Pkd-tree: Parallel kd-tree with Batch Updates

Ziyang Men
zmen002@ucr.edu
UC Riverside

Zheqi Shen
zshen055@ucr.edu
UC Riverside

Yan Gu
ygu@cs.ucr.edu
UC Riverside

Yihan Sun
yihans@cs.ucr.edu
UC Riverside

Abstract

The kd-tree is one of the most widely used data structures to manage multi-dimensional data. Due to the ever-growing data volume, it is imperative to consider parallelism in kd-trees. However, we observed challenges in existing parallel kd-tree implementations, for both constructions and updates.

The goal of this paper is to develop efficient in-memory kd-trees by supporting high parallelism and cache-efficiency. We propose the Pkd-tree (Parallel kd-tree), a parallel kd-tree that is efficient both in theory and in practice. The Pkd-tree supports parallel tree construction, batch update (insertion and deletion), and various queries including k-nearest neighbor search, range query, and range count. We proved that our algorithms have strong theoretical bounds in work (sequential time complexity), span (parallelism), and cache complexity. Our key techniques include 1) an efficient construction algorithm that optimizes work, span, and cache complexity simultaneously, and 2) reconstruction-based update algorithms that guarantee the tree to be weight-balanced. With the new algorithmic insights and careful engineering effort, we achieved a highly optimized implementation of the Pkd-tree.

We tested Pkd-tree with various synthetic and real-world datasets, including both uniform and highly skewed data. We compare the Pkd-tree with state-of-the-art parallel kd-tree implementations. In all tests, with better or competitive query performance, Pkd-tree is much faster in construction and updates consistently than all baselines. We released our code.

1 Introduction

The kd-tree is one of the most widely-used data structures for managing multi-dimensional data. A kd-tree maintains a set of points in D dimensions[1], and supports various queries such as k-nearest neighbor (k-NN), orthogonal range count and range report. Compared to other counterparts, the kd-tree has its unique advantages, such as linear space, simple algorithms, being comparison-based (and thus resistant to skewed data), scaling to reasonably-large dimensions (being efficient up to $D \approx 10$) and supporting a wide range of query types. Due to these advantages, the kd-tree is the choice of data structure in many applications. Indeed, after its invention by Bentley in 1975 [11], kd-tree has been widely used and cited by over ten thousand times across multiple areas such as databases [23, 40, 44, 59], data science [30, 63, 80, 87], machine learning [28, 56, 57, 74], clustering algorithms [55, 58, 61, 72, 76], and computational geometry [21, 43, 60, 78].

Due to the ever-growing data volume, it is imperative to consider parallelism in kd-trees. For instance, the North American region of OpenStreetMap [46] contains 1.29 billion nodes, and building a kd-tree for this dataset on a single core using CGAL [81], a widely-adopted geometry library, takes over 2000 seconds. However, we observe a *significant gap* between the *wide usage* of kd-trees, and a *lack of high-performance parallel implementation* of kd-trees for all three aspects of construction, updates, and queries. Some existing parallel implementations (e.g. [69]) are static and do not support updates. The two parallel libraries for dynamic kd-trees that we are aware of, CGAL [81], and ParGeo [83] (which includes two kd-tree implementations Log-tree and BHL-tree), both have difficulties scaling to today's large-scale data size (see a summary of results in Tab. 1). CGAL and the BHL-tree do not support parallel updates. Even in the sequential updates, they fully rebuild the tree for rebalancing, which is inefficient. The Log-tree parallelizes updates using the classic *logarithmic method* [2, 11, 67]. The logarithmic method avoids fully rebuilding the tree upon update by maintaining $O(\log n)$ perfectly balanced trees with different sizes, such that an update reorganizes the trees by merging some of them in parallel. Accordingly, a query processes all $O(\log n)$ trees and combines the results, which can be significantly more expensive than it on a single kd-tree. As shown in Tab. 1, the Log-tree, despite being faster on updates, can be up to an order of magnitude slower than the BHL-tree or CGAL on k-NN queries. In addition, the construction for these kd-trees is also much slower than the time reported in recent works of other parallel tree structures, such as binary search trees [31, 79] and quad/octrees [15][2], indicating significant space for improvements on the construction algorithm.

In this paper, we overcome the above challenges in existing work by proposing the *Pkd-tree* (Parallel kd-tree), a parallel in-memory kd-tree that is efficient both in theory and in practice. Pkd-tree supports efficient construction, batch-update, and various query types. Our algorithms have strong theoretical bounds in work (sequential time complexity), span (parallelism), and cache complexity. Our key techniques include 1) an efficient construction algorithm that optimizes work, span, and cache complexity simultaneously, and 2) reconstruction-based update algorithms that guarantee the tree to be weight-balanced. With the new algorithmic insights and careful engineering effort, we achieved a highly optimized implementation of the Pkd-tree.

Construction. Our first contribution is a new parallel algorithm to construct weight-balanced kd-trees, given in Sec. 3. To the best of our knowledge, this is the first kd-tree construction algorithm with optimal $O(n \log n)$ work and $O((n/B) \log_M n)$ cache complexity, and polylogarithmic span, all with high probability[3], where n is the tree size, M is the cache size, and B is the cacheline size. To achieve good bounds on all three metrics simultaneously, the algorithmic

To appear: International Conference on Management of Data (SIGMOD), 2025

[1] Based on the original terminology, kd-tree deals with k-dimensional data. To avoid overloading k in different scenarios such as the "k-NN" query (i.e., finding k nearest neighbors of a given point), we use D as the number of dimensions in this paper.

[2] For example, the construction time of the parallel binary search tree reported in [79] is 28s on $n = 10^{10}$ elements on a similar machine, which is faster than all previous kd-tree implementations on $n = 10^9$ shown in Tab. 1.

[3] The *work* of a parallel algorithm is the total number of operations (i.e., sequential time complexity), and its *span* is the longest dependence chain. All the terms here are formally defined in Sec. 2.

Benchmark (10^9-2D)	Baselines	Build	Batch Insert				Batch Delete				10-NN	Range Report
			0.01%	0.1%	1%	10%	0.01%	0.1%	1%	10%	10^7 queries	10^4 queries
`Uniform`	Ours	3.15	.004	.020	.104	.495	.004	.022	.121	.526	.381	.391
	Log-tree	37.9	.008	.059	2.16	30.7	.436	.168	.396	3.01	2.96	2.62
	BHL-tree	31.7	31.0	31.2	31.4	39.9	30.8	31.1	30.9	30.6	.487	2.06
	CGAL	1147	1614	1562	1631	1660	.400	3.89	41.2	427	1.04	311
`Varden`	Ours	3.66	.002	.007	.055	.473	.002	.006	.049	.477	.172	.382
	Log-tree	34.2	.008	.057	2.01	28.0	.799	.848	1.06	3.47	2.05	2.63
	BHL-tree	30.2	29.1	29.2	29.4	37.3	29.3	29.2	29.0	28.0	.239	1.95
	CGAL	429	867	867	849	836	.113	1.06	13.0	153	.511	296

Table 1: Running time (in seconds) for Pkd-tree and other baselines on 10^9 points in 2 dimensions. Lower is better. "Log-tree": the parallel *k*d-tree using logarithmic method from the ParGeo library [83]. "BHL-tree": the single parallel *k*d-tree from ParGeo library [83]. "CGAL": the *k*d-tree from the CGAL library [81]. "Varden": a skewed distribution from [38]. "10-NN": 10-nearest-neighbor queries on 10^7 points. "Range report": orthogonal range report queries on 10^4 rectangles, with output sizes in 10^4–10^6. Experiments are run on a 96-core machine. More details are in Sec. 6. The fastest time for each test is underlined.

highlight here is to 1) determine the splitting hyperplane using a carefully designed sampling scheme, and 2) a ***sieving algorithm*** to partition all points into subspaces of λ levels in the *k*d-tree by *one round of data movement*. By picking $\lambda = \Theta(\log M)$, we achieve strong theoretical bounds for the construction algorithm and good performance in practice due to the saving of memory accesses.

Updates. The *k*d-tree differs from other classic trees and does not support rebalancing primitives for updates, such as overflow/underflow (as in B-trees), or rotations (as in binary search trees). Our idea is to keep the tree weight-balanced, and use a ***lazy strategy*** that tolerates the difference of sibling subtree sizes by a predefined and controllable weight-balancing factor of α before invoking rebalancing by ***locally reconstructing*** the affected subtrees. The idea of rebalancing via local reconstruction was originally proposed by Overmars from early 80s, and has been studied in the sequential setting [6, 20, 37, 64, 65]. However, it remained previously unknown about the efficiency on parallelism and cache complexity. Interestingly, the efficiency of our update algorithm is achieved by making use of our new construction algorithm—first, rebalancing the tree relies on efficient reconstruction; second, both insertion and deletion use the *sieving process* in construction as a subroutine to also achieve good cache complexity and parallelism. We present our update algorithms and the cost analysis in Sec. 4.

Queries. Since the Pkd-tree remains as a single *k*d-tree, the same query algorithms for static *k*d-trees can directly work on Pkd-trees without any modifications. In Sec. 6, we will show that the query performance for Pkd-trees is faster or as fast as existing solutions.

We implemented the Pkd-tree and conducted extensive experiments in Sec. 6. A short summary is given in Tab. 1 on two datasets with 10^9 2D points. In a nutshell, Pkd-trees are much faster in all aspects. For construction, the performance gain is from better cache complexity—data movement can be greatly saved by constructing multiple levels in one round. Compared to the logarithmic method (Log-tree), the Pkd-tree is 2.02–62.0× faster on insertion, 3.27–400× faster on deletion, and 6.71–11.9× faster on queries by avoiding keeping $O(\log n)$ trees. Compared to full reconstruction on updates (BHL-tree and CGAL), the Pkd-tree is orders of magnitude faster on updates and has better query performance. We show running time on real-world datasets, and in-depth experiments with varying dimensions, query types and parameters, individual techniques, scalability, and more, in Sec. 6. We believe that the Pkd-tree is the first *k*d-tree that is highly performant, parallel, and dynamic. We release our code at [62].

2 Preliminaries

We present a table of notations used in this paper in Tab. 2. We use ***with high probability*** (*whp*) in terms of n to mean probability at least $1 - n^{-c}$ for any constant $c > 0$. With clear context, we omit "in terms of n". We use $\log n$ as a short term of $\log_2(1 + n)$.

Computational Model. We analyze our algorithms using the ***work-span model*** in the classic fork-join paradigm with binary-forking [9, 17, 22]. We assume multiple threads that share memory. Each thread is a sequential RAM augmented with a fork instruction, which spawns two child threads that run in parallel. The parent thread resumes execution upon the completion of both child threads. A parallel for-loop can be simulated by a logarithmic number of steps of forking. A computation can be viewed as a directed acyclic graph (DAG). The ***work (W)*** of a parallel algorithm is the total number of operations within its DAG (aka. time complexity in the sequential setting), and the ***span (S)*** depicts the longest path in the DAG. Using a randomized work-stealing scheduler, a computation with work W and span S can be executed in $W/\rho + O(S)$ time *whp* (in W) with ρ processors [9, 22, 42].

We use the classic ***ideal-cache model*** [36] to measure the cost of memory accesses, which is widely used to analyze the cache complexity of algorithms [7, 18, 19, 32]. In this model, the memory is divided into two levels. The CPU is connected to the small-memory (aka. the cache) of finite size M, which is connected to a large-memory (the main memory) of infinite size. Both memories are organized as ***blocks*** with size B. The CPU can only access the data in small-memory with free cost, and there is a unit cost to transfer one block from large-memory to small-memory, assuming the optimal offline cache replacement policy. The cache complexity of an algorithm is number of block transfers during the algorithm.

The ideal-cache model assumes optimal eviction strategy for theoretical analysis. It has been shown that practical cache policies (e.g., LRU) enables the same asymptotic cache I/Os as the optimal strategy [36, 77]. The ideal-cache model is only used for theoretical analysis. In our implementation, we do not control the cache, so the optimal eviction strategy is guaranteed. In reality, the eviction strategy is usually a combination of multiple strategies, considering more complicated components such as set associativity, parallelism, and a few optimizations. However, the theoretical model is still extremely widely used as a good estimation of the practice.

The *k*d-Tree. We study points in Euclidean space in D dimensions, and the distance between two points is their Euclidean distance. A partition hyperplane can be represented by a pair $\langle d, x \rangle$, where d

$\boldsymbol{T}$	a (sub-)kd-tree, also the set of points in the tree		
$\boldsymbol{\phi}$	leaf wrap threshold (leaf size upper bound)		
$\boldsymbol{k}$	required number of nearest neighbors in a query		
$\boldsymbol{\lambda}$	number of levels in a tree sketch (i.e., that are built at a time)		
$\boldsymbol{\mathcal{T}}$	tree skeleton at T with maximum levels λ		
$\boldsymbol{P}$	input point set (for updates, P is the batch to be updated)		
$\boldsymbol{T.lc}$	left child of T	$\boldsymbol{T.rc}$	right child of T
$\boldsymbol{n}$	tree size	$\boldsymbol{m}$	batch size for batch updates
$\boldsymbol{D}$	number of dimensions	$\boldsymbol{d}$	a certain dimension
$\boldsymbol{S}$	samples from P	$\boldsymbol{s}$	size of the S
$\boldsymbol{\sigma}$	oversampling rate	$\boldsymbol{\alpha}$	balancing parameter
$\boldsymbol{M}$	small-memory (cache) size	$\boldsymbol{B}$	block (cacheline) size

Table 2: Notations used in this paper.

($1 \le d \le D$) is the ***splitting dimension*** and $x \in \mathbb{R}$ is the ***splitting coordinate***. We refer to such a pair s as a ***splitter***.

The ***kd-tree*** (k-dimensional tree), is a spatial-partitioning binary tree data structure. To avoid overloading the commonly-used parameter k in k-NN query, we use D to refer to the number of dimensions of the dataset. Each interior (non-leaf) node in a kd-tree signifies an axis-aligned splitting hyperplane $\langle d, x\rangle$. Points to the left of the hyperplane (those with the d-th dimension coordinates smaller than x) are stored in the left subtree and the remaining are in the right subtree. Each subtree is split recursively until the number of points drops below a *leaf wrap* threshold ϕ (a small constant), where all the points are directly stored in a leaf. Common approaches for choosing the dimension of the splitting hyperplane include cycling among the D dimensions [11], choosing the dimension with the widest stretch [35], etc. Pkd-tree also uses the widest dimension as the cutting dimension. The cutting coordinate x is usually the median of the points on the d-th dimension, yielding two balanced subtrees. Given n points in D dimensions, a balanced kd-tree has a height of $\log_2 n + O(1)$ and can be constructed in $O(n \log n)$ work using $O(n)$ space.

