty interpretation. We believe our concept can be extended to a broader range of tasks, datasets, and visualizations in the near future, serving as an effective means of achieving trustworthiness in PVA.

Chapter 6

Discussion

In this chapter, we discuss lessons learned from the process of seeking answers to the research questions and the limitations of this dissertation.

6.1 Lessons Learned

6.1.1 Challenges in Realizing "Strict" Progressiveness

In Section 2.1.1, we covered three definitions of PVA with different levels of strictness. For example, the definition given by Stolper et al. [120] does not require a guarantee on latency, which may be the "loosest" definition, while the definition made in the Dagstuhl seminar [44] would be the strictest since it requires the response time of computation to be bounded. Such a latency-guaranteed progressive system would be ideal from the users' perspective. However, through the course of this research, we identified three real-world hurdles to achieving such strict progressiveness as follows.

First, the parameters of a progressive system are often indirect estimators of the system latency. In SwiftTuna and ProReveal, the number of rows processed in each iteration was an important parameter that users can tune to achieve a certain level of latency. PANENE has a few parameters, such as the

number of operations allowed in each iteration (ops) and the fraction of time used for insertion tasks and rebuilding tasks (τ), to strike a balance between latency and accuracy. Although these parameters can be seen as estimators for the system latency, they are not enough to predict the actual latency of a system. For example, even though one finds a reasonable combination of parameters to achieve a specific level of latency, the combination would not work for different datasets or hardware settings. In this sense, the parameters should be tuned considering target data and hardware or dynamically adjusted during execution monitoring the actual latency of a system if possible. One possible remedy to this problem is the Time Predictor, presented by Fekete and Primet [45], which dynamically infers the number of operations that an algorithm executes per second.

Second, the throughput of a progressive system can be severely degraded to achieve progressiveness. Providing intermediate outcomes during computation imposes an extra cost that often involves input and output (IO). Indeed, in the performance benchmark of SwiftTuna, we found that processing ten times more rows in an iteration only doubles the average response time of the system, which is a 5x speedup. The reason may be that even though we use smaller blocks, there is fixed overhead, such as a delay from sending partial responses to a client, so that the system latency cannot be reduced below the overhead. For better throughput, it would be possible to allow users to dynamically adjust the progressiveness required for the system so that, in an extreme case, they can completely turn off progressive computation for the highest throughput.

Third, to support progressive analytics, every component in its computation pipeline must be progressive, which is a strong requirement. As an example, let us suppose a task where an analyst wants to visualize high-

dimensional data into a heatmap using Responsive *t*-SNE progressively. The computation pipeline for this task consists of four stages: 1) reading data from disks, 2) embedding the data into the 2D space, 3) measuring the density of data points at each cell in a grid on the 2D space, and 4) drawing a heatmap by interpolating the densities between the cells. All four stages should be done progressively, which is possible in this case. However, in practice, there can be a sequential computation that can block the whole pipeline. This problem calls for an ecosystem of progressive components that span all layers of visual analytics from the lowest layer (e.g., reading from a disk) to the highest layer (e.g., visualization).

6.1.2 Visualization in Progressive Systems

Visualization in PVA should be 1) scalable and 2) capable of representing the uncertainty of data. In SwiftTuna and ProReveal, we chose to use perceptually effective visualization techniques: gradient plots [35] for univariate visualization and heatmaps with the Value-Suppressing Uncertainty Palettes (VSUP) [36] for bivariate visualization. Both visualization techniques can encode uncertainty; gradient plots show the confidence intervals of values as gradients and VSUPs represent uncertainty through the lightness and saturation channels. However, even with these techniques, we had to design user interactions for better scalability, such as zooming and brushing. We focused on univariate or bivariate visualizations for tabular datasets. However, for other types of data, we found that very few visualization techniques satisfy both requirements, and developing such visualization techniques would be a promising future direction.

6.1.3 Trustworthiness and Accessibility, Two New Considerations in Visual Analytics

We put forward trustworthiness and accessibility as two new considerations for visual analytics with exploding data volumes. Measuring the trustworthiness of progressive results has been considered as one of the major hurdles of PVA [8, 88]. With the increasing size and complexity of data, we believe it will be more prevalent to make decisions based on uncertain results. Designing our concept, PVA with Safeguards, we discussed how PVA-Guards can be used to represent, verify, and correct such uncertain findings. Further investigating this concept, we believe it would be interesting to explore the opportunities in design choices different from the ones we made in the ProReveal design (Section 5.1.3).

The other consideration that we want to emphasize is the accessibility of intermediate results. Previously, the length of a visual analytics session is usually up to humans; users can pause or stop analysis when they want. However, in PVA sessions, computation is not completed due to the sheer size of data or the complexity of algorithms even after the users left the sessions. To connect the users and systems, we envision a PVA system that allows users to access intermediate results ubiquitously, for example, by providing mobile interfaces. With PVA-Guards, the system may notify users when there are changes in the validity of intermediate knowledge.

6.2 Limitations

We would like to mention the limitations of the three studies included in this dissertation. For all three systems and algorithms, we introduced parameters that can control the computation time of an iteration, such as the number of

rows in a batch in ProReveal and the number of allowed operations in an iteration (ops) in PANENE. These parameters should be adjusted depending on the amount of available computational resources and datasets dynamically if possible.

