并行动态空间索引

Ziyang Men
加州大学河滨分校
河滨市 , USA
ziyang.men@email.ucr.edu

Bo Huang
加州大学河滨分校
河滨市 , USA
bo.huang@email.ucr.edu

Yan Gu加州
大学河滨分校 河滨
市，美国
ygu@cs.ucr.edu

Yihan Sun
加州大学河滨分校
河滨市，美国
yihans@cs.ucr.edu

arXiv:2601.05347v1 [cs.DB] 8 Jan 2026

摘要

维护空间数据（二维或三维空间中的点）至关重要，应用范围广泛，包括计算机图形学、GIS 和机器人学。为了高效地更新和查询空间数据，人们提出了许多数据结构，称为空间索引，例如 kd 树、八叉树/四叉树（也称为 Orth 树）、R 树和包围体层次结构（BVH）。在实际应用中，空间数据集往往高度动态，需要以低延迟批量更新点。因此，需要能够高效并行地批量更新空间索引。遗憾的是，迄今为止很少有工作实现了这一目标。

本文系统地研究并行空间索引，特别关注如何在高度动态工作负载下实现高性能更新。我们选择了两类被认为适合低延迟更新的空间索引：Orth 树和 R 树/BVH。我们提出了两种数据结构：并行 Orth 树 P-Orth 树，以及并行 R 树/BVH SPaC 树系列。与现有的并行 kd 树和 Orth 树相比，P-Orth 树和 SPaC 树在批量更新方面均表现出更优的性能，同时相较于各自对应的 Orth 树和 R 树，保持了更好的或具有竞争力的查询性能。我们还通过全面的实验比较了各种并行空间索引的性能，并在论文末尾分享了研究发现。

1 引言

空间数据广泛存在于地理信息系统（GIS）、空间数据库、计算机图形学、机器人学及其规划等诸多领域。高效处理二维或三维空间中的此类几何对象（通常为点）至关重要，无论是维护（构建、插入、删除）还是执行查询（范围查询、最近邻查询等）都如此。

鉴于其广泛的适用性，人们提出了许多广为人知的数据结构（通常称为“空间索引”）来处理空间数据，例如 kd 树[11], 八叉树/四叉树[25]（统称为*Orth* 树）、范围树、 [12], R 树 [32],以及包围体层次结构（BVH） [4]。空间索引通常将点组织成树，每棵子树对应一个子空间（不一定互不重叠）。子空间的包围盒可用于在查询时剪枝子树。例如，在最近邻搜索中，当搜索到达某个子树时——如果该子区域距查询点比当前最近邻更远，就可以剪枝该子树。尽管这些树维护着不同的不变式，但它们都遵循相同的总体思路：通过剪枝，在查询时跳过大多数对象，从而实现高效的查询性能。

现实应用可能涉及高度动态的数据，更新可能对延迟或吞吐量十分敏感。例如，在 3D 游戏中，必须快速反映移动对象的位置变化，以影响光照和碰撞检测；而 GIS 应用通常会接收海量传感器流，此时总更新吞吐量至关重要。在这两种场景中，更新往往以批次形式到达，必须及时纳入索引。为高效处理更新和查询，不同的空间索引提供了不同的权衡。传统上， kd 树因其最强的不变量（按对象中位数划分）而被认为查询效率极高，但更新代价高昂。Orth 树凭借更简单的不变量（按空间中位数划分），能够提供有竞争力的查询性能和更快的更新。R 树/包围体层次结构涵盖了大量不同方案；它们通常支持最简单、最快的更新，但查询速度较慢。

