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

Jaemin Jo, Jinwook Seo, Jean-Daniel Fekete

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

Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete, *Senior Member, IEEE*

Abstract—We present PANENE, 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 interactive 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 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.

Index Terms—Approximate *k*-Nearest Neighbors, Progressive Data Analysis, Algorithm, Real-Time

1 Introduction

Progressive data analysis has recently gained in popularity due to its ability to deliver ongoing results before the whole computation is completed [1], [2]. 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 article, we address one important problem: progressively finding *k*-nearest neighbors of a given point in a multidimensional space, i.e., the *k*-nearest neighbor problem. We present a novel progressive algorithm, PANENE, for Progressive Approximate k-NEarest NEighbors, broadening the boundary of progressive visual analytics. 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 [3], into interactive 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 [4], classification [5], embedding [6], and non-parametric density estimation [7]. 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 structure 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 [8]. 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) [9], 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

- *Jaemin Jo is with Seoul National University, Korea, E-mail: jmjo@hcil.snu.ac.kr*
- *Jinwook Seo is with Seoul National University, Korea, E-mail: jseo@snu.ac.kr*
- *Jean-Daniel Fekete is with Inria, France, E-mail: Jean-Daniel.Fekete@inria.fr*

Manuscript received April 19, 2005; revised August 26, 2015.

a precomputation leads to long loading time, hampering the interactivity of the entire system. Only few AKNN techniques, such as FLANN [10], support *online* updates; they allow inserting new points even after an index is built. However, this is not sufficient for interactive 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 [11].

In this paper, we present a *progressive* algorithm for indexing and querying approximate k-nearest neighbors. Our algorithm computes k-nearest neighbors iteratively with each iteration finishing in a given number of operations (i.e., progressively). The contributions of this article 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 [3].

This paper 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 3, 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, 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 5) and present applications in interactive analysis (Section 6). Finally, we discuss the limitations of this work and future work.

2 Related Work

In this section, we first introduce previous approaches to the k-nearest neighbor problem in parallel with progressive systems for interactive analysis of large-scale data. Then, we present the challenges and opportunities in designing a progressive algorithm for the k-nearest neighbor problem.

2.1 The k-Nearest Neighbor 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 [12], 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 [13]. At each level in the tree, data is divided into two groups along the dimension in which the data has the highest variance. 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" [14]. 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 [5] and OpenCV [15]. 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 [16] 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 [17] 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 [18] 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 [10].

Hash-based techniques use a set of *locality-sensitive hashing (LSH) functions* [19]. 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 [20], multi-probe LSH [21], kernelized LSH [22], and circular random variable-based matchers [23].

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 [24] 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. [25] provided theoretical guarantees for the accuracy and the computational complexity of such a greedy method. Recently, more sophisticated graph structures such as navigable small world graphs are used for KNN queries. In addition to short-range links in a traditional neighbor graph, navigable small world graphs have long-range links that connect two distant points. Malkov et al. [14] showed that these long-range links can be used for logarithmic scaling of neighbor exploration. 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, 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 [8], [26]. 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 paper, inspired by Progressive Visual Analytics [2], 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 [18].

2.2 Progressive Systems

The latency of interactive systems has become a primary concern for visualization researchers and practitioners. As the size of datasets increases and analytic methods become more sophisticated, delivering results within a time limit becomes a new challenge in designing visual exploration systems. Reducing the latency is an essential problem in visual analytics, because long latency not only delays analysis but also outpaces humans' ability to focus their attention, eventually degrading the quality of the analysis [27]. Nielsen [11] distinguishes 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 been also suggested by other researchers [28], [29]. However, sequential systems (i.e., systems that hang until the entire computation is done) cannot be guaranteed to comply with such time constraints, leading analysts to wait for a response without bounds and hurting their analytical capabilities.

Initially introduced by Stolper et al. [2], the Progressive Visual Analytics (PVA) paradigm considers humans' perceptual and cognitive constraints as the primary concern in system designs. PVA systems deliver the partial results of computation on the fly without blocking analysis until the complete result becomes available. It is akin to the online computation paradigm [30] in that the user can grasp meaningful partial results before the whole computation is finished. PVA has significant benefits: it guarantees that the latency to have a new partial result will be bounded in time, meaning that analysts can expect to receive the new result within the human attention span.