SwiftTuna and ProReveal do not resort to a preprocessing scheme, such as data cubes, but work on data chunks (i.e., blocks). Usually, data chunking does not cost very much and is more effective for cold-start analysis than preprocessing schemes. However, if data is skewed, creating sequential chunks would introduce sampling bias, so randomization is needed, for example, by shuffling the order of chunks or rows.

In the benchmark of PANENE, we used datasets with 100 dimensions. To handle a huge number of dimensions, one can apply a linear dimensionality reduction method first (e.g., PCA [133]) to reduce the number of dimensions to a reasonable number. PANENE has linear loops inside, for example, to split a node in k-d trees. These loops are $O(N)$, but if data is too large, they can eventually block the algorithm. We leave separating the execution of those loops as future work.

In ProReveal, we design and implement seven PVA-Guards on two visualization techniques (i.e., gradient plots and heatmaps). However, our implementation can be extended to different types of knowledge or visualization techniques. Furthermore, although we present initial user study results, we believe that evaluating our concept through a deployment study is necessary and can be done soon.

Chapter 7

Conclusion

Concluding the dissertation, this chapter reviews the contributions that we made and presents future research agendas for PVA.

7.1 Thesis Contributions Revisited

This dissertation endeavored to support the thesis statement (Well-designed progressive systems and algorithms can overcome the scalability and reliability challenges of visual analytics universal in modern data science, enabling the responsive and trustworthy exploration of large-scale multidimensional data.), and present the outcomes of three research projects with each answering one of our research questions.

The first research question was **"Vertical Scalability: how do we enable interactive visual exploration of large-scale data with scalability in both data processing and visualization?"** As an answer to the question, we present SwiftTuna, an interactive system that streamlines the visual information seeking process on large-scale multidimensional data. We design an interactive interface for multidimensional data exploration, seeking the scalability of visualization through interaction techniques, such as Overview+Detail

and zooming as well as introducing tailed charts. To support responsive querying on large-scale data, SwiftTuna leverages an incremental processing approach and provides immediate low-fidelity responses (i.e., prompt responses) as well as delayed high-fidelity responses (i.e., incremental responses). For scalability in data processing, we exploit an in-memory computing engine, Apache Spark [137], to achieve both scalability and performance without building precomputed data structures. Our performance benchmark demonstrates that SwiftTuna can respond to various visualization queries on a real-world dataset with four billion records while preserving the latency between incremental responses within a few seconds.

The second research question was **"Horizontal Scalability: how do we responsively embed and visualize high-dimensional data on a 2D space without long initial computation delays?"** To address the question, we present a progressive algorithm, PANENE, for approximate k-nearest neighbor indexing and querying. PANENE conforms to the most strict definition of progressive computation, i.e., the bounded time between consecutive responses, which contrasts itself with previous online algorithms. PANENE can also incrementally build and maintain a cache data structure, a KNN lookup table, to enable constant-time lookups for KNN queries. With the KNN lookup table, we demonstrated three progressive applications of PANENE, such as regression, density estimation, and Responsive t-SNE. Our benchmarks revealed that PANENE and the KNN lookup table can guarantee the insertion time in each iteration under a specified threshold as well as providing tunable parameters for the trade-off between latency and accuracy.

The third research question was **"Knowledge Trustworthiness: how do we improve the trustworthiness of intermediate knowledge gained from progressive visual analytics?"** The importance of providing trustworthy re-

sults has been recognized in the PVA community [8, 88]. Our approach is based on a novel concept, Progressive Visual Analytics with Safeguards, where we allow people to leave PVA-Guards on uncertain intermediate knowledge that should be verified later. We also present ProReveal, a proof-of-concept PVA system that supports seven safeguards that we derived based on previous task taxonomies. Our user study with 14 participants revealed that people voluntarily employed PVA-Guards to safeguard their findings, and ProReveal's PVA-Guard view provides an overview of uncertain intermediate knowledge. We also found heterogeneity in uncertainty interpretation and believe our new concept can also offer better consistency in PVA.

7.2 Future Research Agenda

Inspired by the lessons learned, we present promising future research agendas for PVA.

7.2.1 Exploring the Design Space of Progressive Visual Analytics with Safeguards

In Chapter 5, we presented Progressive Visual Analytics with Safeguards and designed a proof-of-concept system, ProReveal. ProReveal was built based on our choices of design considerations; for example, it supports explicit knowledge representation with passive approaches to violations. However, we can see the potential of other design choices. For example, if it takes very long to validate PVA-Guards, it would be more useful to take active approaches (e.g., alert an analyst) for handling violations. Another example is to take a more implicit approach to elicit knowledge from visualization, for example, by predicting the knowledge that users can gather from a specific visualization. A sophisticated system would automatically recommend

possible intermediate knowledge from progressive visualization and allow users to confirm and safeguard the knowledge simply.

In addition to investigating various combinations of design choices for PVA with Safeguards, it would be also interesting to design PVA-Guards for a different set fo visualizations. For example, on progressive parallel coordinates [109], one may want to leave a safeguard on the correlation coefficient between two dimensions.