随着数据量不断增长，并行性对于设计高效的数据结构变得至关重要。遗憾的是，关于支持批量更新的并行空间索引，相关研究鲜有报道。在 CGAL [24] 和 Boost [51], 这两个著名库中，大多数空间索引都是串行的。唯一的例外是 CGAL 的 kd 树，但它存在已知的可扩展性问题 [16, 43]。文献 [57],介绍了范围树的并行构建，但不支持批量更新。2022 年，Blelloch 和 Dobson [16] 提出了 Zd 树，这是首个并行四叉树。其思路是利用莫顿曲线——一种将二维或三维点映射为一维整数的空间填充曲线（SFC）——并借助这些信息来实现构建和批量更新。然而，尽管 Zd 树在更新方面具有更好的理论界（每个更新点的工作量为 $O(\log n)$，而非 $O(\log^2 n)$），其速度仍慢于后来提出的并行 kd 树（Pkd 树） [43], 。我们认为，主要原因在于 Pkd 树采用了 I/O（缓存）优化。更值得关注的是，尽管 R 树/BVH 的结构最简单，但我们几乎没有发现关于它们的并行批量更新研究。事实上，大多数现有方法要么基于单次插入/删除 [7, 14, 32, 51, 54], ，要么在更新时完全重建树 [30, 46, 59]。唯一相关的研究 [49] 采用了对数方法，这会显著增加查询时间（详见第 2.3 节）。因此，很自然会提出这样的问题：Orth 树和 R 树/BVH 能否在高度动态工作负载下继续发挥其优势

即将发表于：ACM SIGPLAN并行编程原理与实践研讨会（PPoPP），2026年。

在并行环境中。具体而言，我们研究它们能否在并行环境下实现比 kd 树快得多的构建和批量更新（并提供更好的理论保证），同时保持与其串行版本相当的查询性能。

本文系统地研究了并行空间索引，重点关注如何在高度动态工作负载下实现高吞吐量更新和高效查询。我们提出了两类新的数据结构：**P-Orth 树** 和**SPaC 树系列**。我们将这些数据结构集成到一个名为并行空间索引库 [42], 的库中，简称 PSI-Lib 或 Ψ-Lib。

我们首先介绍一种名为 P-Orth 树的并行 Orth 树设计。几乎所有现有的 Orth 树 [16, 38, 39, 46, 61]都使用空间填充曲线（SFC）来加速构建与更新。然而，仅计算并排序各点的 SFC 代码，就需要多次读取和移动所有数据，耗时较长。本文介绍了不使用 SFC 的 P-Orth 树设计。

根据 orth 树的定义，在 D 维空间中，我们的主要思路是将空间直接均匀划分为 2^D 个桶（子空间），并相应地将输入点划分到各个桶中。为此，我们借鉴了 Pkd 树 [43], 中的筛分算法，该算法直接重新排列输入点，并将属于同一桶的点聚集在一起。随后并行处理每个桶。这使得 Orth 树能够实现 I/O 高效构建和批量更新算法。

从概念上讲，我们的算法等价于对 SFC 代码进行整数排序，但不会生成、存储或使用这些代码。我们认为这一算法思想很有意思，读者可参阅第 3 节，了解算法细节与分析。

接下来的问题是，空间填充曲线是否仍可用于构建空间索引。正如前文所述，空间填充曲线已用于 Orth 树和 R 树/BVH 中 [16, 30, 37–39, 46, 50, 58, 61]。然而，据我们所知，目前尚无任何实现能够在更新性能上与 Pkd 树和 P-Orth 树相媲美，这主要是因为并行支持有限或缺失。

本文提出了SPaC 树系列，它在保持查询性能与现有 R 树/BVH 相当的同时，支持极快的更新（正如 R 树所应具备的）。为此，我们采用PaC 树[23], 作为核心结构，这是一种并行平衡二叉搜索树。PaC 树的关键思想是使用基于连接的算法[2, 3, 17, 57]在并行更新期间高效地重新平衡，并通过叶节点分块（在每个叶节点的平坦数组中维护 16–32 个对象）来改善缓存局部性。为了支持空间查询，一种简单的方法是使用点的 SFC 编码作为键，将点存储在PaC 树中，并为每个树节点添加边界框。然而，这种直接改造会导致更新速度较慢（最高可比 3.5× 慢Pkd 树；见图 3 中“CPAM-H”和“CPAM-Z”两列）。我们发现，主要瓶颈在于维护由 SFC 诱导的全序

T	一棵（子）kd 树，也就是树中的点集		
ϕ	叶节点包裹阈值（叶节点大小上界）		
k	查询中所需的最近邻数量		
λ	树草图中的层数		
$\mathcal{T}$	树骨架位于 T，最大层数为 λ		
P	输入/更新点序列		
$P[l, r)$	P 中索引范围 $[l, r-1]$ 内的点		
T_p	与非叶节点 T关联的枢轴点		
T_l	T 的左子节点	T_r	T的右子节点
n	树大小	m	批次大小
D	维数	d	特定维度
M	小内存（缓存）大小		
B	内存块（缓存行）大小		

