CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Dynamic Kernel Density Estimation using Approximate Nearest Neighbor Search

A Dynamic Extension of the DEANN Algorithm

Master's thesis in Computer science and engineering

GUSTAF ALGESKOG
ZACKARIAS LEESMENT

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Dynamic Kernel Density Estimation using Approximate Nearest Neighbor Search

A Dynamic Extension of the DEANN Algorithm

GUSTAF ALGESKOG
ZACKARIAS LEESMENT

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Dynamic Kernel Density Estimation using Approximate Nearest Neighbor Search
A Dynamic Extension of the DEANN Algorithm
Gustaf Algeskog & Zackarias Leesment

Supervisor: Matti Karppa, Data Science and AI
Examiner: Matti Karppa, Data Science and AI

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Dynamic Kernel Density Estimation using Approximate Nearest Neighbor Search
A Dynamic Extension of the DEANN Algorithm
Gustaf Algeskog & Zackarias Leesment
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Kernel Density Estimation (KDE) is a nonparametric method for estimating an unknown density from observed data. Exact KDE evaluates every data point for each query, which becomes expensive for large datasets. DEANN reduces this cost in the static setting by combining approximate nearest neighbor search with random sampling, but assumes that the dataset is fixed after preprocessing. This thesis extends DEANN to a dynamic setting where points can be inserted and deleted. The implemented Dynamic ANN Estimator (DAE) maintains a dynamic ANN component together with a dynamic random-sampling structure, allowing KDE queries to be evaluated against the current active dataset. The implementation is evaluated against an exact dynamic naïve KDE baseline on several benchmark datasets. The results show that DAE can reduce query time once the active dataset is sufficiently large. However, this comes at the cost of substantially higher construction and update time due to ANN maintenance. Experiments with longer update sequences indicate that query time and approximation error remain stable over repeated updates. These results suggest that dynamic DEANN is most suitable for large, query-heavy dynamic KDE workloads where the maintained estimator can be reused for many queries.

Keywords: Kernel density estimation, dynamic kernel density estimation, approximate nearest neighbor search, random sampling, DEANN

Acknowledgements

We would like to thank Matti Karppa, our supervisor and examiner, for introducing us to an interesting thesis project and for his patient and pedagogical guidance throughout the work. His explanations, feedback, and support were valuable at every stage of the project.

We would also like to thank Martin Aumüller for his helpful input, enthusiasm, and encouragement during the project.

Gustaf Algeskog & Zackarias Leesment, Gothenburg, 2026-06-14

Contents

List of Figures

List of Tables

1

Introduction

Kernel Density Estimation (KDE) is a nonparametric method for obtaining a smooth estimate of an unknown probability density function from observed data [1]. The basic idea is to place a smooth kernel, which is a probability distribution such as a Gaussian, on each data point and to average their contributions to obtain a continuous density estimate. The spread of each kernel is controlled by a bandwidth parameter, which determines how broadly each data point influences the estimate. Figure 1.1 illustrates this construction in one dimension, where each sample contributes a smooth bump and the resulting density is their aggregate.

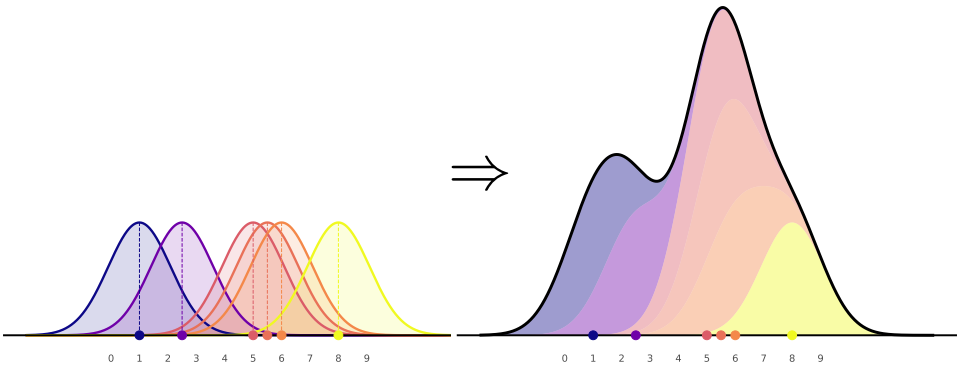

Figure 1.1: One-dimensional kernel density estimation. Each sample contributes a localized Gaussian bump (left), and the KDE is formed by summing and normalizing these contributions to obtain a smooth estimate of the underlying probability density (right).

Although the full mathematical formulation is deferred to Section 2.1, the estimator can be understood as an average of contributions from all data points. For a dataset X and a query point q, the estimate takes the form

$$\mathrm{KDE}_X(q) = \frac{1}{|X|} \sum_{x \in X} K_h(x, q),$$

where $\mathrm{KDE}_X(q)$ denotes the kernel density estimate at q computed from the dataset X, and the factor $\frac{1}{|X|}$ corresponds to averaging over all data points, while $K_h(x, q)$ measures the contribution of a data point x to this estimate. Intuitively, points closer

to q contribute more, while distant points contribute less. A high value of $\mathrm{KDE}_X(q)$ therefore indicates that many data points lie close to q, corresponding to a region of high data density, whereas a low value indicates that q lies in a sparse region.

This interpretation makes KDE particularly useful in settings where the structure of the underlying distribution is unknown or complex, for example when it is multimodal, that is, when it contains multiple local maxima. In contrast, many traditional approaches assume that the data follow a specific distribution, such as a Gaussian distribution, which can be limiting when the true distribution does not match such a simple shape. KDE instead constructs the density estimate directly from the data, without requiring such assumptions.

Consequently, KDE is applicable in a wide range of settings where understanding the shape of an unknown distribution is essential. One application of KDE is in environmental data analysis, for example in the examination of river-flow variability, where it provides smoother and more informative summaries than histograms [2]. In machine learning, KDE has been applied to tasks such as multiclass quantification, where modeling the distribution of classifier outputs enables accurate estimation of class prevalences under label shift [3]. In general, this shows that KDE is effective in situations where the underlying data may be complex or high-dimensional.

To better understand how these contributions are computed in practice, it is useful to consider how a kernel compares a data point with a query point. Depending on the application, closeness can be described by a distance measure, where smaller values indicate more similar points, or by a similarity measure, where larger values indicate more similar points.

The kernel function then converts this notion of closeness into a contribution to the density estimate. A Gaussian-type kernel, for example, decreases exponentially with distance, so that nearby points contribute strongly while distant points contribute very little:

$$K_h(x, y) = \exp\left(-\frac{\|x - y\|_2^2}{2h^2}\right).$$

Here, $\|x - y\|_2$ denotes the Euclidean distance between x and y, and the bandwidth h controls how quickly the contribution decreases as the distance increases.

However, computing KDE exactly is computationally expensive for large datasets or high-dimensional data. Evaluating the density at a single query point requires comparing that query point to all data points, which leads to a computational cost that scales linearly with the dataset size. This makes exact evaluation impractical in many real-world applications.

To reduce this computational cost, a variety of approaches have been proposed to accelerate KDE. These include methods such as *Approximate Skeletonization Kernel-Independent Treecode in High Dimensions* (ASKIT) and approaches based on *Locality Sensitive Hashing* (LSH), as well as other approximation techniques designed to reduce the computational cost [4, 5].

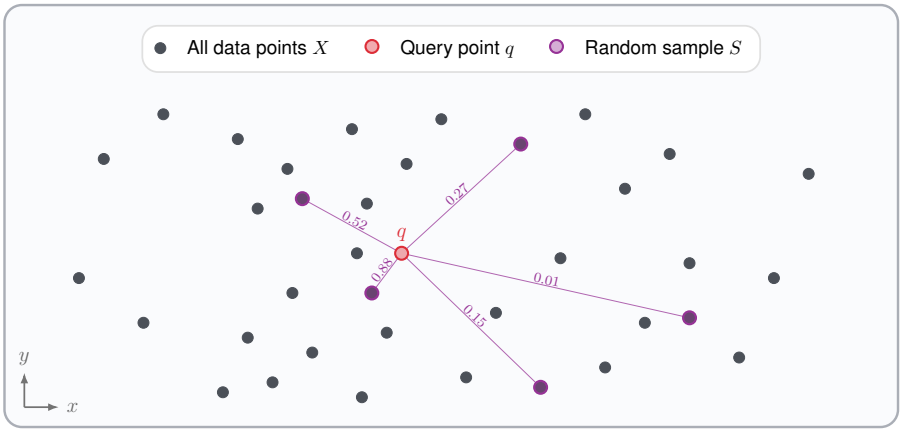

Figure 1.2: Random sampling for a KDE query. The purple points are selected uniformly at random from the dataset. The line segments indicate the distances to the query point q, which determine the KDE contributions of the sampled points.

Another approximation strategy is random sampling. Instead of evaluating the kernel contribution of every point in the dataset, the KDE value is estimated from a uniformly selected subset of points [6]. This reduces the number of kernel evaluations. The resulting estimator is unbiased, meaning that its expected value equals the full KDE value. Equivalently, over repeated independent samples, the average of the estimates converges to the full KDE value.

As illustrated in Figure 1.2, the sampled points are chosen uniformly at random, but their contributions to the KDE estimate are not equal. Nearby sampled points contribute more strongly, while distant sampled points contribute less. For this reason, an estimate can be inaccurate if the random sample misses points close to the query. In that case, the sampled points may contribute only weakly to the estimate, even though the full dataset contains nearby points with large contributions. This can happen even though the estimator is unbiased, since unbiasedness describes the behavior of the estimator averaged over repeated independent samples, not the accuracy of each individual estimate.

One way to reduce this variance is to increase the number of sampled points. However, increasing the sample size also increases the query cost, and in the limit this approaches exact KDE evaluation. This motivates combining random sampling with a method that explicitly identifies points likely to have large kernel contributions. Since nearby points often contribute most to the KDE value, approximate nearest neighbor search provides a natural way to find such points efficiently.

Density Estimation from Approximate Nearest Neighbors (DEANN) builds on this idea by combining Approximate Nearest Neighbor (ANN) search with random sampling. The ANN component is used to identify points close to the query, whose KDE contributions are evaluated directly. The contribution of the remaining points is then estimated using random sampling. This allows DEANN to focus computation on high-contribution points while still accounting for the rest of the dataset, resulting in efficient and unbiased estimators [7]. The structure of the method is illustrated in Figure 1.3.

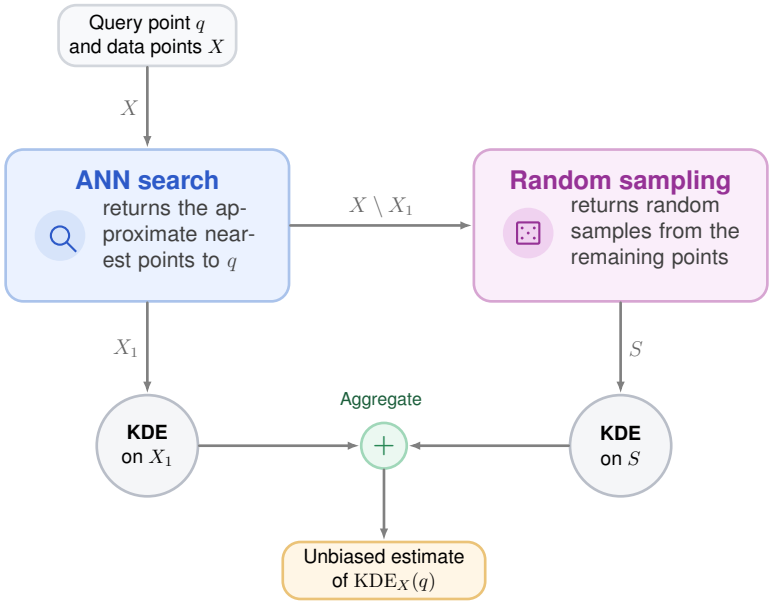

Figure 1.3: Conceptual structure of DEANN. For a query point q, an ANN search over the dataset X identifies a subset X_1 of approximate nearest neighbors, whose KDE contribution is evaluated directly. The remaining points $X \setminus X_1$ are approximated by KDE evaluation on a uniform random sample S. The two contributions are then combined with the appropriate weights to obtain an unbiased estimate of $\mathrm{KDE}_X(q)$.

Given a query point q, the goal is to estimate the density value $\mathrm{KDE}_X(q)$ at that specific point. The estimator returns a single scalar value for each query point, representing the estimated density at that point. To obtain a representation of the density function over a region, this evaluation can be repeated at multiple query points.

To evaluate this estimate efficiently for a given query, DEANN partitions the dataset into two disjoint subsets. An approximate nearest neighbor search identifies a subset of points close to the query, and their KDE contribution is evaluated directly. The contribution of the remaining points is estimated by evaluating the KDE on a uniform random sample drawn from this remainder. The final estimate is obtained by combining the two contributions with weights proportional to the sizes of the two subsets. This allows the method to approximate the density without evaluating all data points, while remaining unbiased. In contrast, a biased estimator tends to consistently over- or under-estimate the true value.

Most existing approaches to accelerating KDE, including DEANN, assume a *static* dataset, where the data are fixed after preprocessing and no insertions or deletions occur [4, 5, 7]. In *dynamic* settings, the data instead arrive as a stream of points or change over time through additions or removals, which requires the data structure to support updates. As described above, DEANN relies on a partition of the dataset into two subsets. Each data point must belong to exactly one of these subsets, and this partition must therefore be maintained under insertions and deletions without duplication or omission.

The dynamic extension is studied through an implementation and an empirical evaluation. The implementation maintains the DEANN query structure under insertions and deletions by combining a dynamic ANN component with a dynamic random-sampling structure. The experiments evaluate query time, construction time, update cost, approximation error, and stability under sliding-window updates. The results show that the dynamic extension can reduce query time compared with an exact dynamic baseline once the active dataset is sufficiently large, but that this comes at the cost of substantially higher construction and update time. They also show that query time and approximation error remain stable over the tested sliding-window update sequences.

1.1 Contributions

This thesis extends the DEANN framework to dynamic kernel density estimation under insertions and deletions of data points.