7.2.2 Alleviating Performance Degradation Resulting from Progressive Outputs

As our benchmarks demonstrated, reporting progressive outputs during computation degrades the throughput of a system. One exciting direction for future research is to find a way to minimize such performance degradation. A system can allow users to dynamically adjust the required response time according to the stage of analysis; for example, one wants to see ongoing results more frequently for the early stage of analysis but lessens the latency requirement after hypotheses are made. PVA-Guards can be used in such a scenario by alerting analysts only when PVA-Guards are invalidated instead of giving them intermediate outcomes every a few seconds. Similarly, we can imagine a system that can report intermediate results when it is focused (i.e., when analysts get back to the analysis).

7.2.3 Enhancing the Accessibility of Progressive Results

Validating the findings from PVA sessions on the entire data can take a long time and persist even after analysts left the sessions. In order to help analysts be aware of the progress of the validation process, it is essential to make the process accessible, for example, through mobile interfaces. Beyond merely

showing the progress, such interfaces can serve as a means of steering the validation process. For example, when intermediate knowledge turned out to be wrong, one can remotely refine the knowledge and re-run the validation process. Mobile devices can be insufficient to perform full-featured visual analytics, but we see opportunities to leverage them to alleviate analysts' burden of monitoring the validation process.

7.2.4 Enriching the Ecosystem for Progressive Visual Analytics

Progressive systems require every stage in computation pipelines, such as loading, processing, and visualizing data, to be progressive. Although there have been attempts to modularize PVA components [45], most contributions to PVA are still made in a fragmentized manner. One important research direction is to consolidate such contributions and enrich an ecosystem for PVA with reusable analytic components consisting of progressive modules that span all layers from the lowest layer (e.g., loading data from a disk) to the highest layer (e.g., visualization and interaction).

7.3 Final Remarks

Throughout this dissertation, we designed and evaluated systems and algorithms to interact with large-scale data by developing SwiftTuna and PANENE for responsive visualization and ProReveal for trustworthy visual exploration. Since its introduction, visualization has served a crucial role in data analysis, bridging the gap between humans and computers. With the unprecedentedly increasing size of data and analytic models, it will become ever more critical to empower humans to understand and interact with data.

Data sizes matter.

Bibliography

[1] Kulsoom Abdullah, Christopher P. Lee, Gregory J. Conti, John A. Copeland, and John T. Stasko. Ids rainstorm: visualizing ids alarms. In *VizSEC*, page 1. Citeseer, 2005.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, Prague, Czech Republic. ACM, 2013.

[3] Faraj Alhwarin, Alexander Ferrein, and Ingrid Scholl. Crvm: circular random variable-based matcher - a novel hashing method for fast nn search in high-dimensional spaces. In *ICPRAM*, 2018.

[4] Robert Amar, James Eagan, and John Stasko. Low-level components of analytic activity in information visualization. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* Pages 111–117, October 2005.

[5] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 459–468, October 2006.

[6] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. *CoRR*, abs/1509.02897, 2015.

[7] Marco Angelini, Thorsten May, Giuseppe Santucci, and Hans-Jörg Schulz. On Quality Indicators for Progressive Visual Analytics. In Tatiana von Landesberger and Cagatay Turkay, editors, *EuroVis Workshop on Visual Analytics (EuroVA)*. The Eurographics Association, 2019.

[8] Marco Angelini, Giuseppe Santucci, Heidrun Schumann, and Hans-Jörg Schulz. A review and characterization of progressive visual analytics. *Informatics*, 5:31, 2018.

[9] Angular. https://angular.io/. Last accessed: 2019-11-20.

[10] Erik Bernhardsson. Benchmarking nearest neighbors. https://github.com/erikbern/ann-benchmarks. Last accessed: 2019-11-20.

[11] Francis J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.

[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, et al. Spark sql: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[13] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435. ACM, 2002.

[14] Sriram Karthik Badam, Niklas Elmqvist, and Jean-Daniel Fekete. Steering the craft: ui elements and visualizations for supporting progressive visual analytics. *Computer Graphics Forum*, 36(3):491–502, June 2017.

[15] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[16] Josh Barnes and Piet Hut. A hierarchical $o(n \, o \, limits@logN)$ force-calculation algorithm. *nature*, 324(6096):446, 1986.

[17] Mike Barnett, Badrish Chandramouli, Robert DeLine, Steven Drucker, Danyel Fisher, Jonathan Goldstein, Patrick Morrison, and John Platt. Stat!: an interactive analytics environment for big data. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1013–1016. ACM, 2013.

[18] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH Forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660. ACM, 2005.

[19] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *cvpr*, pages 1000–1006. IEEE, 1997.

[20] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[21] Erik Bernhardsson. Annoy. `https://github.com/spotify/annoy`. Last accessed: 2019-11-20.

[22] Barnes-Hut t-SNE. `https://github.com/lvdmaaten/bhtsne`. Last accessed: 2019-11-20.

[23] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[24] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[25] Nadia Boukhelifa, Anastasia Bezerianos, Tobias Isenberg, and Jean-Daniel Fekete. Evaluating sketchiness as a visual variable for the depiction of qualitative uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2769–2778, December 2012.

[26] Gary Bradski. OpenCV. *Dr. Dobb's Journal of Software Tools*, 2000.

[27] John Brooke et al. SUS-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[28] AGA Brown, A. Vallenari, T. Prusti, JHJ De Bruijne, C. Babusiaux, CAL Bailer-Jones, M. Biermann, Dafydd Wyn Evans, L. Eyer, Femke Jansen, et al. Gaia data release 2-summary of the contents and survey properties. *Astronomy & astrophysics*, 616:A1, 2018.