The kd-tree can answer various types of queries. Since the Pkd-tree remains a single kd-tree, the same query algorithms for classic kd-trees also work on Pkd-trees. In our experiments, we focus on *k-NN queries* (finding the k nearest points to a query point), rectangle *range report queries* (reporting all points within an axis-aligned bounding box) and rectangle *range count queries* (reporting the number of points within an axis-aligned bounding box).

We use the (subtree) root pointer T to denote a (sub-)kd-tree. With clear context, we also use T to represent the set of all points in T. Every interior node in T maintains two pointers $T.lc$ and $T.rc$ to its left and right children, respectively. As we mentioned, Pkd-tree is *weight-balanced.* Given the balancing parameter $\alpha \in [0, 0.5]$, we say a kd-tree is (weight-)balanced if $0.5 - \alpha \le |T.lc|/|T| \le 0.5 + \alpha$, and *unbalanced* otherwise. Essentially, this means that the two subtrees can be off from perfectly balanced by a factor of α.

3 Parallel Algorithm for Tree Construction

We start with our parallel kd-tree construction algorithm. Constructing a kd-tree requires partitioning the points into nested subspaces recursively based on the median of the splitting dimension. It directly implies a parallel construction algorithm with $O(n \log n)$ work and $O(\log^2 n)$ span using the standard parallel partition algorithm ($O(n)$ work and $O(\log n)$ span). However, it requires $O(\log n)$ rounds of data movement and is not cache-efficient when the input is larger than the cache size. We will refer to this algorithm as the *plain parallel kd-tree construction* algorithm. This is also the algorithm used by the BHL-tree from ParGeo [83].

Instead of partitioning all points into the left and right subtrees and pushing the points to the next level in the recursive calls, the high-level idea of our approach is to build λ levels at a time by one round of the data movement. To avoid the data movement for finding splitting coordinates, our algorithm uses samples to decide all splitters for λ levels. It then distributes all points into the corresponding subtrees (2^{λ} of them) and recurses.

The main challenges here are 1) to use only a subset of the points (samples) to decide the splitters and make the tree nearly balanced, and 2) to move each point exactly once to its final destination in a cache-efficient and parallel manner. Below, we will first elaborate on our parallel and cache-efficient construction algorithm in Sec. 3.1, and show the cost analysis in Sec. 3.2.

3.1 Algorithms Description

We present our algorithm in Alg. 1 and an illustration in Fig. 1. The algorithm $T = \textsc{BuildTree}(P)$ builds a kd-tree T on the input points in array P. As mentioned, our main idea is to use samples to decide all splitters in λ levels. We define the *skeleton* at T, denoted as $\mathcal{T}$, as the substructure consisting of all splitters (and thus interior nodes) in the first λ levels. We use the samples to build the skeleton. In particular, we will uniformly take $2^{\lambda} \cdot \sigma$ samples from P, where σ is the *over sampling rate.* In Sec. 3.2 we will show how to choose the parameter σ to achieve strong theoretical guarantees. Let S be the set of sample points. The skeleton will be the first λ levels of the kd-tree on S. As we will show in Sec. 3.2, we will keep S small and fit in cache, so that the skeleton can be built by the plain parallel kd-tree construction algorithm at the beginning of Sec. 3.

The skeleton depicts the first λ levels of the tree, and splits the space into 2^{λ} subspaces, corresponding to the external nodes (leaves) of the skeleton. We call each such external node a ***bucket*** in this skeleton. We label all buckets from 0 to $2^{\lambda} - 1$. The problem then boils down to sieving the points into the corresponding bucket, so that we can further deal with each bucket recursively in parallel. We first note that the target bucket for each point can be easily looked up in $O(\lambda)$ cost by searching in the skeleton. Sequentially, one can simply move all points one by one to their target bucket, by maintaining a pointer to the (current) last element in each bucket. In parallel, the key challenge is to independently determine the "offset" of each point, so the points can be moved to their target buckets in parallel without introducing locks or data races.

We borrow the idea from the cache-efficient parallel sorting algorithm [18, 32, 33], which also involves redistributing elements into $\omega(1)$ buckets. Our goal is to reorder array P and make all points belonging to the same bucket to be contiguous, so that the next recursion receives a consecutive input slice. To do this, we divide the array into chunks of size l, and process them in parallel. We first count the number of elements in chunk i that fall into bucket j in $A[i][j]$. Note that there is no data race since we count all points sequentially within each bucket. Then we compute the exclusive prefix sum of matrix A in column-major and get the offset matrix B—i.e., we consider storing matrix A in column major in an array, and compute the exclusive prefix sum at each element. This can be done by a parallel cache-efficient matrix transpose [18] and a standard parallel prefix-sum [13]. As such, $B[i][j]$ implies the offset when

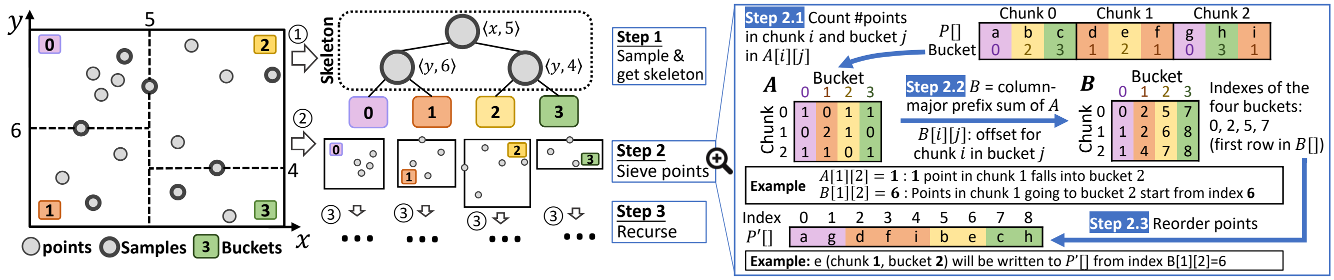
y
5
6
4
x
0
1
2
3
①
②
③
Skeleton
⟨x, 5⟩
⟨y, 6⟩
⟨y, 4⟩
Step 1
Sample & get skeleton
Step 2
Sieve points
Step 3
Recurse
points
Samples
3 Buckets
Step 2.1 Count #points in chunk i and bucket j in $A[i][j]$
Chunk 0
Chunk 1
Chunk 2
$P[]$
Bucket
a b c d e f g h i
0 2 3 1 2 1 0 3 1
A
Bucket
0 1 2 3
Chunk
0 1 0 1 1
1 0 2 1 0
2 1 1 0 1
Step 2.2 B = column-major prefix sum of A
$B[i][j]$: offset for chunk i in bucket j
B
Bucket
0 1 2 3
Chunk
0 0 2 5 7
1 1 2 6 8
2 1 4 7 8
Indexes of the four buckets: 0, 2, 5, 7 (first row in $B[]$)
Example $A[1][2] = 1$: 1 point in chunk 1 falls into bucket 2
$B[1][2] = 6$: Points in chunk 1 going to bucket 2 start from index 6
Index 0 1 2 3 4 5 6 7 8
$P'[]$ a g d f i b e c h
Step 2.3 Reorder points
Example: e (chunk 1, bucket 2) will be written to $P'[]$ from index B[1][2]=6

Figure 1: An illustration of our kd-tree construction algorithm, with a detailed overview on the *sieving step*. In this example, we first sample seven points and construct the tree skeleton using the samples, dividing the plane into four regions (buckets). Next, we sieve all points into the corresponding bucket. Concretely, we divide the points P into chunks of size $l = 3$. All chunks are processed in parallel. For each chunk, we count the number of points for every bucket in array A. We then compute the exclusive prefix sum of A in column-major order to get the offset matrix B. We then move all points from P to P' so that points within each bucket are contiguous. Finally, we recursively construct each subtree (a bucket in P') in parallel.

writing a point in chunk i that belongs to bucket j. We present an illustration for this process in Fig. 1. Then we process all buckets again in parallel, and move each point to its final destination by using the offsets provided from matrix B as the starting pointer. There is still no data race here, since all points that "share" the same offset must be in the same chunk and will be processed sequentially.

After all points in the same bucket are placed consecutively, we recursively build kd-trees for each bucket in parallel. The recursion stops when the number of points is smaller than $2^\lambda \cdot \sigma$. We then switch to the base case and use the plain parallel kd-tree construction to build the subtree. We will later show that by setting proper values for λ and σ, the base case fits into the cache and using the plain parallel construction will not incur extra memory accesses.

3.2 Theoretical Analysis

We now formally analyze our construction algorithm and show its theoretical efficiency. We start with a useful lemma about sampling. Similar results about sampling have also been shown previously [1, 20, 68]. We put it here for completeness.

LEMMA 3.1. *For a Pkd-tree T with size n', for any $\epsilon < 1$, setting $\sigma = (6c\log n)/\epsilon^2$ guarantees that the size of a child subtree is within the range of $(1/2 \pm \epsilon/4) \cdot n'$ with probability at least $1 - 2/n^c$.*

Here we need to distinguish a subtree size n' and the overall tree size n for a stronger high probability guarantee, so we can apply union bounds in the later analysis.

Proof. Alg. 1 guarantees that each leaf in a skeleton has at least σ sampled points. Therefore, every time we find a splitter, it is the median of at least 2σ sampled points. Let s ($\ge 2\sigma$) be the number of samples for this Pkd-tree T, and $\Lambda \subset T$ contains the smallest $(1/2 - \epsilon/4) \cdot n'$ points in the cutting dimension. We want to show that the chance we have more than $s/2$ samples in Λ (i.e., the left side has fewer than $(1/2-\epsilon/4)\cdot n'$ points) is small. Since all samples are picked randomly, we denote indicator variable X_i, where $X_i = 1$ if the i-th sample is in Λ and 0 otherwise. Let $X = \sum X_i$ for $i = 1..|\Lambda|$, and $\mu = \mathbb{E}[X] = (1/2 - \epsilon/4)s$. Let $\delta = \epsilon/(2-\epsilon)$. Then $(1+\delta)\mu = (1 + \frac{\epsilon}{2-\epsilon})(\frac{1}{2} - \frac{\epsilon}{4})s = \frac{s}{2}$. Using the form of Chernoff Bound $\Pr[X \ge (1+\delta)\mu] \le \exp(-\delta^2\mu/(2+\delta))$, we have:

Algorithm 1: Parallel kd-tree construction

Input: A sequence of points P.
Output: A kd-tree T on points in P.
Parameter: λ: the height of a tree skeleton.

```
Function BuildTree($P$)
    // Base case
    if $|P| < 2^\lambda \cdot \sigma$ then Use the plain parallel construction and return
    $S \leftarrow$ Uniformly sample $2^\lambda \cdot \sigma$ points on $P$ with replacement
    Build tree skeleton $\mathcal{T}$ by constructing the first $\lambda$ levels of a $k$d-tree on $S$
    // Sieve each point to their corresponding bucket (external node) in $\mathcal{T}$.
       This is performed by reordering all points in $P$ to make points in the
       same bucket consecutive.
    $R[] \leftarrow$ Sieve($P, \mathcal{T}$)  // $R[i]$: the sequence slice for all points in bucket $i$
    parallel-foreach external node $i$ do
        $t \leftarrow$ BuildTree($R[i]$)        // Recursively build a $k$d-tree on $R[i]$
        Replace the external node with $t$
    return The root of $\mathcal{T}$

// Sieve points in $P$ to the buckets (external nodes) in $\mathcal{T}$
Function Sieve($P, \mathcal{T}$)
    (Conceptually) divide $P$ evenly into chunks of size $l$
    parallel-foreach chunk $i$ do
        for point $p$ in chunk $i$ do
            $j \leftarrow$ the bucket id for $p$ by looking up $p$ in $\mathcal{T}$
            $A[i][j] \leftarrow A[i][j] + 1$
    Get the column-major prefix sum of $A[i][j]$ as matrix $B$
    parallel-foreach bucket $j$ do
        Let $s_j \leftarrow B[0][j]$ be the offset of bucket $j$
    parallel-foreach chunk $i$ do
        for point $p$ in chunk $i$ do
            $j \leftarrow$ the bucket id for $p$ by looking up $p$ in $\mathcal{T}$
            $P'[B[i][j]] \leftarrow p$
            $B[i][j] \leftarrow B[i][j] + 1$
    Copy $P'$ to $P$
    parallel-foreach bucket $j$ do
        $R[j] \leftarrow$ the slice $P[s_j..s_{j+1} - 1]$  // for the last bucket, $s_{j+1} = |P|$
    return $R[]$
```

$$\Pr\left[X \ge \frac{s}{2}\right] \le \exp\left(-\frac{\delta^2\mu}{2+\delta}\right) = \exp\left(-\frac{\epsilon^2 s}{16-4\epsilon}\right) \qquad \text{(plug in } s \ge 2\sigma)$$
$$\le \exp\left(-\frac{12c\log n}{16-4\epsilon}\right) = \frac{1}{n^{c\cdot\frac{12\log_2 e}{16-4\epsilon}}} \le \frac{1}{n^c}.$$

The right subtree has the same low probability of being unbalanced, so taking the union bound gives the state bound. □

LEMMA 3.2 (TREE HEIGHT). *The total height of a Pkd-tree with size n is $O(\log n)$ for $\sigma = \Omega(\log n)$, or $\log n + O(1)$ for $\sigma = \Omega(\log^3 n)$,* *both* whp.

Proof. To prove the first part, we will use $\epsilon = 1$ in Lem. 3.1. In this case, for $\sigma = 6c \log n$, one subtree can have at most 3/4 of the size of the parent with probability $1 - 1/n^c$, which means that the tree size shrinks by a quarter every level. This indicates that the tree heights is $O(\log n)$ *whp* for any constant $c > 0$.

We now show the second part of this lemma. For leaf wrap $\phi \geq 4$, the tree has height 1 for $n \leq 4$. We will show that using $\epsilon = 4/\log n$ (i.e., $\sigma = O(\log^3 n)$), the tree height h is $\log n + O(1)$. Similar to the above, here in the worst case, for a subtree of size n', the children's subtree size is at most $(1/2 + \epsilon/4) \cdot n' = (1/2 + 1/\log n) \cdot n'$. Hence, the tree height satisfies $(1/2 + 1/\log n)^h = 1/n$, so $h = -\log n/\log(1/2 + 1/\log n)$. Here, let $\delta = h - \log n = -\log n/\log(1/2 + 1/\log n) - \log n$. It solves to $\delta = O(1)$ for $n > 4$. Although complicated, the analysis primarily employs some algebraic methods. Due to the space limit, we put it in Appendix A. The high-level idea is to replace $t = \log n$, so $\delta = f(t) = -t/(\log(1/2 - 1/t)) - t = t/(1 + \log t - \log(t - 2)) - t$. We show that $f(t)$ is decreasing for $t \geq 2$ by proving $f'(t) < 0$ for $t \geq 2$. Since we have several logarithmic functions in the denominator, we computed the second and third derivatives and used a few algebraic techniques to remove them. □

Later, we will experimentally show that maintaining strong balancing criteria (tree height of $\log_2 n + O(1)$) is not necessary for most kd-tree's use cases. Hence, in the rest of the analysis, we will use $\sigma = \Theta(\log n)$ and assume the tree height as $O(\log n)$.

