

# KNN Implementation Details Can Dramatically Change Performance: An Example from Cover Trees

**Amol Khanna**  
CrowdStrike  
amol.khanna@crowdstrike.com

**Edward Raff**  
CrowdStrike  
University of Maryland, Baltimore County  
Syracuse University  
edward.raff@crowdstrike.com  
raff.edward@umbc.edu  
edraff@syr.edu

## Abstract

Nearest neighbor search algorithms are widely used and have a long history, but we find that key implementation details are seldom reviewed in articles about them. These details can materially impact performance, and thus change the conclusions one makes about a given algorithm’s superiority. In this article, we highlight these impacts by reviewing the Cover Tree as an example, producing RUSTKNN, a unified Rust implementation of three construction variants of the Cover Tree with three query modes. Our analysis reveals: (1) the optimal query strategy depends on  $k$  and on the dataset—single-tree search is fastest at  $k=1$  on three of our four largest datasets, while dual-tree search is fastest at  $k \geq 5$  on the three largest—and choosing the wrong query mode can cost up to a  $7.5\times$  factor in runtime; (2) the cover tree base parameter, set to 2 in theory and 1.3 in practice, is not a safe default for dual-tree search: 1.3 is optimal on only one of seven datasets, and on covtype it is  $1.6\times$  slower than the optimum; (3) among five micro-optimizations, only early-exit distance computation has a consistent effect, and that effect grows with dimension (none at  $d=32$ ,  $1.31\times$  at  $d=54$ , and  $1.49\times$  at  $d=784$ ); and (4) all-nearest-neighbor and held-out benchmarks yield different implementation rankings. All main timing tables report medians over 20 independent allocations with bootstrap 95% confidence intervals. RUSTKNN is open source at <https://github.com/CrowdStrike/RustKNN>.

## 1 Introduction

Nearest neighbor algorithms, and data structures to accelerate/improve them, exist in many forms. There are algorithms for pruning the search space like the KD-tree [1], sketches for calculating approximations of distance functions faster [2], and more. This line of research has been highly impactful in real-world systems, from information retrieval to clustering/visualization and even in Large Language Models via retrieval-augmented generation and vector databases. While *approximate* methods have dramatically gained popularity in recent years, exact metric-space  $k$ -NN remains necessary when the distance itself is the object of interest or approximate recall is unacceptable—for example, in cases like anomaly detection over raw features, protein-structure comparison under graph kernels, or retrieval under Wasserstein distance. However, when we survey papers (see Appendix K) in this broad area, we find that few consider basic details, such as distinguishing between a hold-out set’s query performance and the original training data (26%). Fewer still report *both* distance-comparison counts and runtime (15%), even though the two are not always correlated, document micro-optimization details (11%)—the low-level implementation choices we isolate in Section 4.2—or batch vs. single-instance queries (10%).

We argue that papers evaluating the performance of nearest neighbor dependent systems need to document and evaluate these factors carefully, or give a justified reason why only a subset matters for their use case. We support that argument by developing RUSTKNN as an example, re-evaluating the methods behind the seminal cover tree data structure—a structure with multiple proposed variants and one of the few with both empirical success and theoretical grounding for reducing nearest neighbor queries. We re-implement a cache-efficient cover tree with three query settings (single, batch, and dual-tree). Working in a compiled language allows both effective runtime evaluation and easy isolation of six micro-optimizations used in the literature (five of which we ablate), along with poorly documented nuances of the cover tree “base” parameter, which to our knowledge has never been independently tested.

Our contributions are:

1. **Optimization ablation.** We quantify the individual and cumulative impact of five micro-optimizations via feature-flag ablation, revealing that early-exit distance computation is the only one with a consistent effect, and that its benefit grows with dimension (none on corel,  $d=32$ ;  $1.31\times$  on covtype,  $d=54$ ; and  $1.49\times$  on mnist,  $d=784$ ), while the traversal-level optimizations sit within about 10% of  $1.0\times$  (Section 4.2).
2. **Query mode  $\times k$  interaction.** We show that the optimal query strategy depends on both  $k$  and the dataset: single-tree search is fastest at  $k=1$  on mnist yet  $3.3\times$  slower than dual-tree search at  $k=10$  on that same dataset, and choosing the wrong query mode costs up to  $7.5\times$  in runtime (Section 5.2).
3. **Base parameter analysis.** We show that the conventional base 1.3 is not a safe default for dual-tree search: the optimum is 2.0 on four of seven datasets at  $k=1$  (five at  $k=10$ ) and 3.0 on the two smallest, and the default is  $1.6\times$  slower than the optimum on covtype, the largest dataset (Section 5.3).
4. **All-NN vs. held-out divergence.** We demonstrate that the standard all-nearest-neighbor benchmark produces different implementation rankings than held-out evaluation, with implications for how the community evaluates nearest neighbor methods (Section 5.4).

In doing so, we find that no single method dominates; the results are dependent on dataset and query settings, and thus far more complicated and involved than the literature has previously implied. This complexity exists within just the cover tree literature alone, and we find it instructive to the questions we may have of the broader literature.

Sections 2–5 cover related work, background, our implementation, and the evaluation.

## 2 Related Work

**Nearest neighbor data structures.** Spatial indexing methods for nearest neighbor search can be organized along two axes: exact versus approximate, and coordinate-based versus general metric space. Among exact methods for coordinate spaces, KD-trees [3] achieve  $O(\log n)$  query time in low dimensions but degrade to  $O(n)$  as dimensionality grows, and their axis-aligned splits tie them to a coordinate representation; ball trees [4] improve on KD-trees through hierarchical ball decomposition, extending efficient search to moderate dimensions; because they partition by distance to centers rather than along axes, they apply to any proper metric. For general metric spaces, vantage-point trees [5] use binary partitions around single pivot points but offer only  $O(n)$  worst-case query time. Navigating nets [6] introduced the theoretical framework of the doubling dimension  $c$  (also called the “expansion constant”), which cover trees [7] made practical with an  $O(c^6 \log n)$  query algorithm, later improved by Izbicki and Shelton [8] with simplified and nearest ancestor variants (though a history of nuanced theoretical runtime errors and fixes [9–12] is beyond this study), tree merging for parallel construction, and cache-optimized packing. On the approximate side, locality-sensitive hashing (LSH) [13] provides sublinear search but requires metric-specific hash families, while hierarchical navigable small world graphs (HNSW) [14] achieve state-of-the-art recall–speed tradeoffs for approximate search but offer no exactness guarantees. Cover trees are a *practical* structure combining exact search, arbitrary metric spaces, and query complexity parametrized by intrinsic rather than ambient dimensionality. They are not alone in that combination—navigating nets, cited above, share it—but they are the variant with the broadest empirical track record, which is why we take them as our case study.

**Implementations and libraries.** The original cover tree C++ reference implementation [7], last updated in 2014, is single-threaded with distance-level optimizations (batched computation, aligned memory) but no tree-level cache packing. mlpack [15] provides the most actively maintained cover tree implementation, including dual-tree traversal and extensive test coverage. FLANN [16] offers randomized KD-trees and is widely used in robotics and computer vision, though it targets coordinate spaces only. Among approximate and GPU-accelerated systems, FAISS [17] scales to billions of vectors with quantization-based search on GPUs; Annoy [18] uses random projection trees with memory-mapped files for production use; and ScaNN [19] employs learned anisotropic vector quantization. Existing cover tree implementations span C++, Haskell, and Java [20, 21] (e.g., JSAT & WEKA) -but no two of them implement the same set of variants, query modes, and micro-optimizations, and none is Rust-native. Critically, cross-language and cross-codebase comparisons confound algorithmic effects (e.g., nearest ancestor vs. simplified) with implementation effects (e.g., compiler optimizations, memory layout, distance computation strategy), making it difficult to attribute performance differences to their true cause. Our work focuses on cover trees specifically, but we believe the insight of delineating implementation details applies to other search structures and can reveal new research directions; Section 5.5 gives direct evidence for a second structure. RUSTKNN is the first Rust-native cover tree, integrating directly with the Rust data-processing ecosystem without foreign-function interface overhead; more importantly for this study, it implements all variants and query modes in a single codebase, enabling controlled comparisons that isolate one factor at a time. These relate to issues in both maintainability and replicability [22], to which a larger literature is relevant but beyond our scope [23–29, 26].

**Rust in the ML ecosystem.** Rust adoption in machine learning infrastructure has accelerated for performance-critical components. Hugging Face tokenizers [30] use a Rust backend; candle [31] and burn [32] provide Rust deep learning frameworks; Qdrant [33] and LanceDB [34] are vector databases built in Rust using HNSW for approximate search; and kiddo [35] and hora [36] provide KD-tree and approximate nearest neighbor search, respectively. These projects address approximate search or coordinate-specific structures. RUSTKNN provides *exact* search over *arbitrary* metric spaces with the full algorithmic suite from Izbicki and Shelton [8], filling a gap in the Rust ML toolchain.

### 3 Background

#### 3.1 Metric Spaces and the Doubling Dimension

A *metric space*  $(\mathcal{X}, d)$  is a set  $\mathcal{X}$  equipped with a distance function  $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_{\geq 0}$  satisfying non-negativity, identity of indiscernibles ( $d(x, y) = 0 \iff x = y$ ), symmetry, and the triangle inequality. Given a finite dataset  $S \subset \mathcal{X}$ , the *doubling constant*  $c$  is the minimum value such that every ball of radius  $r$  centered on a point in  $S$  can be covered by at most  $c$  balls of radius  $r/2$  [6, 37]. The *doubling dimension* is  $\dim_D = \log_2 c$ . Crucially,  $c$  is a property of the dataset, not of the ambient space: data lying on a low-dimensional manifold embedded in  $\mathbb{R}^D$  can have small  $c$  even when  $D$  is large. Cover tree query complexity is  $O(c^6 \log n)$ , parametrized by  $c$  rather than  $D$ —this is what makes cover trees effective in high dimensions, unlike KD-trees whose performance degrades exponentially with  $D$ .

#### 3.2 Cover Tree Invariants

A cover tree is a leveled tree in which each node stores a single data point. Intuitively, each level represents a different *resolution*: the root sees the dataset at coarse scale, and deeper nodes refine the picture. Three structural invariants govern parent-child relationships (Definition 1 of Beygelzimer et al. [7]):

1. **Leveling invariant.** Every node  $p$  has an integer  $\text{level}(p)$ , and every child  $q$  satisfies  $\text{level}(q) = \text{level}(p) - 1$ . Each level down is a finer resolution.
2. **Covering invariant.** Define the covering distance  $\text{covdist}(p) = b^{\text{level}(p)}$ . For each child  $q$  of  $p$ ,  $d(p, q) \leq \text{covdist}(p)$ . No child lies beyond its parent’s covering radius.
3. **Separating invariant.** Define the separation distance  $\text{sepdist}(p) = b^{\text{level}(p)-1}$ . For distinct children  $q_1, q_2$  of  $p$ ,  $d(q_1, q_2) > \text{sepdist}(p)$ . No two siblings are too close together.

The base parameter  $b$  controls the ratio between covering and separating distances. The original formulation uses  $b = 2$ , but Beygelzimer et al. [7] found empirically that  $b = 1.3$  reduces distance computations by 10–71% on benchmark datasets. Our experiments default to  $b = 1.3$ .

Each node also caches  $\text{maxdist}(p) = \max_{q \in \text{descendants}(p)} d(p, q)$ , the greatest distance from  $p$  to any of its descendants, which is upper bounded by  $\text{covdist}(p) \cdot b/(b-1)$  (a geometric series over descendant levels; this equals  $2 \cdot \text{covdist}(p)$  when  $b=2$  and  $\approx 4.3 \cdot \text{covdist}(p)$  at the default  $b=1.3$ ). This value enables the key pruning rule during queries: a subtree rooted at  $q$  can be skipped entirely when  $d(\text{query}, q) - \text{maxdist}(q) > d_{\text{best}}$ , because even the closest descendant of  $q$  must be farther than the current best candidate.

From Beygelzimer et al. [7], the maximum number of children per node is  $O(c^4)$  and the maximum tree depth is  $O(c^2 \log n)$ .

### 3.3 Algorithmic Variants

The original cover tree [7] was defined as an implicit infinite structure in which every internal node also appears as its own child—the *nesting invariant*. The explicit representation stores  $O(n)$  nodes. Izbicki and Shelton [8] observed that the nesting invariant is unnecessary for the runtime analysis and introduced two improved variants that eliminate self-children, reducing the node count to exactly  $n$ .

**Simplified cover tree construction.** The simplified insertion of Izbicki and Shelton [8] builds a tree via depth-first search: starting from the root, the algorithm finds the first child whose covering distance accommodates the new point and recurses; if no child qualifies, the point becomes a new child at level  $-1$ . If the new point lies beyond the root’s covering distance, the root level is raised. The tree contains exactly  $n$  nodes, one per data point, eliminating the self-children of the original formulation. Construction runs in  $O(c^6 n \log n)$  time.

**Nearest ancestor cover tree construction.** This variant adds a fourth invariant: for every point  $p$  with ancestor  $q_1$  and sibling  $q_2$  of  $q_1$ , we require  $d(p, q_1) \leq d(p, q_2)$ —every point’s ancestor at each level is the closest possible. Insertion sorts children by distance to the new point and invokes a *rebalance* procedure that extracts descendants violating the invariant and reinserts them under the new point. Construction is costlier, but the resulting tree has tighter  $\text{maxdist}$  values, leading to more aggressive pruning; the trees are also deeper, so querying is not always quicker than the simplified variant.

**Parallel cover tree construction.** Two cover trees are merged by matching children between them at each level, enabling parallel construction: partition the dataset across processors, build independent trees, then merge via binary reduction in  $O(\log p)$  rounds. Roughly 1% of distance computations occur during the merge; the bottleneck is cache performance in the parallel construction phase. We implement this but do not time it in our experiments; Table 9 reports its distance counts.

**Single-tree query.** Nearest-neighbor search traverses the tree depth-first from the root, visiting children in order of increasing distance to the query. A child  $q$  is pruned when  $d(\text{query}, q) - \text{maxdist}(q) > d_{\text{best}}$ , yielding  $O(c^6 \log n)$  runtime.

**Batch single-tree query.** This algorithm, derived from the code of Beygelzimer et al. [7] but not described in detail in their paper, organizes the query set into its own cover tree and maintains *cover sets*—lists of candidate reference nodes, indexed by scale level—that are passed top-down through the query tree hierarchy. At each scale level, three cases arise (Figure 20 in the appendix): (i) *reference descent*—expand each candidate reference node by replacing it with its children, computing distances and pruning; (ii) *query split*—when a query node has children at the current scale, each query child receives a *filtered copy* of the parent’s cover sets with distances recomputed and entries pruned against the child’s bound; the parent point continues as a “virtual self-child,” inheriting the cover sets directly; (iii) *base case*—at the leaf scale, score remaining candidates. The key mechanism is *cover-set inheritance*: once a reference candidate is pruned from a query node’s cover set, no descendant query point will ever see it. A mutable upper bound threaded through the recursion means any improvement from scoring one query point immediately tightens the threshold for siblings and descendants. The amortization scales with query tree depth: in all-NN mode (query = reference) the query tree has full depth  $O(c^2 \log n)$  with many split points, maximizing amortization,

whereas a small held-out query set yields only 2–3 levels, reducing most queries to independent base-case evaluations.

**Dual-tree traversal.** Dual-tree traversal [38, 39] simultaneously descends query and reference trees, pruning entire (query-subtree, reference-subtree) pairs via the triangle inequality—a joint pruning that can discard more work per distance computation than batch single-tree’s per-query bounds, at the cost of more bookkeeping per node pair. We summarize this algorithm in Figure 21 in the Appendix.

**Why interactions matter.** These three query modes, combined with the base parameter  $b$  and micro-optimizations, create a multi-dimensional performance landscape. As we show in Section 5, the optimal configuration depends on  $k$ , dataset geometry, and evaluation protocol in ways that are not apparent from any single comparison.

## 4 Implementation and Optimization Ablation

### 4.1 Implementation Overview

RUSTKNN comprises approximately 13,000 lines of Rust organized around a generic `Distance<T>` trait that enables arbitrary metric spaces. After cover tree construction, pointer-based trees are converted to a contiguous `PackedCoverTree` in depth-first order (a simpler alternative to the van Emde Boas layout of Izbicki and Shelton [8]) for cache locality: each packed node occupies  $\sim 40$  bytes, fitting within a single 64-byte cache line. The dual-tree traversal adapts the framework of Curtin et al. [39] to cover trees without self-children, using a three-phase expansion with triangle inequality pre-filters (details in Appendix C).

### 4.2 Optimization Ablation

Five micro-optimizations are implemented as compile-time feature flags, enabling controlled ablation experiments.<sup>1</sup> These affect query algorithms and are evaluated in Table 1:

1. **Early-exit distance:** Euclidean distance computation with four independent accumulators and periodic bound checks; exits early when the partial sum exceeds the pruning bound, avoiding the full  $O(d)$  computation.
2. **Halfsort:** Partial sort of children,  $O(n)$  versus  $O(n \log n)$  full sort.
3. **SmallVec:** Stack-allocated child arrays for the typical 3–5 children per node at base 1.3.
4.  $k=1$  **fast path:** Dedicated single-comparison path in the  $k$ -NN state machine.
5. **Triangle pre-filters:** Each reference child caches its parent distance  $d_{\text{parent}}(r_c)$  at construction time. Before computing  $d(q, r_c)$ , the triangle inequality gives a free lower bound  $|d(q, r_p) - d_{\text{parent}}(r_c)|$  using the already-known parent score  $d(q, r_p)$ . If this lower bound minus the child’s `maxdist` exceeds the current  $k$ th-best distance, the entire subtree is skipped without a metric evaluation.

Table 1 shows the slowdown factor when each optimization is individually disabled, measured at  $k=1$  (all-NN benchmark) on five datasets chosen to span the dimensionality axis (pima  $d=8$ , pendigits  $d=16$ , corel  $d=32$ , covtype  $d=54$ , mnist  $d=784$ ), since the headline question here—whether an optimization’s benefit scales with  $d$ —is a claim about that axis. We exclude the five smallest UCI sets ( $n \leq 351$ ): their millisecond-scale runtimes are dominated by measurement noise, as the intervals below make concrete. Each cell is the median over **20 independent SLURM allocations**, with a bootstrap 95% confidence interval; Section 5 (Methodology) explains why we report between-allocation rather than within-allocation spread. The “all off” row disables all five optimizations simultaneously. Full per-dataset, per- $k$  ablation tables appear in Appendix D.

<sup>1</sup> A sixth optimization—sorted-vector batch tracking—replaces a hash map with a sorted vector for mapping held-out batch results back to their inputs. Because the ablation infrastructure targets the all-NN benchmark in dual-tree mode (RUSTKNN Packed), this batch-only optimization is not evaluated; its impact is expected to scale with the number of concurrent query points in batch mode.