[29] Mackinlay Card. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[30] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with Roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[31] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[32] Ed Huai-hsin Chi and John T. Riedl. An operator interaction framework for visualization systems. In *Proceedings IEEE Symposium on Information Visualization (Cat. No. 98TB100258)*, pages 63–70. IEEE, 1998.

[33] Jaegul Choo, Changhyun Lee, Hannah Kim, Hanseung Lee, C.K. Reddy, B.L. Drake, and Haesun Park. Pive: per-iteration visualization environment for supporting real-time interactions with computational methods. In *Visual Analytics Science and Technology (VAST), 2014 IEEE Conference on*, pages 241–242, October 2014.

[34] Kristin A. Cook and James J. Thomas. Illuminating the path: The research and development agenda for visual analytics. Technical report, Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2005.

[35] Michael Correll and Michael Gleicher. Error bars considered harmful: exploring alternate encodings for mean and error. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2142–2151, December 2014.

[36] Michael Correll, Dominik Moritz, and Jeffrey Heer. Value-suppressing uncertainty palettes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '18, 642:1–642:11, Montreal QC, Canada. ACM, 2018.

[37] Criteo's terabyte click logs. `http://labs.criteo.com/downloads/download-terabyte-click-logs`. Last accessed: 2019-11-20.

[38] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[39] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[40] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[41] Philipp Eichmann, Emanuel Zgraggen, Zheguang Zhao, Carsten Binnig, and Tim Kraska. Towards a benchmark for interactive data exploration. *IEEE Data Eng. Bull.*, To Appear.

[42] Alex Endert, Patrick Fiaux, and Chris North. Semantic interaction for visual text analytics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 473–482, Austin, Texas, USA. ACM, 2012.

[43] Cliff Engle, Antonio Lupher, Reynold S. Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 689–692. ACM, 2012.

[44] Jean-Daniel Fekete, Danyel Fisher, Arnab Nandi, and Michael Sedlmair. Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411). *Dagstuhl Reports*, 8(10):1–40, 2019. Jean-Daniel Fekete, Danyel Fisher, Arnab Nandi, and Michael Sedlmair, editors.

[45] Jean-Daniel Fekete and Romain Primet. Progressive analytics: a computation paradigm for exploratory data analysis. *ArXiv e-prints*, July 2016.

[46] Danyel Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 73–80. IEEE, 2011.

[47] Danyel Fisher, Steven M. Drucker, and A Christian König. Exploratory visualization involving incremental, approximate database queries and uncertainty. *IEEE Computer Graphics and Applications*, 32(4):55–62, 2012.

[48] Danyel Fisher, Igor Popov, Steven Drucker, and M. Schraefel. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1673–1682, Austin, Texas, USA, 2012.

[49] Marius Muja. FLANN - Fast Library for Approximate Nearest Neighbors. `https://github.com/mariusmuja/flann`. Last accessed: 2019-11-20.

[50] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[51] Keinosuke Fukunaga. *Introduction to statistical pattern recognition*. Academic press, 2013.

[52] Michael Glueck, Azam Khan, and Daniel J. Wigdor. Dive in!: enabling progressive loading for real-time navigation of data visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 561–570, Toronto, Ontario, Canada. ACM, 2014.

[53] Parke Godfrey, Jarek Gryz, and Piotr Lasek. Interactive visualization of large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2142–2157, 2016.

[54] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[55] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, page 1312, 2011.

[56] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J Haas. Interactive data analysis: the control project. *Computer*, 32(8):51–59, 1999.

[57] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, Tucson, Arizona, USA. ACM, 1997.

[58] Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. Toward scalable systems for big data analytics: a technology tutorial. *IEEE access*, 2:652–687, 2014.

[59] Jean-François Im, Felix Giguere Villegas, and Michael J McGuffin. Visreduce: fast and responsive incremental information visualization of large datasets. In *2013 IEEE International Conference on Big Data*, pages 25–32. IEEE, 2013.

[60] Young-Min Jang, Rammohan Mallipeddi, Sangil Lee, Ho-Wan Kwak, and Minho Lee. Human intention recognition based on eyeball movement pattern and pupil size variation. *Neurocomputing*, 128:421–432, 2014.

[61] You Jia, Jingdong Wang, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3392–3399. IEEE, 2010.

[62] Jaemin Jo, Wonjae Kim, Seunghoon Yoo, Bohyoung Kim, and Jinwook Seo. Swifttuna: responsive and incremental visual exploration of large-scale multidimensional data. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*, pages 131–140, April 2017.

[63] Jaemin Jo, Sehi L'Yi, Bongshin Lee, and Jinwook Seo. Touchpivot: blending wimp & post-wimp interfaces for data exploration on tablet devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '17, pages 2660–2671, Denver, Colorado, USA. ACM, 2017.

[64] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. A progressive k-d tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*, pages 1–5, October 2017.

[65] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. PANENE: a progressive algorithm for indexing and querying approximate k-nearest neighbors. *IEEE Transactions on Visualization and Computer Graphics*:preprint, 2018.

[66] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Profiler: integrated statistical analysis and visualization for data quality assessment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 547–554. ACM, 2012.

[67] Daniel A. Keim, Florian Mansmann, Jörn Schneidewind, Jim Thomas, and Hartmut Ziegler. *Visual analytics: scope and challenges*. In *Visual Data Mining: Theory, Techniques and Tools for Visual Analytics*. Simeon J. Simoff, Michael H. Böhlen, and Arturas Mazeika, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pages 76–90.