With these lemmas, we now show that Alg. 1 is theoretically efficient in work, span, and cache complexity, if we plug in the appropriate parameters. Recall that M is the small memory size. We will set 1) skeleton height $\lambda = \epsilon \log M$ for some constant $\epsilon < 1/2$; and 2) chunk size $l = 2^\lambda$, so array A and B have size $O(2^\lambda \times |P|/l) = O(|P|)$, and operations on A and B will have $O(1)$ cost per input point on average. We use $O(\mathit{Sort}(n)) = O((n/B) \log_M n)$ to refer to the best-known cache complexity of sorting n keys, which is also a lower bound for kd-tree construction—consider that the input points are in one dimension, then building a kd-tree is equivalent to sorting all points by their coordinates. We also assume $M = \Omega(\text{polylog}(n))$, which is true for realistic settings.

Theorem 3.3 (Construction cost). *With the parameters specified above, Alg. 1 constructs a Pkd-tree of size n in optimal $O(n \log n)$ work and $O(\text{Sort}(n)) = O((n/B) \log_M n)$ cache complexity, and has $O(M^\epsilon \log_M n)$ span for constant $0 < \epsilon < 1/2$, all with high probability. Here M is the small-memory size and B is the block size.*

Proof. We start with the work bound. Although the entire algorithm has several steps, each input point is operated for $O(1)$ times in each recursive level, except for lines 14 and 21. For these two lines, looking up the bucket id has $O(\lambda)$ work. Since the total recursive depth of Alg. 1 is $O(\log n)/\lambda$ *whp*, the work is $O(\lambda \cdot (\log n)/\lambda) = O(\log n)$ *whp* per input point, leading to total $O(n \log n)$ work *whp*.

We now analyze the span of Alg. 1. The algorithm starts with sampling $2^\lambda \cdot \sigma$ points and building a tree skeleton with λ levels. Taking samples and building the skeleton on them can be trivially parallelized in $O(\lambda \log n)$ span (using the plain algorithm at the beginning of Sec. 3). In the sieving step, each chunk has $l = 2^\lambda$ elements that are processed sequentially, and all chunks are processed in parallel. This gives $O(2^\lambda)$ span. The column-major prefix sum on line 16 can be computed in $O(\log n)$ span [18], and all other parts also have $O(\log n)$ span. The total span for one level of recursion is therefore $O(l + \log^2 n) = O(M^\epsilon)$, assuming $M = \Omega(\mathit{polylog}(n))$. Since Alg. 1 have $O(\log n)/\lambda$ recursive levels *whp*, the span for Alg. 1 is $O(M^\epsilon \log_M n)$ *whp*.

We finally analyze the cache complexity. Based on the parameter choosing, the samples fully fit in the cache. In each sieving step, since $l = 2^\lambda = M^\epsilon \leq \sqrt{M}$, the array $A[i][\cdot]$ and $B[i][\cdot]$ fits in cache, so the loop bodies on lines 13 and 20 will access the input points in serial, incurring $O(n/B)$ block transfers. All other parts cost $O(n/B)$ block transfers, including the column-major prefix sum on line 16 [18]. Hence, the total cache complexity for Alg. 1 is $O(n/B)$ per recursive level, multiplied by $O((\log n)/\lambda) = O(\log n/\log M) = O(\log_M n)$ levels *whp*, which is $O(n/B \cdot \log_M n)$. □

The work and cache bounds in Thm. 3.3 are the same as sorting (modulo randomization) [3] and hence optimal. The span bound can also be optimized to $O(\log^2 n)$ *whp*, using a similar approach in [18], with the details given in the proof of the theorem below.

Theorem 3.4 (Improved span). *A Pkd-tree of size n can be built in optimal $O(n \log n)$ work and $O(\text{Sort}(n)) = O((n/B) \log_M n)$ cache complexity, and has $O(\log^2 n)$ span, all with high probability.*

Proof. The $O(2^\lambda) = O(M^\epsilon)$ span per recursive level is caused by the two sequential loops on lines 13 and 20. These two loops can be parallelized by a sorting-then-merging process as in [18]. The high-level idea is to first sort (instead of just count) the entries in the loop on line 13 based on the leaf labels. Once sorted, we can easily get the count of the points in each leaf. Then on line 20, once the array is sorted, points can be distributed in parallel. We refer the readers to [18] for more details. The span bound for each level is $O(\log n)$ [17, 18] for the sieving step. For the rest of the part, the span is $O(\lambda \log n)$ caused by skeleton construction. Altogether, the span per level is $O(\lambda \log n)$, and there are $O(\log n/\lambda)$ recursion levels *whp*. Therefore, the total span is $O(\log^2 n)$ *whp*. □

In practice we still use the previous version because $O(M^\epsilon \log_M n)$ span can enable sufficient parallelism, and the additional sorting to get the improved span may lead to performance overhead.

4 Parallel Algorithms for Batch Updates

In this section, we present our parallel update algorithms for Pkd-trees. Here we consider the batch-parallel setting that inserts or deletes a batch of points P to the current Pkd-tree T. Pkd-trees do not require the tree to be perfectly balanced as in existing parallel implementations [53, 75, 81]. Our key idea is to make the tree *weight-balanced* and to *partially reconstruct* the tree upon imbalance.

Fig. 2 illustrates the high-level idea. We allow the sizes of the two subtrees to be off by at most a factor of α, i.e., the size of a subtree can range from $(0.5 - \alpha)$ to $(0.5 + \alpha)$ times the size of its parent. Such a relaxation allows most updates to be performed lazily, until at least a significant fraction of a subtree has been modified. If such a case happens, we rebuild the unbalanced subtree using Alg. 1. The rebuild cost can be amortized to the updated points in all batches. This idea of lazy updates with reconstruction has been studied sequentially on various trees for point updates [37, 64]. The key challenge here is to adapt this idea to the batch-parallel setting while maintaining theoretical and practical efficiency. Theoretically, we show efficient work and cache bounds, and high parallelism

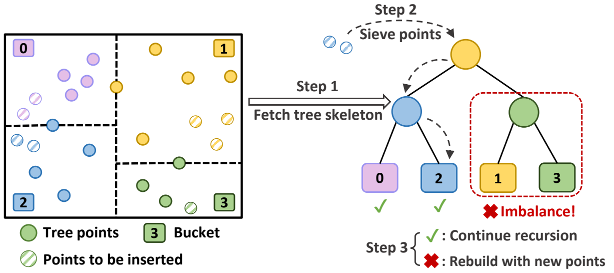

Figure 2: Illustation of our batch insertion to a kd-tree. Our algorithm first fetches the tree skeleton from the kd-tree, sieves the points into the corresponding bucket as in Alg. 1, then processes each buckets in parallel, and finally rebuilds the subtrees that become imbalance after insertion.

Algorithm 2: Batch insertion

```
Input: A sequence of points $P$ and a $k$d-tree $T$.
Output: A $k$d-tree with $P$ inserted.
Parameter: $\lambda$: the maximum height of a fetched tree skeleton.
           $\phi$: the leave wrap of $T$.

// Insert points $P$ into $k$d-tree $T$
1  Function BatchInsert($T, P$)
2      if $P = \emptyset$ then return $T$
3      if $T$ is leaf then return BuildTree($T \cup P$)        // Insert into a leaf
4      $\mathcal{T} \leftarrow$ the skeleton at $T$
5      Sieve points to the corresponding bucket in $\mathcal{T}$ using Sieve($P, \mathcal{T}$) from
         Alg. 1. Let $R[i]$ be the sequence of all points sieved to bucket $i$.
6      $t \leftarrow$ The root of skeleton $\mathcal{T}$
7      return InsertToSkeleton($t, R[0..2^\lambda)$)

// Insert the buckets $R[l]$ to $R[r-1]$ to a node $t$ in the skeleton
8  Function InsertToSkeleton($t, R[l..r)$)
9      if $t$ is an external node in the skeleton then
10         $x \leftarrow$ the subtree at $t$
11         return BatchInsert($x, R[l]$)
12     else
13         if after insertion, the two subtrees at $t$ are weight-balanced then
14             $m \leftarrow$ number of buckets in $t.lc$
15             In Parallel:
16                 $t.lc \leftarrow$ InsertToSkeleton($t.lc, R[l,m)$)
17                 $t.rc \leftarrow$ InsertToSkeleton($t.rc, R[m,r)$)
18             return $t$
19         else return BuildTree($t \cup (\bigcup_{i=l}^{r-1} R[i])$)        // Rebuild the subtree
```

for our new batch-update algorithm. In practice, we conduct an in-depth performance study with the relaxation of balancing criteria. In Sec. 6.5, we show that the query performance of Pkd-tree remains fairly stable for $\alpha \le 0.4$ (the two subtree sizes differ by up to 9×). The weight-balanced feature of Pkd-tree allows it to significantly outperform all existing counterparts.

4.1 Batch Insertion

We show our insertion algorithm in Alg. 2, which takes as input a kd-tree T and a set of points P, and inserts P to T. There are two base cases: 1) if $P = \emptyset$ no insertion is needed (line 2), and 2) if T is a leaf, the algorithm will construct a tree based on $P \cup T$ (line 3).

Otherwise, we will first grab the skeleton $\mathcal{T}$ at T at line 4. Here we will also apply the *sieving algorithm* in construction to sieve all points in P based on the skeleton $\mathcal{T}$ (line 5). Based on the partition of the buckets $R[]$, we will apply the insertions and rebalance the tree in function InsertToSkeleton. This algorithm not only processes the skeleton top down to perform the insertions of each bucket to the corresponding subtrees, but also identifies the unbalanced subtrees to reconstruct them. In particular, with the set of points in each bucket and the original subtree size, we can compute the sizes of each subtree in $\mathcal{T}$ after insertion, and thus determine whether any of these subtrees are unbalanced. If we encounter the node t in $\mathcal{T}$ that will become unbalanced after insertion (the else-condition at line 19), we will directly reconstruct the subtree using all original points in t and the points in P that belong to this subspace. A reconstruction can be performed by flattening all points in the current subtree with the points to be inserted, and applying the construction algorithm to create a (almost) perfectly balanced tree. Note that in our case, it is not perfectly balanced due to our sampling-based construction algorithm, but in Sec. 4.3 we will show our update algorithms are still theoretically efficient. If a reconstruction is triggered at subtree t, we do not need to further process the subtrees of t in this case.

4.2 Batch Deletion

Given a set of points P and a kd-tree T, the batch deletion algorithm removes P from T. Compared with batch insertion, the challenge of batch deletion is the additional step of handling points that are not in the tree, i.e., $P \setminus T \neq \emptyset$. Due to these absent points, we can no longer identify the unbalanced subtrees before we traverse into these subtrees and mark all the points to be deleted.

Therefore, our algorithm for batch deletion goes in two rounds. In the first round, all points in P are sieved into the corresponding leaves in T using the sieving algorithm from Alg. 1. By doing this, we identify all points in P that are not in T, and discard them. After this, we know the exact size of each subtree after deletion, and we then identify the unbalanced ones after deletion. This process is similar to the batch insertion algorithm, and thus the asymptotic cost also remains the same.

4.3 Theoretical Analysis

We now show that our conceptually simple batch update algorithms also have good theoretical guarantees. Since the update algorithms use the construction algorithm as a subroutine, we need to accordingly set up the parameters for both algorithms. In particular, we select $\sigma = (6c \log n)/\alpha^2$ for some constant $c > 0$ to ensure a low amortized cost in Thm. 4.1. Here we assume α is a constant and $\sigma = \Theta(\log n)$.

Theorem 4.1 (Updates). *Using $\sigma = (6c \log n)/\alpha^2$, the update (insertions or deletions) of a batch of size $m = O(n)$ on a Pkd-tree of size n has $O(\log^2 n)$ span* whp; *the amortized work and cache complexity per element in the batch is $O(\log^2 n)$ and $O(\log(n/m) + (\log n \log_M n)/B)$* whp, *respectively.*

For simplicity, Thm. 4.1 assumes the batch size $m = O(n)$. If $m = \omega(n)$, we just need to replace the term n by $m+n$ in the bounds for batch insertions (no change needed for batch deletion).

Due to the space limit, we defer the full proof in Appendix B. The overall structure of this analysis is similar to Thm. 3.3, with the additional information that traversing m leaves in a binary tree of size n touches $O(m \log(n/m))$ tree nodes [16]. Again in practice, we use the sieving approach in Alg. 1, which leads to $O(M^\epsilon \log n)$ span and supports sufficient parallelism.

The update cost bound for Pkd-tree is higher than using the logarithmic method—e.g., the work per point is $O(\log^2 n)$ instead of $O(\log n)$. However, we note that the bound for Pkd-tree is not tight.

Unless in the adversarial case, the update cost per point is more likely to be $O(\log n)$ when subtree rebuild is less frequent. In Sec. 6, we will experimentally show that the update is faster than the logarithmic method practically. Meanwhile, since Pkd-tree only keeps a single tree rather than $O(\log n)$ trees, the query performance on Pkd-tree is significantly better.

In addition, Pkd-tree can support stronger balancing criterion for by setting $\alpha = o(1)$. In this case, the amortized work and cache complexity per point will increase to $O((\log^2 n)/\alpha)$ and $O(\log(n/m) + (\log n \log_M n)/B\alpha)$ *whp*, respectively. For example, we can enable $\log n + O(1)$ tree height by using $\alpha = O(1)/\log n$. However as mentioned, our experimental results show that using tree height as $O(\log n)$ is good enough to give overall good performance for both updates and queries in practice.

5 Implementation Details

Avoid the Extra Copies. For simplicity, in Alg. 1, we assume copying the array of points in P' back to P (line 24) after distributing the points. In practice, this copy can be avoided by swapping P and P' in each recursive call. This significantly saves unnecessary memory accesses in the algorithm.