The benefits of PVA have been advocated through the user experiments in the HCI field. For example, Fisher et al. [31] investigated how the user understands and interacts with incremental approximate visualizations. Their result suggested that users were capable of using incremental visualizations to make their decisions faster. More recently, Zgraggen et al. [27] compared progressive visualizations to blocking and instantaneous visualizations in terms of the insights that the user generated. They found progressive visualizations outperformed blocking visualizations, even giving comparable performance to ideal instantaneous visualizations. Similar results can be elicited from Badam et al.'s user study [32], where a progressive interface, InsightsFeed, and its instantaneous version showed similar performance in terms of answer accuracy and user preferences.

Despite the benefits and established grounds of progressive computation, it is not always easy or even possible to convert a sequential algorithm into a progressive one. A vast body of research has attempted to provide the progressive version of sequential algorithms, widening the boundary of progressive visual analytics. The pioneering work of Stolper et al. [2] provided a progressive implementation of the SPAM algorithm [33] for extracting common patterns in event sequences. Pezzotti et al. [6] presented approximate t-Distributed Stochastic Neighbor Embedding (A-tSNE) which can be used to progressively visualize multidimensional data in 2D spaces. Similarly, Turkay et al. [34] proposed DimXplorer, which enabled interactive exploration using the incremental PCA and mini-batch k-clustering algorithms. Fekete and Primet implemented the Progressivis Toolkit [1] to provide a general and systematic platform for progressive implementations; we rely on the data structures and mechanisms they describe. Finally, Mühlbacher et al. [35] characterized strategies for increasing user involvement in existing sequential algorithms.

In this paper, we focus on one important yet underexplored problem of developing a progressive algorithm for the k-nearest neighbor problem. To our best knowledge, no previous research has studied this problem in spite of its wide use in data analysis.

3 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 [18] and 2) an online version of the algorithm is available as open-source [36], so we could directly compare our progressive version to the online version.

3.1 A Sequential Algorithm

A k-d tree is a binary tree built by recursively partitioning a multidimensional space using axis-aligned hyperplanes [12] 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 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
```

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.

3.2 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 [10]. 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.

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.

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
```

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.

3.3 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

$$c(T) = \sum_{p \in T} \phi_p \times \mathrm{depth}(p) \tag{1}$$

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 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.

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
```

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 quantum 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 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.

3.4 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 [37]: 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.

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 ← dataSource.loadNewPoints(ops)
3:   trees ← a new k-d tree forest built with initPoints
4:   updating ← false
5:   loss ← 0
6:   N ← 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 ← τ × ops
16:     updateOps ← (1 − τ) × ops
17:   else
18:     insertionOps ← ops
19:     updateOps ← 0
20:   end if
21:   newPoints ← dataSource.loadNewPoints(insertionOps)
22:   for p in newPoints do
23:     N ← 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 > α × N log2 N then        ▷ loss is accumulated by a search procedure
30:     updating ← true
31:     loss ← 0
32:     newTree←InitializeBuild(dataSource)
33:   end if
34:   if updateOps > 0 then
35:     done ← 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
```

4 *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* [38]. 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) [6] 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 approximation in both k-d trees and dirtiness propagation. In Section 5.2, 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 update queue and updated if dirty. Algorithm 5 shows the complete algorithm for progressive KNN lookup tables.