[68] Hannah Kim, Jaegul Choo, Changhyun Lee, Hanseung Lee, Chandan K Reddy, and Haesun Park. Pive: per-iteration visualization environment for real-time interactions with dimension reduction and clustering. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[69] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2130–2137, September 2009.

[70] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[71] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[72] Bongshin Lee, Catherine Plaisant, Cynthia Sims Parr, Jean-Daniel Fekete, and Nathalie Henry. Task taxonomy for graph visualization. In *Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization*, pages 1–5. ACM, 2006.

[73] Bastian Leibe, Krystian Mikolajczyk, and Bernt Schiele. Efficient clustering and matching for object class recognition. In *BMVC*, pages 789–798, 2006.

[74] Jianping Kelvin Li and Kwan-Liu Ma. P5: portable progressive parallel processing pipelines for interactive data analysis and visualization. *IEEE Transactions on Visualization and Computer Graphics*:1–1, 2019.

[75] Lennart Lindegren, Carine Babusiaux, C. Bailer-Jones, Ulrich Bastian, Anthony GA Brown, Mark Cropper, Erik Høg, Carme Jordi, David Katz, F. Van Leeuwen, et al. The gaia mission: science, organization and present status. *Proceedings of the International Astronomical Union*, 3(S248):217–223, 2007.

[76] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, December 2013.

[77] Can Liu, Cong Wu, Hanning Shao, and Xiaoru Yuan. Smartcube: an adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[78] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, December 2014.

[79] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: real-time visual querying of big data. In *Proc. of the 15th Eurographics Conference on Visualization*,

EuroVis '13, pages 421–430, Leipzig, Germany. The Eurographs Association & John Wiley & Sons, Ltd., 2013.

[80] Sharon L. Lohr. *Sampling: Design and Analysis: Design and Analysis*. Chapman and Hall/CRC, 2019.

[81] William E. Lorensen and Harvey E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA. ACM, 1987.

[82] Min Lu, Shuaiqi Wang, Joel Lanir, Noa Fish, Yang Yue, Daniel Cohen-Or, and Hui Huang. Winglets: visualizing association with uncertainty in multi-class scatterplots. *IEEE Transactions on Visualization and Computer Graphics*:1–1, 2019.

[83] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.

[84] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[85] Alan M. MacEachren, Robert E. Roth, James O'Brien, Bonan Li, Derek Swingley, and Mark Gahegan. Visual semiotics & uncertainty visualization: an empirical study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2496–2505, 2012.

[86] Yu A. Malkov and DA Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv preprint arXiv:1603.09320*, 2016.

[87] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[88] Luana Micallef, Hans-Jörg Schulz, Marco Angelini, Michaël Aupetit, Remco Chang, Jörn Kohlhammer, Adam Perer, and Giuseppe Santucci. The Human User in Progressive Visual Analytics. In Jimmy Johansson, Filip Sadlo, and G. Elisabeta Marai, editors, *EuroVis 2019 - Short Papers*. The Eurographics Association, 2019.

[89] Robert B. Miller. Response time in man-computer conversational transactions. In *Proc. of the Fall Joint Computer Conference, Part I*, pages 267–277, San Francisco, California. ACM, 1968.

[90] Dominik Moritz, Danyel Fisher, Bolin Ding, and Chi Wang. Trust, but verify: optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '17, pages 2904–2915, Denver, Colorado, USA. ACM, 2017.

[91] Dominik Moritz, Bill Howe, and Jeffrey Heer. Falcon: balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 694. ACM, 2019.

[92] Mpl colormaps. `https://bids.github.io/colormap/`. Last accessed: 2019-11-20.

[93] MRPT performance comparison. `https://github.com/ejaasaari/mrpt-comparison`. Last accessed: 2019-11-20.

[94] Thomas Mühlbacher, Harald Piringer, Samuel Gratzl, Michael Sedlmair, and Marc Streit. Opening the black box: strategies for increased user involvement in existing algorithm implementations. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1643–1652, December 2014.

[95] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[96] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36, 2014.

[97] Tamara Munzner. *Visualization analysis and design*. AK Peters/CRC Press, 2014.

[98] Jakob Nielsen. Response times: the 3 important limits. `https://www.nngroup.com/articles/response-times-3-important-limits/`, March 2016.

[99] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[100] Hannu Oja. Descriptive statistics for multivariate distributions. *Statistics & Probability Letters*, 1(6):327–332, 1983.

[101] Cicero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. Hashedcubes: simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):671–680, 2016.

[102] Alex T. Pang, Craig M. Wittenbrink, and Suresh K. Lodha. Approaches to uncertainty visualization. *The Visual Computer*, 13(8):370–390, 1997.

[103] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[104] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[105] Nicola Pezzotti, Thomas Höllt, Jan Van Gemert, Boudewijn Lelieveldt, Elmar Eisemann, and Anna Vilanova. Deepeyes: progressive visual analytics for designing deep neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 24, January 2018.

[106] Nicola Pezzotti, Boudewijn P. F. Lelieveldt, Laurens van der Maaten, Thomas Höllt, Elmar Eisemann, and Anna Vilanova. Approximated and user steerable tsne for progressive visual analytics. *CoRR*, abs/1512.01655, 2015.