Table 1: Ablation: slowdown ( $\times$  baseline) when each optimization is disabled ( $k=1$ , all-NN, base 1.3). Median [95% CI] over 20 independent allocations. Values  $>1$  mean the optimization helps; intervals spanning 1.0 indicate no reliable effect. Computed from the same pooled sample as the waterfall in Table 7, so the “all off” row here and that table’s “baseline” row are identical by construction.

| Disabled optimization | pima<br>( $d=8$ ) | pendigits<br>( $d=16$ ) | corel<br>( $d=32$ ) | covtype<br>( $d=54$ ) | mnist<br>( $d=784$ ) |
|-----------------------|-------------------|-------------------------|---------------------|-----------------------|----------------------|
| Early-exit distance   | 0.89 [0.73, 1.24] | 0.82 [0.63, 0.99]       | 0.98 [0.97, 1.00]   | 1.31 [1.27, 1.36]     | 1.49 [1.39, 1.73]    |
| Halfsort              | 1.28 [1.02, 1.42] | 0.96 [0.75, 1.10]       | 1.10 [1.09, 1.11]   | 1.03 [1.03, 1.04]     | 1.02 [1.02, 1.03]    |
| SmallVec              | 0.97 [0.89, 1.09] | 0.65 [0.61, 0.95]       | 0.99 [0.98, 1.04]   | 1.01 [1.00, 1.02]     | 1.02 [1.01, 1.03]    |
| $k=1$ fast path       | 1.06 [0.95, 1.28] | 0.78 [0.69, 0.99]       | 1.04 [0.97, 1.06]   | 1.03 [1.03, 1.04]     | 1.01 [1.01, 1.02]    |
| Triangle pre-filter   | 1.40 [1.28, 1.52] | 0.75 [0.71, 0.79]       | 1.01 [0.98, 1.03]   | 0.98 [0.97, 0.99]     | 1.03 [0.96, 1.04]    |
| All off               | 1.11 [0.99, 1.31] | 0.78 [0.72, 0.83]       | 1.06 [1.03, 1.07]   | 1.33 [1.31, 1.34]     | 1.58 [1.39, 1.60]    |

Early-exit distance is the one optimization with a consistent effect, and its benefit grows with dimension:  $1.31\times$  on covtype ( $d=54$ ) and  $1.49\times$  on mnist ( $d=784$ ), but none on corel ( $d=32$ , CI [0.97, 1.00]). The effect is purely per-call work: disabling early exit leaves the dual-tree distance count unchanged on every ablation dataset (159.6M calls on corel either way), so the traversal is identical and only the cost of each rejected candidate differs. In fact none of the five flags changes the count by more than 0.1%; their time effects are constant-factor effects.

The confidence intervals make the noise floor explicit, which is essential to reading the table correctly. On corel, covtype, and mnist they are tight, so the effects there are measurable. On pima and pendigits they are not: their per-allocation timings are *bimodal*—on pendigits, allocations of the same binary cluster near 170 ms or near 260 ms, and the same split appears in an independent earlier sweep—so a ratio of two medians reflects which mode each binary’s allocations happened to land in rather than the optimization. The sub-1.0 entries in those columns (e.g., pendigits SmallVec  $0.65\times$ ) are of this kind. Conclusions about micro-optimizations on small datasets are therefore not supportable, and we draw none.

Among the other optimizations, none has a consistent effect: halfsort, SmallVec, the  $k=1$  fast path, and the triangle pre-filter are within 4% of  $1.0\times$  on covtype and mnist, and on corel only halfsort ( $1.10\times$ ) exceeds that. Differences of this size also arise between builds that should behave identically—for example, disabling the  $k=1$  fast path makes corel’s  $k\geq 5$  queries 6–9% faster (Table 8), although that path is never taken at  $k>1$ —so we treat effects of up to about 10%, corel’s halfsort included, as code-layout noise rather than as properties of the optimization.

## 5 Experiments

### 5.1 Experimental Setup

We describe the hardware and software setup for our experiments in Appendix B.

**Datasets.** Table 2 lists the 12 benchmark datasets. Eleven originate from the UCI Machine Learning Repository [40]; MNIST is from the original LeCun distribution [41]. Sizes span three orders of magnitude ( $n = 150$  to  $581,012$ ) and dimensionalities range from 4 to 784, providing broad coverage of the regimes where cover trees are applicable. The smallest sets are included for two reasons: coverage of the  $(n, d)$  plane, and direct comparability with the seminal cover tree evaluations [7, 8], which use exactly these sets—re-evaluating that literature under controlled conditions is part of this paper’s purpose. They are not the high-performance target, and as Section 4.2 quantifies, their millisecond-scale timings are noise-dominated; the performance conclusions rest on corel, covtype, and mnist. Appendix J adds non-tiny synthetic data ( $n=20,000$ ) whose intrinsic dimension is known by construction, which real datasets cannot provide. All experiments use the Euclidean ( $L_2$ ) distance.

**Methodology.** Every configuration (dataset  $\times$  algorithm  $\times$  variant  $\times$  query mode  $\times k$ ) is executed as **20 independent SLURM allocations**—20 separately scheduled jobs, not 20 repetitions inside one job. We reduce each allocation to one number (the median of its internal timed runs) and report

Table 2: Benchmark datasets used in all experiments.  $n$  is the number of points and  $d$  is the dimensionality.

| Dataset    | $n$ | $d$ | Source | Dataset   | $n$     | $d$ | Source |
|------------|-----|-----|--------|-----------|---------|-----|--------|
| iris       | 150 | 4   | UCI    | optdigits | 5,620   | 64  | UCI    |
| wine       | 178 | 13  | UCI    | pendigits | 10,992  | 16  | UCI    |
| glass      | 214 | 9   | UCI    | letter    | 20,000  | 16  | UCI    |
| bupa       | 345 | 6   | UCI    | corel     | 37,749  | 32  | UCI    |
| ionosphere | 351 | 34  | UCI    | mnist     | 60,000  | 784 | LeCun  |
| pima       | 768 | 8   | UCI    | covtype   | 581,012 | 54  | UCI    |

the *median over the 20 allocations* with a bootstrap 95% confidence interval (4,000 resamples, seeded).

This design captures a distinction that matters more than we initially expected: repeated runs within one allocation agree to 1–2%, but two allocations of a byte-identical binary can differ by 16–29% on small, fast datasets; our intervals report the between-allocation component. Section 4.2 shows a case where this is decisive. Wall-clock time is measured with `std::time::Instant` (Rust) and `std::chrono::high_resolution_clock` (C++). Construction and query times are reported separately; query time is further decomposed into tree-build, traversal, and result-collection phases where applicable. Distance computation counts are obtained from a separately instrumented binary (compiled with `--features instrument`), which increments a thread-local counter on every call to the distance function—including early-exited calls that return before computing all dimensions. Because the counter counts *invocations*, not full  $O(d)$  evaluations, early-exit distance reduces the per-call *work* but not the count; in the dual-tree traversal it leaves the number of invocations unchanged (Section 4.2). Table 9 reports counts for the dual-tree all-NN mode, the mode the ablation (Section 4.2) measures. Results were collected in several measurement *campaigns* (separately queued batches of allocations); between campaigns, medians can shift by 20–40% due to node placement. Every ranking or ratio from which we draw a conclusion is computed within a single table and therefore within a single campaign. Absolute large-dataset times also depend on node load: on an idle node, an mnist all-NN slows  $1.4\text{--}1.5\times$  when 15 copies share the node and  $2.0\text{--}2.1\times$  with 60, about equally for RUSTKNN and C++. All campaigns therefore ran at a similar, high load, so compared implementations saw comparable conditions (Appendix B).

**Baselines.** We compare against three established baselines:

1. **C++ reference:** the original cover tree implementation of Beygelzimer et al. [7], adapted to produce JSON output; its batch all-nearest-neighbor query (from which RUSTKNN’s batch single-tree mode derives).
2. **mlpack single-tree:** mlpack 4.7.1’s cover tree with single-tree  $k$ -NN search and base 1.3.
3. **mlpack dual-tree:** mlpack 4.7.1’s cover tree with dual-tree traversal and base 1.3.

All baselines consume the same CSV data files and use Euclidean distance. These three baselines are compared against RUSTKNN across all 12 datasets.

## 5.2 Query Mode $\times k$ Interaction

Table 3 presents the all-nearest-neighbor benchmark ( $k=1$ ): for each point in the dataset, find its nearest neighbor. This is the standard cover tree benchmark from Izbicki and Shelton [8], exercising tree construction and exhaustive querying. RUSTKNN results use packed simplified cover trees in three query modes (dual-tree, single-tree, batch single-tree).

The key finding is that no single implementation or query mode dominates. At  $k=1$ , the C++ reference has the lowest median on 9 of 12 datasets—every dataset with  $n \leq 20,000$  except optdigits, plus covtype, where its interval overlaps RUSTKNN Dual—while RUSTKNN single-tree search wins on optdigits, corel, and mnist, by  $1.7\times$ ,  $1.6\times$ , and  $3.7\times$  over the C++ reference. At  $k=10$  (Appendix Table 10) the C++ reference still wins every dataset with  $n \leq 20,000$ , but RUSTKNN dual-tree search wins the three largest, by  $1.9\times$  on corel,  $2.2\times$  on covtype, and  $3.6\times$  on mnist over the C++ reference.

Table 3: All-nearest-neighbor query time (ms,  $k=1$ , base 1.3). Median [95% CI] over 20 independent SLURM allocations. RUSTKNN results show three query modes: dual-tree, single-tree, and batch single-tree. Best result per row in **bold**.  $k=10$  results in Appendix Table 10. Values  $\geq 10^3$  are abbreviated K/M.

| Dataset    | $d$ | RUSTKNN            |                           |                    | C++                      | mlpack             | mlpack             |
|------------|-----|--------------------|---------------------------|--------------------|--------------------------|--------------------|--------------------|
|            |     | Dual               | Single                    | Batch              | Ref                      | Single             | Dual               |
| iris       | 4   | 0.82 [0.80, 0.84]  | 0.37 [0.36, 0.38]         | 0.71 [0.68, 0.72]  | <b>0.33 [0.28, 0.35]</b> | 1.24 [1.03, 1.30]  | 1.32 [1.16, 1.35]  |
| wine       | 13  | 0.82 [0.74, 0.85]  | 0.66 [0.57, 0.68]         | 0.74 [0.66, 0.76]  | <b>0.35 [0.27, 0.35]</b> | 1.18 [0.94, 1.20]  | 1.01 [0.81, 1.02]  |
| glass      | 9   | 1.77 [1.41, 1.86]  | 0.96 [0.83, 1.05]         | 1.36 [1.15, 1.50]  | <b>0.54 [0.48, 0.60]</b> | 2.51 [2.23, 2.75]  | 2.22 [2.02, 2.46]  |
| bupa       | 6   | 4.02 [3.48, 4.21]  | 2.07 [1.77, 2.21]         | 3.13 [2.58, 3.23]  | <b>1.27 [1.18, 1.41]</b> | 6.02 [5.64, 6.47]  | 6.17 [5.60, 6.77]  |
| pima       | 8   | 7.29 [6.62, 8.66]  | 9.05 [8.09, 10.8]         | 6.68 [5.67, 7.38]  | <b>2.43 [2.26, 2.50]</b> | 13.5 [12.5, 13.9]  | 13.2 [11.9, 13.3]  |
| ionosphere | 34  | 4.50 [4.25, 4.76]  | 2.54 [2.44, 2.68]         | 4.13 [3.91, 4.38]  | <b>1.16 [1.09, 1.19]</b> | 6.92 [6.59, 7.23]  | 6.61 [6.41, 6.98]  |
| optdigits  | 64  | 887 [886, 888]     | <b>160 [159, 165]</b>     | 979 [973, 1,129]   | 264 [260, 284]           | 1.23K [1.2, 1.31]  | 1.68K [1.66, 1.79] |
| pendigits  | 16  | 174 [173, 191]     | 144 [143, 153]            | 230 [228, 242]     | <b>96.6 [87.1, 97.9]</b> | 560 [546, 601]     | 584 [583, 618]     |
| letter     | 16  | 1.13K [1.12, 1.18] | 644 [640, 680]            | 2.02K [1.98, 2.15] | <b>576 [568, 605]</b>    | 3.78K [3.68, 3.9]  | 6.44K [6.17, 6.6]  |
| corel      | 32  | 4.96K [4.88, 5.17] | <b>2.93K [2.69, 2.96]</b> | 17.2K [17.1, 17.4] | 4.69K [4.31, 4.86]       | 25.1K [20.9, 25.2] | 52K [40.7, 52.2]   |
| covtype    | 54  | 13.6K [12, 14.3]   | 102K [88.8, 103]          | 54.8K [45.6, 57.7] | <b>13K [11.8, 13.2]</b>  | 68.9K [62, 70.3]   | 80.9K [72.6, 89]   |
| mnist      | 784 | 673K [584, 705]    | <b>381K [307, 460]</b>    | 2.34M [2.06, 3.09] | 1.40M [1.15, 1.83]       | 4.86M [3.85, 5.89] | 3.85M [3.25, 4.54] |

**$k$ -scaling dynamics.** Table 4 traces the best RUSTKNN mode across five values of  $k$  for four representative datasets. The crossover is sharpest on mnist: single-tree search is  $1.8\times$  faster than dual-tree search at  $k=1$  but  $3.1\text{--}3.5\times$  slower at every  $k\geq 5$ . Corel shows the same pattern at smaller scale (single-tree  $1.7\times$  faster at  $k=1$ , dual-tree  $1.2\text{--}1.6\times$  faster at  $k\geq 5$ ); on optdigits single-tree leads through  $k=10$ , the two tie at  $k=25$ , and dual-tree leads at  $k=50$ . On covtype dual-tree search is best at every  $k$ , and the largest mode-choice penalty in the table is there: single-tree search is  $7.0\text{--}7.5\times$  slower at every  $k$ . Batch single-tree is never the fastest mode on these four datasets; at  $k\geq 5$  it is within 30% of the best only on optdigits. On mnist and covtype the gaps are well outside the confidence intervals; on corel they overlap slightly at  $k=5$ , and on optdigits the modes are within their intervals throughout.

Table 4: Query mode  $\times k$  interaction (ms, all-NN, RUSTKNN Packed). Median [95% CI] over 20 independent allocations. Bold = best mode per (dataset,  $k$ ). Values  $\geq 10^3$  are abbreviated K/M.

| Dataset   | Mode   | $k=1$                     | $k=5$                    | $k=10$                    | $k=25$                    | $k=50$                    |
|-----------|--------|---------------------------|--------------------------|---------------------------|---------------------------|---------------------------|
| optdigits | Dual   | 887 [886, 888]            | 1.09K [1.08, 1.1]        | 935 [837, 1,184]          | 979 [963, 1,106]          | <b>1.13K [1.12, 1.2]</b>  |
|           | Single | <b>160 [159, 165]</b>     | <b>933 [885, 1,156]</b>  | <b>772 [736, 902]</b>     | <b>976 [962, 1,077]</b>   | 1.21K [1.2, 1.26]         |
|           | Batch  | 979 [973, 1,129]          | 1.16K [1.06, 1.19]       | 987 [969, 1,143]          | 1.14K [1.13, 1.21]        | 1.34K [1.32, 1.42]        |
| corel     | Dual   | 4.96K [4.88, 5.17]        | <b>6.55K [6.3, 6.84]</b> | <b>7.58K [7.27, 7.86]</b> | <b>9.42K [8.97, 9.75]</b> | <b>11.5K [10.8, 11.9]</b> |
|           | Single | <b>2.93K [2.69, 2.96]</b> | 8.12K [6.78, 8.32]       | 9.88K [8.22, 10.2]        | 13.1K [10.8, 13.5]        | 18.1K [13.5, 18.4]        |
|           | Batch  | 17.2K [17.1, 17.4]        | 21.4K [16.9, 21.5]       | 23.6K [19, 24]            | 28.1K [22, 28.9]          | 32.8K [25.5, 34.1]        |
| covtype   | Dual   | <b>13.6K [12, 14.3]</b>   | <b>20.4K [17.7, 21]</b>  | <b>25.2K [22.1, 25.7]</b> | <b>37K [32.9, 37.2]</b>   | <b>53.2K [47.9, 53.4]</b> |
|           | Single | 102K [88.8, 103]          | 151K [131, 154]          | 185K [160, 190]           | 262K [226, 273]           | 374K [315, 387]           |
|           | Batch  | 54.8K [45.6, 57.7]        | 91.7K [77, 94.4]         | 121K [103, 123]           | 182K [153, 182]           | 252K [209, 254]           |
| mnist     | Dual   | 673K [584, 705]           | <b>811K [741, 864]</b>   | <b>886K [766, 950]</b>    | <b>956K [890, 1,073]</b>  | <b>1.07M [0.94, 1.16]</b> |
|           | Single | <b>381K [307, 460]</b>    | 2.49M [2.46, 3.60]       | 2.90M [2.62, 3.96]        | 3.38M [2.90, 3.99]        | 3.49M [2.83, 4.63]        |
|           | Batch  | 2.34M [2.06, 3.09]        | 2.52M [2.42, 3.53]       | 2.98M [2.70, 3.76]        | 3.12M [2.66, 4.07]        | 3.63M [2.88, 4.61]        |

**Why the best mode changes with  $k$ .** Single-tree search runs one independent descent per query, pruning against that query’s current  $k$ th-best distance. At  $k=1$  that radius is the nearest-neighbor distance, which is small, and the dedicated  $k=1$  path keeps the per-candidate bookkeeping to a single comparison: on mnist the single-tree all-NN issues  $6\times$  fewer distance computations than the dual-tree traversal (312M vs. 1.87B, Table 9). As  $k$  grows, the pruning radius widens to the  $k$ th-nearest distance and every one of the  $n$  independent descents pays for it separately: from  $k=1$  to  $k=5$ , single-tree time on mnist grows  $6.5\times$  while dual-tree time grows  $1.2\times$  (Table 4).

Dual-tree traversal [39] amortizes that cost by pruning entire (query-subtree, reference-subtree) pairs, but it pays a per-pair overhead in exchange. When both nodes are internal it runs a three-phase expansion, producing  $O(|\text{children}_q| \times |\text{children}_r|)$  candidate pairs per internal node—typically 9–25 at base 1.3, each a recursive call—and it maintains a per-query-node bound cache with epoch invalidation, and computes phase-1/2 distances that are wasted whenever the pairs they would pre-filter are pruned anyway. Where pruning is weak relative to that overhead—small datasets, or  $k=1$ , where

single-tree search already prunes tightly—the overhead dominates; on the three largest datasets at  $k \geq 5$ , and on covtype at every  $k$ , joint pruning wins. Batch single-tree search, which shares reference expansion across a query subtree but scores each query separately, gets neither benefit here: it is the slowest of the three modes on corel and optdigits at every  $k$  and on mnist at four of five, and lands between the other two only on covtype and on mnist at  $k=25$ . Appendix C and Figures 20–21 give the algorithms and worked diagrams.

The held-out results (Table 5) show a different pattern: with a shallow query tree ( $n_q \leq 512$ ), dual-tree’s construction overhead is reduced but so is its pruning leverage.

### 5.3 Base Parameter Analysis

Beygelzimer et al. [7] recommend base  $b=1.3$  based on empirical evaluation. For RUSTKNN’s dual-tree all-NN (Table 6, Appendix A.4), our data do not support it as a default.

Three patterns emerge. First, **coarser bases win**: the optimum is  $b=2.0$ —the value used in the original analysis—on four of the seven datasets in Table 6 at  $k=1$  and five at  $k=10$ , and  $b=3.0$  on the two smallest (glass and ionosphere,  $n \leq 351$ ), where shallower trees reduce the traversal overhead that dominates at small scale. Across all twelve datasets (Appendix A.4),  $b=1.3$  is optimal in only one of 24 (dataset,  $k$ ) cases. Second, **the optimum rarely shifts with  $k$** : in Table 6 only mnist moves (1.3 at  $k=1$ , 2.0 at  $k=10$ ), and there the optimum is shallow: base 2.0 is 10% slower than 1.3 at  $k=1$  and within 1% of it at  $k=10$ . Third, **the default’s cost is real where it matters**:  $b=1.3$  is  $1.57\times$  slower than the optimum on covtype at  $k=1$  ( $1.51\times$  at  $k=10$ ) and  $1.11$ – $1.17\times$  slower on corel and pendigits; it is within 2% of the optimum only on optdigits at  $k=1$  and on mnist. We therefore recommend sweeping the base rather than adopting 1.3 unchecked: a sweep over  $b \in \{1.3, 2.0, 3.0\}$  costs seconds on most datasets (roughly 10–15 minutes per base on mnist). We do not generalize this beyond dual-tree search; the per-implementation curves in Appendix A.4 show how the other implementations respond to the base. Full base sensitivity curves appear in Appendix A.4.

### 5.4 All-NN vs. Held-Out Divergence

The all-nearest-neighbor benchmark, in which query and reference sets coincide, is the standard evaluation in KNN literature [7, 8]. This setting can enable some memory and algorithmic optimizations since the query and reference sets can use the same cover tree. However, most practical workloads involve held-out queries against a pre-built index. Table 5 (Appendix A.2) shows held-out performance diverges from all-NN rankings.