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
```

5 BENCHMARK

Online algorithms are usually evaluated using competitive analysis [39]: 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. [40] 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.

5.1 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 [41] 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 100 isotropic Gaussian blobs, 10,000 points for each. For reproducibility, the Blob dataset was generated through scikit-learn's blob generator [5]. 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).

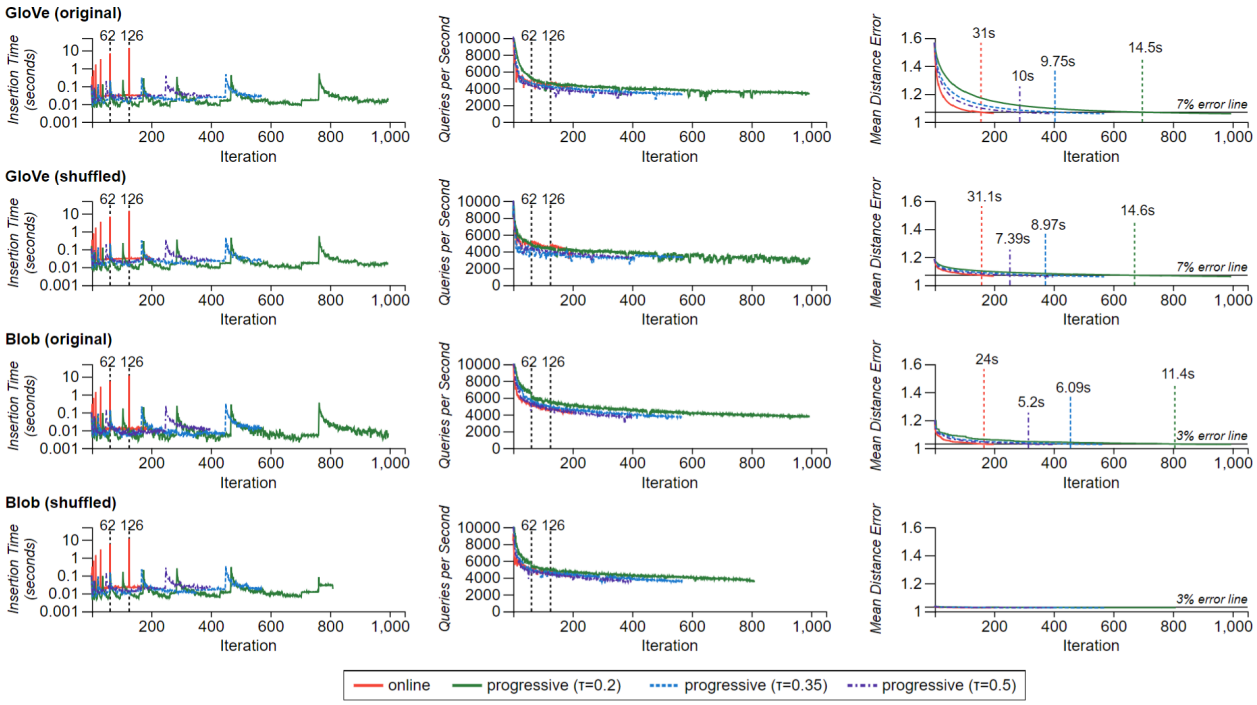

Fig. 1: **Benchmark results of online and progressive k-d trees** according to datasets (*GloVe* [41] and *Blob*) and ordering conditions (*original* and *shuffled*). FLANN's online k-d tree (the red line) rebuilt the k-d trees each time the data size doubles, producing insertion times longer than 10 seconds (e.g., leftmost charts at the 62nd and 126th iterations). In contrast, progressive k-d trees (the green, blue, and purple lines) alleviated the spikes in insertion time by spreading the reconstruction load. Our progressive k-d trees yield a small loss in queries per second (middle column), yet with faster convergence to a mean distance error of 7% for the GloVe dataset and 3% for the Blob dataset (rightmost column).

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* [10], 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 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 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 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 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.

5.2 *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 2 shows the result of the second benchmark. Overall, the insertion time of progressive KNN lookup tables increased compared to that of a 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.

6 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 [3], adding them to the toolbox of interactive analysis tools.

6.1 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:

$$y_{new} = \sum_{p \in KNN_K(x_{new})} w_p y_p \tag{2}$$

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 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 $\rho(p)$:

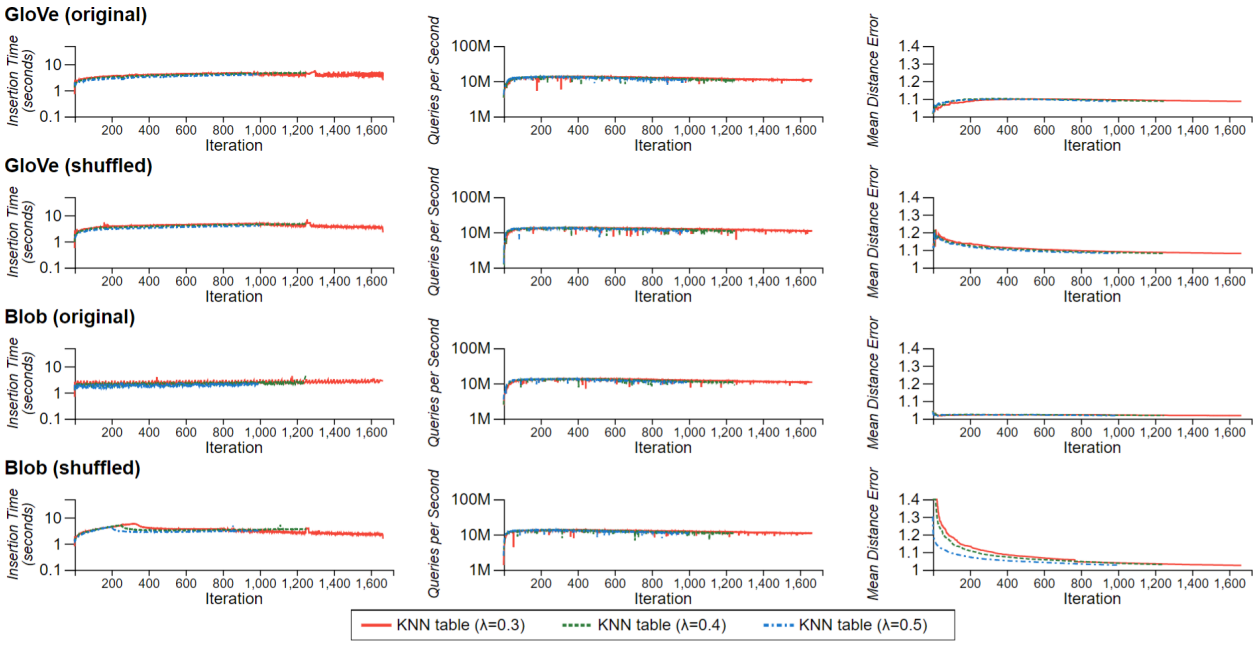

Fig. 2: **Benchmark results of *k*-nearest neighbor lookup tables** according to datasets (*GloVe* [41] and *Blob*) and ordering conditions (*original* and *shuffled*). Designed for repeated KNN queries for points in training data, KNN lookup tables can answer a KNN query in constant time (i.e., one table lookup operation).

$$\rho(p) \propto \sum_{x \in X} e^{-\frac{(p-x)^2}{2h^2}} \qquad \hat{\rho}_K(p) \propto \sum_{x \in KNN_K(p)} e^{-\frac{(p-x)^2}{2h^2}} \tag{3}$$

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 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 [42] 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.

6.2 Responsive t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) [3] 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) [43] reduces the complexity to $O(N \log N)$ by applying the Barnes-Hut approximation [44] 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 [11].