Parameter Choosing. Our theoretical analysis in Sec. 3.2 suggests $\lambda = \epsilon \log M$ for some constant $\epsilon < 1/2$. In practice, we observed that using $\lambda = 4$ to 10 generally gives good performance. We use $\lambda = 6$ for Pkd-trees in our experiments. We use $\phi = 32$ for the leaf warp size, and over sampling rate $\sigma = 32$. We set the balancing parameter $\alpha = 0.3$, and further explain our choice in Sec. 6.5.

Reduce the Memory Usage. A key effort in implementing Pkd-tree is to minimizing the memory usage. Reducing the memory footprint is crucial in at least two aspects. First, it allows the Pkd-tree to handle larger inputs. Second, a smaller memory footprint generally means better cache utilization, leading to better performance.

There are a few approaches in the design of the Pkd-tree to reduce memory usage. The first is the leaf wrapping as mentioned, which creates a flat tree leaf when the subtree size drops below a certain threshold (32 in Pkd-trees). We also contract the leaves when all points are duplicates, and we refer the audience to Appendix C for details. Second, we try to keep each interior node as small as possible to fit more tree nodes in the cache. The only additional information we keep for each tree node is the subtree size, which is needed in our weight-balance scheme and is used in range count queries. Namely, unlike ParGeo and CGAL, the Pkd-tree does not store the *bounding box* of each tree node, which is the smallest box containing all points in this subtree. This box can be used in queries to prune the subtree: when the query does not overlap with the box, the entire subtree can be skipped. In Pkd-tree, instead of storing the bounding box, the query will compute the subspace of each subtree on-the-fly: the function will pass the subspace of the current tree node to recursive calls at its children, so the subspaces for the children can be further computed with the splitter. This is not as tight as the bounding box, but in our experiments, we observed that avoiding explicitly storing bounding boxes gives better overall performance for Pkd-tree.

Queries. Since the Pkd-tree is a single kd-tree, we can use all standard kd-tree query algorithms on Pkd-trees.

In our k-NN query, we use the standard depth-first search algorithm. When searching the query point q in a non-leaf subtree T, if q is to the left of the splitter of T, it will visit the $T.lc$ first, and vice versa. After the recursion returns, we prune the visit to the other child of T by the distance between q and the splitter in T. If T is a leaf, we traverse all points stored in T and add them to the candidate container. The range query is to traverse the tree recursively, checking if the subspaces associated with nodes fall within the query box and pruning branches that do not intersect the query region. The only strong query bounds we know of for the standard kd-tree are for orthogonal range queries and range counts. A range count on D dimensions takes $O(2^{h(D-1)/D})$ work on a kd-tree of height h [2, 20], which is $O(n^{(D-1)/D})$ if the tree height is $\log n + O(1)$. The bound for a range query has an additive term k where k is the output size. We can set the parameters of Pkd-tree accordingly as in Lem. 3.2 to achieve this bound in theory, although later in Sec. 6.5 we show that the query performance does not degenerate by a slightly larger tree height. While no strong bounds are known for k-NN queries on kd-tree on general distributions, previous work has shown that kd-tree is highly practical for such queries, and it is the main use case for kd-trees in the real world.

Parallel Granularity Control. As standard parallel granularity control, for tree construction and batch update, when the input size is smaller than 1024, we will continue the process using the standard sequential algorithm.

6 Experiments

We conducted extensive experiments to demonstrate the efficiency of the Pkd-tree. For both synthetic (Sec. 6.1) and real-world (Sec. 6.2) datasets, the Pkd-tree shows better performance than other baselines in construction, batch updates, and various queries.

We also provide in-depth studies to further understand the performance gains of Pkd-trees. Sec. 6.3 measures the number of cache misses in different algorithms. Our results show that the theoretical guarantee for Pkd-trees (Thm. 3.3 and 4.1) indeed allows for better cache-efficiency and leads to good performance in practice. Sec. 6.4 studies the two techniques in our tree construction algorithm, sampling and constructing multiple levels, and show that they lead to roughly 2× and 4× performance gains, respectively. Since Pkd-trees are weight-balanced, in Sec. 6.5 we show how the balancing criterion affects the update and query performance, and explain how we choose the parameters in the Pkd-tree. Finally, we show that the Pkd-tree has good parallel scalability in Sec. 6.6.

Setup. We use a machine with 96 cores (192 hyperthreads) with four-way Intel Xeon Gold 6252 CPUs and 1.5 TB RAM. Our implementation is in C++ using ParlayLib [14] to support fork-join parallelism. The reported numbers are the average of three runs after a warm-up run. This approach ensures that all timed runs begin with a consistent cache configuration, resulting in more stable performance. We use $\lambda = 6$ as explained in Sec. 5. We set $\alpha = 0.3$, and discuss the choice of this parameter in Sec. 6.5.

Baselines. We compare the Pkd-tree with three existing implementations, described as follows.

- **BHL-tree** [83]. The plain implementation of a parallel kd-tree from ParGeo uses a single tree structure with the binary heap layout. BHL-tree uses the *plain parallel kd-tree* construction algo-

Bench.	D	Construction				Batch Insertion (1%)				Batch Deletion (1%)				10-NN (1%)				Range Report (10K)			
		2	3	5	9	2	3	5	9	2	3	5	9	2	3	5	9	2	3	5	9
Uniform 1000M	Ours	3.15	3.65	5.67	9.66	.104	.107	.123	.152	.121	.134	.171	.232	.381	.822	4.58	108	.391	.706	2.31	16.2
	Log-tree	37.9	45.4	58.0	92.7	2.16	2.66	3.67	6.19	.396	.485	1.94	2.39	2.96	4.48	20.2	879	2.62	4.14	8.94	31.6
	BHL-tree	31.7	40.5	58.4	104	31.4	40.3	57.1	103	30.9	39.3	68.7	114	.487	1.02	7.38	448	2.06	2.94	6.53	23.2
	CGAL	1147	1079	1217	1412	1660	1815	1863	2145	41.2	41.3	45.0	40.2	1.04	2.30	12.5	189	311	282	249	184
Varden 1000M	Ours	3.66	4.78	6.27	11.2	.055	.107	.157	.350	.049	.112	.127	.237	.172	.210	.336	.433	.382	.745	2.24	13.1
	Log-tree	34.2	41.8	57.8	92.6	2.01	2.60	3.72	6.07	1.06	1.14	1.92	2.30	2.05	2.29	23.3	2225	2.63	4.25	7.95	14.1
	BHL-tree	30.2	39.2	58.7	104	29.4	39.1	57.3	102	29.0	38.4	67.0	123	.242	.324	.456	.535	1.95	3.03	5.72	9.96
	CGAL	429	390	372	438	849	700	582	599	13.0	9.53	23.1	3.90	.511	.217	.318	.392	296	283	253	278

Table 3: Running time (in seconds) for Pkd-tree and other baselines. Lower is better. D: dimensions. Baselines are introduced in Sec. 6. The fastest runtime for each benchmark is underlined. Batch insertion is on 10M points from same distribution of the points in the tree, and batch deletion removes 10M points from the tree. "10-NN": 10-nearest neighbor queries on 1% (10M) of the points in the tree. "Range Report": 10K orthogonal rectangle report queries with output size between 10^4–10^6. For queries, we run all of them in parallel and each query itself is run sequentially.

rithm described in Sec. 3, which is work-efficient but not cache-optimized. Its batch update simply rebuilds the whole tree.

- **Log-tree** [83]. The kd-tree implementation based on the logarithmic method from ParGeo. The Log-tree keeps $O(\log n)$ static cache-oblivious kd-trees (using the vEB layout) with exponentially increasing sizes. A batch update is performed by merging and rebuilding a subset of the trees. Queries have to be performed on all $O(\log n)$ trees and the results need to be combined.
- **CGAL** [81]. The kd-tree in the computational geometry library CGAL. CGAL supports parallelism using the threading building blocks (TBB) [52]. During construction, CGAL partitions the points sequentially, then builds two subtrees in parallel. A batch insertion rebuilds the whole tree with the inserted points; a batch deletion removes the points one by one.

Datasets. We test both synthetic and real-world datasets. We introduce the real-world datasets in Sec. 6.2. For synthetic datasets, we use 64-bit integer coordinates with two distributions: Varden and Uniform. Varden is a skewed distribution from [38]. It generates points by a random walk with a low probability of restarting from a random place. Therefore, it contains some very dense subareas that can be far from each other, which can be used as pressure tests for the quality of kd-trees as well as the performance for frequent rebalancing. Uniform draws points within a box uniformly at random. For simplicity, we shorthand each dataset by the dimension, distribution and size. For instance, "3D-V-1000M" stands for 1000 million points in 3 dimensions from Varden.

6.1 Operations on Synthetic Datasets

Overall Performance. We summarize the performance for the Pkd-tree and other baselines in Tab. 3 with tree size of 10^9, in dimensions $D \in \{2, 3, 5, 9\}$. We also include an experiment on a synthetic dataset with 12 dimensions for all baselines, as detailed in Appendix I. Each batch for insertion or deletion contains 10^7 (1% of the tree size) points from the same distribution. We test two query types: 1) 10^7 of 10-NN queries, 2) 10^4 range report queries (output sizes 10^4–10^6). Queries are performed directly after construction. Different queries run in parallel, and each query runs in serial. No other baselines support range count queries, so we test this query on Pkd-tree separately in Fig. 5.

For construction, the Pkd-tree is the fastest in all tests, which is 8.26–12.5× faster than Log-trees, 8.20–11.1× faster than BHL-trees, and 39.1–363× faster than CGAL. The high performance is mainly from good *cache-efficiency* and *scalability*, which will be studied in more details in Sec. 6.3 and 6.6, respectively. CGAL has a known scalability issue [15] (also see the scalability curve in Fig. 10), making it much slower than other implementations in construction.

The Pkd-tree also outperforms all baselines in batch updates. BHL-trees and CGAL always rebuild the entire tree after updates, so Pkd-trees are orders of magnitude faster than them, especially on small batches. Compared to the fastest baseline Log-tree, Pkd-trees are 17.4–40.7× faster in insertions and 3.27–21.7× faster in deletions, mainly due to two reasons: 1) Pkd-trees sieve the updated points to the subtrees in a cache-efficient way, and 2) both Pkd-trees and Log-trees may reconstruct some (sub)trees in batch updates, and Pkd-trees are faster in tree construction as discussed above.

For both k-NN and range queries, the Pkd-tree is the fastest except for three cases, all in high-dimensional queries. It is within 1.1× slower than CGAL in two cases, and 1.32× slower than the BHL-tree in one case. As mentioned in Sec. 5, Pkd-trees do not store the bounding boxes to optimize memory usage, but compute the subspaces for each subtree on-the-fly in queries. This approach trades off (slower) query performance for (faster) construction and updates, which can be more pronounced in higher dimensions. Even so, Pkd-tree is still the fastest in queries for 13 out of 16 instances. Therefore, we choose not to maintain the bounding boxes in tree nodes to achieve better performance for both updates and queries.

Another interesting finding is that almost all kd-trees perform better on Varden than Uniform for k-NN queries. Since Varden datasets contain dense subareas, the neighbors are usually in these regions, resulting in more effective pruning than the uniform datasets.

In the following, we present an in-depth performance study for updates and queries. We use synthetic datasets with size 10^9 points in 3D as the benchmark for the rest of this section.

Batch Updates. To further understand the performance of batch updates, we vary the batch sizes from 10^5 to 10^9, and show the results in Fig. 3. We omit smaller batch sizes because they can be completed quickly anyway and do not have high demand for parallelism. We first construct a tree with 10^9 points. Then a batch insertion inserts a batch from the same distribution into the tree; the batch deletion removes a batch of points from the tree.

Pkd-trees have the best performance for all instances on all distributions. Both the BHL-tree and CGAL fully rebuild the tree on insertions, showing a flat curve of running time with varying

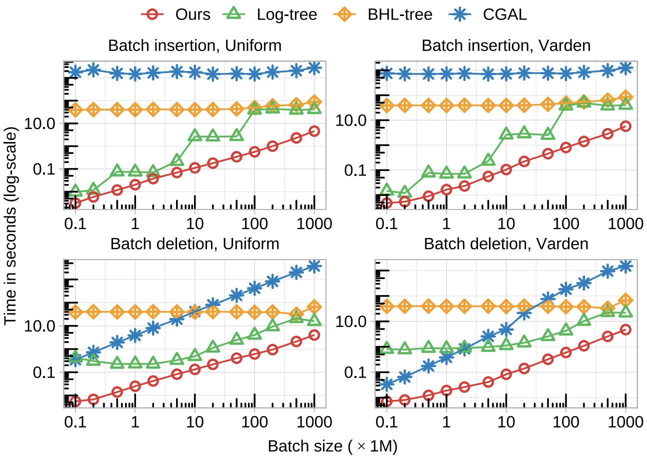

Figure 3: Time required for batch update on points from `Varden` and `Uniform` on a tree with 1000M points in 3 dimensions. Lower is better. The batch size is the number of points (× 1M) in the batch. The time is measured in seconds. Both axes are in log-scale.

batch sizes, which are much slower than the Pkd-tree based on rebalancing. A batch deletion in CGAL removes the points one by one, which performs well for small batches, but is very inefficient for large batches. The Log-tree's performance sits in the middle. It avoids fully rebuilding the tree by merging a subset of the trees for batch insertions. One may notice that there are several jumps for the Log-tree in batch insertions. This is because when the batch size reaches certain threshold, a reconstruction for a large tree may be triggered, causing significant more time. Pkd-trees have the most stable and efficient performance across tests.

***k*-NN Queries.** We now study k-NN queries on `Uniform` and `Varden` with $k \in \{1, 10, 100\}$. We consider 10^9 input points in 3D and call k-NN queries on the first 10^7 input points in parallel. Results are presented in Fig. 4. We also measure the out-of-distribution k-NN queries in Appendix F. The Pkd-tree is always among the fastest. The performance of the BHL-tree and CGAL is similar since they also keep a single tree. The Pkd-tree is slightly faster due to not storing bounding boxes in the tree nodes (see Sec. 5)—it saves the memory footprint at the cost of less efficient pruning. Overall it gives some small advantages on k-NN query performance in low dimensions. Log-trees have significantly worse performance for k-NN queries—5.85–11.7× slower on `Uniform` and 12.0–21.9× slower on `Varden` than Pkd-trees. This is because a query needs to search in all the $O(\log n)$ trees and merge the results.