The implemented estimator, referred to as the Dynamic ANN Estimator (DAE), combines a dynamic approximate nearest neighbor (ANN) structure, treated as a *black-box* component, with a dynamic random sampling scheme implemented as part of this work. The estimator preserves the unbiasedness property of DEANN while allowing the active dataset to change over time.

The implementation is evaluated empirically against a dynamic naïve KDE baseline developed as part of this work. The evaluation measures approximation error relative to the exact baseline and compares construction time, query time, insertion time, deletion time, and stability under repeated updates.

1.2 Limitations

The scope of the work is limited to one prototype implementation and one dynamic ANN backend. The implementation is intended for controlled experimental evaluation rather than deployment as a general-purpose KDE system. All experiments are conducted on the Minerva computing cluster at Chalmers [8].

The thesis does not compare multiple dynamic ANN backends, study alternative KDE formulations, or provide new formal theoretical guarantees. It also does not include extensive parameter tuning, distributed execution, or real-time deployment.

2

Theory

Kernel density estimation was previously introduced as an average of local contributions from the data, forming a smooth approximation of an unknown distribution. This interpretation can be made precise by defining the estimator formally.

2.1 Kernel Density Estimation

Let $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ be a dataset consisting of independent and identically distributed samples drawn from an unknown probability density function f. Here, f denotes the true underlying density, while $\hat{f}_h$ denotes an estimate constructed from the observed data. Given a kernel function $K : \mathbb{R}^d \to \mathbb{R}$ satisfying

$$\int_{\mathbb{R}^d} K(u)\,\mathrm{d}u = 1$$

and a bandwidth $h > 0$, the kernel density estimator is defined as

$$\hat{f}_h(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big).$$

The condition on K ensures that $\hat{f}_h$ integrates to one, so that it defines a valid probability density function [1]. Equivalently, and following the notation used by DEANN [7], this estimator can be written in the form

$$\mathrm{KDE}_X(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x_i, x),$$

where

$$K_h(x, y) = \frac{1}{h^d} K\Big(\frac{x - y}{h}\Big).$$

In this form, the estimate is expressed directly as an average of contributions from all data points.

The bandwidth h controls the smoothing of the estimator. Decreasing h results in a more variable estimate, whereas increasing h produces a smoother estimate with greater bias. This reflects a trade-off between variance and bias, where small bandwidths capture fine structure in the data while large bandwidths emphasize overall trends. Examples of both undersmoothing and oversmoothing are shown in Figure 2.1.

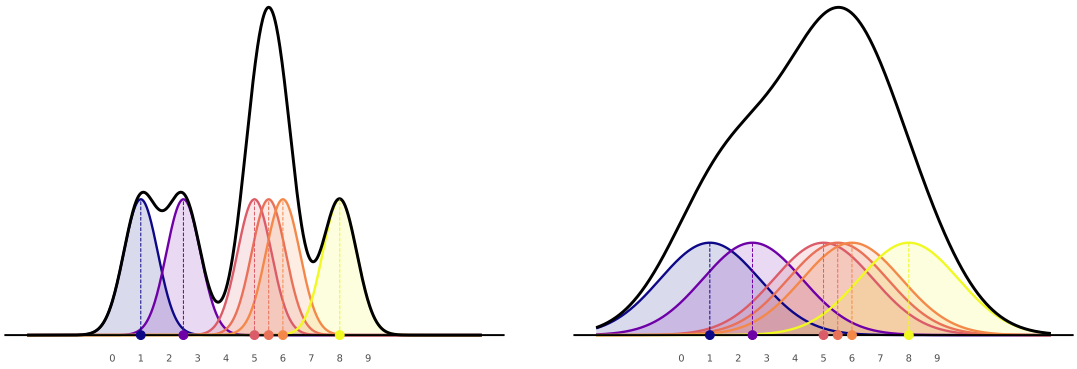

Figure 2.1: Effect of bandwidth selection in one-dimensional kernel density estimation. The KDE on the left uses a bandwidth that is too low, whereas the KDE on the right uses a bandwidth that is too high.

A commonly used kernel is the Gaussian kernel. In its fully normalized form, the bandwidth-scaled Gaussian kernel in d dimensions is

$$G_h(x, y) = \underbrace{\frac{1}{(2\pi h^2)^{d/2}}}_{\text{normalization factor}} \exp\left(-\frac{\|x - y\|_2^2}{2h^2}\right).$$

The leading factor makes the kernel integrate to one, which is required when the KDE is interpreted as a normalized probability density.

For fixed bandwidth h and dimension d, the normalization factor $(2\pi h^2)^{-d/2}$ multiplies every kernel value by the same constant. It therefore changes the overall scale of the KDE values, but not which points contribute more or less to the estimate. When only relative kernel contributions or relative density values are needed, this constant can be omitted. Following the DEANN formulation, this gives

$$K_h(x, y) = \exp\left(-\frac{\|x - y\|_2^2}{2h^2}\right).$$

In many settings, kernel functions depend on a notion of similarity or distance between points. Let $D : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ denote a distance between two points. For the Gaussian kernel, this distance is the Euclidean distance,

$$D(x, y) = \|x - y\|_2.$$

The kernel contribution can then be written as

$$K_s(D(x, y)) = \exp\left(-\frac{D(x, y)^2}{2h^2}\right).$$

With this notation, the kernel contribution can be written compactly as

$$K_h(x, y) = K_s(D(x, y)).$$

Since $D(x, y)$ is a distance, smaller values give larger kernel contributions, while larger values give smaller contributions. This distance-dependent structure is important for methods based on Approximate Nearest Neighbor (ANN) search, since it suggests that points close to the query often contribute most to the density estimate [7].

Although nearby points often contribute most strongly for distance-based kernels, exact KDE still computes the contribution from every data point. Evaluating $\mathrm{KDE}_X(q)$ for a query point q requires computing $K_h(x_i, q)$ for every point $x_i \in X$. For a distance-based kernel, evaluating one contribution requires computing a distance between two points in $\mathbb{R}^d$. Assuming that distance computation scales linearly with the dimensionality, one kernel evaluation costs $\Theta(d)$. Thus, evaluating all n data points for one query costs $\Theta(nd)$. For large datasets, high-dimensional data, or workloads with many queries, this cost motivates accelerated methods for KDE.

2.1.1 Dynamic KDE Problem Setting

In static instances of KDE, the dataset X is fixed. Dynamic KDE instead considers the same estimator when the dataset changes over time through update operations, a formulation studied both as a data-structure problem with update and query operations [9] and in settings where KDE-based structures are maintained as points are added over time [10].

To describe this setting, let X_t denote the active dataset at time step t, with active size $n_t = |X_t|$. When the time step is not important, the active size is denoted simply by n. A query at time step t is evaluated with respect to X_t, rather than with respect to a single fixed dataset. For a query point q, this gives

$$\mathrm{KDE}_{X_t}(q) = \frac{1}{n_t} \sum_{x \in X_t} K_h(x, q).$$

The active dataset is modified by update operations, while queries leave it unchanged. An update is either the insertion of a new point or the deletion of a currently active point. The transition from X_t to X_{t+1} may consist of one or more such updates. After this transition, any number of queries may be evaluated with respect to the same active dataset X_{t+1}, until the next update transition occurs. Thus, a query operation only evaluates, or estimates, $\mathrm{KDE}_{X_t}(q)$ for the active dataset at the current time step. This distinction is important for approximate KDE methods, since their auxiliary data structures must remain consistent with the active dataset as updates are applied.

2.2 Curse of Dimensionality

KDE estimates local density by comparing a query point with the surrounding data. This makes its behavior strongly dependent on the geometry of the data space. As the dimension of the data increases, geometric behavior becomes less intuitive compared with two- and three-dimensional spaces. Points close to the query are expected to provide local information, while distant points contribute less. In high dimensions, this separation becomes harder to rely on.

One relevant effect is sparsity. In high dimensions, finite samples occupy an increasingly sparse subset of the domain. Consequently, regions close to a query point contain fewer observations unless the sample size grows very rapidly, making KDE harder to estimate accurately [11].

Another relevant effect is distance concentration, illustrated in Figure 2.2. In low dimensions, distances can often separate nearby samples from distant samples. In high-dimensional spaces, distances between points can become increasingly similar, reducing the ability of a distance metric to distinguish between near and far points [12].

Using $d_{\min}(d)$ and $d_{\max}(d)$ to denote the nearest and farthest distances from a query point in dimension d, this behavior can be expressed through the bounded relative gap

$$\lim_{d\to\infty} \mathbb{E}\left[1 - \frac{d_{\min}(d)}{d_{\max}(d)}\right] = 0.$$

This means that the nearest and farthest distances become increasingly similar relative to their magnitude. The effect has been shown under broad distributional assumptions, and empirical results suggest that it can already appear in dimensions as low as 10–15 [13].

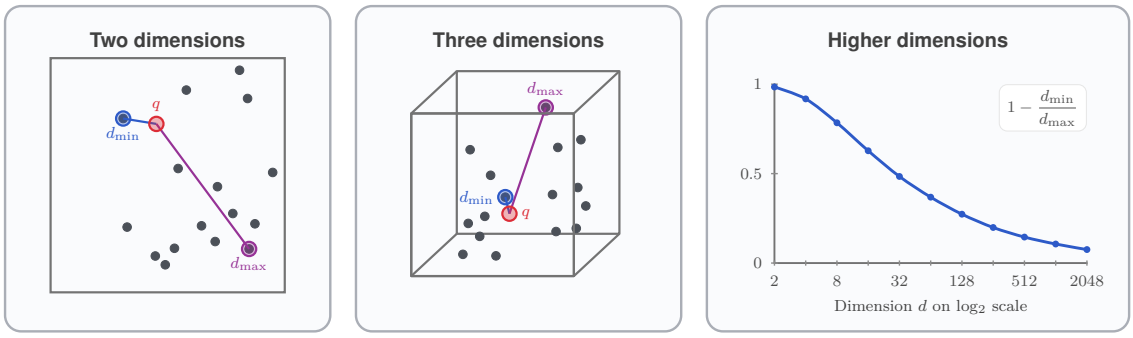

Figure 2.2: Illustration of distance concentration. The left two panels show nearest and farthest distances from a query point in two and three dimensions. The right panel shows that the empirical relative distance gap decreases as the dimension increases, making nearest and farthest distances less distinguishable.

2.3 Approaches to Accelerating KDE

The computational cost of exact KDE has motivated several approaches to accelerating kernel density estimation. These approaches can be grouped according to the part of the KDE computation they reduce. Some exploit spatial structure in the data, some approximate the sum using sampled or representative points, and some prioritize points expected to contribute most strongly to a query.

Tree-based and dual-tree methods organize the data hierarchically, allowing parts of the computation to be bounded or summarized without evaluating every point individually [14, 15]. Related fast kernel summation methods use similar ideas to reduce the cost of large kernel sums [4, 16]. These methods rely on useful geometric structure in the data, and their effectiveness can decrease in high-dimensional spaces where distances become less informative.

Another approach is to approximate the KDE sum using only part of the dataset. For example, sampling-based methods use randomly selected points, while representative-subset methods replace the full dataset with a smaller set chosen to preserve the estimate approximately [6]. These methods reduce the number of kernel evaluations, but their accuracy depends on how well the selected points represent the full dataset.

A third approach is to focus computation on points that are expected to contribute most strongly to the query. For distance-based kernels, nearby points often have large kernel values, making nearest-neighbor and hashing-based methods natural candidates for accelerating KDE [5, 17, 18, 19]. However, considering only retrieved nearby points does not by itself account for the contribution of the remaining data.

One approach that combines two of these ideas is DEANN, which evaluates points expected to contribute strongly to the query and estimates the remaining KDE sum from sampled points [7].

2.4 DEANN

Density Estimation from Approximate Nearest Neighbors (DEANN) is a method for estimating KDE by combining approximate nearest neighbor search with random sampling [7]. The method can be understood as a way to improve over pure random sampling by evaluating likely high-contribution points directly, while still estimating the remaining contribution by sampling.

Random sampling approximates sums over the dataset using randomly selected points. Given a set $X = \{x_1, \ldots, x_n\}$, a function $g : X \to \mathbb{R}$ evaluated on each point, and samples $s_1, \ldots, s_m$ drawn uniformly from X, the full average can be approximated as

$$\frac{1}{n}\sum_{i=1}^{n} g(x_i) \approx \frac{1}{m}\sum_{j=1}^{m} g(s_j).$$

In the KDE setting, $g(x)$ corresponds to the kernel contribution $K_h(x, q)$ for a fixed query q. This estimator is unbiased, meaning that its expected value equals the true average over X. However, unbiasedness only describes the average behavior of the estimator over repeated samples. It does not guarantee that each individual estimate is accurate.

In particular, random sampling can have high variance when the kernel contributions are highly skewed. If a small number of points contribute heavily to the KDE value, a uniform random sample may omit these points. The resulting estimate can then be inaccurate for that query, even though the estimator remains correct in expectation. Larger samples reduce this risk and can provide stronger approximation guarantees, but they also increase query cost [6]. In particular, Karppa et al. show that sufficiently large random samples approximate the full KDE value with high probability [7, Lemma 2].

This motivates evaluating likely high-contribution points directly rather than relying only on random sampling. For distance-based kernels, nearby points often contribute most to the KDE value, so nearest-neighbor search provides a natural way to identify such points. For comparison, exact k-nearest neighbor search returns the k points in the dataset with smallest distance to a query point under a chosen distance measure. Approximate Nearest Neighbor (ANN) search relaxes this requirement in order to improve efficiency.

Given a query point $q \in \mathbb{R}^d$, an ANN data structure returns a set of candidate points that are intended to be close to q, but they are not required to be the exact k nearest points. Efficient ANN implementations often achieve substantial speedups over exact search with only a small loss in retrieval quality in practice [20]. In this work, the ANN component is treated as a black-box subroutine, and the term approximate nearest neighbors refers to the candidate points returned by the ANN backend for a given query.