**Ranking divergence.** The clearest case is mlpack’s two query modes. In the all-NN benchmark they are within about  $2\times$  of each other on every dataset, and dual-tree is the faster of the two on five of them (Table 3); held out, mlpack-Dual is  $7.8\times$  slower than mlpack-Single on pendigits and  $250\times$  slower on covtype at  $n_q=512$  (Table 5). An evaluation that used only the all-NN benchmark would never see a gap of that size. The best RUSTKNN mode changes in the same way: on mnist, single-tree search wins the all-NN benchmark at  $k=1$ , but held out at  $n_q=512$  dual-tree search is  $1.8\times$  faster than single-tree (8.4 vs. 15.1 s; Appendix A.2). And on covtype, the C++ reference ties RUSTKNN’s best mode in all-NN at  $k=1$  (13.0K vs. 13.6K ms, intervals overlapping) but is  $1.5\times$  faster held out at  $n_q=512$  (53.0 vs. 81.1 ms, intervals disjoint). This divergence is systematic: in all-NN mode the query tree is the reference tree, so dual-tree and batch methods amortize their tree-building and expansion costs over all  $n$  points; held out, the query tree is shallow and that amortization is small.

**Where our implementation loses.** Table 5 also contains a result worth stating plainly rather than glossing: on covtype held-out—arguably the most practically relevant cell in the paper, being the one dataset with both large  $n$  and moderate  $d$ —the C++ reference is fastest at every  $n_q$ , and RUSTKNN’s best mode (dual-tree) is  $1.2$ – $1.5\times$  slower. RUSTKNN-Single is also slower than mlpack-Single at  $n_q \geq 128$  (176 vs. 142 ms at  $n_q=512$ ). But single-tree is the wrong RUSTKNN mode here: RUSTKNN-Dual is  $2.2\times$  faster than RUSTKNN-Single at  $n_q=512$  and  $1.75\times$  faster than mlpack-Single. Both facts are the same lesson from opposite directions—an implementation evaluated in only one query mode gets misranked, whichever way the misranking happens to fall.

**Implications for evaluation.** We recommend reporting both all-NN and held-out results: the former stresses scalability, the latter approximates serving latency. Full held-out scaling curves appear in Appendices A.2–A.3.

## 5.5 Beyond Cover Trees: A KD-Tree Under the Same Harness

The factors we isolate above are cover-tree-specific in their details, which raises the question of whether the *methodological* finding—that implementation choices can dominate algorithmic ones—holds for other index structures. To test this without rewriting the study, we added a median-split KD-tree *inside* RUSTKNN, reusing the identical kernel, candidate structure, distance counter, and timing code, so it runs the same five feature-flag ablations. We also benchmarked kiddo, an independent Rust KD-tree crate, under the same 20-allocation protocol, separating the effect of the codebase from that of the index. Full tables are in Appendix H.

**The better index reverses with dimension.** On all-NN at  $k=1$ , the KD-tree is  $13\times$  faster than single-tree cover trees on corel (254 vs. 3,248 ms; same-allocation values from Table 11) and  $23\times$  faster on covtype (4,465 vs. 100,590 ms), because axis-aligned box pruning is highly effective at moderate  $d$ . At  $d=784$  that pruning collapses—the KD-tree issues 594M distance computations—and it becomes  $12\times$  slower than the single-tree cover tree (4.09M vs. 334,487 ms): a textbook curse-of-dimensionality crossover, measured with no cross-codebase confounders.

**The same algorithm in a different codebase differs by up to  $9\times$ .** Our KD-tree and kiddo implement the same algorithm, yet kiddo is  $9\times$  faster on pendigits (7.8 vs. 72 ms) while being  $2.9\times$  slower on covtype and  $2.0\times$  slower on mnist (8.31M vs. 4.09M ms): each wins on different data. A practitioner comparing “KD-tree vs. cover tree” would reach the same verdict with either library here, but not the same magnitude: on covtype the KD-tree beats the best cover-tree mode by  $3.1\times$  with ours but by only 7% with kiddo—precisely the confounding this paper argues against, now shown outside cover trees. The five-flag KD-tree ablation (Table 12) shows the same structure: only early exit helps beyond noise, and only at  $d=784$  ( $1.11\times$ ).

## 6 Conclusion

Using RUSTKNN as a controlled experimental platform, we have shown that cover tree performance depends on interactions between query strategy,  $k$ , base parameter, and micro-optimizations in ways not apparent from cross-codebase comparisons. The best query mode depends on  $k$  and on the dataset—single-tree search at  $k=1$  on most of our larger datasets, dual-tree search at  $k\geq 5$  on the largest—and choosing the wrong mode costs up to  $7.5\times$ ; base 1.3 is not a safe default for dual-tree search (2.0 or more is optimal in 22 of 24 dataset- $k$  cases); early-exit distance is the one micro-optimization that matters, and its benefit grows with dimension ( $1.31\times$  at  $d=54$ ,  $1.49\times$  at  $d=784$ ) while leaving the traversal unchanged; and all-NN and held-out benchmarks produce different implementation rankings, so both should be reported. Reporting the main results over 20 independent allocations with bootstrap intervals is what separated these effects from measurement noise.

Our conclusions are scoped to exact nearest neighbor search under Euclidean distance, single-threaded, on datasets up to  $n\approx 581\text{K}$ . We do not study approximate methods; whether these effects transfer to HNSW/IVF-style indexes is open, as are non-Euclidean metrics, parallel scaling, and  $n > 10^6$ . Timings on the smallest datasets are noise-limited (Section 4.2); cross-codebase timings carry an instruction-set disparity (AVX-512 vs. SSE2; Appendix B), though within-RUSTKNN comparisons are unaffected. Large-dataset times were measured on fully loaded nodes, at least  $\sim 2\times$  an idle machine’s (Appendix B); we interpret only ratios measured under the same load.

## Acknowledgment

We would like to thank Ryan R. Curtin for reviewing early results and helping us sanity check unexpected differences in runtime. Indeed, such issues and discrepancies on what method is fastest, when, and why — is the kind of confusion we hope to mitigate with this research.

## References

- [1] Andrew Moore. A tutorial on kd-trees. Technical report, University of Cambridge, 1991.
- [2] Sergey Ioffe. Improved Consistent Sampling, Weighted Minhash and L1 Sketching. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, pages 246–255, Washington, DC, USA, December 2010. IEEE Computer Society. doi: 10.1109/ICDM.2010.80. URL <http://ieeexplore.ieee.org/document/5693978/>.
- [3] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
- [4] Stephen M. Omohundro. Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA, 1989.
- [5] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321. SIAM, 1993.
- [6] Robert Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 798–807. SIAM, 2004.
- [7] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 97–104. ACM, 2006.
- [8] Mike Izbicki and Christian R. Shelton. Faster cover trees. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, volume 37 of *Proceedings of Machine Learning Research*, pages 1162–1170. PMLR, 2015.
- [9] Edward Raff and Charles Nicholas. Toward Metric Indexes for Incremental Insertion and Querying. *arXiv*, 2018. URL <http://arxiv.org/abs/1801.05055>. arXiv: 1801.05055.
- [10] Yury Elkin and Vitaliy Kurlin. A new near-linear time algorithm for k-nearest neighbor search using a compressed cover tree. In *Proceedings of the 40th International Conference on Machine Learning*, pages 9267–9311. PMLR, July 2023. URL <https://proceedings.mlr.press/v202/elkin23a.html>.
- [11] Yury Elkin and Vitaliy Kurlin. Counterexamples expose gaps in the proof of time complexity for cover trees introduced in 2006. In *2022 Topological Data Analysis and Visualization (TopoInVis)*, pages 9–17, October 2022. doi: 10.1109/TopoInVis57755.2022.00008. URL <https://ieeexplore.ieee.org/abstract/document/9975821>.
- [12] Ryan R Curtin. *Improving Dual-Tree Algorithms*. PhD thesis, Georgia Institute of Technology, 2015.
- [13] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 604–613. ACM, 1998.
- [14] Yu 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*, 42(4):824–836, 2020.
- [15] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, Parikshit Ram, Nishant A. Mehta, and Alexander G. Gray. mlpack: A scalable C++ machine learning library. In *Journal of Machine Learning Research*, volume 14, pages 801–805, 2013.
- [16] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

- [17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.
- [18] Erik Bernhardsson. Annoy: Approximate nearest neighbors in C++/Python. <https://github.com/spotify/annoy>, 2013. Spotify.
- [19] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119, pages 3887–3896. PMLR, 2020.
- [20] Edward Raff. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research*, 18(23):1–5, 2017. URL <http://jmlr.org/papers/v18/16-131.html>.
- [21] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update Mark. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009. ISSN 19310145. doi: 10.1145/1656274.1656278. ISBN: 1931-0145.
- [22] Edward Raff, Michel Benaroch, Sagar Samtani, and Andrew L. Farris. What Do Machine Learning Researchers Mean by “Reproducible”? *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(27):28671–28683, April 2025. ISSN 2374-3468. doi: 10.1609/aaai.v39i27.35093. URL <https://ojs.aaai.org/index.php/AAAI/article/view/35093>.
- [23] Mohammad Arvan, Luís Pina, and Natalie Parde. Reproducibility in Computational Linguistics: Is Source Code Enough? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2350–2361, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.150. URL <https://aclanthology.org/2022.emnlp-main.150>.
- [24] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416545. URL <https://doi.org/10.1145/3324884.3416545>. Series Title: ASE ’20.
- [25] Upulee Kanewala and James M Bieman. Testing Scientific Software: A Systematic Literature Review. *Information and software technology*, 56(10):1219–1232, October 2014. ISSN 0950-5849. doi: 10.1016/j.infsof.2014.05.006. URL <https://pubmed.ncbi.nlm.nih.gov/25125798>.
- [26] Edward Raff and Andrew L. Farris. A Siren Song of Open Source Reproducibility. In *ML Evaluation Standards Workshop at ICLR 2022*, 2022. doi: 10.48550/arXiv.2204.04372. URL <https://arxiv.org/abs/2204.04372>.
- [27] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *29th International Conference on Software Engineering (ICSE’07)*, pages 550–559. IEEE, May 2007. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.77. URL <https://ieeexplore.ieee.org/document/4222616/>.
- [28] Odd Erik Gundersen, Saeid Shamsaliei, and Richard Juul Isdahl. Do machine learning platforms provide out-of-the-box reproducibility? *Future Generation Computer Systems*, 126:34–47, January 2022. ISSN 0167-739X. doi: 10.1016/j.future.2021.06.014. URL <https://www.sciencedirect.com/science/article/pii/S0167739X21002090>.
- [29] Edward Raff. Research Reproducibility as a Survival Analysis. In *The Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021. URL <http://arxiv.org/abs/2012.09932>. arXiv: 2012.09932.

- [30] Anthony Moi and Nicolas Patry. HuggingFace tokenizers: Fast state-of-the-art tokenizers. <https://github.com/huggingface/tokenizers>, 2023.
- [31] Hugging Face. Candle: Minimalist ML framework for Rust. <https://github.com/huggingface/candle>, 2024.
- [32] Nathaniel Simard, Louis Fortier-Dubois, Dilshod Tadjibaev, and Burn Contributors. Burn: A dynamic deep learning framework in Rust. <https://github.com/tracel-ai/burn>, 2024.
- [33] Andrey Vasnetsov and Qdrant Team. Qdrant: High-performance vector search engine. <https://github.com/qdrant/qdrant>, 2024.
- [34] LanceDB Contributors. LanceDB: Embedded vector database for AI applications. <https://github.com/lancedb/lancedb>, 2024.
- [35] sdd. Kiddo: High-performance  $k$ -d tree library for Rust. <https://github.com/sdd/kiddo>, 2024.
- [36] hora-search Contributors. Hora: Approximate nearest neighbor search library. <https://github.com/hora-search/hora>, 2021.
- [37] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. In *Discrete & Computational Geometry*, volume 22, pages 63–93. Springer, 1999.
- [38] Parikshit Ram, Dongryeol Lee, William B. March, and Alexander G. Gray. Linear-time algorithms for pairwise statistical problems. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 22, 2009.
- [39] Ryan R. Curtin, William B. March, Parikshit Ram, David V. Anderson, Alexander G. Gray, and Charles L. Isbell. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 1435–1443. PMLR, 2013.
- [40] Dheeru Dua and Casey Graff. UCI machine learning repository. <https://archive.ics.uci.edu/ml>, 2017.
- [41] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [42] Alexander G Gray and Andrew W Moore. ‘N-Body’ Problems in Statistical Learning. In T K Leen, T G Dietterich, and V Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 521–527. MIT Press, 2001.
- [43] Ping Li. 0-Bit Consistent Weighted Sampling. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 665–674, New York, NY, USA, 2015. ACM. doi: 10.1145/2783258.2783406. URL <http://doi.acm.org/10.1145/2783258.2783406>.
- [44] K. Fukunage and P.M. Narendra. A Branch and Bound Algorithm for Computing k-Nearest Neighbors. *IEEE Transactions on Computers*, C-24(7):750–753, 1975. doi: 10.1109/T-C.1975.224297. URL [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1672890](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1672890).
- [45] Evgeniy A. Petrov and Ruslan R. Salimov. A note on generalized four-point inequality. *Journal of Mathematical Sciences*, 273(3):414–426, June 2023. doi: 10.1007/s10958-023-06507-w. URL <https://doi.org/10.1007/s10958-023-06507-w>.
- [46] Lawrence Cayton. Accelerating Nearest Neighbor Search on Manycore Systems. *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 402–413, May 2012. doi: 10.1109/IPDPS.2012.45. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6267877>.

- [47] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In L K Saul, Y Weiss, and L Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 825–832. MIT Press, 2005.
- [48] Frank Nielsen, Paolo Piro, and Michel Barlaud. Bregman vantage point trees for efficient nearest Neighbor Queries. In *2009 IEEE International Conference on Multimedia and Expo*, pages 878–881. IEEE, June 2009. doi: 10.1109/ICME.2009.5202635. URL <http://ieeexplore.ieee.org/document/5202635/>.
- [49] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Consistent Weighted Sampling Made More Practical. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1035–1043, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. doi: 10.1145/3038912.3052598. URL <https://doi.org/10.1145/3038912.3052598>.
- [50] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based Indexing for High-dimensional Metric Spaces. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 357–368, New York, NY, USA, 1997. ACM. doi: 10.1145/253260.253345. URL <http://doi.acm.org/10.1145/253260.253345>.
- [51] Mei-kang Wu. *Evaluation of R-trees for Nearest Neighbor Search Evaluation of R-trees for Nearest Neighbor Search*. PhD thesis, University of Houston, 2006.
- [52] Peter N Yianilos. Excluded Middle Vantage Point Forests for Nearest Neighbor Search. Technical report, NEC Research Institute, Princeton, NJ, July 1998.
- [53] Ryan R. Curtin, Javier Echaz, and Andrew B. Gardner. Exploiting the structure of furthest neighbor search for fast approximate results. *Information Systems*, 80:124–135, February 2019. doi: 10.1016/j.is.2017.12.010. URL <https://www.sciencedirect.com/science/article/pii/S0306437917301552>.
- [54] Ryan R. Curtin and Andrew B. Gardner. Fast Approximate Furthest Neighbors with Data-Dependent Candidate Selection. In Laurent Amsaleg, Michael E. Houle, and Erich Schubert, editors, *Similarity Search and Applications*, pages 221–235, Cham, 2016. Springer International Publishing. doi: 10.1007/978-3-319-46759-7\_17.
- [55] Ryan R. Curtin, Parikshit Ram, and Alexander G. Gray. Fast Exact Max-kernel Search. In *Proceedings of the 2013 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 1–9. Society for Industrial and Applied Mathematics, May 2013. doi: 10.1137/1.9781611972832.1. URL <https://epubs.siam.org/doi/abs/10.1137/1.9781611972832.1>.
- [56] Ryan R. Curtin. Faster Dual-Tree Traversal for Nearest Neighbor Search. In Giuseppe Amato, Richard Connor, Fabrizio Falchi, and Claudio Gennaro, editors, *Similarity Search and Applications*, pages 77–89, Cham, 2015. Springer International Publishing. doi: 10.1007/978-3-319-25087-8\_7.
- [57] David R Karger and Matthias Ruhl. Finding Nearest Neighbors in Growth-restricted Metrics. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, pages 741–750, New York, NY, USA, 2002. ACM. doi: 10.1145/509907.510013. URL <http://doi.acm.org/10.1145/509907.510013>.
- [58] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [59] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. URL <http://dl.acm.org/citation.cfm?id=645920.673001>.

- [60] Jeffrey K. Uhlmann. Implementing Metric Trees to Satisfy General Proximity / Similarity Queries. Technical report, Naval Research Laboratory, Washington, D.C., USA, 1991.
- [61] Gisli Hjaltason and Hanan Samet. Incremental Similarity Search in Multimedia Databases. Technical report, University of Maryland, College Park, College Park, 2000.
- [62] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404, 1999. URL <http://dl.acm.org/citation.cfm?id=328959>.
- [63] Hanan Samet. K-nearest neighbor finding using MaxNearestDist. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):243–252, 2008. doi: 10.1109/TPAMI.2007.1182.
- [64] Leonid Boytsov and Bilegsaikhan Naidan. Learning to Prune in Metric and Non-Metric Spaces. In C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.
- [65] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11): 1348–1358, August 2010. doi: 10.1016/j.patrec.2010.04.004. URL <http://linkinghub.elsevier.com/retrieve/pii/S0167865510001169>.
- [66] A Andoni, M Datar, N Immorlica, and P Indyk. Locality-sensitive hashing using stable distributions. *Nearest Neighbor Methods in*, 2006. URL <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Locality-sensitive+hashing+using+stable+distributions#1>.
- [67] Peter N Yianilos. Locally Lifting the Curse of Dimensionality for Nearest Neighbor Search (Extended Abstract). In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 361–370, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. URL <http://dl.acm.org/citation.cfm?id=338219.338273>.
- [68] Y Kalantidis and Y Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR 2014)*, Columbus, Ohio, June 2014. IEEE.
- [69] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. URL <http://dl.acm.org/citation.cfm?id=645923.671005>.
- [70] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975. doi: 10.1145/361002.361007. URL <http://doi.acm.org/10.1145/361002.361007>.
- [71] Sergey Brin. Near Neighbor Search in Large Metric Spaces. In *International Conference on Very Large Data Bases*, pages 574–584, San Francisco, CA, 1995. Morgan Kaufmann Publishers.
- [72] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *of Computer Science, 2006. FOCS'06.*, 2006. URL [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=4031381](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4031381).
- [73] Kenneth L. Clarkson. Nearest neighbor searching in metric spaces: Experimental Results for sb(S). Technical report, Bell Laboratories, Lucent Technologies, New Jersey, 2002.
- [74] Parikshit Ram, Dongryeol Lee, and Alexander G. Gray. Nearest-Neighbor Search on a Time Budget via Max-Margin Trees. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 1011–1022. Society for Industrial and Applied Mathematics, April 2012. doi: 10.1137/1.9781611972825.87. URL <https://epubs.siam.org/doi/10.1137/1.9781611972825.87>.