Range Queries. In Tab. 3, we test range reports with relatively large output sizes 10^4–10^6 to make the queries more adversarial. Since the performance of range report is proportional to the output size, in this section, we conduct additional tests on range-count and range-report queries with a variety of output sizes. Note that although small query sizes are more frequently encountered in practice, range queries with large output sizes are also prevalent in various applications, including dynamic programming [41], etc. The Pkd-tree is the only one that supports range count. We run all queries sequentially to measure the query time w.r.t. the output size in Fig. 5. The Pkd-tree (red circles in Fig. 5) generally have the best performance across a wide range of output sizes from 1 to 10^6. The BHL-tree is competitive. Log-tree is slower than Pkd-tree and BHL-tree since it has to query $O(\log n)$ trees. Note that when a subtree is fully contained in the query box, one can output all points in the subtree in parallel. This has been incorporated into all baselines except for CGAL, which makes CGAL particularly slow with large output sizes. When the query only requires the count of points in the range, the range count function on Pkd-trees (blue rectangles in Fig. 5) can be much faster than reporting all the points, especially with large output sizes. This indicates the necessity of including range-count in the interface of a kd-tree.

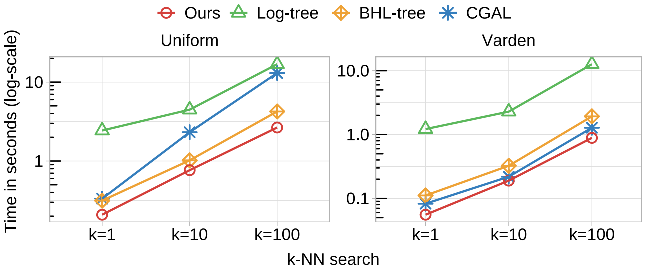

Figure 4: Running time (in seconds) of k-NN queries for $k \in \{1, 10, 100\}$. Lower is better. The dataset contains 1000M points in 3 dimensions. The test contains k-NN queries from 10^7 points in the input. Plots are in log-log scale.

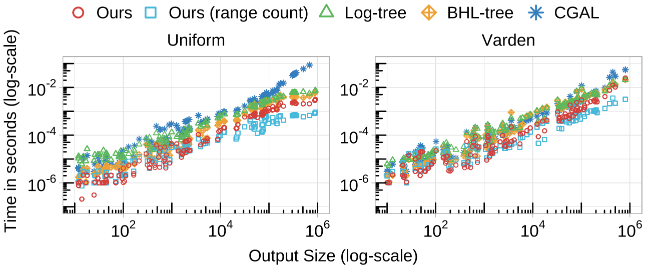

Figure 5: Running time (in seconds) of range queries w.r.t. output sizes. Lower is better. The dataset contains 1000M points in 3 dimensions. Plots are in log-log scale.

6.2 Real-World Datasets

We test our Pkd-tree and other baselines on real-world datasets, including very large ones `COSMOS` (`CM`) [73] and the Northern American region of `OpenStreetMap` (`OSM`) [46] with up to 1.298 billion points, high-dimensional datasets `HT` [48], `CHEM` [34, 84], and `HouseHold` (`HH`) [47] with up to 16 dimensions, and `GeoLife` (`GL`) [88], with highly duplicated points. All coordinates are 64-bit real numbers. Experiments on these real-world datasets show similar trends as in the synthetic datasets in Sec. 6.1.

Tree Construction, *k*-NN and Range Report. We present the running time for tree construction, 1-NN and 10-NN queries, and range report queries for all input points in Tab. 4. CGAL fails to build `HH` and `GL` due to the inability to handle heavy duplicates. BHL-trees and Log-trees cannot process 10-NN queries on `CM` due to high memory usage.

The Pkd-tree shows the best performance in all but three instances of queries in high dimensions, which is consistent with the results in Tab. 3. CGAL is faster in 10-NN on `HT` ($D = 10$) and 1-NN on `CHEM` ($D = 16$). The BHL-tree is also slightly faster than Pkd-tree in 10-NN on `CHEM`. The BHL-tree is the best baseline on

	Points	Dims	Op.	Ours	Log-tree	BHL-tree	CGAL
HT	928K	10	Build	$\underline{.008}$	.678	.061	.472
			1-NN	$\underline{.008}$	2.53	.015	.015
			10-NN	.043	2.81	.059	$\underline{.020}$
			Range	$\underline{.095}$	.651	.478	1.38
HH	2.04M	7	Build	$\underline{.054}$	.716	.102	t.o.
			1-NN	$\underline{.058}$	1.26	1.60	-
			10-NN	$\underline{.229}$	2.60	3.19	-
			Range	$\underline{.080}$	.819	.564	-
CHEM	4.21M	16	Build	$\underline{.059}$	7.07	.786	2.52
			1-NN	.042	16.1	.123	$\underline{.035}$
			10-NN	3.53	17.3	$\underline{3.32}$	3.95
			Range	$\underline{.412}$	4.28	2.64	3.14
GL	24M	3	Build	$\underline{.256}$	1.34	.792	s.f.
			1-NN	$\underline{.274}$	3.74	1.31	-
			10-NN	$\underline{.775}$	14.4	9.37	-
			Range	$\underline{.192}$	1.40	1.30	-
CM	321M	3	Build	$\underline{1.54}$	16.7	13.3	184
			1-NN	$\underline{2.79}$	25.9	5.24	5.94
			10-NN	$\underline{9.09}$	s.f.	s.f.	33.0
			Range	$\underline{.136}$	1.88	1.63	26.0
OSM	1298M	2	Build	$\underline{5.08}$	51.3	56.6	497
			1-NN	$\underline{8.73}$	134	13.0	10.5
			10-NN	$\underline{16.5}$	214	30.6	22.6
			Range	$\underline{.107}$	4.87	3.80	62.9

Table 4: Tree construction and k-NN time on read-world datasets for Pkd-tree and baselines. Lower is better. The "Points" is the number of points in the datasets and "Dim." is the dimension for the points. k-NN queries are performed in parallel on all points in the dataset. "Range" is the time for 10^3 range report queries with output size between 10^4–10^6. The fastest runtime for each benchmark is underlined. "s.f.": segmentation fault. "t.o.": time out (more than 3 hours).

these datasets, but it is still 1.89–13.3× slower in construction and 1.37–27.6× slower in query than the Pkd-tree, except for 10-NN in CHEM, where it is 1.06× faster than the Pkd-tree.

Dynamic Updates. The points from OpenStreetMap (OSM) [46] are associated with time stamps, so we acquire the batch of updates per year (2014 to 2023) to compare the performance of batch updates. We simulate a sliding-window setting. We start with building a tree of the data in 2014. In each year, we insert the corresponding batch, and delete the batch 5 years ago if applicable. After the update of each year, we perform a 10-NN query on another 10^7 points. Fig. 6 presents the performance for all tested algorithms.

Pkd-trees significantly outperform all baselines in all batch updates. For insertions, Pkd-trees is 1.82–9.55× faster than Log-trees, 3.81–12.0× than BHL-trees and 58.3–214× than CGAL. For deletions, Pkd-tree outperforms the fastest baseline Log-tree by 2.44–4.55×. For k-NN queries, Pkd-tree, BHL-tree, and CGAL have similar performance. The Pkd-tree is slightly faster due to various optimizations (e.g., avoiding storing bounding boxes). Log-tree can be much slower than other implementations due to the use of $O(\log n)$ trees. These conclusions are consistent with the results on synthetic datasets shown in Sec. 6.1.

6.3 In-Depth Performance Study

Cache Efficiency and Memory Usage for Tree Construction. One major effort in our work is to achieve low cache complexity for construction of Pkd-trees. In this section, we quantitatively verify this and show that the cache-efficiency indeed contributes to the high performance of Pkd-trees. Fig. 8 shows the numbers of cache misses in tree construction for all implementations, along with the memory usage, which is also crucial for practical performance.

The results verified that our performance gain is consistent with the reduction of the cache misses, demonstrating the importance of cache-efficiency in parallel algorithms. Pkd-trees incur 6–12× less cache misses than BHL-trees or Log-trees, which is close to the speedup of Pkd-tree over BHL-tree or Log-tree in construction. The reported cache misses in 1000M-3D-V construction indicates over 80% of the memory bandwidth usage of the testing machine. We also verify this using the Intel® VTune profiler [51]. On our machine with a peak memory bandwidth of 443.78 GB/s, Pkd-tree has a 327–421 GB/s usage of bandwidth in many stages during construction and update. This indicates that our construction and update algorithms are memory bottlenecked, and optimizing the cache complexity almost linearly contributes the performance.

Study of the Impact of Bounding Boxes. As mentioned, Pkd-trees avoid storing bounding boxes within tree nodes, but compute them on-the-fly, which may be looser than the actual bounding boxes of the subtree. Therefore, Pkd-tree reduces memory accesses and memory usage in construction and batch updates, but may introduce additional computation and tree traversal during queries. To study this trade-off, we measure the Pkd-tree and its variant storing bounding boxes (referred to as Pkd-bb) in terms of range report queries on real-world datasets. Tab. 5 shows the time, instruction per cycle (IPC), cache references (CRs) and cache-misses (CMs), as well as the average number of nodes visited per query. We refer the readers to Appendix H for the full results. Pkd-bb allows for more effective prune, evidenced by the fewer number of nodes visited during search. This benefit is more significant on high-dimensional datasets HT, HH and CHEM. Due to visiting fewer nodes, on these three datasets, Pkd-bb also has lower cache references and cache misses, and is thus faster. On low-dimensional datasets CM and OSM, the difference between the computed subspaces and bounding boxes is small. In this case, Pkd-tree and Pkd-bb visited similar numbers of tree nodes. Thus, Pkd-tree achieves lower time than Pkd-bb in queries due to fewer memory accesses.

6.4 Technique Analysis for Tree Construction

In Alg. 1, we mainly employed two techniques to improve the cache-efficiency: 1) building λ levels at a time to save total data movements, and 2) using sampling to determine splitters to save memory accesses. To test these two techniques, we measure the time and the cache misses in tree construction using three versions of Pkd-trees with different levels of optimizations: 1) the final version with sampling and $\lambda = 6$ (red bars), 2) constructing one level at a time using sampling, i.e., $\lambda = 1$ (blue bars) and 3) constructing one level at a time without sampling, i.e., $\lambda = 1$ and finding exact median in parallel (yellow bars). All benchmarks have 10^9 points in 3 dimensions. Results are presented in Fig. 7. By comparing the red and blue bars, we observe that building multiple levels reduces running time by 2.91–4.31× and cache misses by 3.8× for both distributions. The difference between the blue and yellow bars indicates that sampling improves the time by about 1.86× and reduces cache misses by about 1.9×. The improvement in running time is consistent with the reduction of cache misses. This verifies that the high performance of our construction algorithm indeed comes from the better cache efficiency enabled by the two techniques.

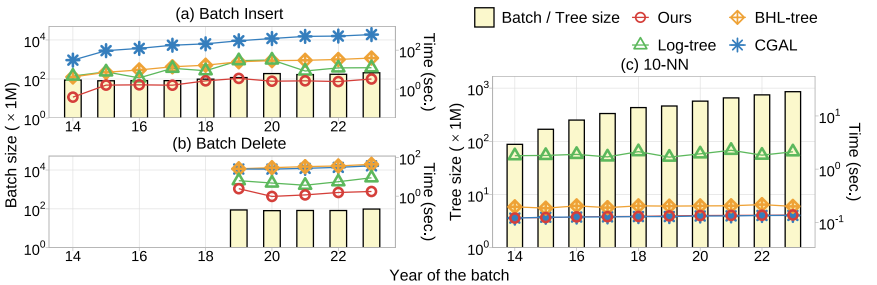

Figure 6: Batch update using a sliding window spanning five years and 10-NN queries on OSM [46]. Lower is better. The input is batched by years from 2014 to 2023. In each year, we insert the corresponding batch, and delete the batch from five years ago if applicable. After the update, we perform 10^7 10-NN queries in parallel. In (a) and (b), bars (left axis) are the batch size, and lines (right axis) depict the time for batch insertion and deletion in every year respectively. In (c), bars (left axis) are tree size, and lines (right axis) show the 10-NN query time. Vertical axes are in log scale.

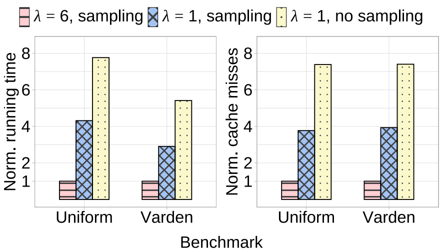

Figure 7: The evaluation of the performance gain for techniques in tree construction. Lower is better. The datasets have 1000M size in 3 dimensions. The y-axis normalized to the final version with $\lambda = 6$ and using sampling.

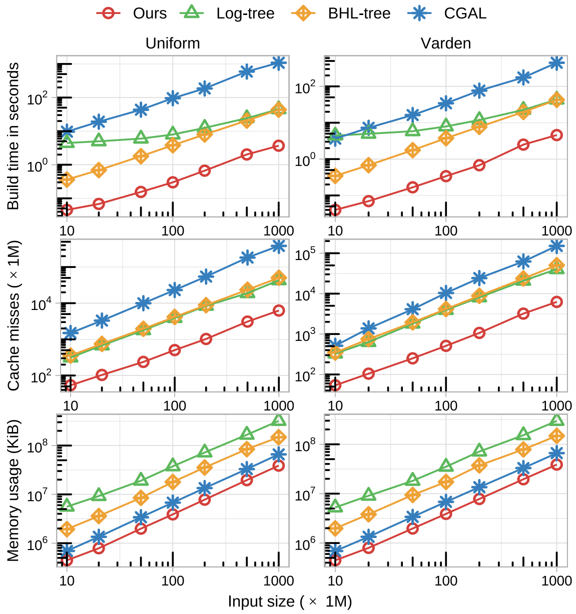

Figure 8: Time, cache misses, and memory usage needed during the tree construction. Lower is better. Plots are in log-log scale. All points are in 3 dimensions.