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 distance 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 the related work section 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 [45] 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 [46] 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 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 [3]. 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 [3]. 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 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.

7 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 [1]. PANENE relies on the OpenMP library [47] to process multiple queries in parallel, and on the Roaring Bitmaps library [37] to support efficient filtering. PANENE, the parameters for benchmarks, and applications presented in this paper are available at `github.com/e-/PANENE` under the BSD 2-clause "Simplified" License.

8 Discussion

In this work, 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) [11], 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 [1], 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 background 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, in this work, we assumed that our algorithms run on a single thread. However, the only constraint we

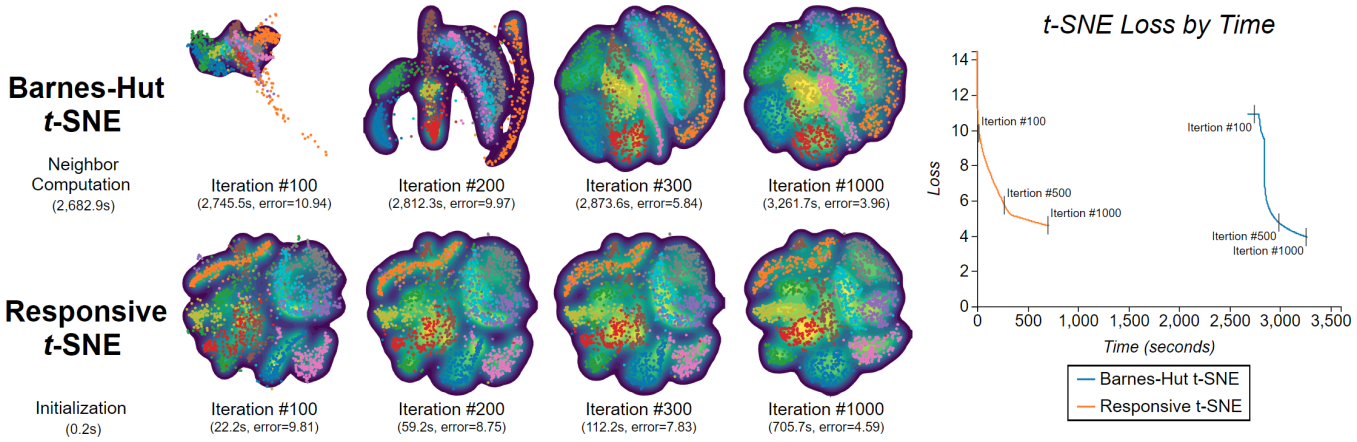

Fig. 3: **Embedding of the MNIST dataset using Barnes-Hut t-SNE and Responsive t-SNE** The Barnes-Hut t-SNE algorithm took about 45 minutes to precompute the nearest neighbors of data points, restricting interactive analysis. In contrast, our responsive t-SNE produced the initial result in a few seconds by computing the nearest neighbors progressively and running the optimization loop of the t-SNE algorithm in an alternate manner. The Barnes-Hut t-SNE took 54.4 minutes to run 1,000 iterations, giving a loss of 3.96, while the responsive t-SNE took 11.8 minutes and yielded a loss of 4.59. Each circle in the scatterplots represents a handwritten letter (28×28 pixels) with a color encoding its digit (0 to 9). Both algorithms were run on a machine equipped with Intel Core i7-7700K CPU (4.2GHz) and 16GB of main memory.

have with the model we rely on [1] 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.

9 Conclusion and Future Work

Although k-nearest neighbor computation is common in data mining algorithms, the long computation time has hindered its application for interactive visual analytics. In this article, we presented 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, triggering 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.

For future work, we can further improve algorithms with tighter real-time constraints. We assumed the costs of an insertion task and a reconstruction task were identical, but we can weight one over the other to estimate the running time more accurately. It would also be interesting to improve other methods for KNN search, such as locality-sensitive hashing (LSH), to be progressive and compare their performance with ours. Finally, a more general and systematic approach to determining the number of operations in one iteration (i.e., *ops*) would benefit future progressive systems.

Acknowledgments

Thanks to Adeline Pierrot for her help in calculating the imbalance of k-d trees. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2016R1A2B2007153).

References