DEANN combines these two ideas by partitioning the dataset into a subset of points returned by ANN and the remaining points [7]. The contributions of the ANN-selected points are evaluated directly, while the contribution of the remaining points is estimated using random sampling. The two contributions are combined using weights proportional to the sizes of the subsets. Figure 2.3 expands on Figure 1.3 from the Introduction by showing the two estimator components and how they are combined.

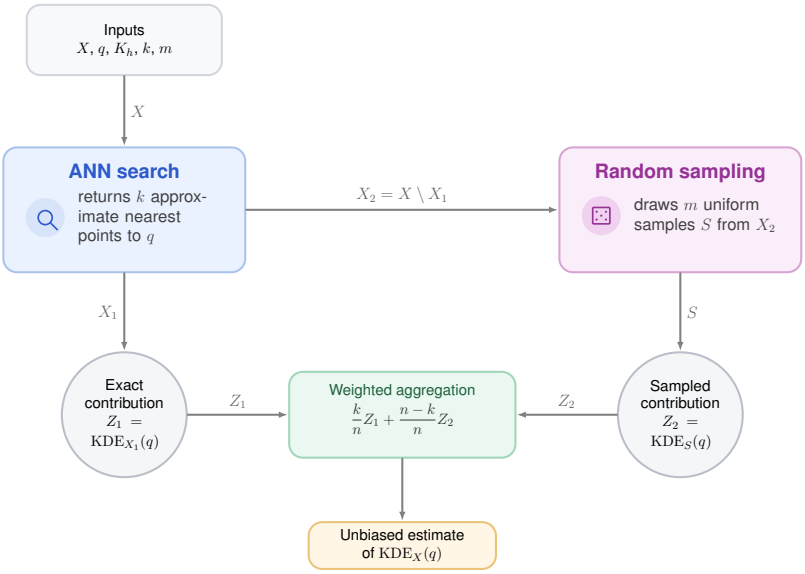

Figure 2.3: Structure of the DEANN estimator. Given inputs X, q, K_h, k, and m, an approximate nearest neighbor query returns a subset $X_1 \subseteq X$ of k points. The remaining points form $X_2 = X \setminus X_1$, from which a uniform random sample S of size m is drawn. The contribution of X_1 is evaluated exactly as Z_1, while the contribution of X_2 is estimated using Z_2 computed on S. The two terms are combined with weights proportional to the sizes of X_1 and X_2 to obtain an unbiased estimate of $\mathrm{KDE}_X(q)$.

To illustrate this process concretely, Figure 2.4 shows a two-dimensional example of a DEANN query. Sampling is performed with replacement, so the same point may be selected multiple times, although this is not shown in the figure. For intuition, the contribution of a point can be understood as decreasing with its distance to the query through the kernel function.

More formally, let $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ denote the dataset and let $q \in \mathbb{R}^d$ be a query point. Let k denote the number of points returned by the approximate nearest neighbor query and let m denote the number of random samples. An approximate nearest neighbor query returns a subset $X_1 \subseteq X$ consisting of k points returned by the ANN procedure. The contribution of these points is evaluated exactly.

The remaining points form the set $X_2 = X \setminus X_1$. Instead of evaluating all contributions from X_2, its average contribution is estimated using m points drawn independently and uniformly at random from X_2, denoted by S. The contribution of X_2 is then estimated based on S.

The final estimate is obtained by combining the exact contribution from X_1 with the scaled estimate of the contribution from X_2. This yields an estimator of the form

$$\widetilde{\mathrm{KDE}}_X(q) = \frac{k}{n} Z_1 + \frac{n-k}{n} Z_2.$$

where Z_1 and Z_2 denote the average kernel contributions over X_1 and S, respectively. The static DEANN procedure is summarized in Algorithm 1, adapted from Karppa et al. [7].

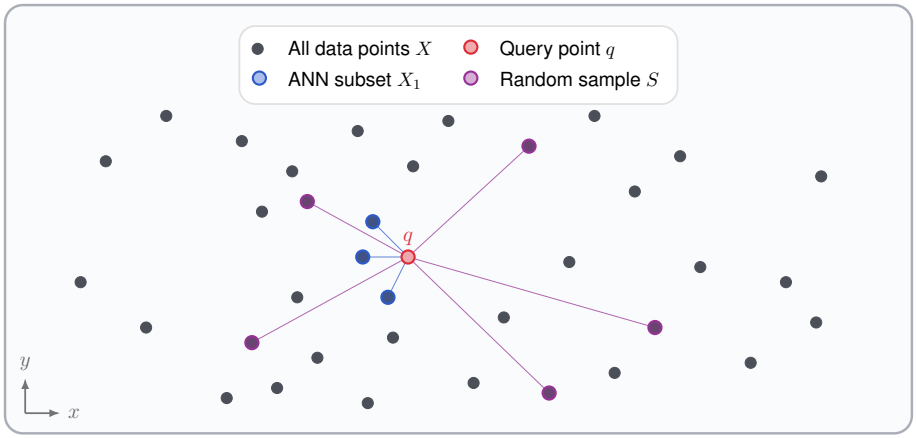

Figure 2.4: Two-dimensional example of a DEANN query with $k = 3$ approximate nearest neighbors (blue) and $m = 5$ random samples (purple).

Algorithm 1 DEANN estimator, adapted from Karppa et al. [7].

Input: Dataset $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d$, query vector $q \in \mathbb{R}^d$, kernel function $K_h : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, approximate nearest neighbor function ANN_X, mapping $\mathbb{R}^d$ to $[n]^k$, number of neighbors k, sample size m.

Output: Unbiased estimate $\widetilde{\mathrm{KDE}}_X(q)$ of $\mathrm{KDE}_X(q)$.

function DEANN($X, K_h, \mathrm{ANN}_X, q, k, m$)
  $X_1 \leftarrow \{x_i : i \in \mathrm{ANN}_X(q)\}$ ▷ Approximate nearest neighbors of q
  $X_2 \leftarrow X \setminus X_1$ ▷ Remaining points
  $Z_1 \leftarrow \mathrm{KDE}_{X_1}(q) = \frac{1}{k} \sum_{x \in X_1} K_h(x, q)$ ▷ Exact average contribution over X_1
  $S \leftarrow$ size-m uniform random sample from X_2
  $Z_2 \leftarrow \mathrm{KDE}_S(q) = \frac{1}{m} \sum_{x \in S} K_h(x, q)$ ▷ Use S to estimate the average contribution over X_2
  $\widetilde{\mathrm{KDE}}_X(q) \leftarrow \frac{k}{n} Z_1 + \frac{n-k}{n} Z_2$ ▷ Combine both parts using their relative sizes
  return $\widetilde{\mathrm{KDE}}_X(q)$
end function

The unbiasedness of the estimator follows from two observations. First, uniform random sampling gives an unbiased estimate of an average over a finite set. In the DEANN setting, this means that the sample average Z_2 over S is an unbiased estimate of the true average kernel contribution over X_2. The number of samples m affects the variance and approximation quality of this estimate, but not its expectation.

Suppose the dataset is split into two disjoint parts, and we want to estimate the average kernel contribution over the entire dataset. If we can estimate the average over each part separately, then the overall average can be obtained by taking a weighted combination of the two, where each part is weighted by its relative size.

Formally, let $X = A \cup B$ with $A \cap B = \emptyset$. If Z_A and Z_B are unbiased estimators of

$\mathrm{KDE}_A(q)$ and $\mathrm{KDE}_B(q)$, respectively, then

$$Z = \frac{|A|}{|X|} Z_A + \frac{|B|}{|X|} Z_B$$

is an unbiased estimator of $\mathrm{KDE}_X(q)$ [7, Lemma 3].

Applying this result with $A = X_1$ and $B = X_2$, the term Z_1 is exact and therefore unbiased, while Z_2 is unbiased by uniform random sampling. Therefore,

$$\mathbb{E}\left[\widetilde{\mathrm{KDE}}_X(q)\right] = \mathrm{KDE}_X(q).$$

This shows that DEANN remains unbiased regardless of the quality of the approximate nearest neighbors returned by the ANN procedure [7, Corollary 4]. If the ANN backend misses points with large kernel contributions, these points are still part of the residual set $X_2 = X \setminus X_1$. They are therefore not discarded, but are included in the quantity estimated by the random sampling term. As a result, missed high-contribution points may increase variance, but they do not introduce bias. The ANN component therefore affects the variance and efficiency of the estimator, but it does not affect unbiasedness.

2.4.1 Challenges in Extending DEANN to Dynamic KDE

To extend DEANN to a dynamic setting, a query on the active dataset X_t requires a partition into ANN-selected points $X_{t,1}$ and a residual subset $X_{t,2} = X_t \setminus X_{t,1}$. This partition must be formed over the current active points. As in the static estimator, the contribution of $X_{t,1}$ is evaluated directly, while the contribution of $X_{t,2}$ is estimated by random sampling [7]. The difficulty is not the query formula itself, but maintaining the data structures that produce this partition as the active dataset changes.

Insertions and deletions impose consistency requirements on both parts of the estimator. After an insertion, the new point must be available to future ANN queries and residual random sampling. After a deletion, the removed point must no longer be returned by the ANN component or sampled as part of the residual contribution. If the ANN component and the sampling component do not represent the same active dataset, the resulting estimate no longer corresponds to $\mathrm{KDE}_{X_t}(q)$. This is particularly challenging for the ANN component, since graph-based approximate proximity graphs are difficult to maintain incrementally as the dataset evolves [21]. Deletions are especially problematic because removing a vertex may require repairing graph connectivity and handling incoming edges that are not directly available in singly linked graph representations [22]. Maintaining a valid DEANN partition over the current active dataset is therefore the central data-structure requirement in dynamic DEANN, with the ANN component posing the main difficulty.

Changes in the active dataset can also change the KDE instance being approximated. If the active dataset changes in size or spatial extent, a previously selected bandwidth h may become less appropriate. The DEANN parameters k and m are also tied to the estimator configuration, since they determine how much of the estimate comes

from ANN-selected points and how much comes from random sampling [7]. The choice of h, k, and m is therefore part of the estimator configuration for a given dataset and KDE value scale, rather than a fixed property of KDE itself [7, 23]. Automatic adaptation of these parameters is not considered here; the experiments instead select parameters before evaluating the dynamic workloads, as described in Section 4.1.3. The focus is instead on maintaining the DEANN estimator over a changing active dataset, using dynamic sampling together with a dynamic ANN component.

2.5 Approximate Nearest Neighbor Search

Nearest neighbor search is the problem of finding the data points closest to a query point under a chosen distance measure. In exact k-nearest neighbor search, the result is the set of k data points with smallest distance to the query. Approximate k-nearest neighbor search relaxes this requirement in order to reduce query time. Given a query point q and a requested number of neighbors k, it returns k existing data points from the indexed dataset that are expected to be close to q, but are not required to be the exact k nearest neighbors. This relaxation is useful in high-dimensional settings, where exact nearest neighbor search may provide little improvement over comparing the query with every point in the dataset [24].

The quality of approximate nearest neighbor results can be defined in different ways. In theoretical work, ANN is often formulated as a mathematical problem with a guarantee on how close the returned point must be to the true nearest neighbor [17]. Practical ANN systems are often evaluated empirically instead, using measures such as query time, memory usage, and recall [20]. For k-nearest neighbor queries, recall measures how many of the true k nearest neighbors are included among the returned dataset points.

Although well-engineered practical ANN methods often provide strong empirical performance, they may lack the rigorous guarantees needed to derive theoretical bounds for every query. As shown in Section 2.4, the estimator separates the ANN-selected points from the residual set. If the ANN backend fails to return some nearby points, those points remain in the residual set, whose contribution is estimated by random sampling. Missed nearby points may increase the variance and reduce the efficiency of the estimate, but they do not introduce bias. For this reason, the ANN backend can be treated as a *black-box* subroutine [7].

ANN systems differ in how they organize the dataset before queries are answered. Hashing-based methods use hash functions designed to make nearby points collide with higher probability. These include both data-independent and data-dependent variants, with data-dependent LSH improving theoretical trade-offs in some settings [24, 25]. Clustering- and quantization-based methods reduce the search space by grouping or compressing vectors [26, 27].

Another important family is graph-based ANN methods. These methods represent the dataset as a proximity graph, where data points are vertices and edges connect nearby points. Examples include HNSW and DiskANN, both of which use graph

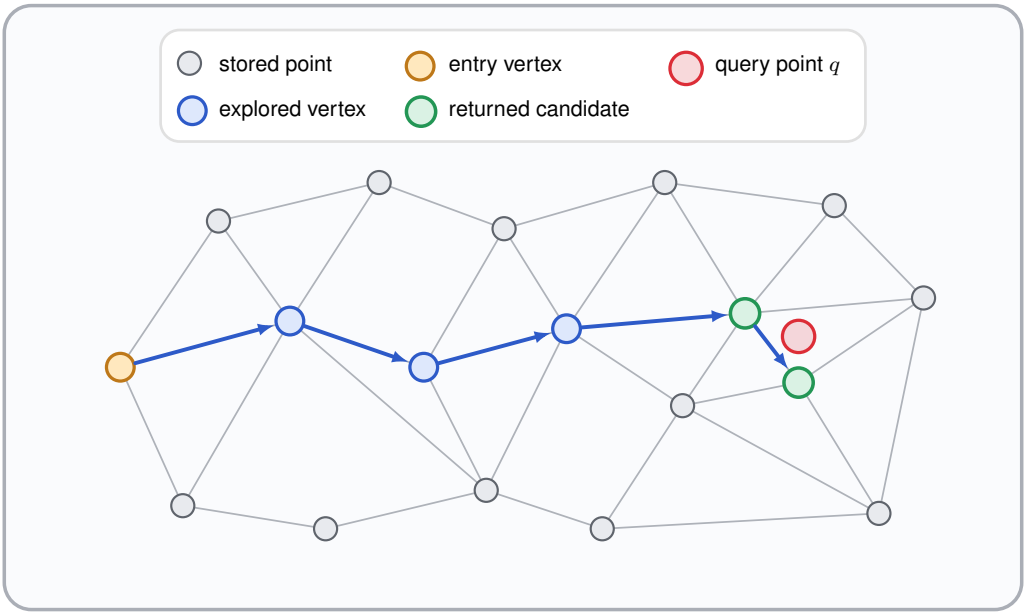

Figure 2.5: Schematic example of graph-based ANN search. The dataset is represented as a proximity graph, and a query is answered by traversing from an entry vertex toward vertices closer to the query point q. Only part of the graph is explored, and the returned candidates are existing data points reached near the end of the search.

traversal to find approximate nearest neighbors efficiently [18, 19]. A query is answered by traversing the graph from one or more entry points, repeatedly exploring vertices whose neighbors may lead closer to the query. The search is approximate because only part of the graph is explored, but a well-constructed graph can guide the search toward nearby points efficiently. Figure 2.5 illustrates this type of graph traversal.