- [75] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. Parallel Cover Trees and their Applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '22, pages 259–272, New York, NY, USA, July 2022. Association for Computing Machinery. doi: 10.1145/3490148.3538581. URL <https://dl.acm.org/doi/10.1145/3490148.3538581>.
- [76] Nazneen Rajani, Rajani McArdle, and Inderjit S. Dhillon. Parallel k nearest neighbor graph construction using tree-based data structures. In *1st High Performance Graph Mining workshop*, pages 1–14. Barcelona Supercomputing Center, 2015. doi: 10.5821/hpgm15.1. URL <http://hdl.handle.net/2117/76382>.
- [77] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel multi-dimensional range query processing with R-trees on GPU. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013. doi: 10.1016/j.jpdc.2013.03.015. URL <http://www.sciencedirect.com/science/article/pii/S0743731513000592>.
- [78] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of Nearest Neighbor Queries in R-trees. In *International Conference on Database Theory*, pages 394–408, London, UK, 1997. Springer-Verlag.
- [79] R A Finkel and J L Bentley. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, March 1974. doi: 10.1007/BF00288933. URL <http://dx.doi.org/10.1007/BF00288933>.
- [80] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. Query-aware locality-sensitive hashing scheme for  $\$L_p\$norm. *The VLDB Journal*, 26(5):683–708, October 2017. doi: 10.1007/s00778-017-0472-7. URL <https://doi.org/10.1007/s00778-017-0472-7>.$
- [81] A Guttman. R-trees: a dynamic index structure for spatial searching. In *1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, 1984. ACM. URL <http://dl.acm.org/citation.cfm?id=602266>.
- [82] Sanjoy Dasgupta and Yoav Freund. Random Projection Trees and Low Dimensional Manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, pages 537–546, New York, NY, USA, 2008. ACM. doi: 10.1145/1374376.1374452. URL <http://doi.acm.org/10.1145/1374376.1374452>.
- [83] Sanjoy Dasgupta and Kaushik Sinha. Randomized Partition Trees for Nearest Neighbor Search. *Algorithmica*, 72(1):237–263, May 2015. doi: 10.1007/s00453-014-9885-5. URL <http://dx.doi.org/10.1007/s00453-014-9885-5>.
- [84] Parikshit Ram and Kaushik Sinha. Revisiting Kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1378–1388, New York, NY, USA, 2019. ACM. doi: 10.1145/3292500.3330875. URL <http://doi.acm.org/10.1145/3292500.3330875>.
- [85] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in Metric Spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001. doi: 10.1145/502807.502808. URL <http://doi.acm.org/10.1145/502807.502808>.
- [86] Anshumali Shrivastava. Simple and Efficient Weighted Minwise Hashing. In D D Lee, U V Luxburg, I Guyon, and R Garnett, editors, *Advances In Neural Information Processing Systems* 29, pages 1498–1506. Curran Associates, Inc., 2016.
- [87] Andrew W Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 397–405, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. URL <http://dl.acm.org/citation.cfm?id=2073946.2073993>.
- [88] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984. doi: 10.1145/356924.356930. URL <http://doi.acm.org/10.1145/356924.356930>.

- [89] Qiang Huang, Jianlin Feng, Qiong Fang, and Wilfred Ng. Two Efficient Hashing Schemes for High-Dimensional Furthest Neighbor Search. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2772–2785, December 2017. doi: 10.1109/TKDE.2017.2752156. URL <https://ieeexplore.ieee.org/abstract/document/8038075>.
- [90] Neeraj Kumar, Li Zhang, and Shree Nayar. What Is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images? In *Proceedings of the 10th European Conference on Computer Vision: Part II*, pages 364–378, Berlin, Heidelberg, 2008. Springer-Verlag. doi: 10.1007/978-3-540-88688-4\_27. URL [http://dx.doi.org/10.1007/978-3-540-88688-4\\_27](http://dx.doi.org/10.1007/978-3-540-88688-4_27).

## NeurIPS Paper Checklist

### 1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [\[Yes\]](#)

Justification: The abstract and introduction list four numbered contributions (optimization ablation, query mode  $\times k$  interaction, base parameter analysis, and all-NN vs. held-out divergence) that directly correspond to the experimental sections of the paper.

Guidelines:

- The answer [\[N/A\]](#) means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [\[No\]](#) or [\[N/A\]](#) answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

### 2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [\[Yes\]](#)

Justification: Yes—the Conclusion contains an explicit “Limitations” paragraph. It states that our conclusions are scoped to exact nearest neighbor search under Euclidean distance, single-threaded, on datasets up to  $n \approx 581\text{K}$ ; that approximate (HNSW/IVF-style) methods, non-Euclidean metrics, parallel scaling and  $n > 10^6$  are not studied; that timings on the smallest datasets are noise-limited (Section 4.2 draws no micro-optimization conclusions from the five UCI sets with  $n \leq 351$  or from pima/pendigits); that cross-codebase timings carry an instruction-set disparity; and that absolute large-dataset times were measured on loaded nodes, so only ratios measured under the same load are interpreted.

Guidelines:

- The answer [\[N/A\]](#) means that the paper has no limitation while the answer [\[No\]](#) means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

### 3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: This is an empirical systems paper. No new theorems or proofs are presented; the theoretical foundations are from prior work [8].

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

### 4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Hardware specifications, software versions, compiler flags, and experimental methodology are described in Section 5.1 and Appendix B. Implementation details (architecture, packed trees, dual-tree traversal, parallel construction) are in Appendix C (Sections C.1–C.4). All algorithms are described in sufficient detail to reimplement.

### 5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: The RUSTKNN library—every tree variant, query mode and optimization flag evaluated in the paper, with its tests—is open source under the MIT license at <https://github.com/CrowdStrike/RustKNN>. The benchmark harness and cluster job scripts are not released; the measurement protocol (20 independent allocations, estimator, node-load control), hardware, compiler flags and baseline versions are documented in Section 5 and Appendix B, and all datasets (UCI, MNIST) are public and cited.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.

- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

## 6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Dataset dimensions and sizes are listed in Table 2, experimental methodology is described in Section 5.1, and hardware/software details including compiler flags are provided in Appendix B (Section B).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

## 7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Every timing table in the main text, and the held-out, waterfall and  $k=10$  tables in the appendices (Tables 5, 7 and 10), reports the median over **20 independent SLURM allocations** with a **bootstrap 95% confidence interval** (4,000 resamples, seeded by the data so a cell repeated across tables is identical). The base-parameter and per- $k$  ablation tables report the same 20-allocation medians without intervals, and the KD-tree ablation, naive and synthetic appendix tables are medians of several runs within one allocation; each caption says which. The factor of variability captured is between-allocation: each allocation is a separately scheduled job, reduced to one number (the median of its internal timed runs), and the interval is taken over those 20 per-allocation values. Intervals are percentile bootstrap and therefore asymmetric; no normality is assumed.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.

- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

## 8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Experiments were run on a SLURM cluster with Intel Xeon Platinum 8490H processors, one physical core per benchmark job, with node load held at a similar level across campaigns. Hardware and software details are described in the Appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

## 9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: The research uses only standard, publicly available machine learning benchmark datasets and raises no ethical concerns.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

## 10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: This work concerns a foundational data structure for nearest neighbor search. It has no direct negative societal impact beyond general-purpose algorithmic research.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

## 11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper presents a data structure library for nearest neighbor search. No models or datasets with misuse risk are released.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

## 12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All benchmark datasets (UCI repository, MNIST) are properly cited. Baseline implementations (mlpack 4.7.1, 3-clause BSD license; the C++ reference implementation of Beygelzimer et al., GPL-2.0) and the kiddo Rust crate (MIT or Apache-2.0) are cited, used unmodified apart from JSON output added to the C++ benchmark driver, and are not redistributed with RUSTKNN.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, [paperswithcode.com/datasets](https://paperswithcode.com/datasets) has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

### 13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [\[Yes\]](#)

Justification: The new asset is the RUSTKNN library, released at <https://github.com/CrowdStrike/RustKNN> under the MIT license, with a README covering installation, usage, the API for each query mode, input limits and feature flags, plus API documentation, tests and runnable examples.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

### 14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [\[N/A\]](#)

Justification: This work does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

### 15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [\[N/A\]](#)

Justification: This work does not involve human subjects research.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

**16. Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: LLMs were not used as a component of the core methodology, algorithms, or experimental evaluation in this research.

## A Experimental Figures

### A.1 All-Nearest-Neighbor Query Comparison

Figures 1–3 show grouped bar charts comparing all implementations on the all-nearest-neighbor benchmark at base  $b=1.3$ . Bars are medians over 20 independent allocations and whiskers are bootstrap 95% confidence intervals, the same estimator used throughout the paper (Section 5); on the smallest datasets the whiskers are wide enough to overlap between implementations, which is the point made in Section 4.2.

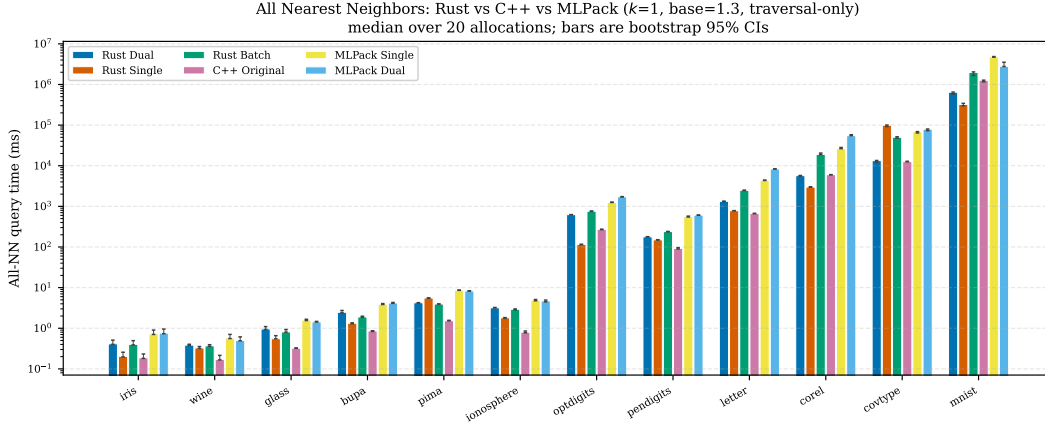


Figure 1: All-nearest-neighbor query time comparison ( $k=1$ , base 1.3). Grouped bar chart across all 12 datasets with all six implementations.

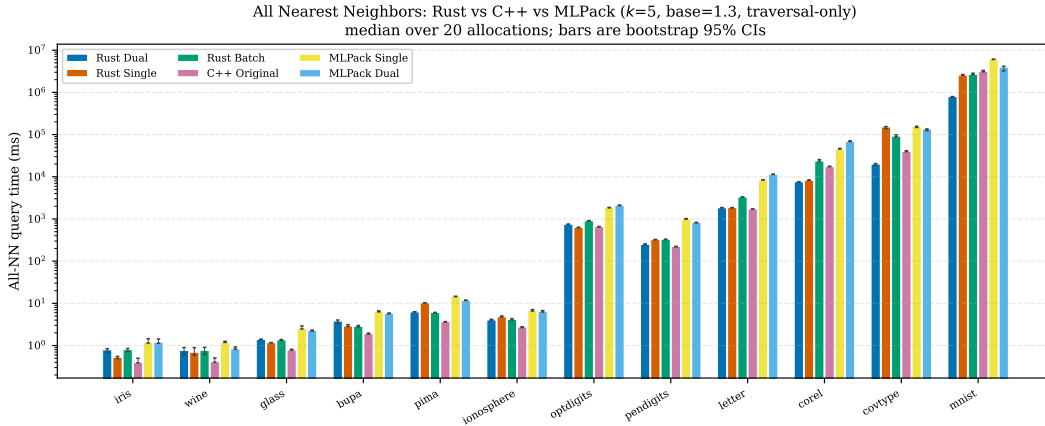


Figure 2: All-nearest-neighbor query time comparison ( $k=5$ , base 1.3).

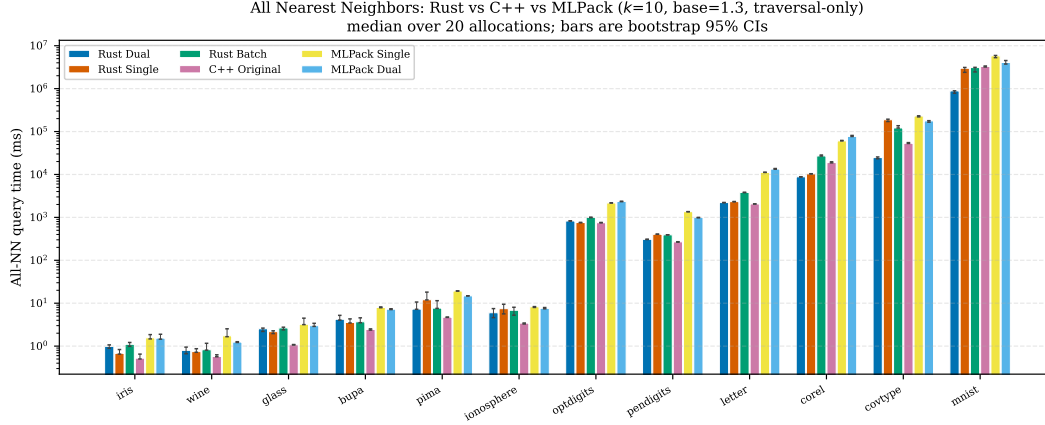


Figure 3: All-nearest-neighbor query time comparison ( $k=10$ , base 1.3).

## A.2 Held-Out Query Scaling (Traversal)

Figures 4–8 isolate traversal-only timing as a function of the number of query points  $n_q$ . Lines are medians over 20 independent allocations; shaded bands are bootstrap 95% confidence intervals.

Table 5: Held-out  $k$ -NN query time (ms,  $k=1$ ) for varying numbers of query points  $n_q$ . Median [95% CI] over 20 independent allocations. Best per row in **bold**. pima has  $n=768$ , so its held-out split caps at  $n_q=384$ . Values  $\geq 10^3$  are abbreviated K/M.

| Dataset   | $n_q$ | RUSTKNN Packed    |                          |                   | C++<br>Ref               | mlpack<br>Single  | mlpack<br>Dual     |
|-----------|-------|-------------------|--------------------------|-------------------|--------------------------|-------------------|--------------------|
|           |       | Dual              | Single                   | Batch             |                          |                   |                    |
| pima      | 32    | 0.13 [0.11, 0.13] | 0.10 [0.09, 0.10]        | 0.17 [0.16, 0.18] | <b>0.09 [0.08, 0.09]</b> | 0.23 [0.22, 0.24] | 0.69 [0.63, 0.70]  |
|           | 128   | 0.57 [0.53, 0.57] | 0.39 [0.36, 0.39]        | 0.52 [0.49, 0.53] | <b>0.31 [0.28, 0.31]</b> | 0.93 [0.86, 0.94] | 1.64 [1.51, 1.66]  |
|           | 384   | 2.17 [2.00, 2.18] | 1.15 [1.05, 1.16]        | 1.43 [1.30, 1.43] | <b>0.79 [0.73, 0.80]</b> | 2.72 [2.51, 2.73] | 3.59 [3.31, 3.64]  |
| pendigits | 32    | 0.93 [0.79, 1.05] | <b>0.59 [0.52, 0.67]</b> | 1.05 [0.93, 1.16] | 0.63 [0.52, 0.67]        | 1.66 [1.41, 1.78] | 31.0 [27.7, 34.1]  |
|           | 128   | 3.35 [2.97, 3.55] | 2.32 [2.01, 2.49]        | 3.92 [3.45, 4.19] | <b>2.22 [1.97, 2.34]</b> | 6.27 [5.51, 6.79] | 76.1 [69.3, 81.7]  |
|           | 512   | 13.2 [13.0, 13.6] | 9.04 [8.86, 9.30]        | 15.1 [14.8, 15.5] | <b>8.11 [7.26, 8.27]</b> | 23.6 [21.2, 25.0] | 183 [166, 193]     |
| corel     | 32    | 5.76 [4.45, 7.28] | 6.10 [4.63, 7.96]        | 8.89 [6.89, 11.0] | <b>4.70 [3.97, 5.59]</b> | 19.2 [15.1, 23.9] | 15.2 [11.9, 18.7]  |
|           | 128   | 19.3 [15.6, 22.7] | 19.4 [15.6, 23.9]        | 35.4 [29.3, 43.0] | <b>19.2 [16.0, 21.6]</b> | 79.4 [62.0, 88.3] | 60.8 [47.5, 66.4]  |
|           | 512   | 102 [92.5, 107]   | <b>65.6 [53.3, 73.4]</b> | 163 [133, 178]    | 104 [83.0, 108]          | 300 [231, 331]    | 196 [161, 227]     |
| covtype   | 32    | 4.59 [3.65, 5.81] | 6.90 [5.44, 8.83]        | 26.5 [24.4, 33.4] | <b>3.83 [3.02, 4.75]</b> | 7.57 [5.65, 9.70] | 5.84K [4.38, 6.76] |
|           | 128   | 21.8 [16.8, 25.1] | 41.4 [30.8, 48.8]        | 37.4 [32.4, 43.6] | <b>14.7 [11.1, 16.9]</b> | 32.9 [23.7, 38.4] | 12.8K [9.54, 14.7] |
|           | 512   | 81.1 [69.6, 90.8] | 176 [127, 203]           | 117 [95.3, 136]   | <b>53.0 [41.6, 60.0]</b> | 142 [105, 166]    | 35.6K [27.4, 41.4] |

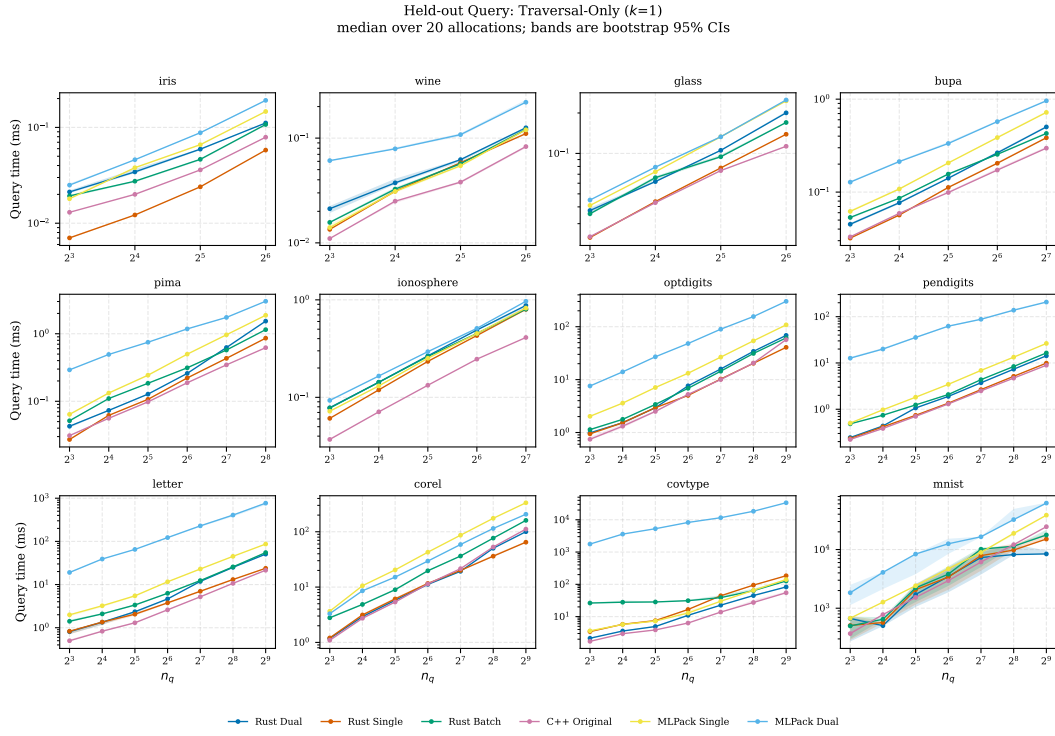


Figure 4: Held-out query scaling, traversal only ( $k=1$ ). Query time vs. number of query points  $n_q$ .