表1. 本文中使用的符号。

PaC 树中的所有点。为应对这一挑战，我们精心重新设计了PaC 树中的基于连接的算法，使其仅在偏序下维护空间数据；也就是说，插入/删除后，叶节点中的点可以无序。更多细节见第4节。我们将这一设计称为空间 PaC 树，简称SPaC 树。在 Ψ-Lib 中，我们同时采用莫顿曲线（SPaC-Z 树）和希尔伯特曲线（SPaC-H 树）。

我们的 P-Orth 树和 SPaC 树都有坚实的理论依据。我们证明，每个对象的更新开销为 $O(\log n)$，对于 SPaC 树 而言；对于 $O(\log \Delta)$，对于 P-Orth 树 (Δ为纵横比，参见第 3.3节)，这比$O(\log^2 n)$，对于 Pkd 树而言，要强得多。我们的批量更新具有多对数跨度，体现出强大且可扩展的并行性。

我们在具有各种输入分布、查询分布、查询类型和更新模式的工作负载上测试了 Ψ-Lib。我们将 Ψ-Lib 与现有的并行和顺序基线进行比较，包括 Pkd 树、Zd 树等。我们的实验模拟了静态设置，以及将更新连续应用于初始树的高度动态设置。这种设置更能反映各数据结构处理高度动态工作负载的能力，尤其能展现索引质量是否以及如何受到数据集逐步演变的影响。借助新算法，P-Orth 树 和 SPaC 树 在构建和更新性能方面均表现出色，同时查询性能与常规 Orth 树和 R 树相当。P-Orth 树 在均匀分布数据的构建和查询中几乎总是最快的，在更新方面也接近最佳。SPaC 树 支持极快的并行批量更新——速度可比 Pkd 树快 2 至 6 倍，尤其适用于输入点、查询或插入/删除顺序呈偏斜分布的情况。通过全面的实验，我们在第 5.4节分享了研究结果，并在图 8中展示了各并行空间索引的查询-更新权衡。我们的代码可在 [42] 获取。

2 预备知识与相关工作

本文中，我们使用 n 表示输入规模或树大小。我们使用$\log n$ 记号表示$\log_2(n+1)$ 对数。本文使用的记号汇总于表1。

2.1 计算模型

我们考虑经典分叉-合并范式下的共享内存多处理器环境，并采用二叉分叉 [8,18, 21]。每个计算线程都是一个顺序执行的随机存取机（RAM），并扩展了分叉指令，可生成两个并行执行的子线程；两个子线程都完成后，父线程恢复执行。通过对数层次的分叉，可以高效模拟并行 for 循环。分析算法时，我们采用工作量-跨度模型，其中工作量是算法中的操作总数，跨度是并行计算中最长的依赖链。使用随机化工作窃取调度器时，工作量为W、跨度为S的计算以$W/\rho + O(S)$时间以高概率（在W内）在ρ个处理器上执行 [8, 21, 31]。

我们使用理想缓存模型 [26] 来分析算法的 I/O 成本。在该模型中，内存分为两层：大小为 M 的快速内存（缓存）和容量任意大的慢速内存。CPU 只能访问快速内存中的数据（无需成本），数据以大小为 B的块在两层之间传输。每次块传输产生单位成本。缓存采用全相联映射，并使用最优离线缓存替换策略。算法的缓存复杂度通过其执行期间两层内存之间的块传输次数来衡量。