Graph-based ANN methods are often designed for static datasets, where the graph is constructed before queries are performed. In a dynamic setting, insertions and deletions change both the set of vertices and the edges used for search. Maintaining search quality under these updates is difficult, since local changes can affect graph connectivity and paths followed during traversal [21, 22].

2.5.1 IP-DiskANN

Dynamic DEANN requires an ANN index that can be updated when points are inserted into or deleted from the active dataset. IP-DiskANN, short for InPlaceUpdate-DiskANN, is a dynamic graph-based ANN index that extends DiskANN with in-place insertions and deletions [22]. These operations make it suitable as the ANN backend for dynamic DEANN, since the ANN index must represent the same active points as the KDE estimator.

DiskANN stores the dataset as a directed proximity graph with bounded out-degree [19]. Each vertex corresponds to a data point, and its outgoing edges point to

selected neighboring vertices. The degree bound limits the number of outgoing edges per vertex, which keeps the index size and traversal cost controlled. Insertions can be handled incrementally by searching the existing graph for candidate neighbors of the new point and pruning the resulting neighbor lists.

Deletions are more difficult. A DiskANN-style graph stores outgoing edges, but not the complete set of incoming edges to each vertex. When a point is removed, vertices that previously pointed to it can be difficult to find. If these references are not repaired, the graph may contain dangling edges or lose connections that were useful for search. Deletion is not only a matter of removing the point itself, but also of preserving the navigability of the graph after removal [22].

FreshDiskANN addresses this problem using lazy deletion [28]. Deleted points are marked rather than immediately removed from the graph, and queries avoid returning them as nearest-neighbor results. The graph is periodically consolidated by removing deleted vertices and repairing affected neighborhoods. This avoids repairing every deletion immediately, but the consolidation step can be expensive and may introduce maintenance overhead when many points have been deleted [22].

IP-DiskANN instead applies updates in place. For insertions, it uses the incremental structure of DiskANN. For deletions, it searches near the deleted point to approximate the vertices whose outgoing edges may be affected, adds a limited number of replacement edges, and prunes updated neighbor lists when they exceed the degree bound [22]. The purpose is to maintain graph connectivity without exhaustively reconnecting all affected neighborhoods, which could introduce many edges, increase update cost, and degrade the graph structure.

Because the in-place deletion procedure may not remove every edge pointing to a deleted vertex, IP-DiskANN also performs a lightweight cleanup step that removes remaining dangling references. Unlike the consolidation used by lazy-deletion approaches, this cleanup does not rebuild neighborhoods using distance computations, which makes it substantially cheaper than full graph consolidation [22].

The parameters used later in the experiments are the graph degree R, the build list size l_b, and the search list size l_s. The graph degree R bounds the number of outgoing edges stored for each vertex. The build list size l_b controls the candidate list used during index construction and insertion, while the search list size l_s controls the candidate list used when answering ANN queries. Larger values generally increase the amount of graph exploration and maintenance work, but may improve ANN recall.

2.6 Related Work

With the KDE estimator, the dynamic problem setting, and the ANN backend established, this section positions the thesis relative to prior work on dynamic KDE. The comparison focuses on differences in update model, query model, and algorithmic approach.

Liang et al. study dynamic KDE as a theoretical data-structure problem [9]. Their

method uses locality-sensitive hashing and importance sampling to support sublinear update and query time, and includes robustness guarantees for adaptive queries. Their formal update operation replaces an existing data point with a new one. This differs from the active-set model considered in this thesis, where the dataset evolves through explicit insertions and deletions.

Laenen et al. also study dynamic KDE, but with a different objective [10]. Their data structure maintains KDE estimates for a set of query points as data points are inserted. These estimates are then used to maintain sparse approximations of fully connected similarity graphs for dynamic spectral clustering. Thus, KDE is a central component of their method, but the main application is dynamic graph construction rather than direct KDE query evaluation over a changing active dataset.

Other work considers dynamic KDE under more specific structural assumptions. For example, Huang et al. study a dynamic low-rank Fast Gaussian Transform [29]. This is related because it also addresses KDE under changes to the data, but it relies on a different acceleration strategy and a different problem structure. Including this line of work highlights that dynamic KDE can be formulated in several ways. The relevant design choices include whether updates are insertions, deletions, or replacements, whether queries are answered on demand or maintained over time, and whether the algorithm relies on hashing, low-rank structure, or nearest-neighbor search.

The work of this thesis considers a different formulation. The goal is to extend the DEANN estimator to a dynamic active dataset, while preserving the query structure of the static estimator [7]. The implemented estimator maintains a dynamic ANN index and a dynamic residual random-sampling structure so that KDE queries can be evaluated against the current active dataset after insertions and deletions. Compared with the dynamic KDE works mentioned, the focus is therefore not on maintaining a fixed set of stored query estimates or on developing a new theoretical data structure, but on studying the implementation and empirical trade-offs of a dynamic DEANN-style estimator.

3

Implementation

A dynamic kernel density estimation framework is implemented in C++ as an extension of the DEANN codebase [30]. The implementation focuses on the data structures, update handling, and query procedures required to support dynamic datasets.

Two methods for evaluating Kernel Density Estimation (KDE) are implemented. The first is a dynamic naïve KDE baseline that evaluates the density exactly over the current active dataset. The second is the main estimator, referred to as the *Dynamic ANN Estimator*, which extends the DEANN framework to dynamic datasets by combining dynamic approximate nearest neighbor search with dynamic random sampling.

3.1 Dynamic Naïve KDE

As a baseline, the static naïve KDE implementation from the DEANN codebase is used for the implementation of the dynamic baseline. Additional logic is introduced to support insertions, deletions, and query handling over the evolving dataset, thereby extending it to a dynamic setting. Although this approach is computationally expensive, it provides a correctness reference and serves as a baseline for evaluating approximate methods.

3.1.1 Data representation and updates

The dynamic naïve KDE baseline stores the dataset as a flat contiguous array of point coordinates. For a dataset of n points in d dimensions, the representation consists of $n \cdot d$ values stored in row-major order. This layout is consistent with the static naïve KDE implementation and allows the existing exact KDE routine to be reused.

Insertions append the coordinates of the new point to the end of the array. This requires copying one d-dimensional point and therefore costs $O(d)$.

Deletions are handled by swap deletion. When the point at index i is deleted, the last point in the array is copied into position i, and the array is shortened by one point. This also costs $O(d)$, since only one d-dimensional point is copied. The operation does not preserve the order of points in memory, but this does not affect

the mathematical definition of the KDE, since the estimator is an unordered sum over the dataset.

3.1.2 Exact query evaluation

Queries are evaluated using the static naïve KDE routine from the original DEANN codebase [30]. For a query point q, the routine evaluates the kernel contribution of every stored point and returns the exact KDE value over the current dataset. The supported update and query operations are summarized in Algorithm 2.

Algorithm 2 Operations in Dynamic Naïve KDE.

Input: Dataset X, query point q, point index i, new point x.
Output: Exact KDE value over the current dataset.
function INSERT(X, x)
    append x to X
end function
function DELETE(X, i)
    $X[i] \leftarrow X[|X| - 1]$
    remove the last point from X
end function
function QUERY(X, q)
    return NAIVEKDE(X, q)
end function

The dynamic naïve KDE estimator is designed to provide an exact reference in the presence of updates. While its computational cost is high, it enables controlled comparison with approximate methods and isolates the effect of update handling from approximation error.

3.2 Dynamic ANN Estimator

The main estimator in the implementation is a dynamic extension of DEANN, referred to here as the *Dynamic ANN Estimator*. Its purpose is to approximate KDE efficiently on a dataset that evolves through insertions and deletions. As in DEANN, the estimate is formed by combining an exact contribution from points returned by an approximate nearest neighbor search with an estimated contribution obtained from the remaining points through random sampling. In the dynamic setting, the query formula remains largely unchanged. The main implementation requirement is instead to keep the point-store, ANN index, and sampling structure synchronized as points are inserted and deleted, while maintaining the efficiency and accuracy of the estimator.

3.2.1 Overall design

The estimator is implemented as a composition of three components. The first component is a `PointStore`, which maintains the current set of points together with their identifiers and deletion status. The second component is an ANN index based on IP-DiskANN, used to retrieve a set of approximate nearest neighbors for a given query. The third component is a dynamic random sampling structure, used to estimate the contribution of the remaining active points.

This separation keeps the implementation modular. The `PointStore` acts as the source of truth for the active dataset, the ANN backend is responsible only for approximate nearest neighbor retrieval, and the sampling component maintains the set of IDs eligible for random sampling. Query evaluation combines these components as in DEANN, where the contribution of ANN-selected points is evaluated exactly, while the remaining contribution is estimated through sampling. Figure 3.1 shows the component structure and the main information exchanged during queries and updates.

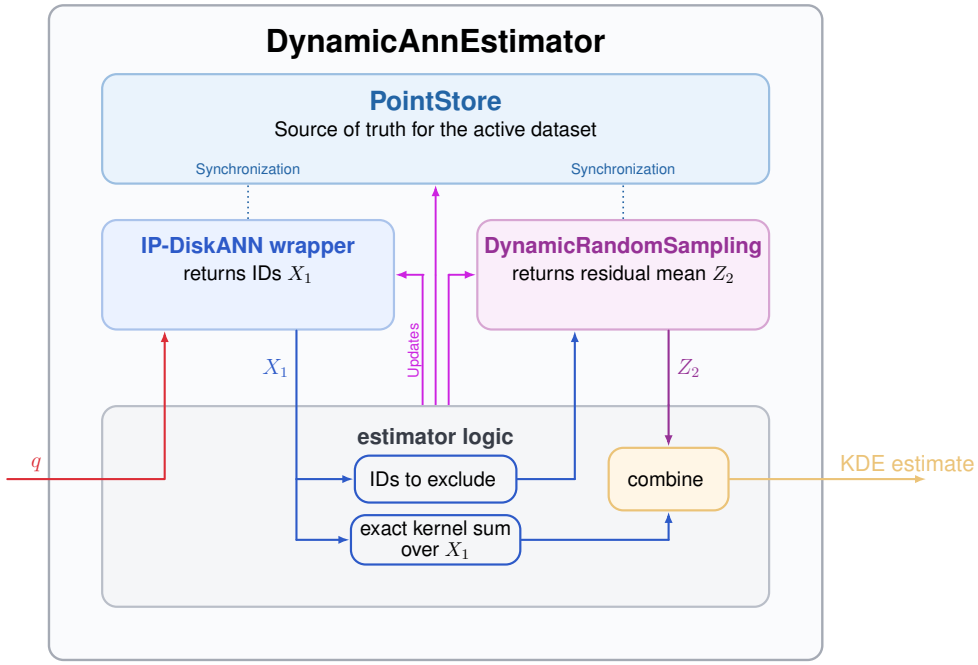

Figure 3.1: Organization of the Dynamic ANN Estimator. The estimator owns a `PointStore`, an IP-DiskANN wrapper, and a `DynamicRandomSampling` structure. The `PointStore` maintains the active dataset, while the estimator logic coordinates queries and updates across the ANN and sampling components.

3.2.2 IP-DiskANN wrapper

The ANN backend is based on the IP-DiskANN implementation available through the Microsoft DiskANN repository. In particular, the implementation uses the Rust `diskann-garnet` interface, which exposes DiskANN functionality through Foreign Function Interface (FFI) endpoints [31]. Since no suitable C++ interface for IP-DiskANN was available, a C++ wrapper was implemented to integrate this backend into the estimator.

The wrapper defines a C++ interface for initialization, insertions, deletions, ANN queries, and cleanup after deletions. It encapsulates the FFI communication with the Rust backend and keeps the rest of the estimator independent of the backend interface. A small extension to the Rust FFI layer was required to expose the lightweight cleanup step described for IP-DiskANN [22]. This step removes dangling edges that still point to deleted vertices after in-place deletions. The operation already existed in the underlying IP-DiskANN implementation, but was not exposed through the existing FFI interface.

The wrapper also defines the ownership boundary between components. The `PointStore` maintains the primary dataset representation, while the ANN backend maintains a separate copy of the data within its internal index structures. These two representations serve different roles in the estimator, with the `PointStore` acting as the source of truth and the ANN backend providing approximate nearest-neighbor queries. The ANN component is treated as a black box that supports dynamic updates and nearest-neighbor queries.

3.2.3 Dynamic Random Sampling

The estimated contribution from the remaining points is handled by a dynamic random sampling structure (DRS). This component maintains a list of currently active IDs together with an auxiliary ID-to-position array, which stores for each ID its current position in the active-ID list. The purpose of this representation is to support efficient updates, since insertion of a new ID appends it to the active-ID list, while deletion is handled by a swap-remove operation in which the deleted ID is replaced by the last active ID and the list is shortened.

At query time, the sampling component can either sample from the entire active set or sample while excluding a specified set of IDs. The latter operation is used in the Dynamic ANN Estimator, since points already accounted for by the ANN contribution must not also be included in the random sampling estimate. Exclusion is implemented through rejection sampling over the active-ID list. Algorithm 3 shows the logic of the exclusion-based query routine.

Algorithm 3 SampleResidual in Dynamic Random Sampling.