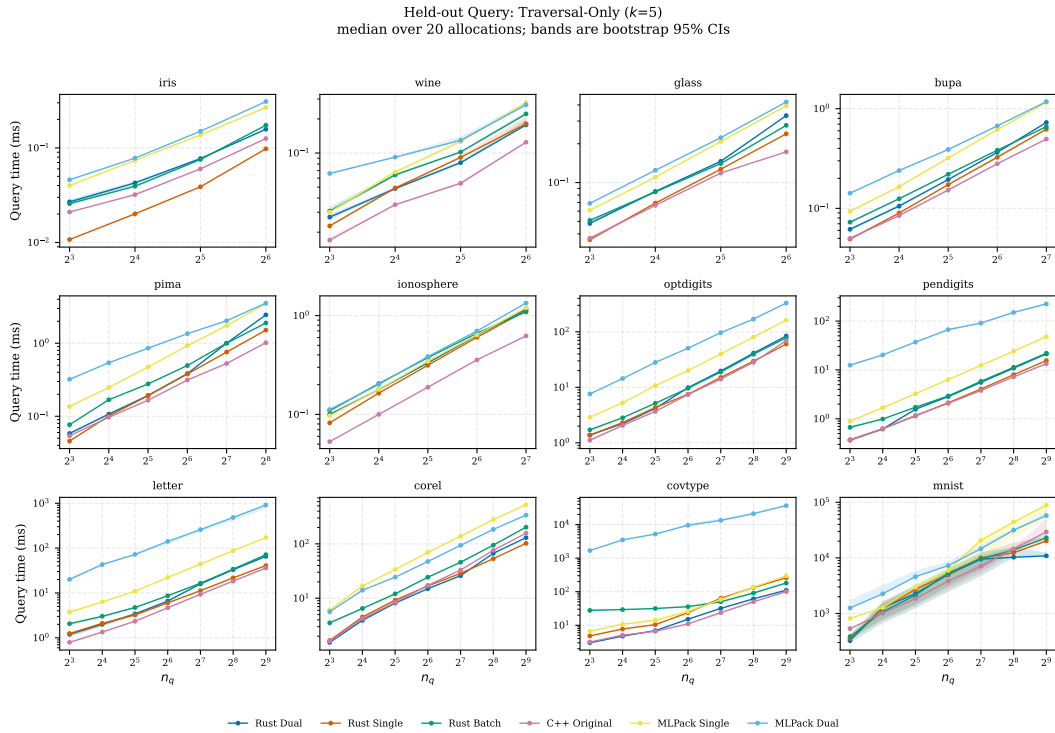


Figure 5: Held-out query scaling, traversal only ( $k=5$ ).

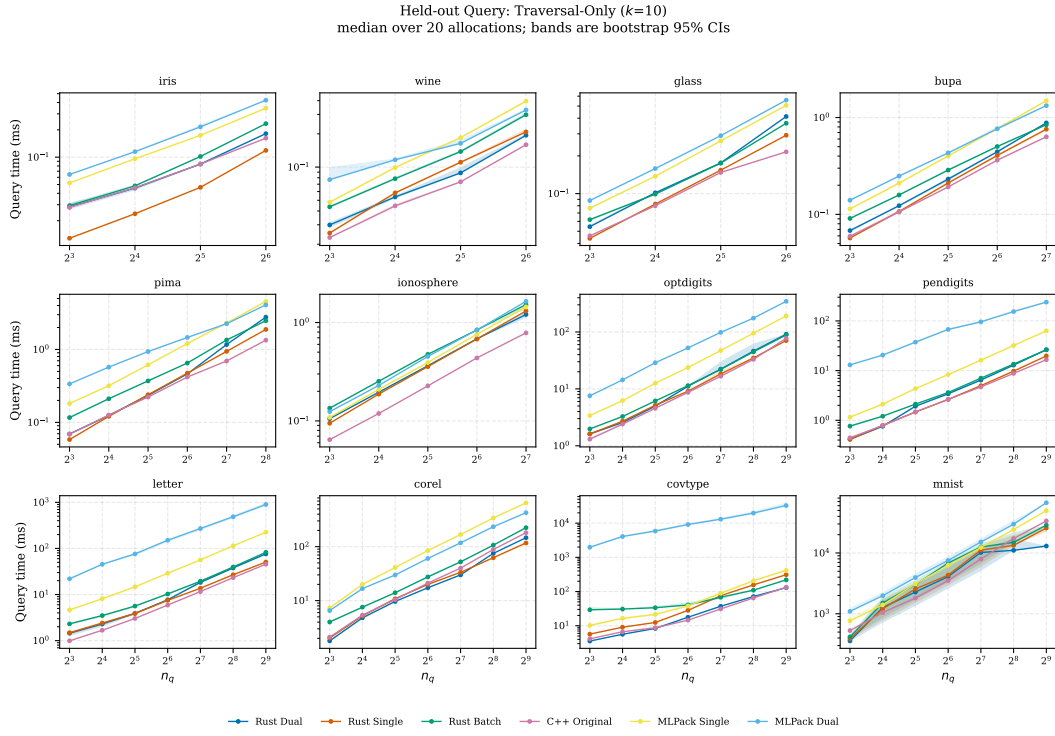


Figure 6: Held-out query scaling, traversal only ( $k=10$ ).

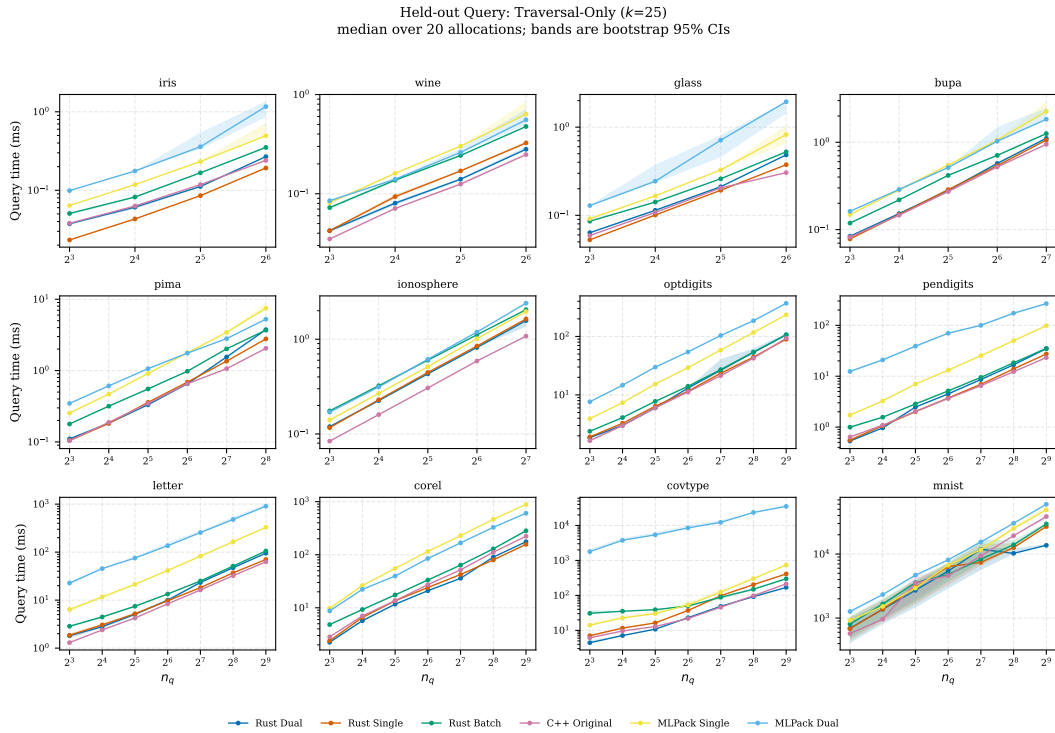


Figure 7: Held-out query scaling, traversal only ( $k=25$ ).

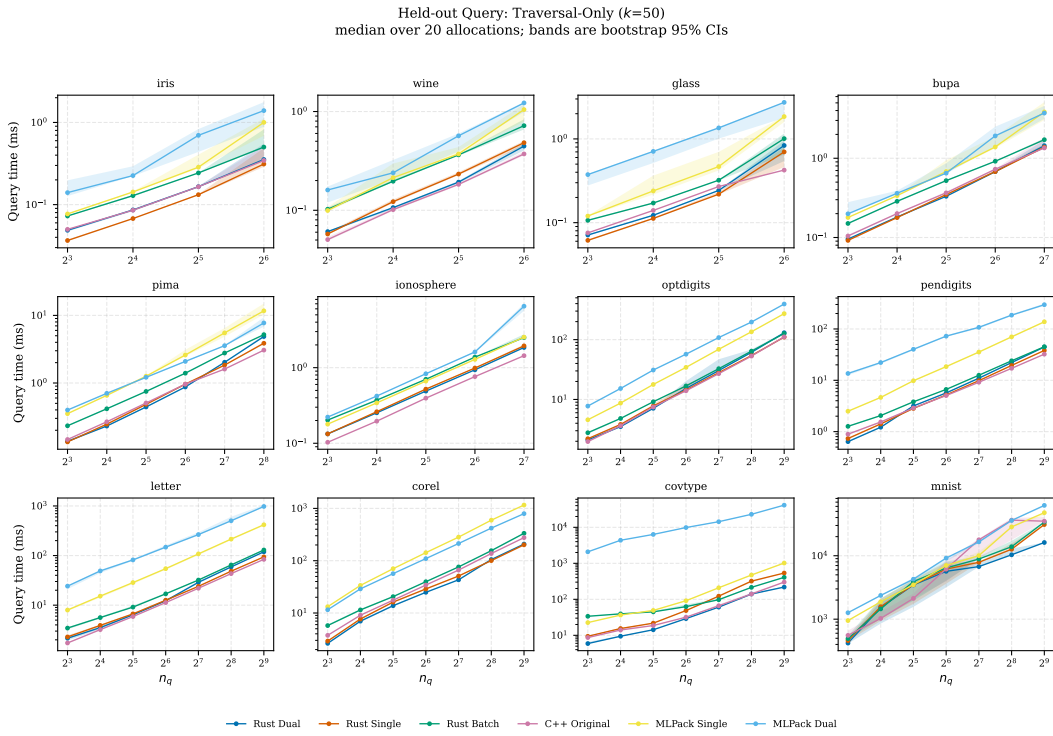


Figure 8: Held-out query scaling, traversal only ( $k=50$ ).

### A.3 Held-Out Query Scaling (End-to-End)

Figures 9–13 show end-to-end timing (including query-tree construction) as a function of the number of query points  $n_q$ . Lines are medians over 20 independent allocations; shaded bands are bootstrap 95% confidence intervals.

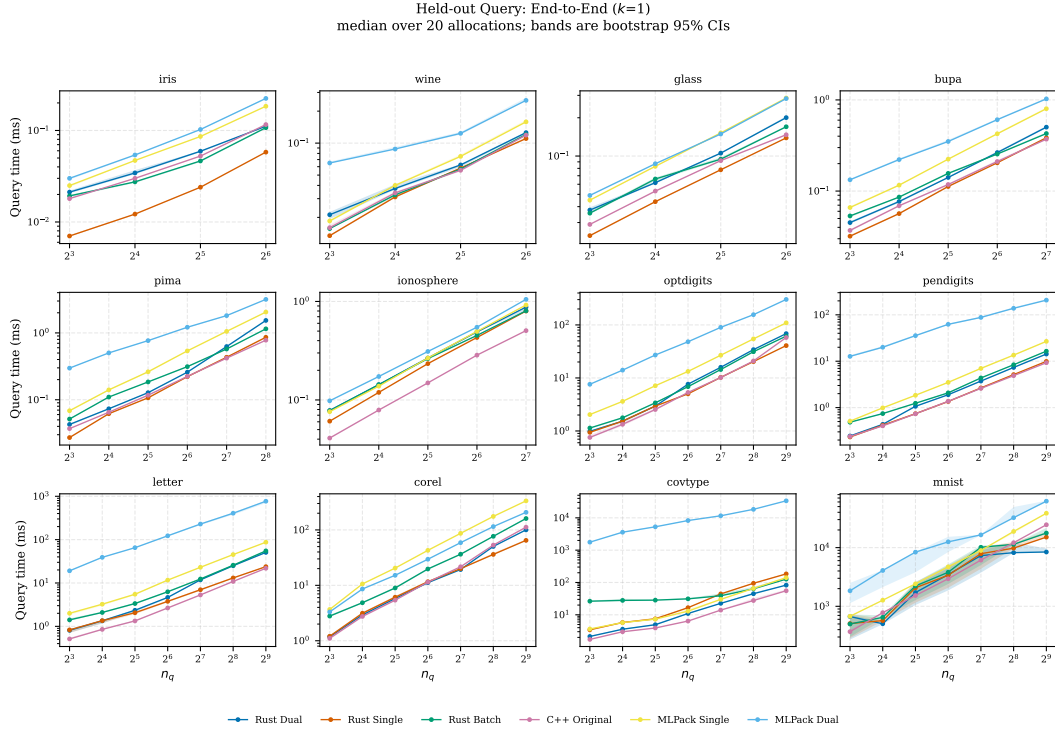


Figure 9: Held-out query scaling, end-to-end ( $k=1$ ). Query time vs. number of query points  $n_q$ .

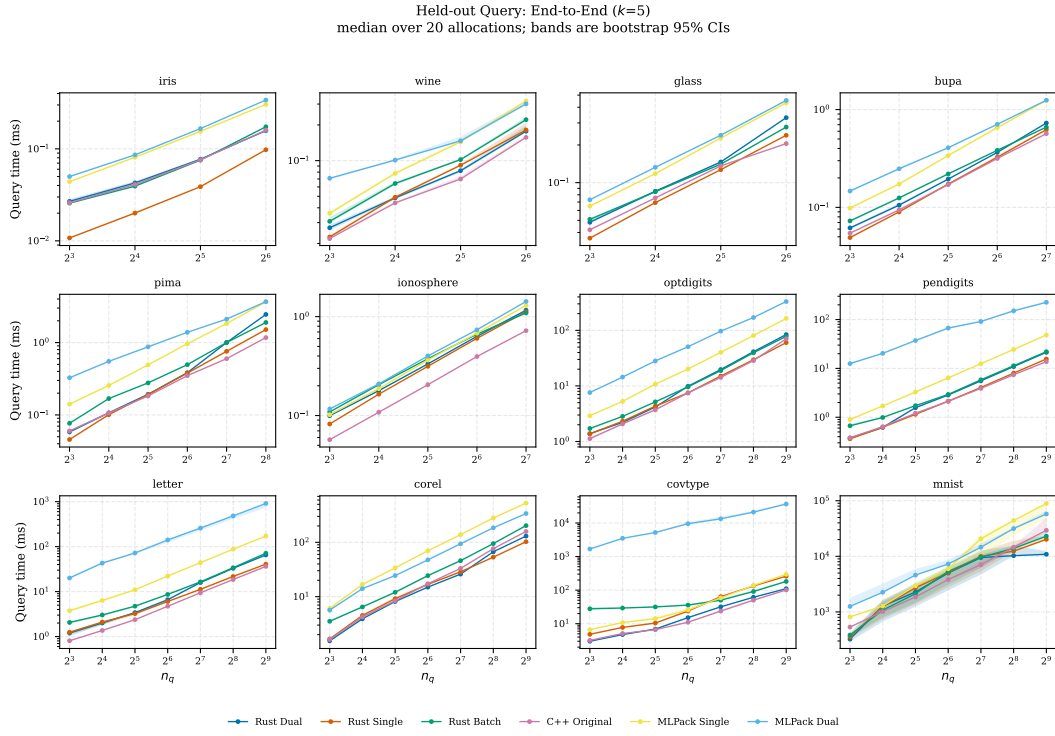


Figure 10: Held-out query scaling, end-to-end ( $k=5$ ).

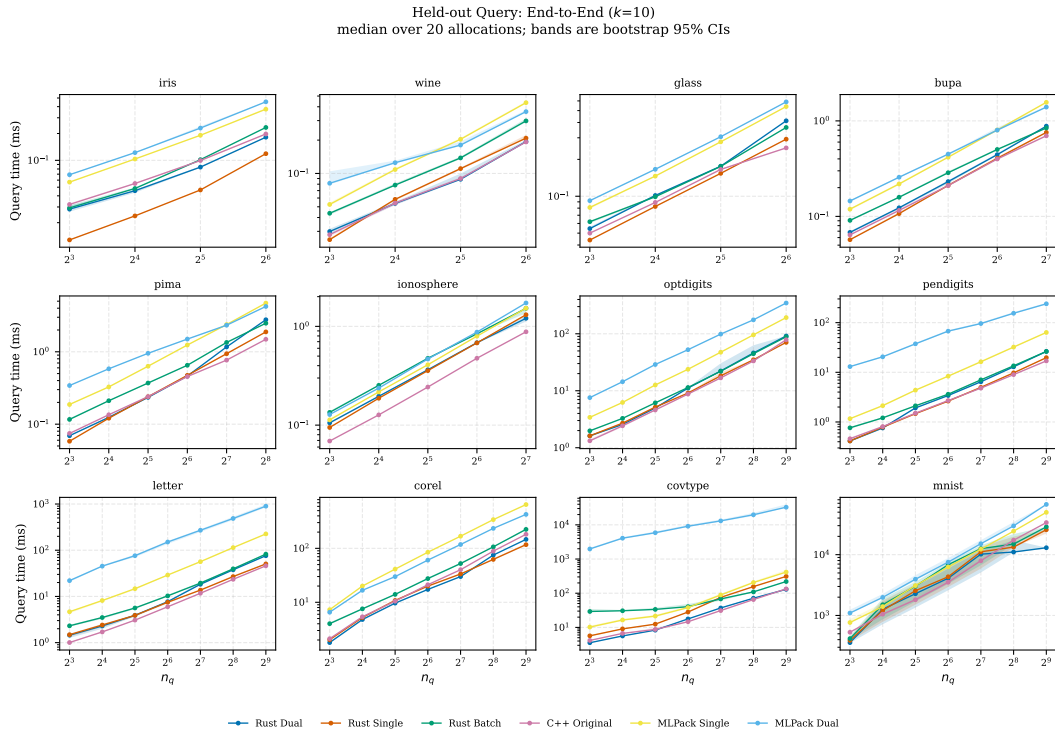


Figure 11: Held-out query scaling, end-to-end ( $k=10$ ).

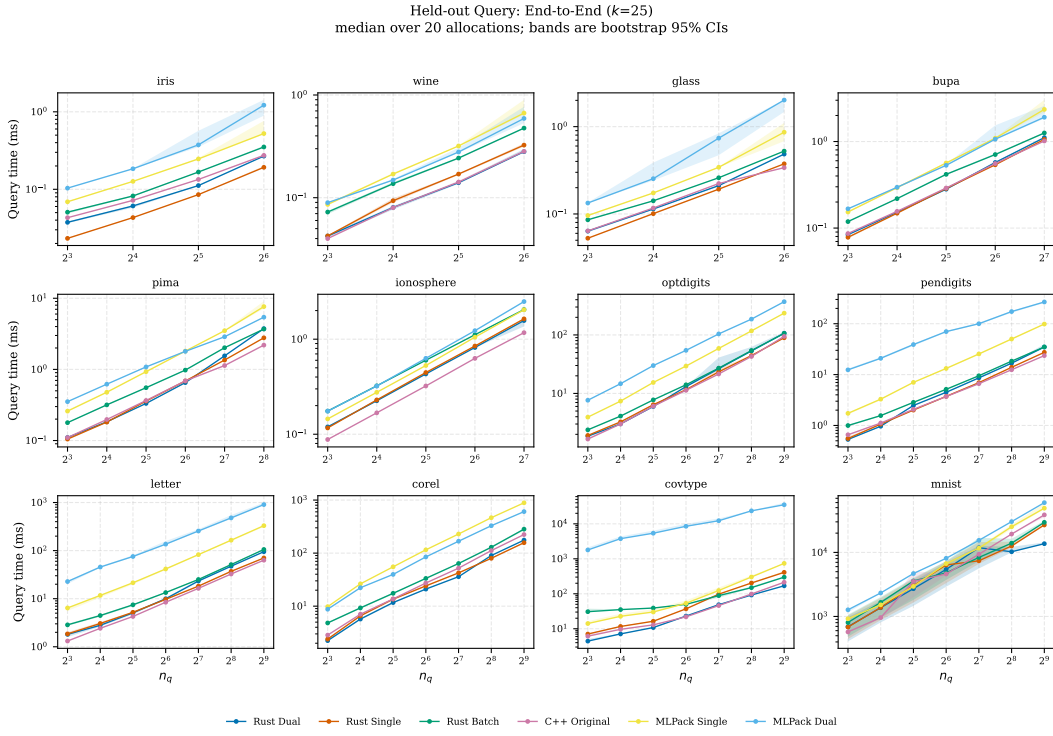


Figure 12: Held-out query scaling, end-to-end ( $k=25$ ).