[1] J.-D. Fekete and R. Primet, "Progressive analytics: A computation paradigm for exploratory data analysis," *ArXiv e-prints*, July 2016. [Online]. Available: http://arxiv.org/abs/1607.05162
[2] C. D. Stolper, A. Perer, and D. Gotz, "Progressive visual analytics: User-driven visual exploration of in-progress analytics," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 1653–1662, Dec 2014.
[3] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
[4] T. N. Tran, R. Wehrens, and L. M. Buydens, "KNN-kernel density-based clustering for high-dimensional multivariate data," *Computational Statistics & Data Analysis*, vol. 51, no. 2, pp. 513–525, 2006.
[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[6] N. Pezzotti, B. P. F. Lelieveldt, L. van der Maaten, T. Höllt, E. Eisemann, and A. Vilanova, "Approximated and user steerable tsne for progressive visual analytics," *CoRR*, vol. abs/1512.01655, 2015.
[7] K. Fukunaga, *Introduction to statistical pattern recognition*. Academic press, 2013.
[8] E. Bernhardsson, "Benchmarking nearest neighbors," https://github.com/erikbern/ann-benchmarks, last accessed: 2018-01-22.
[9] Y. A. Malkov and D. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *arXiv preprint arXiv:1603.09320*, 2016.
[10] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.
[11] J. Nielsen, *Usability engineering*. Elsevier, 1994.
[12] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[13] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.
[14] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.
[15] G. Bradski, "OpenCV," *Dr. Dobb's Journal of Software Tools*, 2000.
[16] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE, 1997, pp. 1000–1006.
[17] C. Silpa-Anan and R. Hartley, "Optimised KD-trees for fast image descriptor matching," in *IEEE Conference on Computer Vision and Pattern Recognition*, June 2008, pp. 1–8.
[18] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.
[19] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 459–468.
[20] M. Bawa, T. Condie, and P. Ganesan, "LSH Forest: self-tuning indexes for similarity search," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 651–660.
[21] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 950–961.
[22] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 2009, pp. 2130–2137.
[23] F. Alhwarin, A. Ferrein, and I. Scholl, "Crvm: Circular random variable-based matcher," 2018.
[24] T. B. Sebastian and B. B. Kimia, "Metric-based shape retrieval in large databases," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3. IEEE, 2002, pp. 291–296.
[25] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, 2011, p. 1312.
[26] "MRPT performance comparison," https://github.com/ejaasaari/mrpt-comparison, last accessed: 2018-01-22.
[27] E. Zgraggen, A. Galakatos, A. Crotty, J.-D. Fekete, and T. Kraska, "How progressive visualizations affect exploratory analysis," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 8, pp. 1977–1987, 2017.
[28] R. B. Miller, "Response time in man-computer conversational transactions," in *Proc. of the Fall Joint Computer Conference, Part I*. ACM, 1968, pp. 267–277.
[29] B. Shneiderman, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 265–285, Sep. 1984.
[30] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