[107] Nicola Pezzotti, Alexander Mordvintsev, Thomas Hollt, Boudewijn PF Lelieveldt, Elmar Eisemann, and Anna Vilanova. Linear tsne optimization for the web. *arXiv preprint arXiv:1805.10817*, 2018.

[108] Sajjadur Rahman, Maryam Aliakbarpour, Ha Kyung Kong, Eric Blais, Karrie Karahalios, Aditya Parameswaran, and Ronitt Rubinfield. I've seen "enough": incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, August 2017.

[109] Rene Rosenbaum, Jian Zhi, and Bernd Hamann. Progressive parallel coordinates. In *2012 IEEE Pacific Visualization Symposium*, pages 25–32, February 2012.

[110] Rpforest. `https://github.com/lyst/rpforest`. Last accessed: 2019-11-20.

[111] Hans-Jörg Schulz, Marco Angelini, Giuseppe Santucci, and Heidrun Schumann. An enhanced visualization process model for incremental visualization. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1, 2015.

[112] Thomas B. Sebastian and Benjamin B. Kimia. Metric-based shape retrieval in large databases. In *Object recognition supported by user interaction for service robots*, volume 3, pages 291–296. IEEE, 2002.

[113] Michael Sedlmair, Miriah Meyer, and Tamara Munzner. Design study methodology: reflections from the trenches and the stacks. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2431–2440, 2012.

[114] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16(3):265–285, September 1984.

[115] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, volume 10, pages 1–10, 2010.

[116] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[117] Nickolay Smirnov et al. Table for estimating the goodness of fit of empirical distributions. *Annals of Mathematical Statistics*, 19(2):279–281, 1948.

[118] Fabian Sperrle, Jürgen Bernard, Michael Sedlmair, Daniel Keim, and Mennatallah El-Assady. Speculative execution for guided visual analytics. *arXiv preprint arXiv:1908.02627*, 2019.

[119] Robert F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991.

[120] Charles D. Stolper, Adam Perer, and David Gotz. Progressive visual analytics: user-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, December 2014.

[121] The movies dataset. `https://www.kaggle.com/rounakbanik/the-movies-dataset`. Last accessed: 2019-11-20.

[122] Thanh N. Tran, Ron Wehrens, and Lutgarde MC Buydens. KNN-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis*, 51(2):513–525, 2006.

[123] Anne Treisman. Preattentive processing in vision. *Computer vision, graphics, and image processing*, 31(2):156–177, 1985.

[124] Cagatay Turkay, Erdem Kaya, Selim Balcisoy, and Helwig Hauser. Designing Progressive and Interactive Analytics Processes for High-Dimensional Data Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):131–140, January 2017.

[125] Typescript: javascript that scales. `https://www.typescriptlang.org/`. Last accessed: 2019-11-20.

[126] Pravin M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(1):101–115, 1989.

[127] Laurens Van Der Maaten. Barnes-Hut t-SNE. *arXiv preprint arXiv:1301.3342*, 2013.

[128] Vega datasets. https://github.com/vega/vega-datasets. Last accessed: 2019-11-20.

[129] Jules Vidal, Joseph Budin, and Julien Tierny. Progressive wasserstein barycenters of persistence diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[130] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1106–1113. IEEE, 2012.

[131] Zhe Wang, Nivan Ferreira, Youhao Wei, Aarthy Sankari Bhaskar, and Carlos Scheidegger. Gaussian cubes: real-time modeling for visual exploration of large multidimensional datasets. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):681–690, 2016.

[132] Colin Ware. *Information visualization: perception for design*. Elsevier, 2012.

[133] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[134] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2015.

[135] Yifan Wu, Larry Xu, Remco Chang, Joseph M. Hellerstein, and Eugene Wu. Making sense of asynchrony in interactive data visualizations. *CoRR*, abs/1806.01499, 2018. arXiv: 1806.01499.

[136] Jia Yu and Mohamed Sarwat. Accelerating spatial data visualization dashboards via a materialized sampling cube approach. In *Proceedings of the 2020 IEEE International Conference on Data Engineering*, to appear.

[137] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proc. of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Boston, MA. USENIX Association, 2010.

[138] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[139] Emanuel Zgraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. How progressive visualizations affect exploratory anal-

ysis. *IEEE Transactions on Visualization and Computer Graphics*, 23(8):1977–1987, 2017.

[140] Zheguang Zhao, Lorenzo De Stefani, Emanuel Zgraggen, Carsten Binnig, Eli Upfal, and Tim Kraska. Controlling false discoveries during interactive data exploration. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 527–540, Chicago, Illinois, USA. ACM, 2017.

국문 초록

현대 데이터 사이언스에서 인터랙티브한 시각화를 통해 데이터를 이해하는 것은 필수적인 분석 방법 중 하나이다. 그러나, 최근 데이터의 크기가 폭발적으로 증가하면서 데이터 크기로 인해 발생하는 지연 시간이 인터랙티브한 시각적 분석에 큰 걸림돌이 되었다. 본 연구에서는 이러한 확장성 문제를 해결하기 위해 점진적 시각적 분석(Progressive Visual Analytics)을 지원하는 일련의 시스템을 디자인하고 개발한다. 이러한 점진적 시각적 분석 시스템은 데이터 처리가 완전히 끝나지 않더라도 중간 분석 결과를 사용자에게 제공함으로써 데이터의 크기로 인해 발생하는 지연 시간 문제를 완화할 수 있다.