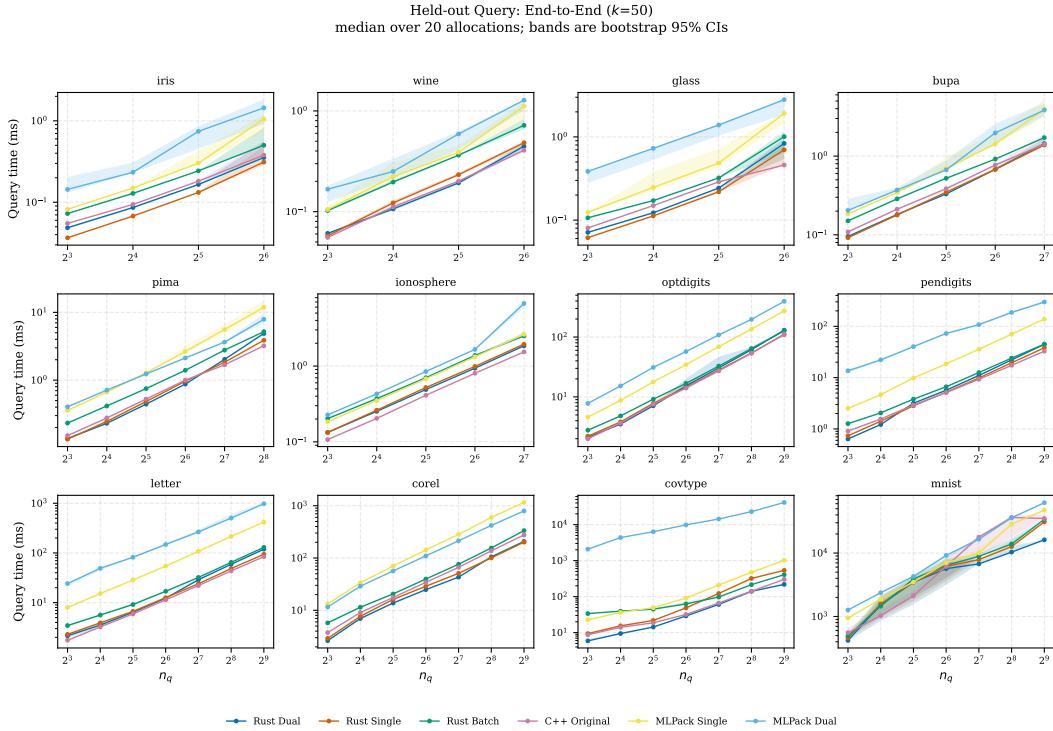


Figure 13: Held-out query scaling, end-to-end ( $k=50$ ).

#### A.4 Base Parameter Sensitivity

Figures 14–19 show query time as a function of the base parameter  $b \in \{1.1, 1.3, 1.5, 2.0, 3.0, 5.0\}$  for all-nearest-neighbor and held-out queries. Every panel reports the median over 20 independent allocations with a bootstrap 95% confidence interval, the same estimator used throughout the paper. Note that the held-out sweeps use a query set capped at a fraction of each dataset, so the smallest sets are evaluated at  $n_q < 256$  (iris 75, wine 89, glass 107, bupa 172, ionosphere 175).

Table 6: Optimal base for RUSTKNN Packed dual-tree all-NN, with the cost of using the conventional  $b=1.3$  instead:  $t(1.3)/t(b^*)$ , so  $1.00\times$  means 1.3 is itself optimal. Medians over 20 independent allocations from the base-sweep campaign; the absolute times may differ from Table 3 due to cross-campaign variance. Bold = optimal  $b$  differs from 1.3.

| Dataset    | $k=1$       |           |              | $k=10$      |           |              |
|------------|-------------|-----------|--------------|-------------|-----------|--------------|
|            | Optimal $b$ | Time (ms) | $b=1.3$ cost | Optimal $b$ | Time (ms) | $b=1.3$ cost |
| glass      | <b>3.0</b>  | 1.04      | $1.67\times$ | <b>3.0</b>  | 2.08      | $1.55\times$ |
| ionosphere | <b>3.0</b>  | 2.79      | $1.57\times$ | <b>3.0</b>  | 5.00      | $1.39\times$ |
| optdigits  | <b>2.0</b>  | 653       | $1.02\times$ | <b>2.0</b>  | 804       | $1.11\times$ |
| pendigits  | <b>2.0</b>  | 165       | $1.14\times$ | <b>2.0</b>  | 273       | $1.17\times$ |
| corel      | <b>2.0</b>  | 4,773     | $1.11\times$ | <b>2.0</b>  | 6,942     | $1.17\times$ |
| covtype    | <b>2.0</b>  | 8,401     | $1.57\times$ | <b>2.0</b>  | 16,088    | $1.51\times$ |
| mnist      | 1.3         | 638,925   | $1.00\times$ | <b>2.0</b>  | 842,461   | $1.00\times$ |

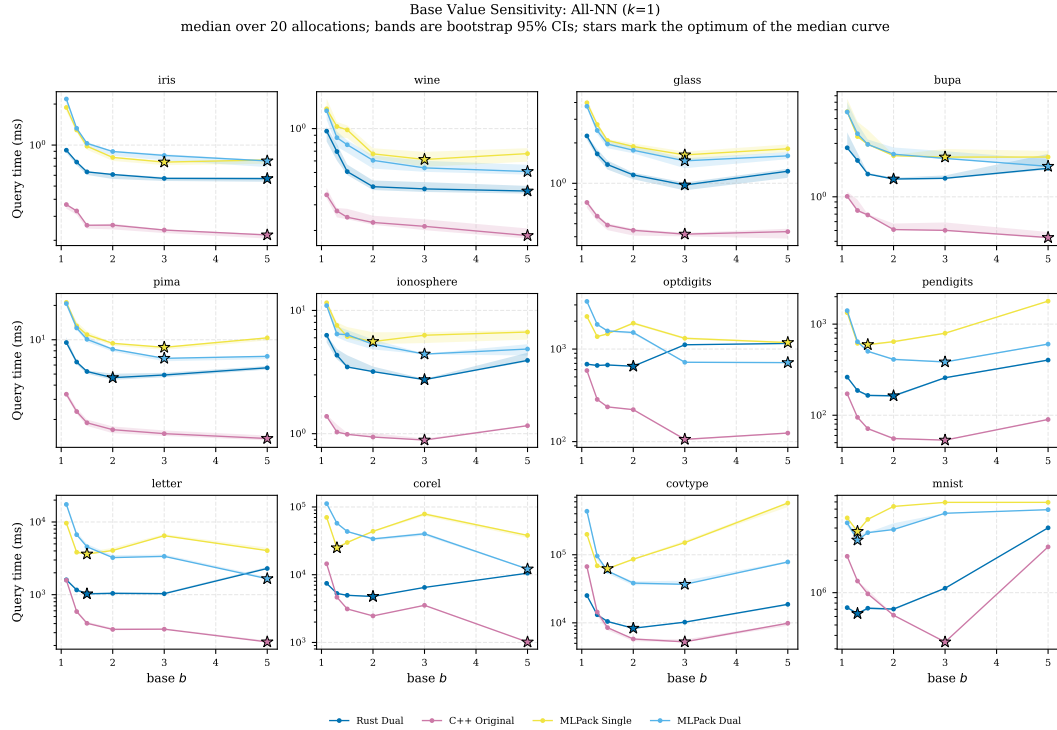


Figure 14: All-nearest-neighbor query time ( $k=1$ ) vs. base parameter. Each panel shows one dataset; implementations are color-coded. Medians over 20 independent allocations; shaded bands are bootstrap 95% CIs; stars mark the optimum of each implementation’s median curve. RUSTKNN curves show traversal time. For RUSTKNN Packed (dual-tree), base 1.3 is optimal on only 1 of 12 datasets (mnist) and within 20% of the best base on 5 of 12; elsewhere the optimum is 2.0, 3.0, or (on the two smallest sets) 5.0, and 1.5 on letter.

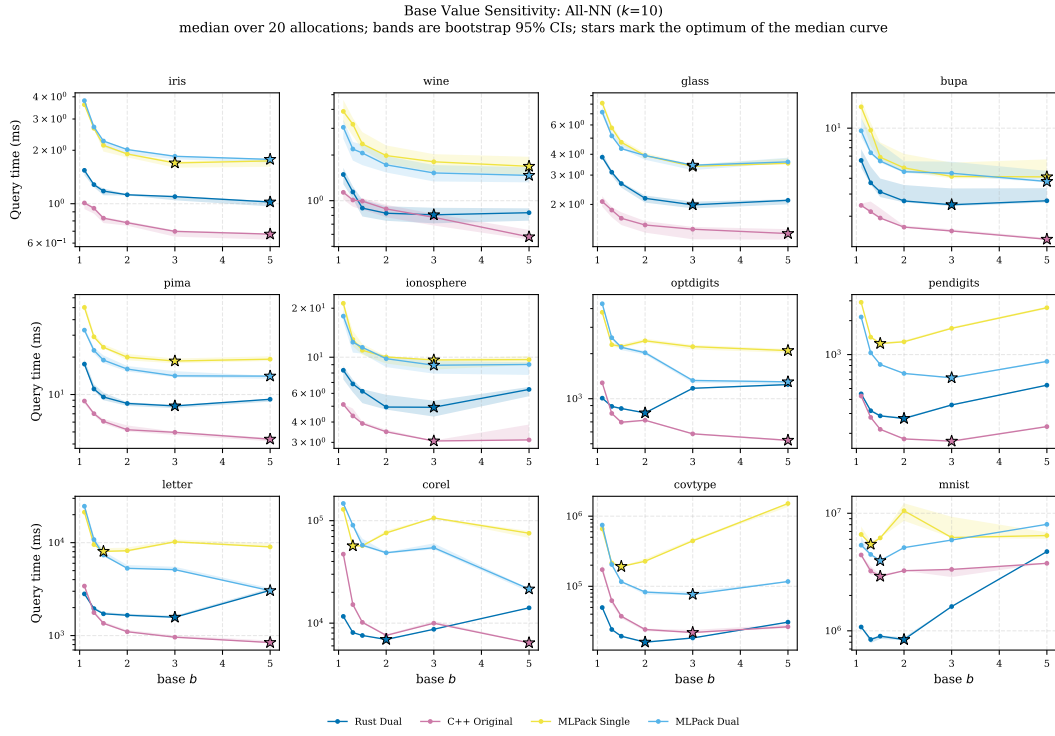


Figure 15: All-nearest-neighbor query time ( $k=10$ ) vs. base parameter.

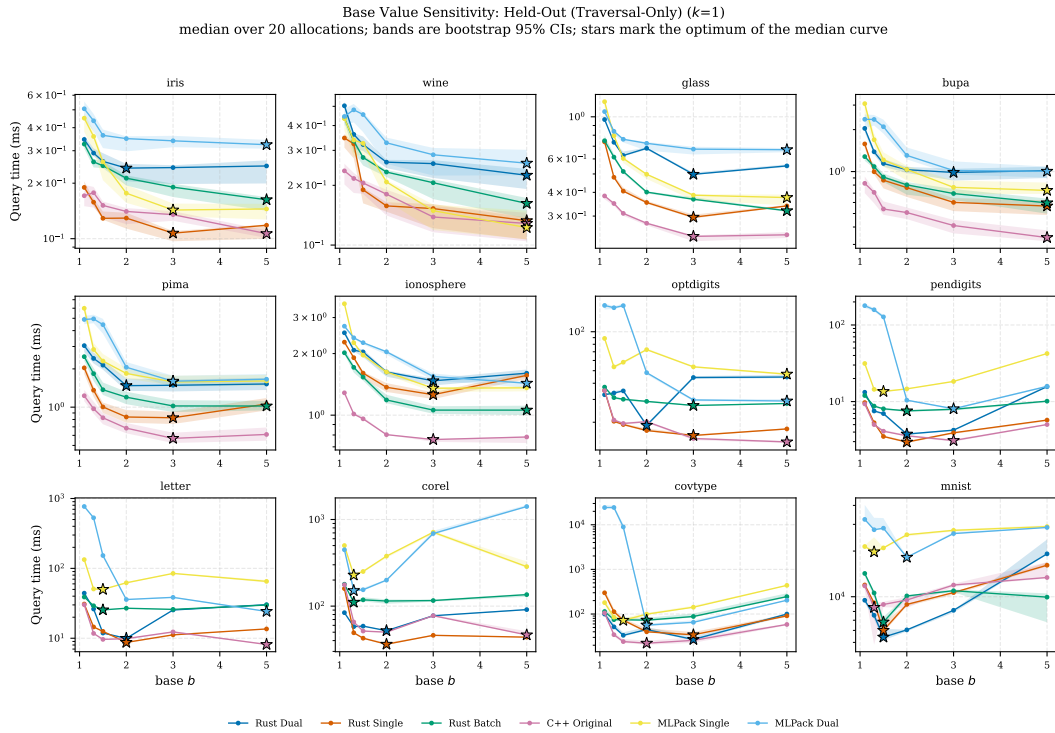


Figure 16: Held-out traversal query time ( $k=1$ ) vs. base parameter.

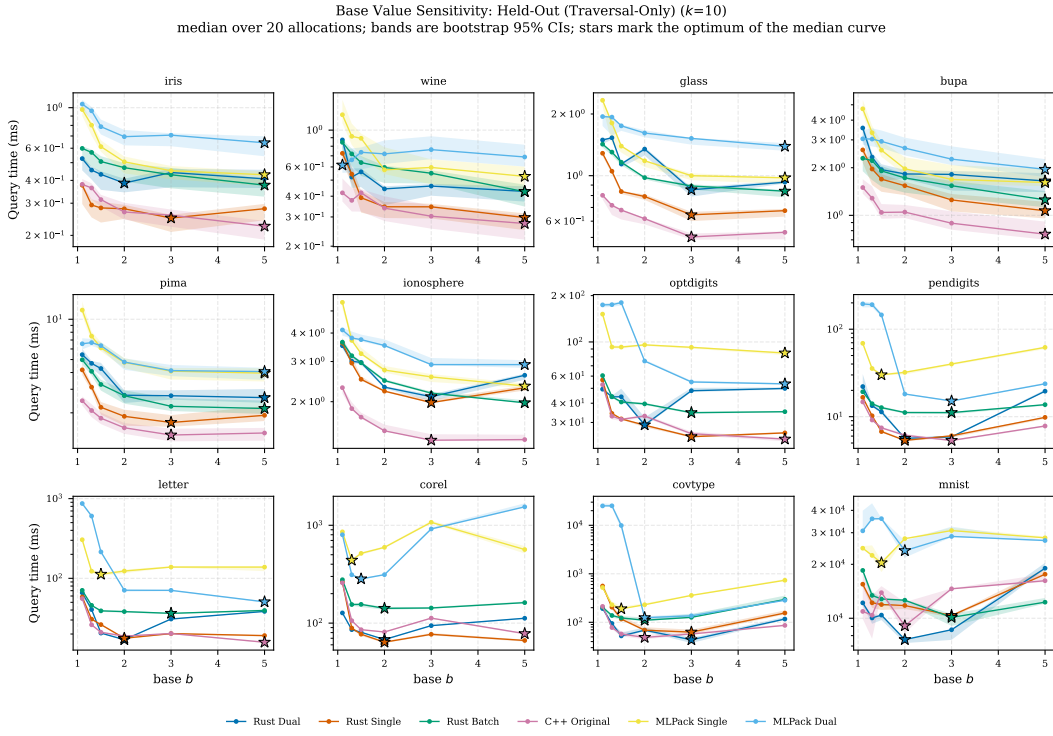


Figure 17: Held-out traversal query time ( $k=10$ ) vs. base parameter.

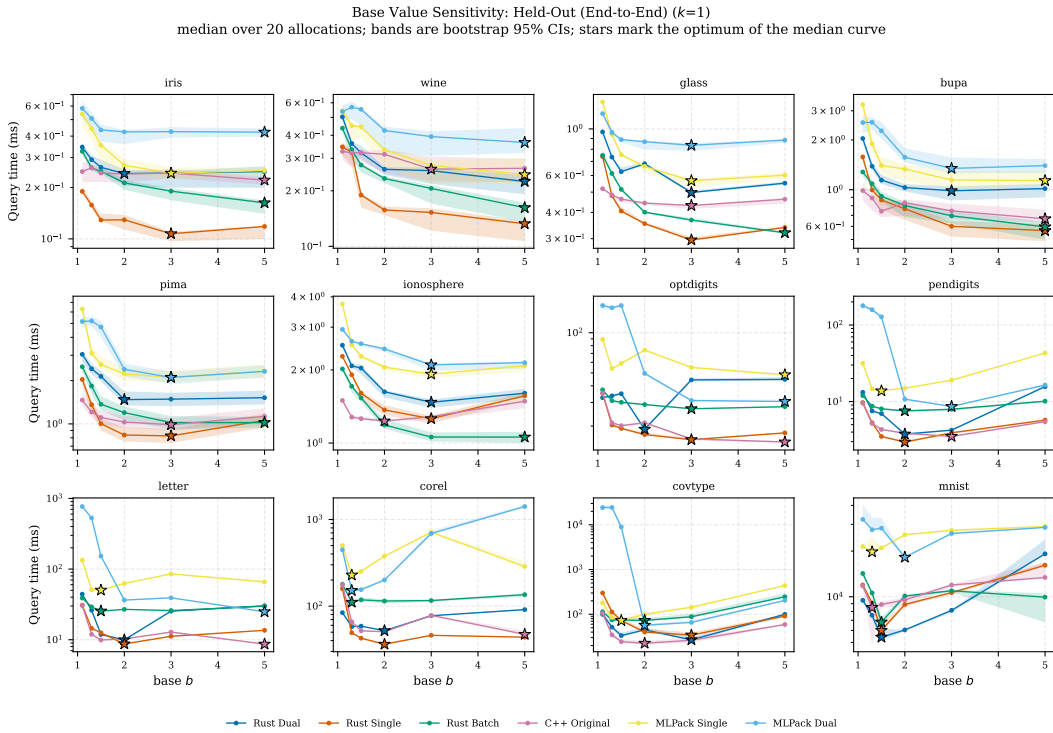


Figure 18: Held-out end-to-end query time ( $k=1$ ) vs. base parameter.

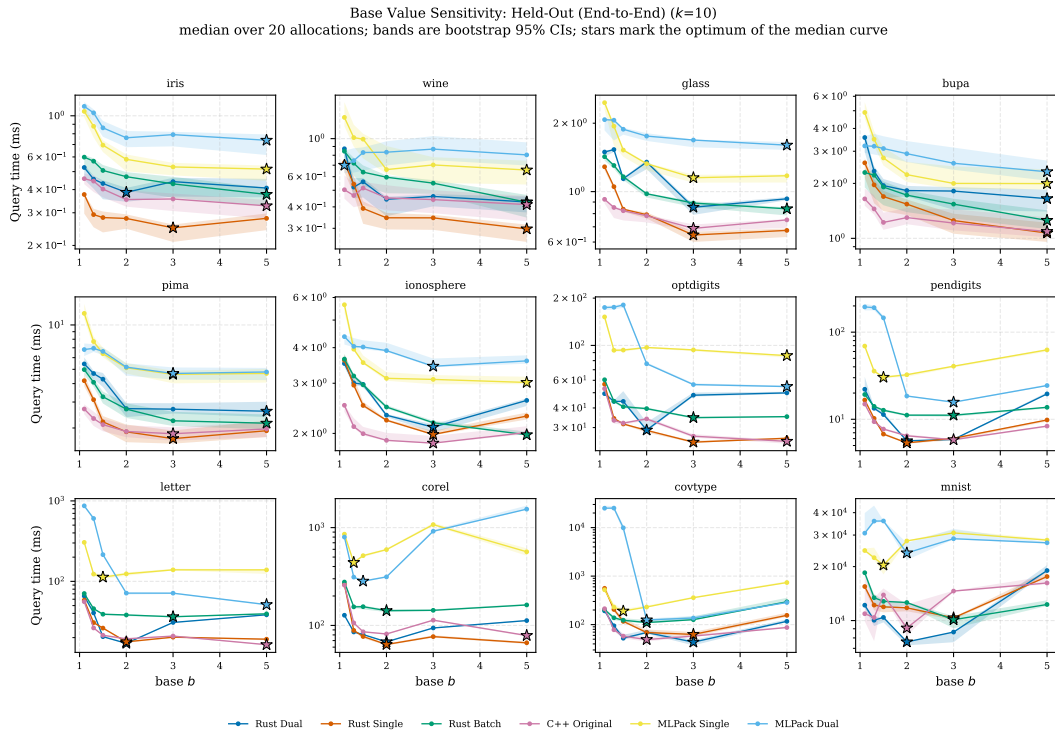


Figure 19: Held-out end-to-end query time ( $k=10$ ) vs. base parameter.