	Tree	Time(sec.)	# Leaf	# Interior	IPC	CRs(M)	CMs(M)
HT	Pkd	.587	2,675	3,957	.326	926	508
	Pkd-bb	$\underline{.268}$	$\underline{207}$	$\underline{891}$	$\underline{.506}$	$\underline{245}$	$\underline{134}$
HH	Pkd	.385	2,817	3,621	.320	557	361
	Pkd-bb	$\underline{.192}$	$\underline{615}$	$\underline{1,135}$	$\underline{.387}$	$\underline{242}$	$\underline{150}$
CHEM	Pkd	1.15	4,276	5,701	$\underline{.271}$	1,662	1,450
	Pkd-bb	$\underline{.837}$	$\underline{1,330}$	$\underline{2,506}$	.235	$\underline{954}$	$\underline{820}$
GL	Pkd	.329	3,268	5,478	$\underline{.303}$	439	345
	Pkd-bb	$\underline{.291}$	$\underline{1,285}$	$\underline{3,484}$	.215	$\underline{407}$	$\underline{317}$
CM	Pkd	$\underline{.531}$	2,456	3,939	$\underline{.186}$	$\underline{692}$	$\underline{649}$
	Pkd-bb	.577	$\underline{2,195}$	$\underline{3,785}$	.165	752	703
OSM	Pkd	$\underline{.326}$	529	1,243	$\underline{.171}$	$\underline{460}$	$\underline{426}$
	Pkd-bb	.363	$\underline{236}$	$\underline{959}$	.148	473	441

Table 5: Performance comparison of the original Pkd-tree (Pkd) and a variant with bounding boxes (Pkd-bb) for range report queries on real-world datasets. The best performance is underlined. The query contains 10^4 range report queries with output size 10^4–10^6. "Time": Time for all queries in seconds, "Leaf": Average number of leaf nodes visited per query, "Interior": Average number of interior nodes visited per query, "IPC": Instructions per cycle, "CR": Cache reference, "CMs": Cache misses.

6.5 Balancing Parameter Revisited

One of the core ideas of the Pkd-tree is to relax the strong balancing criterion to achieve much better performance in construction and updates. This may increase the tree height and affect the query performance to some extent, and the trade-off is controlled by the parameter α. In this section, we systematically study the choice of α and the corresponding impact on the performance.

To do this, we create adversarial inputs such that belated rebalancing may result in a large tree height. We generate skewed batch sequences and insert them into an empty tree by 1000 batches. After each update, we perform queries to see how the imbalance affects the query performance. We test α in a full range from 0.01 (almost always rebalance on updates) to 0.50 (no rebalancing; siblings can be arbitrarily different). We generated multiple distributions and show two of the most representative ones in the paper:

- TYPE I: one instance of the 3D-V-1000M dataset.
- TYPE II: concatenation of one instance from 3D-U-100M and another one from 3D-V-900M.

We observe that TYPE I is adversarial since the Varden is generated by a random-walk plus random jump process. By cutting the stream into 1000 batches, different dense areas (clusters) are added to the tree in-order, which will trigger frequent rebalancing for the Pkd-trees (otherwise the tree quality can degenerate significantly). TYPE II, as well as most of the other distributions are more resistant to large α values—the initial tree are generated on a Uniform distribution, providing a roughly even partition of the space—belated rebalancing does not affect the tree quality as much as TYPE I.

Fig. 9 demonstrates the construction and 1-NN time w.r.t. the balancing parameter α. The "rebuild size" (yellow bars) denotes the cumulative size of the subtrees that are reconstructed throughout all batch insertions. This value is normalized to the final tree size, 10^9. The "incremental update time" is to construct a tree by inserting 1000 batches incrementally. We normalize the construction time to that if we directly build a tree once using all the points. After each batch insertion, we perform 1-NN queries for a batch of another 1M points generated uniformly at random. We normalize the query

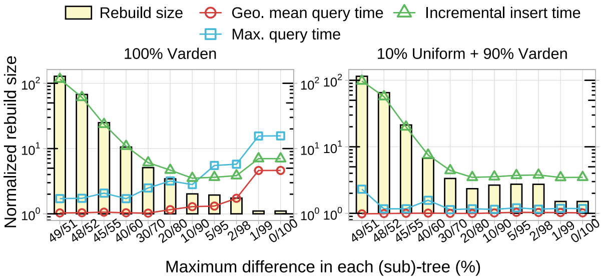

Figure 9: Normalized rebuild size, update and query times with varying balancing parameter α. Lower is better. The dataset contains 10^9 points in 3D, divided into 1000 batches, and incrementally inserted into an initially empty tree. The "rebuild size" (yellow bars, left axis) denotes the total number of tree nodes involved in reconstructions, normalized to the final tree size. The total insertion time (green line, right axis) is normalized to that of building a tree directly from the input. After each of the 1000 batch insertions, we perform 1-NN queries for 1M points from the `Uniform` distribution. The blue and red lines (right axis) show the geometric mean and maximum time among the 1000 queries respectively, normalized to the query time on a perfectly-balanced tree on the same set of points.

time to that on a perfectly balanced tree with the same set of points to illustrate the impact of imbalance. Among all the 1000 batch queries, we record the maximum normalized query time by the blue rectangles in Fig. 9, which roughly represents the "worst-case" query time, and the geometric mean, which represents the "average case". We use $x/y = (0.5 - \alpha)/(0.5 + \alpha)$ in the figure to indicate the degree of balance controlled by α, which means that two sibling subtrees can differ by at most $x : y$. We pick 1-NN query here since the 1-NN query performance is the most sensitive to how balance the kd-tree is, among all queries types tested in this paper.

For both input sequences, the overall trend for the incremental construction time decreases when less rebalance is required, since rebalancing is triggered less frequently. There is a slight rebound when the subtrees are excessively unbalanced (1/99 or worse)—the cost of traversing the tree in batch insertion also increases when the tree becomes skewed. For queries, an unbalanced tree can significantly slow down the performance, as the searches need to go much deeper in the tree to touch the incident points.

Overall, the query performance is stable for a reasonably large range of α. The worst-case performance is negligibly affected all the way up to 30/70 ($\alpha = 0.2$), and the average-case overhead is small until 10/90 ($\alpha = 0.4$). However, when we further relax the balancing criterion, then the performance can degenerate greatly on `TYPE I`—up to 4.48× slowdown on average for $\alpha = 0.5$. The performance may still be reasonable on instances such as `TYPE II`. To ensure better query performance in general, we choose $\alpha = 0.3$ (i.e., 20/80) as the default setting in Pkd-tree since it achieves a good tradeoff for the construction, update and query performance.

We also report the running time of batch updates with three specific values of $\alpha \in \{0.03, 0.1, 0.3\}$ in Appendix G.

6.6 Parallel Scalability

We test the scalability for the tree construction and batch updates of Pkd-tree and other baselines on both 3D-U-1000M and 3D-V-1000M. We normalize all running time to the Pkd-tree on one core, and show the scalability in Fig. 10. The Pkd-tree overall has very good scalability. For `Uniform`, the Pkd-tree achieves 37.3× self-relative speedup in construction, 59.3× in batch insertion and 52.4× in batch deletion using 96 cores. The numbers for `Varden` is 35.9× in tree construction, 25.3× in batch insertion, and 37.8× in batch deletion. The speedup on `Varden` is lower in batch updates since the new points are unevenly distributed in the subtrees, resulting in more challenges in load balancing in constructing subtrees with various sizes. Both Log-trees and BHL-trees have also shown good parallel speedup, while the performance difference is mainly due to the slow sequential (1 core) performance compared to the Pkd-tree. The main reason that causes the advantage is the carefully-designed sieving algorithm introduced in Sec. 3, which is used in both construction and updates. The only implementation that does not scale well is CGAL, which was also observed in previous work [15]. CGAL only parallelizes the process of building two subtrees, but finds the splitters and partitions points into subtrees sequentially, limiting its parallelism.

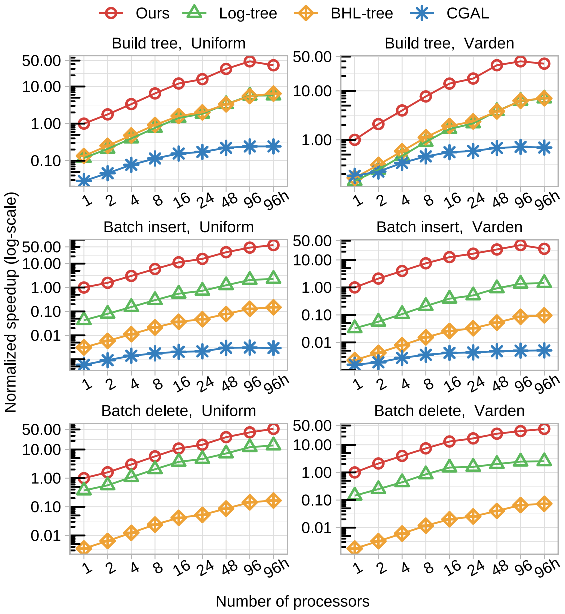

Figure 10: Normalized parallel speedup of operations on `Uniform` and `Varden` for the Pkd-tree and other baselines on varying numbers of processors. Higher is better. The curves show relative running time on different number of threads normalized to the Pkd-tree on one thread. The benchmark contains 1000M points in 3D. The "batch insert" inserts another 10M points from the same distribution into the tree, and the "batch delete" removes 10M points from the tree. "96h": 96 cores with hyper-threads. There is no data for CGAL in batch delete since it deletes points sequentially.

7 Related Work

In this section, we review the literature of the kd-tree, with a summary of the most relevant ones in Tab. 6. The original algorithm proposed by Bentley et al. [11, 35] does not include a rebalancing scheme, and assumes either static data, or inserting keys in a random order. Since then, researchers have been developing rebalancing schemes for kd-trees, mainly in the two categories: *logarithmic method*, and *partial rebuild.* The logarithmic method was first proposed by Bentley in [12], and has been followed up by later work, including optimizing the cache bounds [2, 67] and parallelism [83]. However, as shown in Tab. 3, maintaining $O(\log n)$ trees in the loga-

			Layout & Update	**Balancing Criteria**	**Cache (I/O) Optimizations**	**Notes & More Optimizations**
1975	**Bentley [11]**	Seq.	Single tree; No rebalancing	No balance	-	• Proposed *k*d-tree
1978	**Bentley [12]**	Seq.	Log-method; Tree merging	Perfectly balanced	-	• Proposed logarithmic method
1980	**kdb-tree [70]**	Seq.	B-tree; Overflow/underflow	Not shown	• B-tree layout	-
1983	**Overmars [64]**	Seq.	Single tree; Partial rebuild	Relaxed (weight balanced)	-	-
2003	**Bkd-tree [67]**	Seq.	Log-method; Tree merging	Perfectly balanced	• Cache opt. construction • Cache opt. point update	-
2003	**Agarwal et al. [2]**	Seq.	Log-method; Tree merging	Perfectly balanced	• Cache opt. construction • Cache opt. point update • vEB layout	-
2016	**Agarwal et al. [1]**	Dist.	Single tree; Static (no update)	Relaxed (randomized)	-	• Sampling • Multi-level construction
2020	**CGAL [81]**	Par.	Single tree; Full rebuild (sequential deletion)	Perfectly balanced	• Leaf wrap	• "CGAL" tested in Sec. 6 • Implementation available
2021	**ikd-tree [25]**	Seq.	Single tree; Partial rebuild (lazy deletion)	Relaxed (weight balanced)	-	-
2021	**ParGeo [83]**	Par.	Log-method; Tree merging (lazy deletion)	Perfectly balanced	• Leaf wrap • vEB layout	• "Log-tree" tested in Sec. 6 • Implementation available
2021	**ParGeo [83]**	Par.	Single tree; Full rebuild	Perfectly balanced	• Leaf wrap	• "BHL-tree" tested in Sec. 6 • Implementation available
	This paper (Pkd-tree)	Par.	Single tree; Partial rebuild	Relaxed (weight balanced, randomized)	• Leaf wrap • Cache opt. construction • Cache opt. batch update	• Sampling • Multi-level construction • Implementation available

Table 6: Summary of related work. "Log-method": logarithmic method (using $O(\log n)$ *k*d-trees). "Seq.": sequential; "Par.": parallel; "Dist.": distributed. "Cache opt.": optimized for cache (or I/O) efficiency. Some of the designs are optimized for disk I/Os, but algorithmically the optimizations for disks or caches are almost identical. Therefore, we also denote both of them as "cache-opt." in this table.

rithmic method hampers the query efficiency significantly. Another issue for this method is that insertions and deletions are asymmetric and need to be handled by different approaches, which is more complicated. An alternative idea is to maintain a single tree and partially rebuild the unbalanced subtrees, which was proposed by Overmars [64]. Many papers followed up this idea, such as the KDB-tree [70], scapegoat k-d tree [37], ikd-tree [25], and the divided k-d tree [82]. Among them, only KDB-tree is cache-optimized. None of them considered parallelism.

There have been many attempts to optimize the cache (or I/O) efficiency for *k*d-trees. Early work simply considers flattening the binary structure into a B-tree-like multiple-way tree [67, 70]. These papers are optimized for disk I/Os, and do not consider parallelism or batch updates. Procopiuc et al. [67] gave a (sequential) cache-efficient *k*d-tree construction algorithm, and dynamized the *k*d-tree using the logarithmic method. Agarwal et al. [2] showed how to construct a cache-oblivious *k*d-tree using the vEB layout [10]. Motivated by Procopiuc et al. [67], Wang et al. [83] proposed parallel batch update algorithms on *k*d-trees, and implemented them as the Log-tree in the ParGeo library (called BDL-tree in their paper). However, Wang et al. [83] did not show the cache complexity for their parallel construction or update algorithms.

There exist parallel *k*d-tree algorithms, but most of them are not cache-friendly or do not support a full interface. Parallel construction for static *k*d-trees has been well studied, mainly in two approaches. The first approach [24, 26, 85] is to presort all points in all D dimensions in parallel. To compute the partition hyperplane, the median of the corresponding dimension is selected, and all elements are stably partitioned into two subtrees and recursively constructed. The second approach [4, 29, 49, 69, 75] finds the median as the splitter on the fly, and then constructs the sub-trees recursively in parallel. Reif and Neumann's work [69] also proposed to support range-join using a *k*d-tree algorithm with the second approach. Some of them also use sampling [4, 49] or constructing multiple levels [1, 39]. Agarwal et al. [1] showed a distributed algorithm for static *k*d-trees, which also uses sampling and multi-level construction to optimize the number of rounds in the MPC model. However, they did not show cache or span bounds, and have no implementations. These approaches do not directly support updates. Although the *k*d-tree in CGAL [81] supports updates, it simply rebuilds the tree after updates, which is inefficient. The ParGeo library [83] provides several *k*d-tree implementations, among which Log-trees and BHL-trees are generally the fastest. The BHL-tree is based on a single *k*d-tree, which fully rebuilds the tree on updates. The Log-tree uses the logarithmic method as discussed above to support parallel batch updates. We compared to both of them in Sec. 6. There also exist concurrent *k*d-trees [27, 50] that achieve linearizability and lock-freedom. Our work focuses on batch-parallel setting, which aims to support a batch of insertions or deletions with good work, span, and cache bounds.