첫째로, 수십억 건의 행을 가지는 데이터를 시각적으로 탐색할 수 있는 Swift-Tuna 시스템을 제안한다. 데이터 처리 및 시각적 표현의 확장성을 목표로 개발된 이 시스템은, 약 40억 건의 행을 가진 데이터에 대한 시각화를 전처리 없이 수 초마다 업데이트할 수 있는 것으로 나타났다. 둘째로, 근사적 k-최근접점(Approximate k-Nearest Neighbor) 문제를 점진적으로 계산하는 PANENE 알고리즘을 제안한다. 근사적 k-최근접점 문제는 여러 기계 학습 기법에서 쓰임에도 불구하고 초기 계산 시간이 길어서 인터랙티브한 시스템에 적용하기 힘든 한계가 있었다. PANENE 알고리즘은 이러한 긴 초기 계산 시간을 획기적으로 개선하여 다양한 기계 학습 기법을 시각적 분석에 활용할 수 있도록 한다. 특히, 유용한 비선형적 차원 감소 기법인 t-분포 확률적 임베딩(t-Distributed Stochastic Neighbor Embedding)을 가속하여 수백 개의 차원을 가지는 데이터를 빠른 시간 내에 사영할 수 있다. 위의 두 시스템과 알고리즘이 데이터의 행 또는 열의 개수로 인한 확장성 문제를 해결하고자 했다면, 세 번째 시스템에서는 점진적 시각적 분석의 신뢰도 문제를 개선하고자 한다. 점진적 시각적 분석에서 사용자에게 주어지는 중간 계산 결과는 최종 결과의 근사치이므로 불확실성이 존재한다. 본 연구에서

는 세이프가드를 이용한 점진적 시각적 분석(Progressive Visual Analytics with Safeguards)이라는 새로운 개념을 제안한다. 이 개념은 사용자가 점진적 탐색에서 마주하는 불확실한 중간 지식에 세이프가드를 남길 수 있도록 하여 탐색에서 얻은 지식의 정확도를 추후 검증할 수 있도록 한다. 또한, 이러한 개념을 실제로 구현하여 탑재한 ProReveal 시스템을 소개한다. ProReveal를 이용한 사용자 실험에서 사용자들은 세이프가드를 성공적으로 만들 수 있었을 뿐만 아니라, 중간 지식의 불확실성을 다루기 위해 세이프가드를 자발적으로 이용한다는 것을 알 수 있었다. 마지막으로, 위 세 가지 연구의 결과를 종합하여 점진적 시각적 분석 시스템을 구현할 때의 디자인적 난제와 향후 연구 방향을 모색한다.

주요어: 정보 시각화; 인간-컴퓨터 상호작용; 시각적 분석; 점진적 시각적 분석; 빅 데이터; 대용량 데이터; 사용자 상호작용; 차원 축소; 사용자 평가
학번: 2014-21782

감사의 글

서울대학교에서 컴퓨터공학도로서 보낸 지난 10년은 저에게 이루 헤아릴 수 없을 만큼 많은 배움을 가져다 준 시간이었습니다. 이 동안 너무나 많은 분들이 저에게 영감과 용기와 가르침을 주셨습니다. 6년 간의 박사과정을 마무리하는 지금 이 자리를 빌어 소중한 분들께 그 동안 하지 못했던 감사의 말씀을 올리고자 합니다.

가장 먼저, 저의 지도교수이신 서진욱 교수님께 감사드립니다. 7년 전 겨울 방학 때 무턱대고 연구실에서 인턴을 하고 싶다고 찾아뵀는데 흔쾌히 받아주신 기억이 아직도 생생합니다. 이 때 교수님과 주고받은 이메일을 보면 지금도 감회가 새롭습니다. 교수님은 아직까지 화를 내시는 모습을 본 적이 없을 정도로 어떤 상황에도 저에게 진심 어린 조언을 먼저 해 주시는 분입니다. 잘하는 것 보다 적극성을 강조하시는 교수님의 가르침은 지식적인 측면뿐만 아니라 인격적인 측면에서도 저를 많이 성장시켰습니다. 교수님을 찾아뵐 때마다 학술적인 이야기뿐만 아니라 제 건강, 경제적인 상황, 인생의 고민 등 사적인 내용까지 터놓고 얘기할 정도로 제가 의지할 수 있는 분이셨습니다. 기회가 된다면 저도 교수님 같은 스승이 되고 싶습니다.

당신의 지도 학생이 아님에도 불구하고 서진욱 교수님 못지 않게 저를 도와주시고 가르쳐 주신 김보형 교수님께 감사의 말씀을 올리고 싶습니다. 그리고 회의 때마다 항상 창의적인 생각으로 저에게 영감을 주시는 이봉신 박사님께도 감사드립니다. 연구 주제를 발전시키는 시작 단계부터 논문을 다듬는 마무리 단계까지, 두 분의 정성 어린 지도 덕분에 연구를 더 잘 마무리할 수 있었습니다. 마지막으로, 제 박사학위논문의 심사 위원을 흔쾌히 수락해주시고 심사 때 다양한 관점에서의 의견을 개진해주신 김명수 교수님과 이영기 교수님께도 감사 인사 드립니다.