[31] D. Fisher, I. Popov, S. Drucker, and M. Schraefel, "Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster," in *CHI '12*, 2012, pp. 1673–1682.
[32] S. K. Badam, N. Elmqvist, and J.-D. Fekete, "Steering the Craft: UI Elements and Visualizations for Supporting Progressive Visual Analytics," *Computer Graphics Forum*, vol. 36, no. 3, pp. 491–502, 2017.
[33] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 429–435.
[34] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser, "Designing Progressive and Interactive Analytics Processes for High-Dimensional Data Analysis," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 131–140, Jan 2017.
[35] T. Mühlbacher, H. Piringer, S. Gratzl, M. Sedlmair, and M. Streit, "Opening the black box: Strategies for increased user involvement in existing algorithm implementations," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 1643–1652, Dec 2014.
[36] M. Muja, "FLANN - Fast Library for Approximate Nearest Neighbors," https://github.com/mariusmuja/flann, last accessed: 2018-01-22.
[37] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Software: practice and experience*, vol. 46, no. 5, pp. 709–719, 2016.
[38] P. M. Vaidya, "An $O(n \log n)$ algorithm for the all-nearest-neighbors problem," *Discrete & Computational Geometry*, vol. 4, no. 1, pp. 101–115, 1989.
[39] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
[40] P. Eichmann, E. Zgraggen, Z. Zhao, C. Binnig, and T. Kraska, "Towards a benchmark for interactive data exploration," *IEEE Data Eng. Bull.*, To Appear.
[41] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162
[42] Lorensen, William E. and Cline, Harvey E., "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. [Online]. Available: http://doi.acm.org/10.1145/37401.37422
[43] L. Van Der Maaten, "Barnes-Hut t-SNE," *arXiv preprint arXiv:1301.3342*, 2013.
[44] J. Barnes and P. Hut, "A hierarchical $o(n \log N)$ force-calculation algorithm," *nature*, vol. 324, no. 6096, p. 446, 1986.
[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[46] "Barnes-Hut t-SNE," https://github.com/lvdmaaten/bhtsne, last accessed: 2018-01-22.
[47] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

Jaemin Jo Jaemin Jo is a Ph.D. student in the Department of Computer Science and Engineering, Seoul National University, Seoul, Korea. His research interests include human-computer interaction and large-scale data visualization. He received the B.S. degree in computer science and engineering from Seoul National University, Seoul, Korea in 2014.

Jinwook Seo Jinwook Seo is a professor in the Department of Computer Science and Engineering, Seoul National University, where he is also the Director of the Human-Computer Interaction Laboratory. His research interests include HCI, information visualization, and biomedical informatics. He received his Ph.D. in computer science from the University of Maryland at College Park in 2005.

Jean-Daniel Fekete Jean-Daniel Fekete is the Scientific Leader of the INRIA Project Team AVIZ that he founded in 2007. He received his PhD in Computer Science in 1996 from University of Paris Sud, France, joined INRIA in 2002 as a confirmed researcher, and became Senior Research Scientist in 2006. His main research areas are Visual Analytics, Information Visualization and Human Computer Interaction. He is a Senior Member of IEEE.