2.2 空间数据

本文研究欧几里得空间 $\mathbb{R}^D$ 中的点，其中$D = 2$ 为 2 或 3，但所提出的技术可以推广到形状以及任意常数整数 $D > 1$。

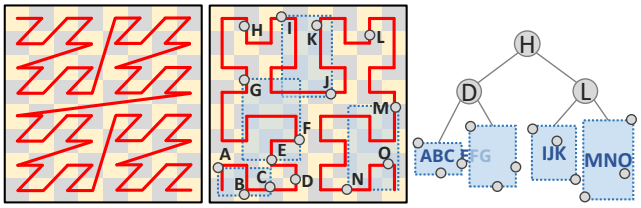
空间数据上的查询。为了评估空间索引的质量，我们使用标准的 k-近邻查询和范围查询。 k近邻（k-NN）查询以点集 P 和查询点 q 作为输入，并在 P中返回距离 q 最近的 k个点。范围查询以点集 P和轴对齐矩形子区域 r 作为输入。范围计数查询返回 r 内 P 中的点数，范围列表查询返回 r内的所有点。

空间填充曲线。空间填充曲线（SFC）将多维点映射到一维序列中。在Ψ-Lib 中，我们使用Z 曲线（莫顿曲线）和希尔伯特曲线，如图1所示。它们都会将每个点编码为一个整数，以确定该点沿曲线的顺序。对于整数坐标，希尔伯特曲线和 Z 曲线都能在常数时间内计算得出。空间填充曲线被广泛用于促进空间
索引 [16, 30, 37, 39, 46, 50]。

2.3 现有的常用空间索引

空间划分树：Orth 树和 kd 树。 在空间划分树中，每个节点表示一个子空间。它的所有子节点共同构成该子空间的一个互不重叠的划分，通常通过轴对齐划分超平面实现，即指定一个划分维度 d和一个坐标 x。因此，空间划分树的区别在于如何选择划分超平面。

作为典型示例， kd 树会 [11] 选取中位数



莫顿曲线 / Z 曲线　　希尔伯特曲线 a 包含 15 个点的 SPaC-H 树

图 1. 空间填充曲线，以及一个包含 15 个点、采用大小为 3 的叶节点封装的 SPaC 树示例。在此例中，每个叶节点包含 3 个点，其包围盒以蓝色标出。

作为划分维度上的坐标，并据此始终将空间均衡地划分为两棵子树。 D 维 Orth 树使用每个维度上的中点，将空间均匀划分为 2^D个子空间（因此是一棵 2^D叉树）。具体而言，二维 Orth 树称为四叉树 [25]，三维 Orth 树称为八叉树 [35]。

并行版本的 kd 树和 Orth 树均已提出。Blelloch 等人 [16] 提出了一种称为 Zd 树的并行 Orth 树，它使用莫顿曲线来辅助构建与更新。Yesantharao 等人 [62] 提出了两种并行kd 树：BHL-tree 和 Log树。只有 Log树支持高效的并行批量更新，其采用对数方法，即维护 O（$\log n$）棵大小分别为 1、2、..... n/2 的树，从而使一次批量更新最多可以分解为 O（$\log n$）次树重构。然而，这种方法会大幅降低查询速度 [43]。近期有一项工作提出了 Pkd 树 [43]，无需采用对数方法，即可在并行构建和批量更新中实现最优工作量与缓存复杂度。其基本思路是通过采样近似对象中位数，并结合筛分算法以 I/O 高效方式划分点。我们的 P-Orth 树也借鉴了这一思路；详情见第 3 节。不过，Pkd 树更新一个大小为 m的批次需要 O（$m \log^2 n$）工作量。我们将展示 Ψ-Lib 如何实现更优的界。

对象划分树：R 树/包围体层次结构。在对象划分树中，每棵（子）树中的对象（点）会被划分为互不相交的子集，每个子集对应一个子节点，并以递归方式构建。每个树节点通常存储一个包围盒（在三维中则为包围体），它是包含其子树中所有对象的最小轴对齐区域。尽管名称各异——数据库中称为 R 树，通常用于二维（“R”代表矩形）；计算机图形学中通常称为包围体层次结构（BVH），通常用于三维（“V”代表体积）——但它们基于相同的概念。为简便起见，我们使用术语“R 树”来指代对象划分树这一通用概念。它们可以是二叉树 [14,30, 60]，也可以具有更大的分支因子 [29, 32, 37, 51]，；既可以离线构建 [30, 46, 51, 59]，也可以增量式构建（从而支持更新） [7, 14, 32, 37, 51, 54]。