학위 과정 동안 수 많은 뛰어난 동료들을 만날 수 있었던 것은 저에게 큰 행운이었습니다. 관심 분야, 나이, 국적은 다르더라도 합심하여 연구하면서 많은 것들을 배우고 성장할 수 있었습니다. 지면의 한계로 모든 분을 언급할 수는 없지만 이 자리를 빌어 이 분들께 감사를 표하고 싶습니다.

연구자로서 미흡한 저의 연구에 시간을 내서 힘을 보태 주신 공저자 분들께 먼저 감사드리려고 합니다. 제가 인턴 시절 수행한 미숙한 연구임에도 불구하고 첫 InfoVis 논문을 낼 수 있게 도와주신 박종헌 교수님과 허재석 교수님께 감사드립니다. 여러 번 학회를 같이 다니면서 책에선 배울 수 없는 것들을 가르쳐 주시고 대학원생의 삶에 대해 아낌없이 조언해 주신 송현주 교수님께도 감사드립니다. 또, 항상 꼼꼼하게 연구를 검토해주는 믿을 수 있는 동료이자 재미있는 룸메이트였던 이세희 형에게 고마움을 표하고 싶습니다. 밤새서 연구 얘기를 하고 게임을 하던 것이 엊그제 같은데 이제는 두 분 다 결혼을 하셔서 그럴 수 없는 것이 아쉽습니다. 항상 성실하게 연구실 자리를 지켰던, 그리고 이제는 학생들을 가르치고 있으실 유승훈 박사님과 훌륭한 머신 러닝 연구자로 성장하고 있는 김원재에게도 감사를 전합니다. I would like to thank Jean-Daniel Fekete, who hosted me at AVIZ as a visiting Ph.D. student and introduced the world of progressiveness. I was also lucky to collaborate with Pierre Dragicevic and Frédéric Vernier, who gave me countless pieces of advice.

박사 과정 동안 사는 곳은 많이 옮겼지만 연구실은 제가 한결 같이 일하고, 먹고, 쉴 수 있는 공간이었습니다. 그리고 이 공간에서 저와 함께 6년이 넘는 시간 동안 생활한 연구실 동료들에게 감사드립니다. 동료들은 연구를 위한 제안서 작업부터 국제 학회를 개최하는 것 까지 정말 다양한 종류의 프로젝트를 함께 하면서 역경을 함께 헤쳐나간 그래서 힘들 때 의지할 수 있는 존재였습니다. 먼저, 연구실에 입학했을 때 부터 저에게 많은 조언을 해주신 Kyle Koh 형과 항상 모범이 되는 이제는 그 누구보다도 컴퓨터공학도 김영호 형에게 감사를 드립니다. 매사에 꼼꼼함을 본받고 싶은 이용석 형, 주제에 대한 통찰이 남다른 한구현 형, 주어진 일을 빈틈 없이 수행하는 신동화 형, 항상 참신한 VR 애플리케이션을 보여주시는 채

한주 형, 노련한 모습이 돋보이는 민구봉 형, 가정과 학업을 병행하시는 모습이 인상적인 김영택 형, 그리고 입학 동기이자 열정적인 장유리에게도 감사를 표합니다. 연구실의 막내였던 것이 얼마 안된 것 같은데 어느새 고참이 된 부족한 저를 잘 따라와주는 후배들도 고맙습니다. 이제는 어엿한 랩장이 된 복진욱, 발표하는 모습이 멋진 남자 김재영, 개발에 대해 남다른 열정을 가진 김준회, 하나의 연구 논문을 석사 과정이라는 짧은 기간동안 완성해낸 김민지, 엉뚱하지만 신선한 자극을 주는 최길웅, 연구실 맨 앞자리에서 항상 열심히 공부하는 융, 묵묵히 할일을 수행하는 박관모, 차분하고 의젓한 정석원, 그리고 장래가 촉망되는 두 신입생 고형권과 안단태에게 고마움을 표하고 싶습니다. 또한, 이러한 좋은 연구실 분위기를 만들어 물려주시고 동참해주신 선배님들께도 감사를 전합니다.

이제는 어엿한 사회인이 되어 각자의 소임을 다하고 있는 친구들에게도 고마움을 표하고 싶습니다. 박사 과정 기간동안 제가 웃음을 잃지 않게 해준 유능하고 유머러스한 컴퓨터공학부 09학번 동기들이 있는 것이 고맙고 자랑스럽습니다. 비록 자주 만나지는 못하지만 아직까지 끈끈한 그래서 저에게는 더 각별한 사람들입니다.

마지막으로, 저의 든든한 후원자인 사랑하는 가족들에게 가장 큰 감사를 올립니다. 엄하지만 따뜻하신 아버지와 지혜로운 어머니의 한결같은 지원과 교육 덕분에 제가 이 자리까지 올 수 있었습니다. 제가 어떤 결정을 하든 존중해 주신 덕분에 제가 자주적으로 인생을 살아가고 있습니다. 항상 그리고 앞으로도 저보다 어릴 동생이지만 누구보다도 든든한 윤겸에게도 고마움을 표하고 싶습니다.

지면의 한계상 여기서 미처 감사를 표하지 못한 분들께도 앞으로 살아가면서 감사를 전할 기회가 있으리라 믿습니다. 감사합니다.

2020년 1월

조재민 올림