Input: Active-ID list A, exclusion set X_1, sample size m, query point q, kernel function K_h.
Output: Sample-based estimate Z_2 of the average kernel contribution over active points not in X_1.

function SampleResidual(A, X_1, m, q, K_h)
  $T \leftarrow 0$ ▷ Accumulate kernel contributions of accepted samples
  $c \leftarrow 0$ ▷ Number of accepted samples
  while $c < m$ **do**
    draw an ID i uniformly from A ▷ Sample from the current active IDs
    if $i \notin X_1$ **then**
      $T \leftarrow T + K_h(x_i, q)$ ▷ Use the sampled point only if it is not excluded
      $c \leftarrow c + 1$
    end if
  end while
  $Z_2 \leftarrow T/m$
  return Z_2 ▷ Average kernel contribution of the accepted samples
end function

This design makes the sampling component fully dynamic while keeping update operations inexpensive. It also makes the residual sampling step independent of the ANN implementation, as the only information passed between the two components is the set of ANN-selected identifiers X_1, which is excluded when sampling.

3.2.4 Update handling

The estimator supports both insertions and deletions after initialization. During `fit`, the points are inserted into the `PointStore`, the ANN wrapper is constructed, and each initial point is inserted into the IP-DiskANN index using its integer identifier. After this, the sampling structure is initialized from the active IDs in the `PointStore`.

Insertions are handled as synchronized updates across all dynamic components. A new point is first inserted into the `PointStore`, which returns a new identifier. This identifier is then inserted into the sampling structure and forwarded to the ANN backend together with the point coordinates. As a result, the point becomes available both for nearest neighbor retrieval and for sampling from the remaining points.

Deletions are handled similarly. If an identifier is valid and still active, it is first marked as deleted in the `PointStore`. The identifier is then removed from the sampling structure and forwarded to the ANN backend for deletion from the IP-DiskANN index. This ensures that deleted points no longer contribute to either part of the estimator. To prevent unbounded growth of inactive entries, the `PointStore` is compacted when the fraction of deleted stored points exceeds a fixed threshold. Compaction scans the stored entries, skips entries marked as deleted, and copies the

remaining active points into a new point-store representation while preserving their existing IDs.

The important property of this update procedure is that all components are modified consistently. The active dataset represented by the `PointStore`, the active IDs maintained by the sampler, and the points indexed by the ANN structure are intended to remain synchronized after each update.

3.2.5 Query evaluation

The query procedure follows the same overall structure as DEANN. A set of k approximate nearest neighbors is first retrieved and used to compute an exact contribution to the estimate. The contribution of the remaining active points is then approximated through random sampling, after which the two terms are combined into the final KDE estimate.

The ANN contribution is obtained in two stages. First, the wrapper queries IP-DiskANN for k neighbors. Then the exact kernel contribution of the returned ANN IDs is computed by iterating over the corresponding points in the `PointStore`. The implementation accumulates the kernel *sum*, not the mean, over the ANN-selected points. This is important because the final estimator combines the exact ANN sum with a sample mean over the residual set.

If $k = 0$, no ANN contribution is computed and the estimator reduces to pure random sampling over the active dataset. If the number of random samples is zero, or if the ANN contribution covers the full active dataset, that is, if $k = n$, no residual sampling is performed and the estimator returns the normalized ANN sum directly. Otherwise, the ANN IDs are inserted into an exclusion set, and the sampling component is queried with exclusion enabled. The sampler then performs rejection sampling over the active IDs until the requested number of accepted samples has been collected.

The final estimate follows the DEANN formulation. Let X_1 denote the set of k ANN-selected active points, let n be the number of active points, and let Z_2 denote the sample mean over the residual set $X_2 = X \setminus X_1$. The implementation computes the exact ANN contribution as the total sum

$$T_{X_1} = \sum_{x \in X_1} K_h(x, q).$$

For $k > 0$, the corresponding average contribution is

$$Z_1 = \frac{T_{X_1}}{k}.$$

The final estimate is computed as

$$\widetilde{\mathrm{KDE}}_X(q) = \frac{T_{X_1}}{n} + \frac{n-k}{n} Z_2 = \frac{k}{n} Z_1 + \frac{n-k}{n} Z_2.$$

When $k = n$, the residual term has weight zero, so the implementation returns T_{X_1}/n without calling the sampler. The full dynamic query procedure is summarized in Algorithm 4.

Algorithm 4 Query evaluation in the Dynamic ANN Estimator.

Input: Query point q, number of ANN points k, sample size m, kernel function K_h.

Output: Dynamic DEANN estimate $\widetilde{\mathrm{KDE}}_X(q)$ over the active dataset X.

function DynamicANNQuery(q, k, m, K_h)
  $n \leftarrow$ number of active points
  $X_1 \leftarrow$ ANN backend query result for q, querying for k neighbors
  $X_2 \leftarrow X \setminus X_1$ ▷ Remaining active points
  $T_{X_1} \leftarrow \sum_{x \in X_1} K_h(x, q)$ ▷ Exact total kernel sum over X_1
  if $m = 0$ **or** $k = n$ **then**
    return T_{X_1}/n ▷ No residual sampling is performed
  end if
  $Z_2 \leftarrow$ DRS.SampleResidual(X_1, m, q, K_h)
  return $\frac{T_{X_1}}{n} + \frac{n-k}{n} Z_2$ ▷ Combine contributions as a weighted average
end function

3.2.6 Design rationale

The Dynamic ANN Estimator is designed to mirror the structure of DEANN while supporting updates. At the query level, the estimator remains conceptually simple, since an exact contribution from ANN-selected points is combined with an estimated contribution from random sampling. Most of the implementation complexity instead arises from maintaining consistency of the active dataset across the dynamic components.

The implementation is based on two main design decisions. The first is the use of a wrapper around the Rust IP-DiskANN implementation rather than a native C++ reimplementation. This keeps the ANN component modular and makes it possible to treat the index as an external black box. It also ensures that the ANN works as the authors intended. The second is the use of an explicit active-ID representation in the random sampling structure, which allows insertions and deletions to be handled efficiently and makes it possible to exclude ANN-selected IDs during query evaluation.

Together, these choices yield an implementation in which updates remain localized to each component, while query evaluation follows the same conceptual decomposition as in static DEANN.

4

Experiments

The empirical evaluation compares dynamic naïve KDE and DAE with respect to approximation error, running time, and stability under repeated updates. Approximation error is measured relative to exact KDE values computed by the dynamic naïve baseline. Running time is measured separately for construction, queries, insertions, and deletions. Construction time captures the upfront cost of initializing an estimator, while query and update times capture the cost after construction. To evaluate whether repeated updates degrade the maintained data structures, the experiments include long update sequences where the active dataset changes repeatedly and both approximation error and running time are measured throughout the sequence.

4.1 Experimental Configuration

All experiments were run on the Minerva compute cluster at the Department of Computer Science and Engineering, Chalmers and the University of Gothenburg. Each benchmark was restricted to a single CPU core on an Intel Xeon Gold 6548N compute node. This was done to make the reported running times reflect the relative cost of the KDE estimators and dynamic update operations, rather than differences in parallel execution or hardware utilization. No GPU acceleration was used. All cluster nodes run Ubuntu Linux 24.04 LTS [8].

4.1.1 Datasets

The experiments use four benchmark datasets: SIFT1M, GIST1M, MNIST, and DEEP10M. For each dataset, a set of base vectors is used as the dynamic dataset, and a separate set of query vectors is used for KDE evaluation. The datasets and their corresponding base-vector counts, query-vector counts, and dimensionalities are summarized in Table 4.1.

SIFT1M and GIST1M were obtained from the TEXMEX corpus, which provides base, query, learning, and ground-truth files for approximate nearest-neighbor evaluation [32]. Both datasets have been widely used in nearest-neighbor benchmarking [27]. MNIST consists of handwritten digit images represented as 784-dimensional vectors [33]. DEEP10M denotes the 10-million-vector subset of the Yandex billion-scale similarity-search benchmark together with its query set [34]. The subset is derived from DEEP1B, a large-scale image descriptor dataset [35].

Table 4.1: Datasets used in the experimental evaluation.

Dataset	Base vectors	Query vectors	Dimensions
MNIST	59,000	1,000	784
SIFT1M	1,000,000	10,000	128
GIST1M	1,000,000	1,000	960
DEEP10M	10,000,000	10,000	96

4.1.2 Compared estimators

Two estimators are evaluated: dynamic naïve KDE and the Dynamic ANN Estimator (DAE). The dynamic naïve KDE estimator computes the exact KDE value over the current active dataset and serves as the reference for accuracy measurements. DAE is the main method evaluated in this thesis. It combines dynamic ANN search with dynamic random sampling to approximate exact KDE while supporting insertions and deletions.

Where relevant, the running time of DAE updates is also reported separately for the point-store, random-sampling component, and ANN component.

4.1.3 Parameter selection

The estimator configuration consists of the bandwidth h, the number of ANN neighbors k, the number of residual random samples m, and the IP-DiskANN parameters R, l_b, and l_s defined in Section 2.5.1. DAE also uses maintenance thresholds for deletion-related cleanup and point-store compaction. All parameters were fixed before the dynamic benchmarks and were kept unchanged during each benchmark run.

The IP-DiskANN cleanup threshold was set to 0.2, following the IP-DiskANN experimental setup [22]. The `PointStore` compaction threshold was set to 0.5 as a fixed implementation setting. Both thresholds were kept constant in all experiments; optimizing cleanup and compaction policies is outside the scope of the evaluation.

For each dataset, bandwidths were selected in a similar manner as in Backurs et al. [23] and DEANN [7]. For a target KDE value μ, the bandwidth h_μ was chosen such that the median exact KDE value $\mathrm{KDE}_{h_\mu}(q)$ over a validation query set was approximately μ. The target values considered in the parameter sweep were $\mu \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Thus, μ is not an accuracy requirement, but determines the scale of the KDE values.

After the bandwidth had been fixed for a given dataset and target value, the remaining parameters were selected by validation. Each validation candidate consisted of one value of k, one value of m, and one IP-DiskANN backend regime. Two backend regimes were considered. The low-recall regime used $R = 32$ and $l_b = l_s = 64$, while the high-recall regime used $R = 64$ and $l_b = l_s = 128$. The individual IP-DiskANN parameters were therefore not swept separately; instead they were varied only through these two regimes.

For each dataset, active size, and target value μ, all validation candidates were evaluated on validation queries. Candidates with mean relative error greater than 0.1 were discarded. Among the remaining candidates, the candidate with the lowest mean query time was selected. In every final benchmark configuration, this procedure selected the low-recall IP-DiskANN regime. The high-recall regime did not reduce validation error enough to offset its additional query-time cost.

In a dynamic setting, the active size may change during evaluation, and the fastest accurate parameter configuration is not necessarily the same for all active sizes or target KDE values. This can be seen from the parameter-sweep tables in Appendix A.1. The experiments do not retune parameters during a dynamic run. Instead, the final configuration for each dataset was selected using the largest active size evaluated for that dataset and the final target value $\mu = 10^{-5}$. This gives one fixed DAE configuration per dataset and avoids changing parameters during the dynamic workload. The resulting evaluation therefore measures DAE with fixed, preselected parameters rather than an adaptive parameter-selection scheme.

The final configurations used in the main benchmarks are shown in Table 4.2.

Table 4.2: Selected DAE configurations for $\mu = 10^{-5}$.

Dataset	n	Bandwidth h	k	m
SIFT1M	1 000 000	78.03	80	5120
GIST1M	1 000 000	0.32	57	1810
MNIST	59 000	418.10	113	1280
DEEP10M	10 000 000	0.24	40	14482

4.1.4 Evaluation metrics

The evaluation reports both approximation error and running time. Accuracy is measured by comparing the approximate KDE estimates with exact KDE values computed by the dynamic naïve KDE baseline. For a set of evaluated queries $Q = \{q_1, \ldots, q_s\}$, let $\mathrm{KDE}_X(q_i)$ denote the exact KDE value of query q_i over the current dataset X, and let $\widetilde{\mathrm{KDE}}_X(q_i)$ denote the corresponding approximate estimate. The relative error for query q_i is

$$e_i = \frac{\left|\widetilde{\mathrm{KDE}}_X(q_i) - \mathrm{KDE}_X(q_i)\right|}{\mathrm{KDE}_X(q_i)}.$$

The relative error measures the error as a fraction of the exact value, rather than as an absolute difference. The mean relative error is the average relative error over all evaluated queries and summarizes the typical error over the query set. The 95th percentile relative error is the relative error below which 95% of the evaluated query errors fall.

Additionally, running time is measured for construction, queries, insertions, and deletions. Construction time measures the time required to build an estimator on an initial active dataset before queries or updates are performed. For insertions

and deletions, the time spent in the point-store, random-sampling component, ANN index update, and ANN cleanup step is measured separately when applicable.

4.2 Benchmarks

To evaluate the dynamic KDE estimators, the experiments are divided into query benchmarks, update benchmarks, and streaming benchmarks. This separation is used because a single benchmark is not sufficient to distinguish query cost, update cost, and long-term behavior under repeated updates. The query benchmarks evaluate density queries on fixed active datasets of different sizes, while the update benchmarks isolate the cost of insertions and deletions. The streaming benchmarks combine queries and updates over longer dynamic sequences to evaluate whether accuracy and running times degrade over time.

4.2.1 Query Benchmarks

The query benchmark evaluates the estimators on fixed active datasets of different sizes. For each dataset, several active sizes are selected. For each active size n, an estimator is constructed using the first n points from the base vector set. The estimator is then evaluated on a fixed set of query points from the query vector set.

For each active size, the benchmark records construction time, query time, and accuracy relative to the dynamic naïve KDE baseline. Since the active dataset is fixed after construction, no insertions or deletions are performed during this benchmark. The benchmark therefore isolates query behavior from update behavior and shows how query performance changes as the active dataset grows. Construction time is kept separate from query time, since a method with faster queries may still require substantial preprocessing before those query savings are realized.

4.2.2 Update Benchmarks

The update benchmarks measure the cost of modifying the active dataset. Insertions and deletions are evaluated separately, since the two operations may have different costs.

In the insert-only benchmark, each estimator is first constructed on an initial active dataset. New points are then inserted in batches. For each batch, the benchmark records insertion time, insertion time per point, and the active dataset size.