There have been other data structures for multi-dimensional data such as R-trees (e.g., [8, 45, 54, 66, 86]) and quad/octrees (e.g., [15]). Most of them do not support parallel updates. There exist papers on parallel R-trees, such as on GPUs [66, 86] and on disks [8, 54]. However, we are unaware of open-source in-memory implementations that support parallel construction and updates. This is probably not surprising, given that the main use cases for R-tree are for external memory while the thread-level in-memory parallelism is a less relevant optimization. For completeness, we compare the tree construction, k-NN and range report time for Pkd-tree with the sequential R-tree in Boost [71] in Appendix D. A recent paper [15] developed batch-parallel quad/octrees called Zd-trees. However, through the correspondence with the authors, we confirmed that the released version has correctness issues in batch updates, so we cannot compare to their batch-update performance. We give a comparison to their construction time and k-NN time in Appendix E.

8 Conclusion

We present Pkd-tree, a parallel kd-tree that has strong theoretical guarantees in work, span, and cache complexity for tree construction and batch update, as well as high performance in practice. Our main techniques include sampling, multi-level construction, the sieving algorithm, and the weight-balance scheme to holistically optimize the work, span, cache-efficiency in both constructions and updates. In this way, our approach relaxes the balancing criteria by a controllable manner, which allows for overall good performance considering construction, update, and various queries. In our experiments, the Pkd-tree significantly outperforms all the existing parallel kd-tree implementations on construction and updates, with competitive or better query performance.

9 Acknowledgements

This work is supported by NSF grants CCF-2103483, TI-2346223 and IIS-2227669, NSF CAREER Awards CCF-2238358 and CCF-2339310, the UCR Regents Faculty Development Award, and the Google Research Scholar Program.

References

[1] Pankaj Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. 2016. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Principles of Database Systems (PODS)*. 429–440.

[2] Pankaj K Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. 2003. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of the nineteenth annual symposium on Computational geometry*. 237–245.

[3] Alok Aggarwal and S Vitter, Jeffrey. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[4] I Al-Furajh, Srinivas Aluru, Sanjay Goil, and Sanjay Ranka. 2000. Parallel construction of multidimensional binary search trees. *IEEE Transactions on Parallel and Distributed Systems* 11, 2 (2000), 136–148.

[5] Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 445–447.

[6] Arne Andersson. 1989. Improving partial rebuilding by using simple balance criteria. Springer, 393–402.

[7] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. 2004. Cache-Oblivious Data Structures. *Handbook of Data Structures and Applications* 27 (2004).

[8] Lars Arge, Klaus H Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. 2002. Efficient bulk operations on dynamic R-trees. *Algorithmica* 33 (2002), 104–128.

[9] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[10] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. 2000. Cache-oblivious B-trees. In *focs*. IEEE, 399–409.

[11] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[12] Jon Louis Bentley. 1979. Decomposable searching problems. *Inform. Process. Lett.* 8, 5 (1979), 244–251.

[13] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Trans. on Comput.* 38, 11 (1989).

[14] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[15] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[17] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[18] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[19] Guy E. Blelloch and Yan Gu. 2020. Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*.

[20] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[21] Benjamin Blonder, Cecina Babich Morrow, Brian Maitner, David J Harris, Christine Lamanna, Cyrille Violle, Brian J Enquist, and Andrew J Kerkhoff. 2018. New approaches for delineating n-dimensional hypervolumes. *Methods in Ecology and Evolution* 9, 2 (2018), 305–319.

[22] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[23] Christian Böhm, Stefan Berchtold, and Daniel A Keim. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)* 33, 3 (2001), 322–373.

[24] Russell A Brown. 2014. Building a balanced kd tree in o (kn log n) time. *arXiv preprint arXiv:1410.5420* (2014).

[25] Yixi Cai, Wei Xu, and Fu Zhang. 2021. ikd-tree: An incremental kd tree for robotic applications. *arXiv preprint arXiv:2102.10808* (2021).

[26] Yu Cao, Xiaojiang Zhang, Boheng Duan, Wenjing Zhao, and Huizan Wang. 2020. An improved method to build the KD tree based on presorted results. In *International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 71–75.

[27] Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas. 2018. Concurrent linearizable nearest neighbour search in lock free-kd-tree. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*. 1–10.

[28] Yifei Chen, Yi Li, Rajiv Narayan, Aravind Subramanian, and Xiaohui Xie. 2016. Gene expression inference with deep learning. *Bioinformatics* 32, 12 (2016), 1832–1839.

[29] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L Bocchino Jr, Sarita V Adve, and John C Hart. 2010. Parallel SAH kD tree construction. In *High performance graphics*. Citeseer, 77–86.

[30] Zhenyun Deng, Xiaoshu Zhu, Debo Cheng, Ming Zong, and Shichao Zhang. 2016. Efficient kNN classification algorithm for big data. *Neurocomputing* 195 (2016), 143–148.

[31] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[32] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[33] Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[34] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

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

[36] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.

[37] Igal Galperin and Ronald Rivest. 1993. Scapegoat Trees.. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 93. 165–174.

[38] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.

[39] Kirill Garanzha, Simon Premože, Alexander Bely, and Vladimir Galaktionov. 2011. Grid-based SAH BVH construction on a GPU. *The Visual Computer* 27 (2011), 697–706.

[40] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

[41] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[42] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[43] Yulan Guo, Ferdous Sohel, Mohammed Bennamoun, Min Lu, and Jianwei Wan. 2013. Rotational projection statistics for 3D local surface description and object recognition. *International journal of computer vision* 105 (2013), 63–86.

[44] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3 (1994), 357–399.

[45] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 47–57.

[46] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated

street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.

[47] Georges Hebrail and Alice Berard. 2012. Individual household electric power consumption. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54.

[48] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. 2016. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems* 157 (2016), 169–176.

[49] Warren Hunt, William R Mark, and Gordon Stoll. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing*. IEEE, 81–88.

[50] Jeffrey Ichnowski and Ron Alterovitz. 2020. Concurrent nearest-neighbor searching for parallel sampling-based motion planning in SO (3), SE (3), and euclidean spaces. In *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer, 69–85.

[51] Intel Corporation. 2024. VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[52] Intel Threading Building Blocks [n. d.]. Intel Threading Building Blocks (TBB). https://www.threadingbuildingblocks.org.

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

[54] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 21, 2 (1992), 195–204.

[55] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence* 24, 7 (2002), 881–892.

[56] Jiaxin Li, Ben M Chen, and Gim Hee Lee. 2018. So-net: Self-organizing network for point cloud analysis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9397–9406.

[57] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.

[58] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. 2003. The global k-means clustering algorithm. *Pattern recognition* 36, 2 (2003), 451–461.

[59] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[60] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[61] Leland McInnes and John Healy. 2017. Accelerated hierarchical density based clustering. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 33–42.

[62] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2024. Pkd-tree: Parallel kd-tree with Batch Updates. https://github.com/ucrparlay/KDtree.

[63] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[64] Mark H Overmars. 1983. *The design of dynamic data structures*. Vol. 156. Springer Science & Business Media.

[65] Mark H Overmars and Jan Van Leeuwen. 1981. Maintenance of configurations in the plane. *Journal of computer and System Sciences* 23, 2 (1981), 166–204.

[66] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 618–627.

[67] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases (SSTD)*. Springer, 46–65.

[68] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing* 18, 3 (1989), 594–607.

[69] Maximilian Reif and Thomas Neumann. 2022. A scalable and generic approach to range joins. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3018–3030.

[70] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 10–18.

[71] Boris Schäling. 2011. *The boost C++ libraries*. Boris Schäling.

[72] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[73] Nick Scoville, H Aussel, Marcella Brusa, Peter Capak, C Marcella Carollo, M Elvis, M Giavalisco, L Guzzo, G Hasinger, C Impey, et al. 2007. The cosmic evolution survey (COSMOS): overview. *The Astrophysical Journal Supplement Series* 172, 1 (2007), 1.

[74] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. 2005. *Nearest-neighbor methods in learning and vision: theory and practice*. Vol. 3. MIT press Cambridge, MA, USA:.

[75] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, Vol. 26. Wiley Online Library, 395–404.

[76] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and Joã o Gama. 2013. Data stream clustering: A survey. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–31.

[77] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 7 pages. https://doi.org/10.1145/2786.2793

[78] Mark William Smith, Jonathan L Carrivick, and Duncan J Quincey. 2016. Structure from motion photogrammetry in physical geography. *Progress in physical geography* 40, 2 (2016), 247–275.

[79] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[80] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th international conference on world wide web*. 287–297.

[81] The CGAL Project. 2020. *CGAL User and Reference Manual* (5.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.1/Manual/packages.html

[82] Marc J van Kreveld and Mark H Overmars. 1991. Divided kd trees. *Algorithmica* 6 (1991), 840–858.

[83] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: a library for parallel computational geometry. In *European Symposium on Algorithms (ESA)*.

[84] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1982–1995.

[85] Hiroki Yamasaki, Atsushi Nunome, and Hiroaki Hirata. 2018. Parallelizing the Construction of a k-Dimensional Tree. In *2018 IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD)*. IEEE, 23–30.

[86] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on gpus using r-trees. In *Proceedings of the 2Nd ACM SIGSPATIAL international workshop on analytics for big geospatial data*. 23–31.

[87] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. 2016. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of medical systems* 40 (2016), 1–8.

[88] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

A Proof for Tree Height

Lemma A.1. *Function $f(n) = -\log n / \log(1/2 + 1/\log n) - \log n = O(1)$ for $n > 4$.*

Proof. Let $t = \log n$, we have:

$$
\begin{aligned}
f(t) &= -\frac{t}{\log(t+2) - \log 2t} - t \\
&= -\left(\frac{t}{\log(t+2) - \log t - 1} + t\right) \\
&= -h(t)
\end{aligned}
$$

Clearly, $f(t) = O(1)$ when $t \longrightarrow 2^+$. We then show $f(t) > 0$ holds for $t > 2$, which is :

$$
\begin{aligned}
\log \frac{t+2}{2t} > \log \frac{1}{2} &\implies \frac{1}{\log(t+2) - \log 2t} + 1 < 0 \\
&\implies h(t) < 0
\end{aligned}
$$

as desired. The remaining is to show $f(t)$ is decreasing, which equivalents to show $h(t)$ is increasing over $t > 2$. The derivative of $h(t)$ is

$$
\begin{aligned}
h'(t) &= \frac{\log(t+2) - \log t - 1 - t\left(\frac{c}{t+2} - \frac{c}{t}\right)}{(\log(t+2) - \log t - 1)^2} + 1 \\
&= \frac{\log \frac{t+2}{t} - 1 - \frac{c \cdot t}{t+2} + c}{\left(\log \frac{t+2}{t} - 1\right)^2} + 1
\end{aligned}
$$

where $c = 1/\ln 2$. Let $k = (t+2)/t$. We wish to show that $h'(k) > 0$ holds for $k \longrightarrow 1^+$, namely,

$$
\begin{aligned}
&\frac{\log k - 1 - c/k + c}{(\log k - 1)^2} + 1 > 0 \\
\iff &\log^2 k - \log k - c/k + c > 0 \\
\iff &g(k) > 0
\end{aligned}
$$

Since $g(1^+) > 0$, therefore, it is sufficient to show $g'(k) > 0$ holds for k, i.e.,

$$
\begin{aligned}
&2c^2 \cdot \ln k / k - c/k + c/k^2 > 0 \\
\iff &2c \cdot k \ln k - k > -1 \\
\iff &k(2c \ln k - 1) > -1
\end{aligned}
$$

The function w.r.t k in LHS is increasing and equals to -1 (the RHS) when $k = 1$. Proof follows then. □

B Proof for Batch Updates

Theorem B.1 (Updates). *A batch update (insertions or deletions) of a batch of size m on a Pkd-tree of size n has $O(\log n \log_M n)$ span* whp*; the amortized work and cache complexity per element in the batch is $O(\log^2 n)$ and $O(\log(n/m) + (\log n \log_M n)/B)$* whp*, respectively.*

Proof. We will start with the span bound. According to Thm. 3.4, the sieve process and trees rebuilding (rebalancing) all have $O(\log n \log_M n)$ span *whp*. Note that the tree rebuilding (line 19) can only be triggered once on any tree path. The span for other parts is $O(\log n)$—the InsertToSkeleton function can be recursively call for $\lambda = O(\log n)$ levels each with constant cost. In total, the span is the same as the construction algorithm, since in the extreme case, the entire tree can be rebuilt.

Then we show the work bound. The cost to traverse the Pkd-tree and find the corresponding leaves to update is $O(\log n)$ per point *whp*, proportional to the tree height. Once a rebuild is triggered (on line 19), the cost is $O(n' \log n')$ where n' is the subtree size. After that (or the original construction), each subtree will contains $(1/2 \pm \sqrt{(12c \log n)/\sigma}/4)n' = (1/2 \pm \alpha/4)n'$ points *whp* (Lem. 3.1). We need to insert at least another $3\alpha n'/4$ points for this subtree to be sufficiently imbalance that triggers the next rebuilding of this subtree. The amortized cost per point in this subtree is hence $O(\log n'/\alpha) = O(\log n')$ on this tree node assuming α is a constant. Note that Pkd-tree has the tree height of $O(\log n)$, so overall amortized work per inserted/deleted point is $O(\log^2 n)$.

We can analyze the cache complexity similarly. We first show the rebuilding cost. For a subtree of size n', the cost is $O((n'/B) \log_M n')$ (Thm. 3.3). The amortized cost per updated point, using the same analysis above, is $O((1/B) \log_M n')$. Again since the tree height is $O(\log n)$, the overall amortized work per inserted/deleted point is $O((\log n \log_M n)/B)$. Then, we consider the cost to traverse the tree and find the corresponding leaves to update. Finding m leaves in a tree of size n will touch $O(m \log(n/m))$ tree nodes [16], so the amortized block transfer per point is $O(\log(n/m))$. Putting both cost together gives the stated cache complexity. □

C Handling of Duplicates

One issue we observed in some existing kd-tree implementations (e.g., CGAL) is the inefficiency in dealing with duplicate points. Because many points may fall onto the split hyperplane and a default approach will put all of them on one side of the tree, the tree height (and thus the query performance) may degenerate significantly. Pkd-tree uses a special *heavy leaf* to handle this. When all points in a node are duplicates, we use a heavy leaf to store the coordinate and the count.

D Comparison to the R-tree

The R-tree is a commonly seen spatial index structure in practice. The Boost library [71] provides an optimized sequential implementation of the R-tree. We compare the Pkd-tree with the Boost R-tree in terms of tree construction, k-NN, and range report. The R-tree in Boost only supports sequential tree construction and updates. For k-NN and range report, we parallelize all queries for both Pkd-trees and R-trees. We note that the R-tree supports more general functionalities than kd-trees, such as range queries with arbitrary shapes, spatial overlap/intersection queries, handling non-point and high-dimensional objects, and may not be specifically optimized for k-NN or range queries. Tab. 7 summarizes the results.