据我们所知，唯一支持批量更新的并行 R 树是 Qi 等人提出的 [50]，，其采用了对数方法。然而，如前所述，对数方法会显著降低查询速度，因此并不理想。此外，还存在基于锁的并发 R 树 [22, 44]。

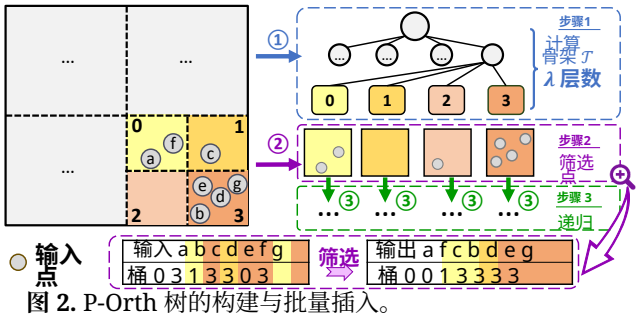


图 2. P-Orth 树的构建与批量插入。

3 并行正交树（P-Orth 树）

本节介绍我们设计的并行正交树（P-Orth 树），该树基于空间中位数递归地将点划分到嵌套区域中。

早期算法。 构建或更新 Orth 树的朴素方法，是从根节点开始，逐层将点分配到子树中，直到到达叶节点 [25, 35]。然而，这种方法速度较慢，因为全局数据移动的轮数与树高成正比，而树高可能很大。因此，几乎所有后续的 Orth 树 [16, 38, 39, 46, 61] 都使用空间填充曲线，尤其是莫顿曲线（见图1），以加快算法速度。其核心思路是按莫顿顺序对所有输入点排序，这只需要 $O(\log_M n)$ 轮全局数据移动，其中 M 表示缓存大小。然后，由于 Orth 树总是在空间中位数处分割，因此可以对排序后的值进行二分搜索，以确定分割超平面；而同一子树中的所有点在莫顿顺序中也构成一个连续范围。Blelloch 和 Dobson 在其 Zd 树论文 [16], 中也采用这一思路来实现并行 Orth 树。

现有工作的不足。 尽管长期以来采用的基于 Morton 的方法在工作量、跨度和缓存界方面表现良好，但仔细观察可以发现，它存在两个主要缺点。

- **性能。** 该方法还必须在预处理阶段为每个点计算 Morton 编码，并对 ⟨编码,点⟩ 对进行排序。这会增加内存占用，并导致对所有数据进行更多轮读写操作，从而带来显著开销（见图 Zd-tree中的“3”）。
- **适用性。** 虽然空间填充曲线可以将高维数据映射到一维，但会受到精度限制。大多数现代机器使用 64位字，这足以支持二维数据（每个维度具有 32 位精度）。然而，三维支持被限制为每个维度 21 位，而处理更高维度（$D > 3$）大多不可行 [1]。即使在较低维度下，当某些子区域的精度耗尽时，也需要回退到朴素的基于分区的解决方案，这并不优雅。

我们的解决方案。 为了解决这些问题，我们设计的 P-Orth 树完全避免使用空间填充曲线。我们证明，基于排序的思想在概念上可以等效地实现，而无需使用空间填充曲线。因此，我们的 P-OrthTree 适用于任何坐标类型和范围（不一定是整数），速度快且灵活。从理论上讲，P-Orth 树在构建和批量更新方面都具有很强的界限。在实践中，P-Orth 树的性能优于Pkd 树和Zd 树，几乎适用于所有情况，但偏斜分布除外；详见第5节。下面，我们将在第3.1节介绍构造算法，在第3.2节介绍更新算法，并在第3.3节进行成本分析。