In the delete-only benchmark, each estimator is first constructed on an initial active dataset. Points are then removed in batches until a specified minimum active size is reached. For each batch, the benchmark records deletion time, deletion time per point, the active dataset size, and the stored-to-active ratio. The stored-to-active ratio is recorded to track whether inactive entries accumulate after deletions.

4.2.3 Streaming Benchmark

The streaming benchmark uses a sliding-window update pattern. The active dataset has a fixed size throughout the benchmark. At each step, a batch of points is deleted and a batch of new points is inserted. Thus, the estimator is repeatedly updated while the number of active points remains constant. Figure 4.1 illustrates one state of the sliding-window update pattern.

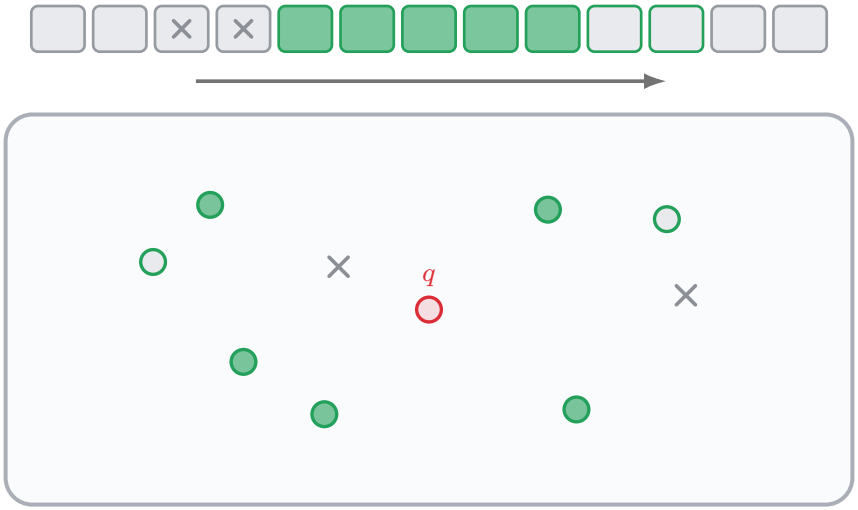

Figure 4.1: Schematic view of one sliding-window state. Green points are active, gray crosses were deleted in the previous batch, and outlined points will be inserted in the next batch.

This benchmark evaluates whether repeated replacements cause degradation that is not visible in the isolated query or update benchmarks. Such degradation may appear as increasing update time, increasing query time, or increasing error relative to the dynamic naïve KDE baseline. The active sets are generated sequentially from the base vector set: the initial active set and subsequent inserted points are taken in dataset order.

5

Results

The empirical comparison covers dynamic naïve KDE and DAE with respect to construction time, query time, approximation error, insertion cost, deletion cost, and sliding-window behavior. Dynamic naïve KDE is used as the exact reference method, while DAE is evaluated using the parameter configurations described in Section 4.1.3.

5.1 Query Benchmark Results

The query benchmark evaluates how estimator construction time, query evaluation, and approximation error vary with active dataset size. Estimator construction is reported separately to distinguish setup cost from the query cost.

The measured construction times for DAE and dynamic naïve KDE are shown across active dataset sizes in Figure 5.1. Across all datasets, DAE is substantially more expensive to build, with differences spanning multiple orders of magnitude.

The query-time scaling with active dataset size is shown in Figure 5.2. At the smallest active sizes, dynamic naïve KDE has lower query time than DAE on all datasets. Dynamic naïve KDE shows approximately linear growth in query time as the active dataset size increases. For DAE, query time initially increases with active dataset size, but the rate of increase becomes less pronounced as the active dataset size grows. On SIFT1M, GIST1M, and DEEP10M, dynamic naïve KDE exceeds DAE after the marked crossover points. For MNIST, the smallest dataset with a maximum active size of 59,000 points, no crossover is observed, and dynamic naïve KDE remains faster throughout the tested active sizes.

The approximation error of DAE in the query benchmark is shown in Figure 5.3. Across the tested active sizes, the mean relative error remains close to the 0.1 validation threshold used during parameter selection. The 95th percentile relative error is consistently higher than the mean, but remains within the same order of magnitude.

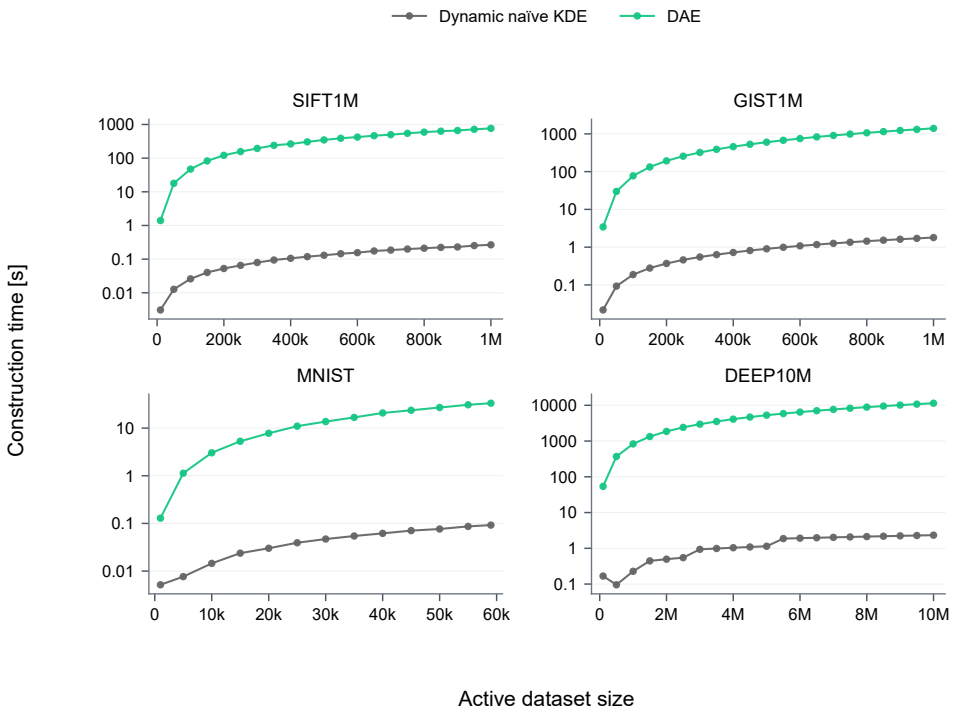

Figure 5.1: Construction time as a function of active dataset size in the query benchmark. The y-axis is logarithmic.

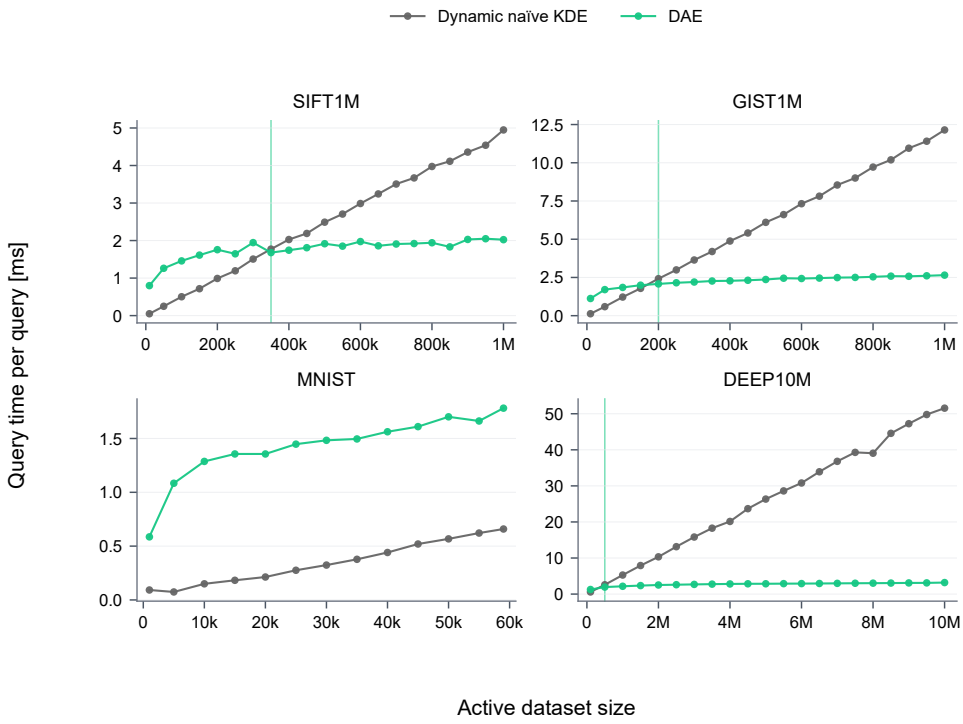

Figure 5.2: Query time per query as a function of active dataset size. The vertical line marks the first tested active size where DAE is lower than dynamic naïve KDE.

Figure 5.3: Mean and 95th percentile relative error for DAE. The dashed line marks relative error 0.1.

5.2 Isolated Update Results

The isolated update benchmarks measure insertion and deletion cost separately from query cost, and show how these costs vary with active dataset size. Across the isolated update benchmarks, the ANN index update is the largest DAE update component for both insertions and deletions. The DEEP10M component breakdown in Figure 5.4 illustrates this pattern for the largest dataset. The corresponding plots for the remaining datasets are provided in Appendix A.2.

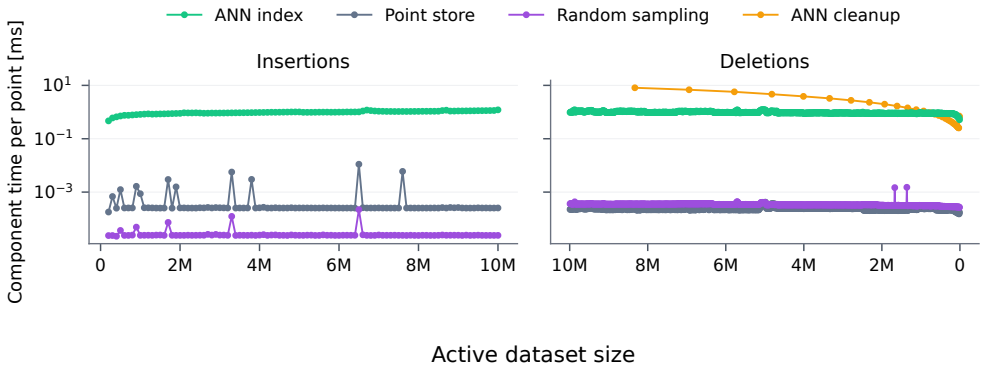

Figure 5.4: Component-level update time per point for DAE on DEEP10M. The y-axis is logarithmic.

Table 5.1: Average update time per point in the isolated update benchmarks.

Operation	Dataset	Dynamic naïve KDE [ms]	DAE [ms]
Insert	SIFT1M	0.0006	0.915
	GIST1M	0.0037	1.424
	MNIST	0.0035	0.545
	DEEP10M	0.0004	0.978
Delete	SIFT1M	0.0006	0.929
	GIST1M	0.0011	1.035
	MNIST	0.0009	0.636
	DEEP10M	0.0005	1.025

The `PointStore` insertion timings contain occasional spikes; these are less frequent and smaller in magnitude than the ANN index insertion cost. For deletions, ANN cleanup and `PointStore` compaction occur intermittently and are visible in the component timings but do not dominate the average deletion time reported in Table 5.1.

The component timings decompose the DAE update cost, while the direct comparison with dynamic naïve KDE is given by the average update time per point. For the insert-only benchmark, this is computed as the total measured insertion time divided by the total number of inserted points. For the delete-only benchmark, it is computed as the total measured deletion time divided by the total number of deleted points. For both insertions and deletions, DAE has higher average update time per point than dynamic naïve KDE across all datasets.

5.3 Sliding-Window Results

The previous benchmarks evaluate queries and updates separately. The sliding-window benchmark combines them by repeatedly updating a fixed-size active set and evaluating queries after each update step. Among the four datasets, DEEP10M has the largest number of points, and therefore provides the most informative case for assessing whether query time, update time, and relative error increase over a long sequence of updates. The DEEP10M sliding-window results are shown in Figure 5.5, where no sustained degradation is observed. Similar results for the remaining datasets are provided in Appendix A.3.

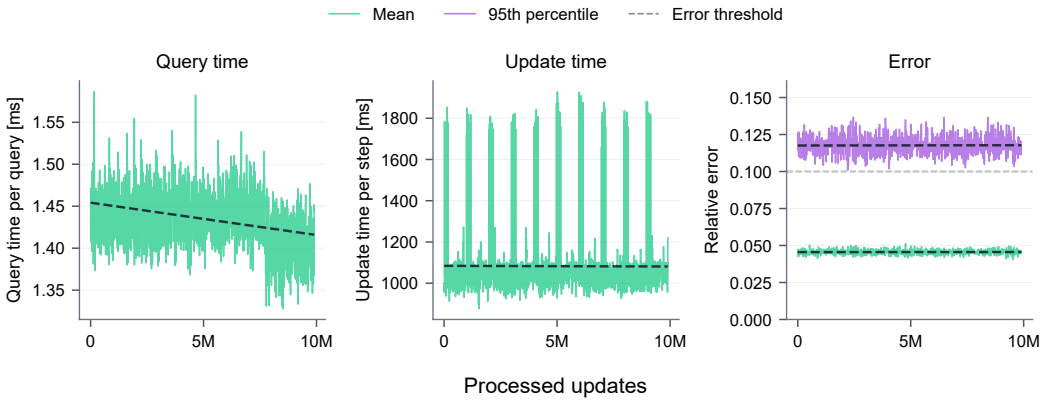

Figure 5.5: Sliding-window benchmark on the full DEEP10M dataset, where one update consists of one insertion and one deletion. The error panel shows both mean and 95th percentile relative error. Dashed lines show linear trends over the processed updates.

6

Discussion

The results show that extending DEANN to a dynamic setting is feasible, but useful only for certain workloads. In dynamic KDE, queries should be fast, while insertions and deletions should be cheap enough that maintaining the estimator is preferable to using a simpler method with slower queries but lower construction and update cost.