With 96 cores, our parallel Pkd-tree is 91.9–115× faster than the sequential Boost R-tree in construction, 30–580× faster in batch insertions, and 133–2259× faster in deletion. Since Boost R-tree only supports point updates, the speedup of Pkd-tree comes from both handling multiple updates in a batch and parallelism. Regarding the k-NN query, the Pkd-tree is 10.4–14.8× faster than the Boost R-tree. For the range report query, the Pkd-tree is 4.13–4.17× faster. One possible reason for the speedup is that Pkd-tree can output the candidates of a subtree in parallel when its associated subspace is fully contained in the query box by running a parallel flatten.

Benchmark (1000M-2D)	Baselines	Build	Batch Insert				Batch Delete				10-NN (1%)	Range Report (10K, 1M]
			0.01%	0.1%	1%	10%	0.01%	0.1%	1%	10%		
Uniform	Ours	3.15	.004	.020	.104	.495	.004	.022	.121	.526	.381	.391
	R-tree (seq.)	363	.282	2.86	28.6	287	1.10	11.6	121	1188	5.64	1.62
	Zd-tree	6.70	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	.870	n.a.
Varden	Ours	3.66	.002	.007	.055	.473	.002	.006	.049	.477	.172	.382
	R-tree (seq.)	336	.060	.606	6.21	64.7	.267	2.23	24.7	308	1.79	1.60
	Zd-tree	6.70	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	.331	n.a.

Table 7: Running time (in seconds) for the Pkd-tree and other baselines on 10^9 points in 2 dimensions. Lower is better. "R-tree (seq.)": the serial R-tree implementation from Boost [71]. "Zd-tree": the parallel Quad/Oct-tree implementation using the Morton order from [15]. "10-NN": 10-nearest-neighbor queries on 10^7 points. "Range report": orthogonal range report queries on 10^4 rectangles, with output sizes in 10^4–10^6. The fastest time for each test is underlined. "n.a.": not applicable.

Tree	Baselines	Query points					
		Uniform			Varden		
		1-NN	10-NN	100-NN	1-NN	10-NN	100-NN
Uniform	Ours	.209	.765	2.66	.086	.162	.774
	Log-tree	2.43	4.48	16.9	1.21	2.00	15.7
	BHL-tree	.315	1.02	4.24	.204	.911	8.89
	CGAL	.333	2.32	13.0	.108	.223	1.57
Varden	Ours	.938	1.71	4.15	.056	.190	.888
	Log-tree	t.o.	t.o.	t.o.	2.43	4.48	16.9
	BHL-tree	t.o.	t.o.	t.o.	.315	1.02	4.24
	CGAL	1.68	3.07	7.80	.083	.219	1.28

Table 8: In-distribution and out-of-distribution k-NN query time (in seconds) for Pkd-tree and other baselines on synthetic datasets with 3 dimensions. Lower is better. The tree contains 10^9 points, and the query points contains 10^7 candidates. "t.o.": time out after 600s.

Bench.	Baselines	α	Batch Insert (1%)				Batch Delete (1%)			
			2	3	5	9	2	3	5	9
Uniform 1000M	Ours	0.03	2.95	4.23	6.05	10.3	3.46	4.60	6.66	10.9
		0.1	.686	.799	1.36	1.97	.776	.986	1.52	2.19
		0.3	.104	.107	.123	.152	.121	.134	.171	.232
	Log-tree	-	2.16	2.66	3.67	6.19	.396	.485	1.94	2.39
	BHL-tree	-	31.4	40.3	57.1	103	30.9	39.3	68.7	114
	CGAL	-	1660	1815	1863	2145	41.2	41.3	45.0	40.2
Varden 1000M	Ours	0.03	2.81	3.05	4.21	9.65	7.79	10.7	10.9	20.5
		0.1	.485	.690	.978	1.83	2.09	2.09	2.65	4.11
		0.3	.055	.107	.157	.350	.049	.112	.127	.237
	Log-tree	-	2.01	2.60	3.72	6.07	1.06	1.14	1.92	2.30
	BHL-tree	-	29.4	39.1	57.3	102	29.0	38.4	67.0	123
	CGAL	-	849	700	582	599	13.0	9.53	23.1	3.90

Table 9: Batch update time (in seconds) for Pkd-tree and other baselines on synthetic datasets with dimensions 2, 3, 5, and 9. Lower is better. The tree contains 10^9 points, and the batch contains 10^7 points from the same distribution as the points in the tree. Parameter α is the imbalance ratio used in Pkd-tree.

E Comparison to the Zd-tree

We also compare the Pkd-tree with the Zd-tree [15], which is a parallel quad/octree based on the Morton order (aka. the space-filling curve). The Zd-tree maps each point to an integer by interleaving the bits of the coordinates and uses integer sort as preprocessing so that the tree construction has $O(n)$ work and $O(n^{\epsilon})$ span, where n is the input size and $\epsilon < 1$ a constant.

The implementation of the Zd-tree is part of the PBBS [5], which should support parallel tree construction, batch updates, and k-NN. Due to integer precision issues, Zd-tree's implementation only supports inputs in 2 or 3 dimensions. We omit the evaluation for batch updates since their code has problems that are causing it to produce incorrect results. Tab. 7 summarizes the comparison between the Pkd-tree and the Zd-tree.

For construction, the Pkd-tree is 1.34–2.12× faster than the Zd-tree on both datasets and dimensions. This is surprising as Zd-tree is a quad/octree based on the space-filling curve, which handles multi-dimensional points as integers. Therefore, the computation is simpler than the kd-tree-based structures in construction and updates (and is thus reasonable to achieve higher performance). We believe the reason is the I/O optimizations we designed in Pkd-trees, and applying them to quad/octree can be an interesting future work. For k-NN query, the Pkd-tree is 1.22–2.40× faster than the Zd-tree on both datasets. The reason is that the Pkd-tree uses object median as a cutting plane, which enables more efficient special prune during the searches. Besides, the low memory usage of the tree structure of the Pkd-tree achieves higher cache utilization and thus better query performance.

F Out-of-distribution Queries

In the context of tree queries, the out-of-distribution (OOD) query refers to the query with a distribution that is significantly different from the data distribution used to construct the tree. This can lead to inefficiencies because the tree structure may not be optimized for such queries. We perform both the in-distribution and out-of-distribution query for all baselines by first constructing the tree using the points from one distribution and then performing the k-NN query using points from another distribution. Tab. 8 illustrates the results.

In general, Pkd-tree still achieves the best performance. Both the Pkd-tree and CGAL are relatively resistant to OOD queries, which is reasonable as the kd-trees use of the object median as the cutting plane is less sensitive to the data distribution. However, both the Log-tree and the BHL-tree suffer from timeouts on one of the OOD queries. This is due to the prune heuristic used in their implementation, which skips a (sub-)tree only when the distance between the query point and the cutting plane (rather than the bounding box) is larger than the current best candidate. While this is a commonly used heuristic that trades off more tree nodes traversed for faster computing per node, we note that this heuristic leads to much worse performance for some OOD queries.

G Imbalance Ratios for Batch Update

We measure the batch update time of the Pkd-tree with different values of imbalance ratio α, which serves as a complementary for Tab. 3. The results are summarized in Tab. 9. Note that we omit the tree construction and queries since these metrics tested in Tab. 3

	Tree	CC(M)	Inst(M)	IPC	CRs(M)	CMs(M)	BR(M)	BMs(M)
HT	Pkd	95,300	31,084	.326	926	508	6,832	58.0
	Pkd-bb	33,885	17,143	.506	245	134	2,279	18.0
HH	Pkd	69,078	22,127	.320	557	361	4,114	117
	Pkd-bb	27,987	10,831	.387	242	150	1,636	44.0
CHEM	Pkd	243,014	65,919	.271	1,662	1,450	14,318	170
	Pkd-bb	139,701	32,887	.235	954	820	6,583	107
GL	Pkd	71,844	21,767	.303	439	345	3,376	118
	Pkd-bb	62,637	13,438	.215	407	317	1,981	101
CM	Pkd	120,478	22,407	.186	692	649	3,397	173
	Pkd-bb	133,104	21,913	.165	752	703	3,296	176
OSM	Pkd	71,679	12,247	.171	460	426	1,366	50.0
	Pkd-bb	75,185	11,124	.148	473	441	1,178	48.0

Table 10: Hardware profiling of vanilla Pkd-tree (Pkd) and a variant with bounding box optimizations (Pkd-bb) for range report query on real-world datasets. Underlined values indicate better performance. The range report query contains 10^4 rectangles each with output size 10^4–10^6. Different queries are performed in parallel, and each query searches the tree in serial. "CC": Cycles, "Inst": Instructions, "IPC": Instructions per cycle, "CR": Cache reference, "CMs": Cache misses, "BR": Branches, "BMs": Branch misses.

	Tree	Time (sec.)	Average # of nodes proceed			
			Leaf	Interior	Skip	Flatten
HT	Pkd	.587	2,675	3,957	1,282	0
	Pkd-bb	.268	207	891	605	79
HH	Pkd	.385	2,817	3,621	802	3
	Pkd-bb	.192	615	1,135	455	65
CHEM	Pkd	1.15	4,276	5,701	1,425	0
	Pkd-bb	.837	1,330	2,506	1,091	84
GL	Pkd	.329	3,268	5,478	2,042	167
	Pkd-bb	.291	1,285	3,484	1,993	206
CM	Pkd	.531	2,456	3,939	1,053	429
	Pkd-bb	.577	2,195	3,785	1,120	470
OSM	Pkd	.326	529	1,243	435	278
	Pkd-bb	.363	236	959	439	284

Table 11: Algorithmic statistic of vanilla Pkd-tree (Pkd) and with bounding-box optimizations (Pkd-bb) for range report on real-world datasets. Underlined values indicate better performance. The range report query contains 10^4 rectangles each with output size 10^4–10^6. Different queries are performed in parallel, and each query searches the tree in serial. "Leaf": average number of leaf nodes visited per query, "Interior": average number of non-leaf nodes visited per query, "Skip": average number of nodes skipped per query, i.e., the associated sub-space does not intersect with the query box, "Flatten": average number of nodes flattened per query, i.e., the associated sub-space is fully contained in the query box.

are not affected by the imbalance ratio.

When $\alpha = 0.03$, batch insertion in the Pkd-tree almost always rebuilds the entire tree, as does batch deletion. However, the Pkd-tree is still two orders of magnitude faster than the BHL-tree, which rebuilds the whole tree for batch insertion as well. The Pkd-tree is also one order of magnitude faster than the BHL-tree for batch deletions. When the imbalance is more tolerated and $\alpha = 0.1$, the Pkd-tree becomes the fastest among all baselines for batch insertion. For batch deletion, however, the Log-tree is faster than the Pkd-tree on 6 out of 8 instances by a factor of 1.39–2.03×, as the Log-tree only needs to build a small portion of the tree when the batch size is small. Finally, with α set to 0.3, the Pkd-tree achieves a speedup of 5.24–67.6× for batch insertion and 6.42–159× for batch deletion, compared to $\alpha = 0.03$ or $\alpha = 0.1$. With $\alpha = 0.3$, the Pkd-tree is also the fastest among all baselines.

H Profiling for Range Report Queries

As we mentioned, Pkd-tree does not store the bounding box for subtrees in each node, but instead computes them on-the-fly during the query. This reduces the memory usage of the tree, while increasing the computation cost and potentially more number of nodes to be explored during the query. To verify this trade-off, we perform the hardware profiling, as well as summarizing the algorithmic statistics, for both the Pkd-tree and the variant with the bounding box stored in each node. The results are illustrated in Tab. 10 and Tab. 11, respectively.

Storing the bounding box within the node can facilitate pruning, resulting in fewer nodes being explored during searches. However, this advantage diminishes as the dimensionality of the input points decreases. On the contrary, reduced memory usage enhances search speed more, as the dynamically computed bounding box is nearly as tight as the exact one when the dimensionality of inputs is low. We have discussed this in the main paper, and show the full results here in Tab. 10 and Tab. 11.

I High-dimensional Synthetic Dataset

We compare the Pkd-tree with other baselines on synthetic datasets with 12 dimensions on 100 million points, and present the results in Tab. 12. Pkd-tree remains its advantage on tree construction and batch update, due to higher cache efficiency. For tree construction, the Pkd-tree is 18.3–21.2× faster than the Log-tree, 13.4–15.4× faster than the BHL-tree and 43.5–120× faster than the CGAL. Regarding batch insertion, Pkd-tree achieves 3.27–240× speedup than Log-tree, and orders of magnitude faster than BHL-tree and CGAL. For batch deletion, compared with the best baseline, Pkd-tree is 1.5–202× faster than the CGAL, 2.53–172× faster than Log-tree and orders of magnitude faster than BHL-tree.

For 10-NN queries, Pkd-tree is 1.24× faster than CGAL on `Uniform`, while it is slightly slower on `Varden`, where CGAL is 1.04× faster. Regarding the range report, Pkd-tree is 6.27–10.9× faster than CGAL. Both Log-tree and BHL-tree timeouts on k-NN queries as explained in Appendix F. They also experience the segmentation fault on the range report for the same reason.

Benchmark (100M-12D)	Baselines	Build	Batch Insert				Batch Delete				10-NN	Range Report
			0.01%	0.1%	1%	10%	0.01%	0.1%	1%	10%	10^7 queries	10^4 queries
Uniform	Ours	.938	.001	.003	.017	.115	.002	.005	.029	.148	51.4	17.4
	Log-tree	19.9	.013	.017	4.09	4.22	.345	.347	.345	.427	t.o.	s.f.
	BHL-tree	14.4	13.1	13.0	13.3	14.6	10.9	11.1	11.4	13.1	t.o.	s.f.
	CGAL	112	164	168	157	171	.032	.278	2.83	30.0	63.7	109
Varden	Ours	1.07	.002	.004	.025	.269	.002	.005	.023	.169	.048	10.5
	Log-tree	19.7	.012	.013	3.99	4.00	.342	.349	.346	.427	t.o.	s.f.
	BHL-tree	14.4	12.8	12.7	12.8	14.1	11.3	11.8	11.9	13.0	t.o.	s.f.
	CGAL	46.7	70.7	63.7	59.5	63.9	.003	.032	.452	4.82	.046	114

Table 12: Running time (in seconds) for the Pkd-tree and other baselines on 10^8 points in 12 dimensions. Lower is better. The 10-NN queries 10^7 points in parallel, and the range report contains 10^4 rectangle range queries in parallel with output size 10^4 to 10^6. All queries searches the tree in serial. “t.o.”: time out after 600s. “s.f.”: segmentation fault.