## B Experiment Details

**Hardware.** All experiments run on a multi-node SLURM cluster. Each compute node contains two Intel Xeon Platinum 8490H processors (Sapphire Rapids, 60 cores per socket, two hardware threads per core, 112.5 MiB of L3 per socket, maximum clock 1.90 GHz, i.e., no turbo boost) with 2TB of RAM, running Ubuntu 22.04 LTS (Linux 5.15). Every benchmark job is allocated one physical core (`--cpus-per-task=1` under core-granularity scheduling) and its own memory, so jobs never share a core; they do share each socket’s L3 cache and memory bandwidth with the other jobs on the node.

**Node load.** That sharing matters on the large datasets. On an otherwise idle node,  $N$  pinned copies of the mnist all-NN, split evenly across the two sockets, slow down by  $1.48\times$  ( $N=15$ ),  $1.61\times$  ( $N=30$ ), and  $1.99\times$  ( $N=60$ ) for the C++ reference, and by  $1.43\times$  ( $N=15$ ) and  $2.07\times$  ( $N=60$ ) for RUSTKNN single-tree search (each copy pinned to its own physical core): the two degrade at the same rate, so ratios between implementations measured under the same load are stable even though absolute times are not. Because the clock is fixed, the slowdown is cache and memory-bandwidth contention rather than frequency scaling. We therefore ran every campaign with the nodes held at a similar, high load (about 110 memory-heavy benchmark jobs per 120-core node), using filler jobs that run the same mnist workloads while a campaign’s last allocations finished, so that no configuration was measured on an emptying node. One of the three nodes ran large-dataset allocations  $1.1\text{--}1.5\times$  slower than the other two at the same job count; allocations were spread across nodes, so this mainly widens the upper confidence bounds on covtype and mnist rather than moving the medians. On pima and pendigits, per-allocation timings are bimodal for reasons we have not isolated (Section 4.2).

**Software.** The Rust implementation is compiled with `rustc 1.94.1` (stable, 2026-03-25) using `--release` with LTO enabled and `codegen-units=1`. The reference C++ cover tree of Beygelzimer et al. [7] is compiled with `g++ 11.4.0 (-O3 -ffast-math -funroll-loops)`. `ml-pack 4.7.1` [15] is compiled with `g++ 11.4.0 (-std=c++17 -O3 -DNDEBUG)`. All implementations use a cover-tree base parameter of 1.3 unless otherwise noted (sensitivity analysis in Appendix A.4). Note that the Rust binaries are built on the cluster’s login node (Xeon Platinum 8358, Ice Lake) with `RUSTFLAGS="-C target-cpu=native"`, i.e., AVX-512 code that also runs on the Sapphire Rapids compute nodes, while the C++ builds use `g++`’s default instruction set (SSE2); this favors the Rust implementation on vectorizable distance computations and is itself a confounder of the kind this paper argues about. Timing runs use un-instrumented builds; distance counts come from a separate pass with the instrument build (Section 5).

## C Implementation Details

This appendix provides full implementation details for RUSTKNN, supplementing the overview in Section 4.

### C.1 Architecture and Trait-Based Generics

The core abstraction enabling arbitrary metric spaces is the `Distance<T>` trait:

```
pub trait Distance<T> {  
    fn distance(&self, p: &T, q: &T) -> f64;  
    fn distance_with_bound(&self, p: &T, q: &T,  
        upper_bound: f64) -> f64 { self.distance(p, q) }  
}
```

Users implement a single method; the library provides optimized built-in implementations for Euclidean and Manhattan metrics. The optional `distance_with_bound` method enables early-exit distance computation: the built-in Euclidean implementation uses four independent accumulators with 8-element unrolled loops for instruction-level parallelism, checking partial squared sums every 16–64 dimensions. If the partial sum exceeds `upper_bound2`, the function returns a sentinel value without computing the final square root.

The generic node structure `Node<T: Clone>` stores the data point, level, cached `maxdist`, cached parent distance `d_parent`, children as `Vec<Box<Node<T>>>`, and an `is_duplicate` flag for merge-created duplicates. Fields are ordered for cache locality: hot query fields (`point`, `maxdist`, `d_parent`) appear first.

The library is organized into three variant modules (`simplified/`, `nearest_ancestor/`, `packed/`) sharing common algorithms in a `core/` directory. A zero-cost `CoverTree` enum wrapper delegates via pattern matching.

### C.2 Packed Trees and Cache Optimization

After construction, the pointer-based tree is converted to a `PackedCoverTree` via a single depth-first traversal. All nodes are stored in a contiguous `Vec<PackedNode<T>>`, with child relationships in a separate shared `Vec<usize>` array. Each `PackedNode` carries u32 fields (`child_index_start`, `child_index_count`) as indices into the shared array.

This yields compact nodes: `PackedNode<f64>` occupies approximately 40 bytes, with hot query fields spanning  $\leq 32$  bytes—fitting within a single 64-byte cache line. Using u32 saves 8 bytes per node versus `usize`. Approximately 1,000 nodes fit in a 32 KB L1 cache.

During packing, children are sorted by `d_parent` (ascending) so closer children appear first in memory, tightening the pruning bound sooner.

### C.3 Dual-Tree Traversal

The dual-tree traversal requires several adaptations for cover trees without self-children. When expanding one side’s children while holding the other constant, the held-constant node’s *point* must still score against the expanded children. Two helper routines (`point_vs_subtree`, `subtree_vs_point`) simulate the missing self-child.

When both nodes have children, the traversal proceeds in three phases: (1) query point  $\times$  reference children, (2) query children  $\times$  reference point, and (3) child–child pairs using triangle inequality lower bounds from phases 1 and 2 as free pre-filters:  $lb = \max(|d(q_p, r_{c_j}) - d_{parent}(q_{c_i})|, |d(q_{c_i}, r_p) - d_{parent}(r_{c_j})|)$ . Pairs whose lower bound minus combined `maxdist` exceeds the pruning threshold are skipped without additional distance computation.

Per-query-point bounds are cached in a flat `Vec` indexed by node index. Two bound modes are supported: full recursive B1/B2 bounds with epoch-based cache invalidation (the default), and a Beygelzimer-style bound. A cheaper per-point *k*th-distance bound was removed: it drops the per-subtree margin and can prune true neighbors.

#### C.4 Parallel Construction

Parallel construction follows the partition-build-merge strategy of Izbicki and Shelton [8]. The dataset is partitioned into equal chunks, each thread builds an independent cover tree, and trees are merged via binary reduction in  $O(\log p)$  rounds. The merge algorithm aligns tree levels, partitions children into covered and uncovered sets, recursively merges covered children, and reinserts uncovered points. Exact `maxdist` recomputation after merge restores optimal pruning bounds.

## D Full Ablation Tables

Table 7 shows the cumulative waterfall: starting from “all off” (all five optimizations disabled), each row adds one optimization. Values are speedup over all-off ( $\times$ ); the final “Baseline” row gives the combined speedup of all optimizations. This table and Table 1 report the same endpoint quantity,  $t(\text{all off})/t(\text{baseline})$ , and are computed from a *single pooled 20-allocation sample*, so the “Baseline” row below is identical to Table 1’s “all off” row by construction rather than merely up to noise.

Table 7: Cumulative waterfall: speedup ( $\times$  all-off) as optimizations are added one at a time ( $k=1$ , all-NN, base 1.3). Median [95% CI] over 20 independent allocations. Optimizations are added in order of increasing sophistication. The “Baseline” row equals the “all off” row of Table 1 by construction (same pooled sample).

| Configuration | pima              | pendigits         | corel             | covtype           | mnist             |
|---------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| All off       | 1.00 [0.83, 1.21] | 1.00 [0.93, 1.08] | 1.00 [0.97, 1.03] | 1.00 [0.98, 1.02] | 1.00 [0.88, 1.13] |
| +Triangle     | 0.88 [0.77, 1.14] | 1.00 [0.92, 1.03] | 1.02 [0.99, 1.03] | 1.03 [1.01, 1.04] | 1.12 [0.98, 1.13] |
| + $k=1$ fast  | 1.14 [0.97, 1.52] | 0.99 [0.92, 1.03] | 1.01 [0.96, 1.02] | 1.02 [1.01, 1.03] | 1.12 [0.98, 1.16] |
| +SmallVec     | 0.91 [0.79, 1.26] | 1.00 [0.93, 1.04] | 1.01 [0.99, 1.03] | 1.06 [1.02, 1.09] | 1.12 [0.98, 1.14] |
| +Halfsort     | 1.24 [0.87, 1.59] | 0.96 [0.78, 1.23] | 1.08 [1.05, 1.09] | 1.02 [0.97, 1.05] | 1.06 [0.86, 1.14] |
| +Early-exit   | 1.11 [0.99, 1.31] | 0.78 [0.72, 0.83] | 1.06 [1.03, 1.07] | 1.33 [1.31, 1.34] | 1.58 [1.39, 1.60] |
| Baseline      | 1.11 [0.99, 1.31] | 0.78 [0.72, 0.83] | 1.06 [1.03, 1.07] | 1.33 [1.31, 1.34] | 1.58 [1.39, 1.60] |

On covtype ( $d=54$ ) and mnist ( $d=784$ ), nearly the entire combined effect comes from the final step: adding early-exit raises the speedup from  $1.02\times$  to  $1.33\times$  on covtype and from  $1.06\times$  to  $1.58\times$  on mnist, while the four traversal-level optimizations before it contribute at most about 12% (on mnist, within intervals that include 1.0). On corel ( $d=32$ ) the combined effect is only  $1.06\times$ , and adding early-exit does not increase it. The composition is largely additive on the large datasets: early-exit targets the cost of each distance computation while the other four target traversal overhead, so they operate on independent bottlenecks. On pima and pendigits the per-allocation timings are bimodal (Section 4.2), so no ordering of the intermediate rows on those two datasets is meaningful; we report them for completeness only.

Table 8 shows the full per- $k$  individual ablation results for the three larger datasets, where the intervals in Table 1 are tight enough for the numbers to carry meaning. Its  $k=1$  column is the same quantity as Table 1’s corresponding entries and is computed from the same 20-allocation source, so the two agree by construction.

Table 8: Per- $k$  ablation slowdown factors ( $\times$  baseline, all-NN, base 1.3) for corel, covtype, and mnist. Values  $>1$  indicate the optimization helps. Median over 20 independent allocations at every  $k$ ; the  $k=1$  column matches Table 1 by construction. Per-cell confidence intervals are omitted for width and are reported in the supplementary analysis output.

| Disabled   | corel ( $d=32$ ) |      |      |      | covtype ( $d=54$ ) |      |      |      | mnist ( $d=784$ ) |      |      |      |
|------------|------------------|------|------|------|--------------------|------|------|------|-------------------|------|------|------|
|            | $k=1$            | 5    | 10   | 50   | $k=1$              | 5    | 10   | 50   | $k=1$             | 5    | 10   | 50   |
| Early-exit | 0.98             | 0.99 | 0.98 | 0.98 | 1.31               | 1.24 | 1.24 | 1.18 | 1.49              | 1.28 | 1.26 | 1.18 |
| Halfsort   | 1.10             | 0.99 | 1.01 | 1.02 | 1.03               | 0.97 | 0.98 | 0.98 | 1.02              | 1.00 | 0.99 | 0.98 |
| SmallVec   | 0.99             | 1.02 | 1.03 | 1.05 | 1.01               | 0.98 | 0.99 | 0.98 | 1.02              | 0.99 | 0.99 | 0.99 |
| $k=1$ fast | 1.04             | 0.91 | 0.91 | 0.94 | 1.03               | 0.96 | 0.98 | 0.98 | 1.01              | 0.99 | 0.99 | 0.99 |
| Triangle   | 1.01             | 1.01 | 1.01 | 1.03 | 0.98               | 0.98 | 0.98 | 0.99 | 1.03              | 0.99 | 0.99 | 0.99 |
| All off    | 1.06             | 0.93 | 0.92 | 0.91 | 1.33               | 1.19 | 1.20 | 1.15 | 1.58              | 1.30 | 1.37 | 1.44 |

Early-exit has no effect on corel at any  $k$  ( $0.98\text{--}0.99\times$ ). On covtype and mnist its benefit shrinks as  $k$  grows, from  $1.31\times$  and  $1.49\times$  at  $k=1$  to  $1.18\times$  on both at  $k=50$ , consistent with a larger  $k$  widening the pruning radius, so that fewer candidate distances exceed the bound and can be cut short. The other optimizations (halfsort, SmallVec, triangle,  $k=1$  fast path) are within a few percent of  $1.0\times$  on covtype and mnist, confirming that traversal-level micro-optimizations have diminishing returns when distance computation dominates. The one sizable entry, corel’s  $k=1$  fast path at  $0.91\text{--}0.94\times$  for  $k\geq 5$ , comes from a path that is never taken at  $k>1$  and so is a build-to-build layout effect, not an effect of the optimization (Section 4.2).

## E Distance Computation Tables

Table 9 presents full distance computation counts across all datasets and variants for the all-NN benchmark at  $k=1$ .

Table 9: Distance computations for all-NN ( $k=1$ , base 1.3, dual-tree mode for Packed/NA Pk/NA, single-tree all-NN for Simplified/Parallel, the reference implementation’s batch query for C++) across all 12 datasets. Values in billions (B), millions (M) or thousands (K). The ablation in Section 4.2 measures the same dual-tree Packed mode, so its baseline counts are the Packed column here.

| Dataset    | Packed | NA Pk | NA    | Simpl. | Parallel | C++   |
|------------|--------|-------|-------|--------|----------|-------|
| iris       | 5.27K  | 5.22K | 5.22K | 6.62K  | 7.63K    | 2.63K |
| wine       | 3.75K  | 4.25K | 4.25K | 9.47K  | 11.7K    | 2.64K |
| glass      | 12.9K  | 11.9K | 11.9K | 19.5K  | 22.2K    | 5.63K |
| bupa       | 36.7K  | 37.6K | 37.6K | 44.6K  | 60.9K    | 17.9K |
| pima       | 77.1K  | 88.8K | 88.8K | 208K   | 192K     | 41.9K |
| ionosphere | 43.2K  | 50K   | 50K   | 45.8K  | 59.7K    | 18.1K |
| optdigits  | 13.2M  | 13.2M | 13.2M | 2.31M  | 5.57M    | 5.23M |
| pendigits  | 5.08M  | 5.85M | 5.85M | 5.07M  | 7.16M    | 2.81M |
| letter     | 42.7M  | 50.1M | 50.1M | 22.3M  | 26.9M    | 18.6M |
| corel      | 160M   | 197M  | 197M  | 50.5M  | 104M     | 76.6M |
| covtype    | 230M   | 483M  | 483M  | 1.55B  | 1.7B     | 136M  |
| mnist      | 1.87B  | 1.96B | 1.96B | 312M   | 510M     | 697M  |

## F Batch Single-Tree and Dual-tree Algorithm Diagrams

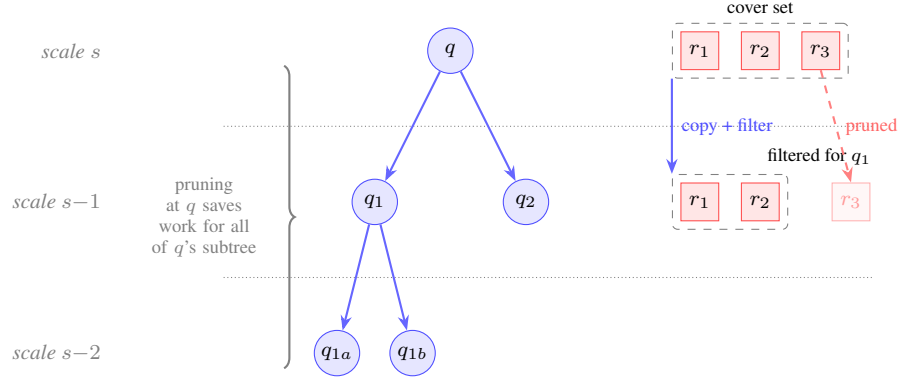


Figure 20: Batch single-tree query. The query tree (blue, left) is traversed top-down by scale level. At each query split, children inherit *filtered copies* of the parent's reference cover sets (red, right). A reference node pruned at  $q$  is never seen by any descendant, amortizing the pruning decision across all queries in  $q$ 's subtree.

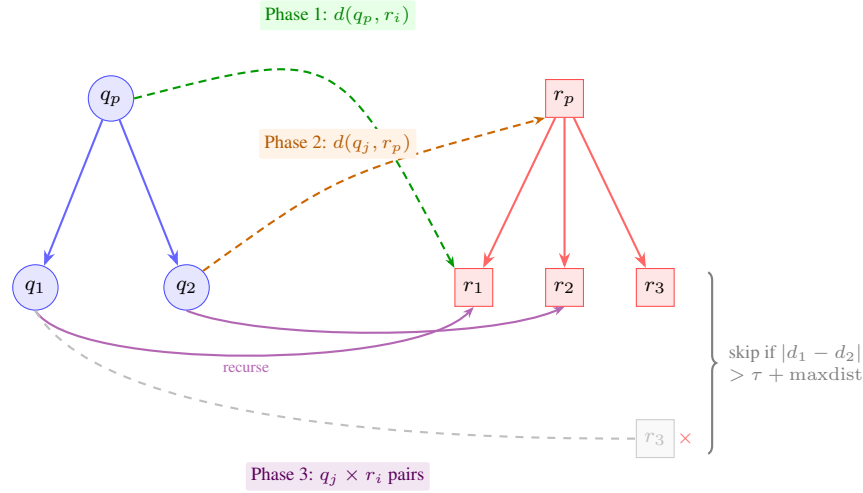


Figure 21: Dual-tree traversal. At each internal pair  $(q_p, r_p)$ , three phases execute: Phase 1 computes  $d(q_p, r_i)$  for each reference child; Phase 2 computes  $d(q_j, r_p)$  for each query child; Phase 3 evaluates all  $q_j \times r_i$  pairs, using triangle inequality bounds from Phases 1–2 to prune without additional distance computations. Pruned pairs (gray) skip entire subtrees. The quadratic  $|\text{children}_q| \times |\text{children}_r|$  expansion in Phase 3 is the main per-pair cost of the traversal.