The Dynamic ANN Estimator improves query performance by using a dynamic ANN component to identify high-contribution points and random sampling to estimate the residual contribution. However, the ANN component that enables faster queries must also be updated after insertions and deletions, making ANN maintenance the main additional cost of the estimator. Consequently, the central question is not whether the DAE is always faster than exact dynamic naïve KDE, but when its lower query time justifies its construction and update costs.

6.1 Query Performance

The query scaling results show that DAE is not uniformly faster than exact naïve KDE, and instead show a clear crossover behavior. For small active datasets, exact dynamic naïve KDE is faster because it has little overhead and simply scans the active points. The DAE must perform an ANN query, evaluate the returned points, and sample from the residual set, so its fixed query overhead is larger. As the active dataset grows, the linear scan in exact KDE becomes increasingly expensive, while DAE grows more slowly because it evaluates only ANN-selected and randomly sampled points, at the cost of introducing approximation error.

This explains why the DAE achieves faster queries than exact KDE on SIFT1M, GIST1M, and DEEP10M, but not on MNIST, where the active dataset is too small for the query-time savings to outweigh the estimator overhead. The effect is clearest on DEEP10M, where avoiding a full scan over the larger active dataset produces a larger benefit. The results therefore indicate that the DAE is primarily useful for large active datasets where exact KDE queries dominate the workload.

The selected ANN configurations also suggest that the high-recall IP-DiskANN regime is not necessary for achieving time-efficient KDE estimates in these experiments. The goal is not necessarily to recover the exact nearest-neighbor set, but to retrieve enough high-contribution points to reduce the variance of the residual sampling estimate.

6.2 Update Costs

The main drawback of the Dynamic ANN Estimator is its update cost. Exact dynamic naïve KDE relies on a simple dataset representation, so insertions and deletions are cheap. In contrast, DAE updates affect the `PointStore` and the dynamic random sampling structure, but most of the additional cost comes from updating the IP-DiskANN component.

Insertions are expensive because each new point must be integrated into the graph used by the ANN component. This requires more work than appending a point to the exact baseline, since the new point must be connected to nearby points while preserving useful search paths. Deletions are more difficult when they are handled in place rather than only through lazy deletion. In a graph-based ANN structure, removing a point can break search paths through the graph. The affected neighborhoods therefore need to be repaired and reconnected so that queries can still reach relevant parts of the dataset, while avoiding unnecessarily long search paths or poorly connected regions.

The IP-DiskANN cleanup step and `PointStore` compaction occur infrequently, so they add little to the average update cost. The dominant cost is instead maintaining the ANN component. Overall, the DAE trades faster query times at large active sizes for higher construction and update costs compared with exact dynamic naïve KDE. It is therefore most suitable when the active dataset is large and many queries are performed between updates. For update-heavy workloads, or for datasets where the DAE does not reach a query-time crossover, exact dynamic naïve KDE remains the more appropriate method.

6.3 Workload Dependence

The practical value of the DAE depends on whether its construction and update costs can be amortized. Building the ANN component is substantially more expensive than constructing the exact dynamic naïve baseline, making DAE unattractive when the estimator is used only briefly. Its query-time advantage emerges when the estimator is reused for many queries, especially on large active datasets, but frequent insertions or deletions can still outweigh these savings because updates are more expensive than in the exact baseline.

The query, construction, and update results support this trade-off, and the corresponding break-even thresholds are provided in Appendix A.4. DAE should therefore not be viewed as universally faster than exact dynamic naïve KDE. Its advantage appears primarily for large dynamic datasets with many density queries between updates, or when the active set changes gradually while the ANN component is reused over a long period.

The results also show that the workload trade-off is dataset-dependent. DEEP10M has the largest per-query saving at the largest tested active size and tolerates the most updates per query, but its high construction cost means that it does not

have the lowest fit-amortization threshold. Instead, GIST1M reaches the lowest fit-amortization threshold among the datasets where the DAE is faster than exact KDE.

Despite GIST1M and SIFT1M both having one million base vectors, GIST1M has much higher dimensionality and a larger query-time saving. This suggests that avoiding full exact KDE scans becomes more valuable when each scan requires more expensive distance computations. MNIST does not reach a query-time crossover within the tested range, so exact dynamic naïve KDE remains preferable for that dataset in this evaluation.

6.4 Stability Under Repeated Updates

The sliding-window benchmark evaluates whether the estimator remains stable when many insertions and deletions are applied over time. Since the active size stays fixed, stable query time and stable approximation error indicate that the maintained structures continue to support the estimator after repeated updates.

The results show no clear long-term degradation in either query time or approximation error. This is important because the DAE has a high construction cost. The estimator only becomes attractive if this construction cost can be amortized over many future queries. If repeated updates degraded the estimator and forced periodic rebuilds, this amortization would be much harder, and the method would be less useful for dynamic workloads.

This also reflects positively on the IP-DiskANN component. Although insertions and deletions are expensive, the ANN graph appears to remain useful after many in-place updates. The update cost is therefore high, but the updates do not appear to gradually damage query performance or approximation quality.

Some variation in timing is visible across the sliding-window runs. Part of this variation is expected, since IP-DiskANN cleanup and `PointStore` compaction are triggered intermittently rather than at every update. At the same time, some apparent disruptions in the timing plots appear to be caused by measurement noise. For example, the DEEP10M query-time plot in Figure 5.5 appears to decrease over time, but repeated runs with identical parameters and random seed produced different fine-grained timing patterns. This suggests that small timing fluctuations should not be interpreted as systematic estimator changes.

6.5 Limitations

The implementation uses IP-DiskANN as its only dynamic ANN backend. Because IP-DiskANN dominates the update cost, the results should be interpreted as an evaluation of the DAE with this backend rather than as a general statement about all possible dynamic ANN approaches. A different backend, or a different implementation of the same backend, could change both query and update performance. The

Rust/C++ FFI integration may also affect the measured timings, since the implementation includes wrapper overhead and is constrained by the interface exposed by the IP-DiskANN codebase.

The experiments use fixed parameter configurations selected before the main benchmarks. This keeps the evaluation controlled, but it means that the full trade-off between speed and accuracy is not explored. The bandwidth, number of ANN neighbors, number of random samples, ANN parameters, cleanup threshold, and compaction threshold could all affect the results. In a long-running dynamic setting, fixed parameters may also become less suitable if the active dataset changes in size or distribution. The evaluation also does not explore active sizes beyond DEEP10M, so the behavior of the DAE on even larger datasets remains an open question.

The timing measurements are also affected by runtime variability on the shared cluster. Repeated runs with identical parameters and random seed sometimes produced different fine-grained timing patterns, which makes millisecond-scale timing effects difficult to interpret precisely. For this reason, the timing results should be interpreted mainly in terms of larger trends, such as query-time crossovers, update-cost dominance, and stability under repeated updates.

Finally, the workloads are controlled benchmarks rather than application traces. They isolate query scaling, update cost, and sliding-window stability, which makes the behavior easier to interpret. However, real workloads may contain bursty updates, changing query distributions, nonuniform deletion patterns, or many queries between update batches. Since the usefulness of the DAE depends strongly on the query-to-update ratio, different workload patterns could change whether the estimator is beneficial in practice.

6.6 Future Work

Future work should first study the parameter space more systematically. The current evaluation uses fixed configurations, but DAE has several interacting parameters that affect query time, update cost, and approximation error. Exploring these parameters more fully could identify configurations with lower update cost or better accuracy for the same query time.

A second direction is to compare alternative dynamic ANN backends. Since ANN maintenance is the dominant cost in the current implementation, backend choice is likely to have a large effect on the overall trade-off. Comparing graph-based, hashing-based, or tree-based dynamic ANN structures would help determine how much of the observed update cost is specific to IP-DiskANN.

Finally, the benchmark workloads could be extended to better reflect application scenarios. Longer dynamic runs, bursty update patterns, changing query distributions, and workloads with many queries between update batches would give a clearer picture of when dynamic DEANN is practically useful. Adaptive parameter selection is another natural extension, especially for settings where the active dataset changes in size, density, or distribution.

7

Conclusion

This thesis investigated whether DEANN can be extended from a static setting to a dynamic one that supports insertions and deletions. The resulting Dynamic ANN Estimator combines a point-store, a dynamic ANN component, and a dynamic random-sampling structure, allowing DEANN-style KDE queries to be evaluated against the current active dataset.

The evaluation shows that the DAE can reduce query time compared with exact dynamic naïve KDE when the active dataset is sufficiently large. Query-time crossovers are observed on SIFT1M, GIST1M, and DEEP10M, while MNIST does not reach a crossover within the tested range. This shows that the benefit of the DAE is strongly dataset-dependent.

The main drawback is the cost of construction and ANN maintenance. Construction, insertions, and deletions are substantially more expensive for the DAE than for the exact baseline. The method is therefore most useful when these costs can be amortized over many queries, especially in large, query-heavy dynamic workloads.

The sliding-window experiments show no clear long-term degradation in query time or approximation error over the tested update sequences. This suggests that the maintained ANN and sampling components remain effective under repeated insertions and deletions, so the estimator can continue to be used without requiring frequent rebuilds.

Overall, the results show that a dynamic extension of DEANN is feasible, but its practical benefit depends on dataset size, parameter choices, ANN backend, and query-to-update ratio.

Bibliography

[1] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. Routledge, 2018.

[2] S. Węglarczyk, "Kernel density estimation and its application," in *ITM Web of Conferences*, vol. 23, EDP Sciences, 2018, p. 00 037.

[3] A. Moreo, P. González, and J. J. del Coz, "Kernel density estimation for multiclass quantification," *Machine Learning*, vol. 114, no. 4, p. 92, 2025.

[4] W. B. March, B. Xiao, and G. Biros, "ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, A1089–A1110, 2015.

[5] M. Charikar, M. Kapralov, N. Nouri, and P. Siminelakis, "Kernel density estimation through density constrained near neighbor search," in *2020 IEEE 61st Annual Symposium on Foundations of Computer Science*, IEEE, 2020, pp. 172–183.

[6] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li, "Quality and efficiency for kernel density estimates in large data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, NY, USA: Association for Computing Machinery, 2013, pp. 433–444. [Online]. Available: `https://doi.org/10.1145/2463676.2465319`.

[7] M. Karppa, M. Aumüller, and R. Pagh, "DEANN: Speeding up kernel-density estimation using approximate nearest neighbor search," in *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics*, 2022, pp. 3108–3137.

[8] M. Karppa, *Minerva computing cluster*, Accessed: 2026-02-16. [Online]. Available: `https://git.chalmers.se/karppa/minerva`.

[9] J. Liang, Z. Song, Z. Xu, J. Yin, and D. Zhuo, *Dynamic maintenance of kernel density estimation data structure: From practice to theory*, 2024. [Online]. Available: `https://arxiv.org/abs/2208.03915`.

[10] S. Laenen, P. Macgregor, and H. Sun, "Dynamic similarity graph construction with kernel density estimation," in *International Conference on Machine Learning*, 2025.

[11] D. W. Scott and S. R. Sain, "Multidimensional density estimation," in *Handbook of Statistics*, vol. 24, Elsevier, 2005, pp. 229–261.

[12] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory*, ser. ICDT 2001, Berlin, Heidelberg: Springer, 2001, pp. 420–434.

[13] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?" In *Database Theory*, ser. ICDT '99, Berlin, Heidelberg: Springer, 1999, pp. 217–235.

[14] A. Gray and A. Moore, "N-body problems in statistical learning," in *Advances in Neural Information Processing Systems*, vol. 13, MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2000/file/7385db9a3f11415bc0e9e2625fae3734-Paper.pdf.

[15] D. Lee, A. Moore, and A. Gray, "Dual-tree fast gauss transforms," in *Advances in Neural Information Processing Systems*, vol. 18, MIT Press, 2005. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2005/file/9087b0efc7c7acd1ef7e153678809c77-Paper.pdf.

[16] P. Ram, D. Lee, W. March, and A. Gray, "Linear-time algorithms for pairwise statistical problems," in *Advances in Neural Information Processing Systems*, vol. 22, Curran Associates, Inc., 2009. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2009/file/2421fcb1263b9530df88f7f002e78ea5-Paper.pdf.

[17] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[18] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[19] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnaswamy, and R. Kadekodi, "DiskANN: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[20] M. Aumüller, E. Bernhardsson, and A. J. Faithfull, *ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms*, 2018. [Online]. Available: https://arxiv.org/abs/1807.05614.

[21] X. Zhao, Y. Tian, K. Huang, B. Zheng, and X. Zhou, "Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces," *Proceedings of the VLDB Endowment*, vol. 16, no. 8, pp. 1979–1991, 2023. [Online]. Available: https://doi.org/10.14778/3594512.3594527.

[22] H. Xu, M. Dobson Manohar, P. A. Bernstein, B. Chandramouli, R. Wen, and H. V. Simhadri, *In-place updates of a graph index for streaming approximate nearest neighbor search*, 2025. [Online]. Available: https://arxiv.org/abs/2502.13826.

[23] A. Backurs, P. Indyk, and T. Wagner, "Space and time efficient kernel density estimation in high dimensions," in *Advances in Neural Information Processing Systems*, vol. 32, 2019, pp. 15 773–15 782.

[24] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98, New York, NY, USA: Association for Computing Machinery, 1998, pp. 604–613. [Online]. Available: https://doi.org/10.1145/276698.276876.

[25] A. Andoni and I. Razenshteyn, *Optimal data-dependent hashing for approximate near neighbors*, 2015. [Online]. Available: https://arxiv.org/abs/1501.01062.

[26] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[27] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011. [Online]. Available: https://inria.hal.science/inria-00514462.

[28] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, *FreshDiskANN: A fast and accurate graph-based ANN index for streaming similarity search*, 2021. [Online]. Available: https://arxiv.org/abs/2105.09613.

[29] B. Huang, Z. Song, O. Weinstein, J. Yin, H. Zhang, and R. Zhang, *A dynamic low-rank fast gaussian transform*, 2024. [Online]. Available: https://arxiv.org/abs/2202.12329.

[30] M. Karppa, M. Aumüller, and R. Pagh, *DEANN: Speeding up kernel density estimation using approximate nearest neighbor search*, Code repository. Accessed: 2026-02-16, 2022. [Online]. Available: https://github.com/mkarppa/deann/.

[31] H. V. Simhadri et al., *DiskANN: Graph-structured indices for scalable, fast, fresh and filtered approximate nearest neighbor search*, Code repository. Accessed: 2026-04-24. [Online]. Available: https://github.com/microsoft/DiskANN.

[32] TEXMEX, *Evaluation of approximate nearest neighbors: Large datasets*, Accessed: 2026-05-26. [Online]. Available: http://corpus-texmex.irisa.fr/.

[33] 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. [Online]. Available: https://doi.org/10.1109/5.726791.

[34] Yandex Research, *Benchmarks for billion-scale similarity search*, Accessed: 2026-05-26. [Online]. Available: https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search.

[35] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *2016 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, 2016, pp. 2055–2063. [Online]. Available: https://doi.org/10.1109/CVPR.2016.226.

A

Additional Results

A.1 Parameter-Sweep Results

Tables A.1 and A.2 show additional parameter-sweep results for SIFT1M and MNIST.

Table A.1: Optimal parameters for $\mu = 10^{-5}$ as the active dataset size changes.

		Optimal parameters		
Dataset	n	Bandwidth h	k	m
SIFT1M	10 000	77.14	57	320
SIFT1M	100 000	77.68	40	2560
SIFT1M	500 000	77.97	40	5120
SIFT1M	1 000 000	78.03	80	5120
MNIST	1 000	426.15	40	28
MNIST	5 000	422.99	57	160
MNIST	20 000	419.90	80	453
MNIST	40 000	419.06	80	1280
MNIST	59 000	418.10	113	1280

Table A.2: Optimal parameters for SIFT1M and MNIST at the largest active dataset size as the target KDE value changes.

			Optimal parameters		
Dataset	n	Target μ	Bandwidth h	k	m
SIFT1M	1 000 000	10^{-2}	147.25	0	320
SIFT1M	1 000 000	10^{-3}	111.98	0	905
SIFT1M	1 000 000	10^{-4}	91.46	0	5120
SIFT1M	1 000 000	10^{-5}	78.03	80	5120
MNIST	59 000	10^{-2}	788.12	0	226
MNIST	59 000	10^{-3}	615.34	0	1810
MNIST	59 000	10^{-4}	504.42	80	1280
MNIST	59 000	10^{-5}	418.10	113	1280

A.2 Update Component Results

Figures A.1, A.2, and A.3 show the component-level DAE update times for the remaining isolated update benchmarks.

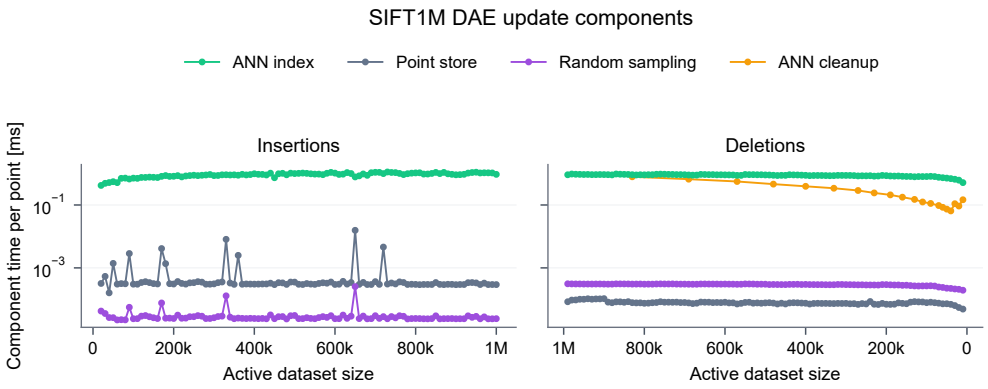

Figure A.1: Component-level update time per point for DAE on SIFT1M. The y-axis is logarithmic.

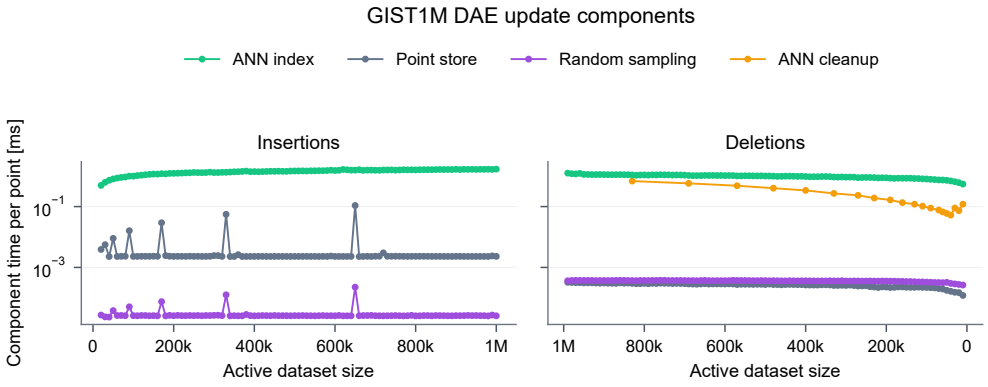

Figure A.2: Component-level update time per point for DAE on GIST1M. The y-axis is logarithmic.

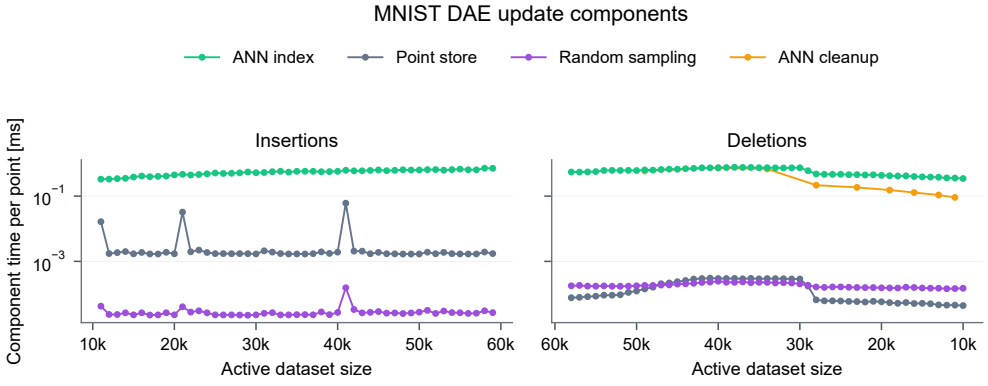

Figure A.3: Component-level update time per point for DAE on MNIST. The y-axis is logarithmic.

A.3 Sliding-Window Results

Figures A.4, A.5, and A.6 show the sliding-window benchmark results for the remaining datasets. These figures complement the DEEP10M sliding-window results in Chapter 5 and show whether update time, query time, and relative error remain stable over repeated updates.

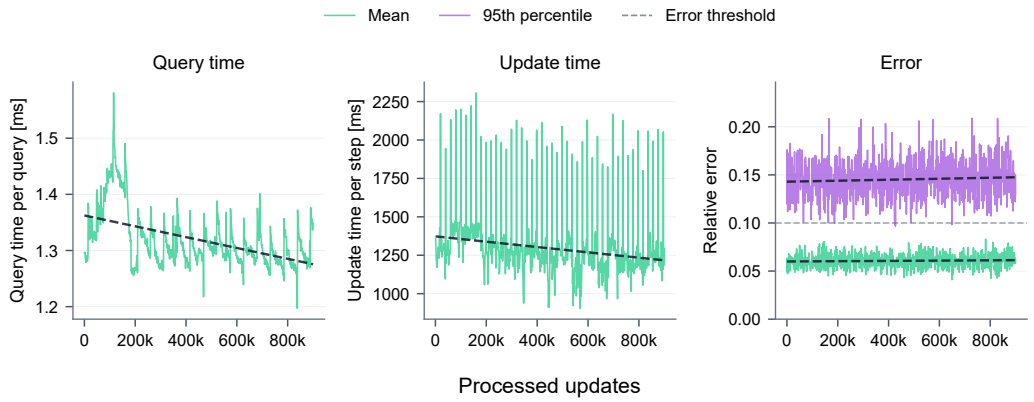

Figure A.4: Sliding-window benchmark on SIFT1M, where one update consists of one insertion and one deletion. The panels show update time, query time per query, and relative error over the processed updates. Dashed lines show linear trends.

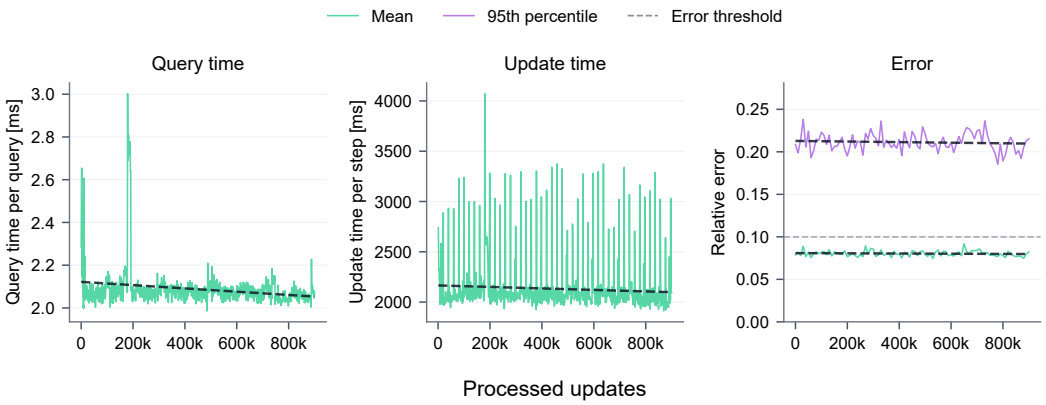

Figure A.5: Sliding-window benchmark on GIST1M, where one update consists of one insertion and one deletion. The panels show update time, query time per query, and relative error over the processed updates. Dashed lines show linear trends.

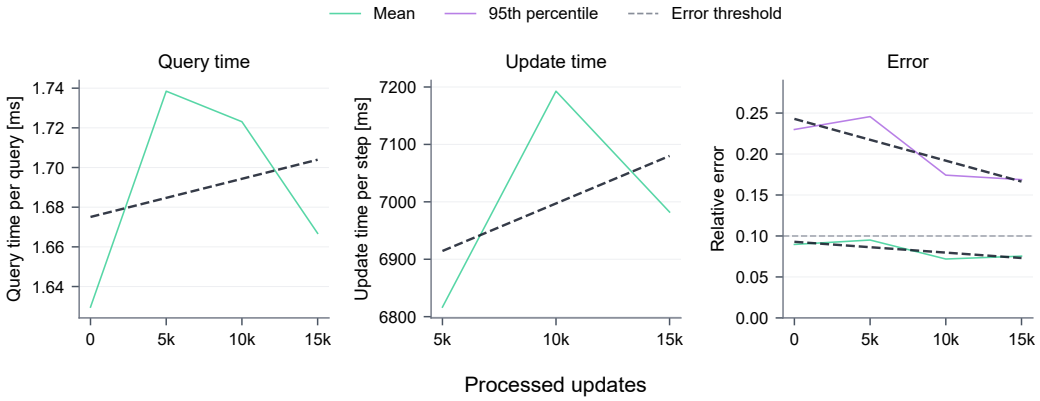

Figure A.6: Sliding-window benchmark on MNIST, where one update consists of one insertion and one deletion. The panels show update time, query time per query, and relative error over the processed updates. Dashed lines show linear trends.

A.4 Break-Even Thresholds

The query and update benchmarks measure construction time, query time, and update time separately. As a supplementary timing-only comparison, these measurements can be combined to estimate when the lower query time of DAE compensates for its higher construction or update cost. The resulting thresholds are shown in Tables A.3 and A.4.

Here, $\mathrm{DAE}_{\texttt{fit}}$ and $\mathrm{Naïve}_{\texttt{fit}}$ denote the measured construction times, $\mathrm{DAE}_{\texttt{query}}$ and $\mathrm{Naïve}_{\texttt{query}}$ denote query time per query, and $\mathrm{DAE}_{\texttt{update}}$ and $\mathrm{Naïve}_{\texttt{update}}$ denote update time per point.

The number of queries needed to amortize the construction-time overhead of DAE is estimated as

$$\frac{\mathrm{DAE}_{\texttt{fit}} - \mathrm{Naïve}_{\texttt{fit}}}{\mathrm{Naïve}_{\texttt{query}} - \mathrm{DAE}_{\texttt{query}}}.$$

The maximum number of updates per query before the update overhead offsets the query-time saving is estimated as

$$\frac{\mathrm{Naïve}_{\texttt{query}} - \mathrm{DAE}_{\texttt{query}}}{\mathrm{DAE}_{\texttt{update}} - \mathrm{Naïve}_{\texttt{update}}}.$$

For Table A.4, $\mathrm{DAE}_{\texttt{update}}$ and $\mathrm{Naïve}_{\texttt{update}}$ are taken from the corresponding insert-only or delete-only benchmark.

These thresholds are not general guarantees. They depend on the measured dataset, active size, parameter configuration, and hardware, and they do not account for approximation error. They are included only to illustrate the workload balance required for DAE to outperform exact dynamic naïve KDE in the measured configurations.

Table A.3: Number of queries needed for DAE to amortize fit-time overhead relative to naïve KDE at the largest tested active size.

Dataset	Active size	Query saving [ms]	Queries needed
DEEP10M	10 000 000	51.07	223 827
GIST1M	1 000 000	9.50	147 049
SIFT1M	1 000 000	2.92	261 852
MNIST	59 000	–	Never

Table A.4: Maximum number of updates per query before naïve KDE becomes faster than DAE, using largest-size query times and average update time per point.

Dataset	Inserts per query	Deletes per query
DEEP10M	up to 52	up to 49
GIST1M	up to 6	up to 9
SIFT1M	up to 3	up to 3
MNIST	Never	Never