## G All-NN $k=10$ Results

Table 10: All-nearest-neighbor query time (ms,  $k=10$ , base 1.3). Companion to Table 3 ( $k=1$ ). Median [95% CI] over 20 independent SLURM allocations. Best result per row in **bold**. Values  $\geq 10^3$  are abbreviated K/M.

| Dataset    | $d$ | RUSTKNN                   |                    |                    | C++                       | mlpack             | mlpack             |
|------------|-----|---------------------------|--------------------|--------------------|---------------------------|--------------------|--------------------|
|            |     | Dual                      | Single             | Batch              | Ref                       | Single             | Dual               |
| iris       | 4   | 1.33 [1.15, 1.37]         | 0.94 [0.77, 0.97]  | 1.46 [1.21, 1.48]  | <b>0.93 [0.80, 0.94]</b>  | 2.67 [2.30, 2.68]  | 2.69 [2.38, 2.72]  |
| wine       | 13  | 1.34 [1.11, 1.39]         | 1.44 [1.18, 1.51]  | 1.69 [1.40, 1.70]  | <b>1.10 [0.91, 1.16]</b>  | 3.27 [2.67, 3.50]  | 2.33 [2.01, 2.53]  |
| glass      | 9   | 2.83 [2.50, 3.28]         | 2.46 [2.14, 2.79]  | 2.86 [2.50, 3.28]  | <b>1.70 [1.51, 1.91]</b>  | 5.51 [4.57, 5.94]  | 5.05 [4.17, 5.31]  |
| bupa       | 6   | 6.46 [5.66, 6.88]         | 5.64 [4.80, 5.85]  | 5.44 [4.80, 5.82]  | <b>3.65 [3.42, 4.09]</b>  | 11.4 [10.7, 13.6]  | 10.3 [9.51, 12.6]  |
| pima       | 8   | 12.3 [10.5, 13.7]         | 20.0 [17.2, 22.5]  | 12.3 [10.9, 13.4]  | <b>7.50 [6.81, 7.55]</b>  | 30.4 [28.7, 31.2]  | 23.6 [23.4, 25.1]  |
| ionosphere | 34  | 6.49 [6.34, 6.88]         | 7.97 [7.93, 8.47]  | 7.39 [7.15, 7.80]  | <b>4.77 [4.52, 5.08]</b>  | 11.5 [10.5, 12.0]  | 10.8 [9.74, 11.2]  |
| optdigits  | 64  | 935 [837, 1,184]          | 772 [736, 902]     | 987 [969, 1,143]   | <b>728 [716, 785]</b>     | 2.13K [2.1, 2.24]  | 2.33K [2.29, 2.44] |
| pendigits  | 16  | 300 [299, 330]            | 383 [382, 409]     | 372 [371, 396]     | <b>260 [256, 284]</b>     | 1.28K [1.26, 1.36] | 949 [945, 1,012]   |
| letter     | 16  | 1.9K [1.89, 1.99]         | 1.9K [1.89, 2.01]  | 3.05K [3.03, 3.24] | <b>1.71K [1.71, 1.79]</b> | 9.19K [8.89, 9.54] | 9.95K [9.8, 10.4]  |
| corel      | 32  | <b>7.58K [7.27, 7.86]</b> | 9.88K [8.22, 10.2] | 23.6K [19, 24]     | 14.5K [12.7, 14.9]        | 49.6K [41.3, 50.3] | 73.2K [56.1, 75.2] |
| covtype    | 54  | <b>25.2K [22.1, 25.7]</b> | 185K [160, 190]    | 121K [103, 123]    | 55.1K [49.8, 55.8]        | 232K [214, 232]    | 177K [156, 180]    |
| mnist      | 784 | <b>886K [766, 950]</b>    | 2.90M [2.62, 3.96] | 2.98M [2.70, 3.76] | 3.20M [3.14, 3.51]        | 5.29M [5.21, 5.34] | 4.25M [3.20, 5.05] |

## H KD-Tree Baseline: Ablation and Cross-Codebase Comparison

This appendix supports Section 5.5. The KD-tree is a median-split tree implemented inside RUSTKNN (`src/kdtree/`), reusing the Distance trait (including `distance.with.bound`, the early-exit kernel), the same  $k$ -NN candidate structure, the same thread-local distance counter, and the same loader and timing code as the cover trees. Correctness was verified against brute-force search on every dataset (maximum coordinate discrepancy  $10^{-16}$ ). Because it shares the kernel, it runs the same five feature-flag ablations.

Table 11: Cross-structure and cross-codebase comparison (all-NN,  $k=1$ , ms). “KD-tree” and the two cover-tree columns share one codebase, kernel, and candidate structure, so their differences are attributable to the index itself. `kiddo` is an independent Rust KD-tree crate (median [95% CI] over 20 allocations) and isolates what changes when only the codebase differs. KD-tree and cover-tree columns come from the same dedicated single-allocation run, so that comparison is controlled same-run; the cover-tree values agree with Table 3 within 2% on covtype and within 7–19% on the other datasets, the differences reflecting cross-campaign variance (Section 5).

| Dataset   | $d$ | Same codebase |           |         | Independent             |
|-----------|-----|---------------|-----------|---------|-------------------------|
|           |     | KD-tree       | CT single | CT dual | <code>kiddo</code>      |
| pendigits | 16  | 71.9          | 171       | 199     | 7.77 [6.27, 9.80]       |
| corel     | 32  | 254           | 3,248     | 5,702   | 59.1 [55.3, 70.8]       |
| covtype   | 54  | 4,465         | 100,590   | 13,722  | 12,846 [12,777, 12,943] |
| mnist     | 784 | 4.09M         | 334,487   | 628,683 | 8.31M [8.29M, 8.34M]    |

Table 12: KD-tree optimization ablation: slowdown ( $\times$  baseline) when each flag is disabled ( $k=1$ , all-NN). Directly comparable to the cover-tree ablation in Table 1, since both share the distance kernel and candidate structure. Median of five runs in a single allocation.

| Disabled optimization | pima<br>( $d=8$ ) | pendigits<br>( $d=16$ ) | corel<br>( $d=32$ ) | covtype<br>( $d=54$ ) | mnist<br>( $d=784$ ) |
|-----------------------|-------------------|-------------------------|---------------------|-----------------------|----------------------|
| Early-exit distance   | 0.90              | 0.98                    | 0.99                | 0.97                  | 1.11                 |
| Halfsort              | 0.62              | 1.04                    | 1.00                | 1.00                  | 1.01                 |
| SmallVec              | 1.02              | 1.01                    | 0.99                | 0.98                  | 1.00                 |
| $k=1$ fast path       | 1.09              | 1.00                    | 1.00                | 1.01                  | 1.00                 |
| Triangle pre-filter   | 0.70              | 1.00                    | 1.00                | 0.99                  | 1.02                 |
| All off               | 0.98              | 0.98                    | 0.96                | 1.02                  | 1.11                 |

The pattern follows the cover-tree result in the way one would predict from first principles: early-exit distance is the only flag with an effect beyond noise, and only at high dimension ( $1.11\times$  at  $d=784$ ;  $\approx 1\times$  at  $d \leq 54$ ), because it acts on per-distance cost, which is what scales with  $d$ . The traversal-level flags designed around cover-tree invariants—halfsort and the triangle pre-filter—sit at  $\approx 1\times$  on a structure whose traversal they do not describe. As with the cover trees, the pima column is noise-dominated and we draw nothing from it.

Table 13 gives the KD-tree’s distance counts, which make the curse-of-dimensionality crossover explicit: on mnist the KD-tree issues 594M distance computations (each up to 784-dimensional), versus 312M for the simplified cover tree in single-tree mode (Table 9), because axis-aligned box pruning stops discriminating at high  $d$ .

For the cross-codebase column we used `kiddo` 5.3.2, an actively maintained independent Rust KD-tree crate, via its `ImmutableKdTree` with `nearest_n::<SquaredEuclidean>`. It was run under the same 20-allocation protocol as the main tables (all-NN,  $k=1$ ); because `kiddo` is compiled as part of the same Rust project, it inherits the same RUSTFLAGS (including `target-cpu=native`), so the ISA disparity noted in Appendix B does not apply to this comparison. `kiddo` necessarily uses its own distance kernel and candidate structure, which is why it cannot run our ablation flags and why we implemented an in-harness tree for the controlled comparison: an external library alone would reintroduce exactly the cross-codebase confounding this paper is about. Used together, the two answer different questions—the in-harness tree isolates the index, and `kiddo` quantifies what the codebase alone contributes.

Table 13: KD-tree all-NN metric evaluations ( $k=1$ ), all 12 datasets. Cover-tree counts for all 12 datasets appear in Appendix E.

| Dataset    | $d$ | KD-tree distance evaluations |
|------------|-----|------------------------------|
| iris       | 4   | 1,126                        |
| wine       | 13  | 1,207                        |
| glass      | 9   | 1,534                        |
| bupa       | 6   | 2,914                        |
| pima       | 8   | 7,605                        |
| ionosphere | 34  | 3,773                        |
| optdigits  | 64  | 417,599                      |
| pendigits  | 16  | 274,892                      |
| letter     | 16  | 2,111,157                    |
| corel      | 32  | 566,391                      |
| covtype    | 54  | 10,982,600                   |
| mnist      | 784 | 594,194,887                  |

## I Naive Search with Partial Distance Computation

A reasonable question about any index is whether it earns its complexity. We added a naive variant—a linear scan—to the same harness, using the same early-exit (“partial distance”) Euclidean kernel and the same distance counter as every other implementation. Building it with no-early-exit yields the full-distance twin. Both perform identical  $n^2$  distance *calls*, so any runtime difference is purely the per-call dimension work that early exit skips—this isolates the partial-distance effect on the simplest possible baseline.

Table 14: Naive all-NN ( $k=1$ ): full vs. partial distance computation. Both columns issue the same number of distance calls; the difference is per-call work avoided. Median of three runs in one allocation; on covtype, the six runs (three per variant) ran concurrently on one node so both variants saw the same load.

| Dataset    | $d$ | Full (ms)  | Partial (ms) | Speedup | $n^2$ distance calls |
|------------|-----|------------|--------------|---------|----------------------|
| pima       | 8   | 5.2        | 4.6          | 1.14×   | 589,824              |
| ionosphere | 34  | 4.3        | 3.0          | 1.41×   | 123,201              |
| pendigits  | 16  | 1,019      | 1,110        | 0.92×   | 120,824,064          |
| optdigits  | 64  | 1,047      | 1,065        | 0.98×   | 31,584,400           |
| letter     | 16  | 3,519      | 3,678        | 0.96×   | 400,000,000          |
| corel      | 32  | 51,901     | 20,603       | 2.52×   | 1,424,987,001        |
| covtype    | 54  | 34,477,113 | 21,721,950   | 1.59×   | 337,574,944,144      |
| mnist      | 784 | 4,206,345  | 2,551,888    | 1.65×   | 3,600,000,000        |

The partial-distance benefit depends on the dataset more than on its dimension: it is large on corel ( $2.52\times$ ,  $d=32$ ), mnist ( $1.65\times$ ,  $d=784$ ) and covtype ( $1.59\times$ ,  $d=54$ ), but negligible or slightly negative on pendigits, letter, and optdigits ( $0.92$ – $0.98\times$ ,  $d=16$ – $64$ ), where the branch checks cost about as much as the work they skip. Corel is the instructive case: early exit is worth  $2.5\times$  in the linear scan but nothing in the cover tree (Table 1), consistent with the tree having already pruned most of the far candidates whose distance computations early exit would otherwise cut short. The benefit of a kernel-level optimization thus depends on the structure that calls the kernel, not only on the data.

The absolute numbers also answer the “is an index worth it?” question directly. Naive all-NN on mnist requires  $3.6\times 10^9$  distance calls (43–70 minutes) and on covtype  $3.4\times 10^{11}$  (6–10 hours), against milliseconds to minutes for the trees. At the same time, on the smallest datasets the naive scan is competitive in absolute terms—another reason the tiny-dataset results in this paper should not drive implementation decisions.

## J Synthetic Manifolds with Known Intrinsic Dimension

The cover tree’s  $O(c^6 \log n)$  query bound is stated in terms of the *doubling* dimension, not the ambient dimension, but on real datasets the intrinsic dimension is unknown, so the distinction cannot be tested directly. We therefore generated synthetic data where it is known by construction:  $n=20,000$  points sampled on a  $d$ -dimensional latent manifold, then linearly embedded in ambient dimension  $D \geq d$ : latent coordinates are uniform on  $[0, 1]^d$ , the embedding is a fixed random  $D \times d$  matrix with  $\mathcal{N}(0, 1)$  entries, and isotropic Gaussian noise ( $\sigma=0.01$ ) puts points near rather than exactly on the manifold, with a fixed seed per configuration. These run through the harness unchanged. Two sweeps separate the two dimensions.

Table 15: Synthetic manifolds ( $n=20,000$ , all-NN  $k=1$ ). Cover-tree cost tracks the *intrinsic* dimension  $d$  and does not grow with the *ambient* dimension  $D$ . KD-tree distance counts are shown for contrast: they are near-identical ( $267K \pm 5$ ) across all configurations. Median of three runs in a single allocation.

| Sweep                   | Configuration | CT time (ms) | CT distance comps. | KD distance comps. |
|-------------------------|---------------|--------------|--------------------|--------------------|
| A: vary $d$ ( $D=100$ ) | $d=2$         | 669          | 5,404,434          | 267,253            |
|                         | $d=5$         | 1,203        | 8,782,805          | 267,248            |
|                         | $d=10$        | 3,182        | <b>22,633,024</b>  | 267,248            |
| B: vary $D$ ( $d=5$ )   | $D=20$        | 501          | 9,765,110          | 267,248            |
|                         | $D=50$        | 918          | 10,985,580         | 267,248            |
|                         | $D=100$       | 1,203        | <b>8,782,805</b>   | 267,248            |

Sweep A raises the intrinsic dimension at fixed ambient dimension: cover-tree distance computations rise steeply,  $5.4M \rightarrow 8.8M \rightarrow 22.6M$ , a  $4.2\times$  increase for a  $5\times$  increase in  $d$ . Sweep B raises the ambient dimension  $5\times$  at fixed intrinsic dimension: distance computations do *not* rise monotonically ( $9.8M \rightarrow 11.0M \rightarrow 8.8M$ —variation is about 25%, consistent with different random embedding matrices across configurations rather than an intrinsic-dimension effect). This is direct empirical support for the claim that cover-tree query cost is governed by intrinsic rather than ambient dimensionality, which the theory asserts but which, to our knowledge, has not previously been isolated experimentally.

Wall-clock behaves differently from distance count in Sweep B, and the difference is informative: cover-tree time still rises with  $D$  ( $501 \rightarrow 1,203$  ms) even though the number of distance computations does not, because each computation is  $D$ -dimensional. This is the distance-count-versus-runtime decoupling discussed in Section 4.2, and it is why we report both throughout: a paper reporting only distance counts here would conclude, incorrectly, that ambient dimension is free.

## K Literature Survey: Evaluation Practices in Metric Space Search

To contextualize our experimental contributions, we surveyed 61 papers from the metric space nearest neighbor search literature and classified each on four evaluation-practice dimensions that our work addresses:

1. **Micro-optimizations** — Does the paper discuss implementation-level micro-optimizations (e.g., SIMD, cache layout, memory coalescing, bit tricks) as opposed to purely algorithmic improvements?
2. **Runtime & distance counts** — Does the paper measure *both* wall-clock runtime and the number of distance computations (or avoided/pruned computations) in its experiments?
3. **Batch vs. single query** — Does the paper compare or test between bulk/batch querying and single-item-at-a-time querying?
4. **Query from set vs. held-out** — Does the paper explicitly differentiate or test between queries drawn from the indexed dataset versus new held-out points not in the data structure?

Table 16 presents the full classification. A “1” indicates the paper reasonably addresses the criterion; “0” indicates it does not.

Table 16: Literature classification of 61 metric space search papers on four evaluation-practice criteria.

| Paper | <i>Micro-opt.</i> | <i>Runtime &amp;<br/>dist. counts</i> | <i>Batch vs.<br/>single</i> | <i>Set vs.<br/>held-out</i> |
|-------|-------------------|---------------------------------------|-----------------------------|-----------------------------|
| [42]  | 0                 | 0                                     | 0                           | 0                           |
| [43]  | 0                 | 0                                     | 0                           | 0                           |
| [44]  | 0                 | 0                                     | 0                           | 0                           |
| [10]  | 0                 | 0                                     | 0                           | 0                           |
| [45]  | 0                 | 0                                     | 0                           | 0                           |
| [1]   | 0                 | 0                                     | 0                           | 1                           |
| [46]  | 1                 | 0                                     | 1                           | 0                           |
| [47]  | 0                 | 1                                     | 0                           | 0                           |
| [13]  | 0                 | 0                                     | 0                           | 0                           |
| [48]  | 0                 | 1                                     | 0                           | 1                           |
| [49]  | 0                 | 0                                     | 0                           | 0                           |
| [11]  | 0                 | 0                                     | 0                           | 0                           |
| [7]   | 0                 | 0                                     | 0                           | 0                           |
| [5]   | 0                 | 1                                     | 0                           | 0                           |
| [50]  | 0                 | 1                                     | 0                           | 0                           |
| [51]  | 1                 | 0                                     | 1                           | 1                           |
| [52]  | 0                 | 0                                     | 0                           | 0                           |
| [53]  | 0                 | 0                                     | 0                           | 1                           |
| [54]  | 0                 | 0                                     | 0                           | 1                           |
| [55]  | 0                 | 0                                     | 0                           | 1                           |
| [8]   | 1                 | 1                                     | 0                           | 1                           |
| [56]  | 0                 | 0                                     | 1                           | 1                           |
| [57]  | 0                 | 0                                     | 0                           | 0                           |
| [4]   | 0                 | 0                                     | 0                           | 0                           |
| [58]  | 0                 | 0                                     | 0                           | 0                           |
| [59]  | 0                 | 0                                     | 0                           | 0                           |
| [12]  | 0                 | 1                                     | 1                           | 1                           |
| [60]  | 0                 | 0                                     | 0                           | 1                           |
| [2]   | 0                 | 0                                     | 0                           | 0                           |
| [61]  | 0                 | 0                                     | 0                           | 0                           |

*continued on next page*

Table 16: (continued)

| Paper | <i>Micro-opt.</i> | <i>Runtime &amp;<br/>dist. counts</i> | <i>Batch vs.<br/>single</i> | <i>Set vs.<br/>held-out</i> |
|-------|-------------------|---------------------------------------|-----------------------------|-----------------------------|
| [62]  | 0                 | 0                                     | 0                           | 1                           |
| [63]  | 0                 | 0                                     | 0                           | 0                           |
| [64]  | 0                 | 0                                     | 0                           | 0                           |
| [65]  | 0                 | 0                                     | 0                           | 0                           |
| [66]  | 0                 | 0                                     | 0                           | 0                           |
| [67]  | 0                 | 0                                     | 0                           | 1                           |
| [68]  | 0                 | 0                                     | 0                           | 1                           |
| [69]  | 0                 | 0                                     | 0                           | 0                           |
| [70]  | 0                 | 0                                     | 0                           | 0                           |
| [71]  | 0                 | 0                                     | 0                           | 0                           |
| [72]  | 0                 | 0                                     | 0                           | 0                           |
| [37]  | 0                 | 0                                     | 0                           | 0                           |
| [73]  | 0                 | 1                                     | 0                           | 0                           |
| [74]  | 0                 | 0                                     | 0                           | 1                           |
| [75]  | 0                 | 0                                     | 1                           | 0                           |
| [76]  | 0                 | 0                                     | 0                           | 0                           |
| [77]  | 1                 | 0                                     | 1                           | 0                           |
| [78]  | 0                 | 0                                     | 0                           | 0                           |
| [79]  | 0                 | 0                                     | 0                           | 0                           |
| [80]  | 0                 | 0                                     | 0                           | 1                           |
| [81]  | 0                 | 0                                     | 0                           | 0                           |
| [82]  | 0                 | 0                                     | 0                           | 0                           |
| [83]  | 0                 | 0                                     | 0                           | 0                           |
| [84]  | 1                 | 0                                     | 0                           | 1                           |
| [85]  | 0                 | 0                                     | 0                           | 0                           |
| [86]  | 1                 | 0                                     | 0                           | 0                           |
| [87]  | 0                 | 1                                     | 0                           | 0                           |
| [88]  | 0                 | 0                                     | 0                           | 0                           |
| [39]  | 0                 | 0                                     | 0                           | 0                           |
| [89]  | 0                 | 0                                     | 0                           | 0                           |
| [90]  | 1                 | 1                                     | 0                           | 0                           |

**Summary.** Of the 61 papers surveyed:

- Only **7** (11%) discuss implementation-level micro-optimizations, indicating that the vast majority of metric space search work operates at the algorithmic level without examining how implementation details interact with algorithmic choices.
- Only **9** (15%) report *both* wall-clock runtime and distance computation counts, despite the fact that these two metrics can diverge significantly (e.g., fewer distance computations does not guarantee faster runtime when per-computation cost varies or overhead dominates).
- Only **6** (10%) compare batch/bulk querying against single-item querying—and these are almost exclusively the dual-tree and GPU-parallelism papers.
- Only **16** (26%) explicitly test with held-out query points distinct from the indexed dataset, despite the fact that production workloads nearly always involve queries unseen at index-construction time.

These gaps motivate the four contributions of this paper: our optimization ablation (addressing gap 1), joint runtime and distance reporting (gap 2), query-mode comparison (gap 3), and all-NN vs. held-out evaluation (gap 4).