[1]可以使用更高精度的整数（即 128位字）对 SFC 进行编码，但考虑到 SFC 编码中的位操作、排序和二分搜索，这样做的成本可能也很高。

算法 1： 并行正交树（P-Orth 树）构建

输入： 点序列 P，区域盒 H。
输出： 区域 P 中点 T 的 P-Orth 树。
参数： λ：树骨架的高度。
ϕ：k维树的叶包装。

函数 BuildOrth(P, H)
  如果 $|P| < \phi$，**则**
    返回 一个包含点 P 及其包围盒的叶节点
  通过构建前 $\lambda = (\log_2 M)/2D$ 层，基于 H
  $B[] \leftarrow$ 划分 H 基于 $\mathcal{T}$ // $B[i]$：桶的子区域 i
  // 重新排列点，使同一桶中的点彼此相邻
  $R[] \leftarrow$ 筛选(P, T) // $R[i]$：桶 i中所有点的切片
  并行循环 外部节点 i of $\mathcal{T}$ **do**
    $t \leftarrow$ BuildOrth($R[i]$, $B[i]$) // 递归构建
    将外部节点 i 替换为 t
  计算 T 中所有内部节点的包围盒，并合并大小不超过 ϕ的非叶子子树
  返回T 的根节点

算法 2： P-Orth 树的批量插入

输入： 一系列点 P，一个包含区域 H 的 P-Orth 树 T。 **输出：** 插入 P 后的 P-Orth 树。 **参数：** λ：获取的树骨架的最大高度。

// 删除操作是对称的。
1 函数 批量正交插入(T, P, H) 2 如果 $P = \emptyset$ 则返回 T3 如果 T 是叶节点则 // 插入叶节点 4 返回 BuildOrth($T \cup P, H$) 5 T ← 在 T处检索骨架6 $B[] \leftarrow$ 根据 T 划分 H // $B[i]$：桶 i的子区域 7 $R[] \leftarrow$ 筛选(P, T) // $R[i]$：桶 i中点的切片 8 对 T 中的外部节点 i 进行并行循环 do9 $t \leftarrow$ 批量正交插入($i, R[i], B[i]$)// 递归插入 10 将 T 的外部节点替换为 t11 更新 T 中所有受影响节点的包围盒 12 返回 T 的根节点

3.1 P-Orth树构建

我们的想法是构建 P-Orth 树时，将“概念上的”排序过程与树的构建协调起来。目标是每轮数据移动都能一次性构建 λ 层树，同时实现高并行性。这里我们采用筛选(P, $\mathcal{T}$) 函数，该函数来自 [43],，并行分配点集 P，分配依据是一个 λ层树骨架$\mathcal{T}$。从较高层次来看，我们的算法等价于每轮对 Morton 编码的最高有效位进行整数排序。不过，无需计算、存储或比较编码。注意，筛选() 也用于更新算法。我们在算法P-Orth 树构建过程见算法1，并在图2中进行说明。

算法1 分为三个步骤，如图2所示。第一步（第4–5行）构建树骨架$\mathcal{T}$，包含 $\lambda = (\log_2 M)/2D$层，其中 M 表示缓存大小。这确保$\mathcal{T}$ 的叶节点（外部节点）数量 $2^{\lambda \cdot D}$ 能够装入缓存。理论上，这一步可以并行执行；不过，由于工作量很小，实际中会按顺序执行。注意，计算骨架$\mathcal{T}$ 需要当前子树的包围区域，因此我们还需要计算$\mathcal{T}$ 所有叶节点对应的子区域（第5行）。

构建好 T 后，第二步是将 P 中的点筛选到 T 的叶节点中。此步骤由算法 1 所示的 Sieve(P, T) 函数实现，该函数由 Pkd 树论文 [43] 提出。此函数会对 P 中的所有点重新排序，使所有点按照其所属的桶排序。输出 $R[]$ 记录了排序后数组 P 中每个桶对应的切片，以便将其传递给递归调用来处理各个桶。由于桶的数量通常较少，Sieve 算法类似于并行计数排序 [18], , 可以通过 I/O 高效的方式执行 [19]。受篇幅限制，我们省略 Sieve 算法的细节，并直接将该技术作为黑盒使用。更多细节请参阅 [43] 。图 2 展示了此步骤结果的示意图；完成后，T 的同一叶节点中的所有点都会聚集在一起，概念上存储于数组 $R[]$ 中。

3.2 P-Orth树的批量更新

P-Orth 树的批量插入和删除都与构造算法非常相似。这里我们首先介绍批量插入算法，见算法 2，随后讨论删除算法。

批量插入算法接收一批点 P，并将它们添加到现有的 P-Orth 树 T中。为此，我们也对这些点进行筛选，得到 λ 层，然后递归地并行地将点插入各个桶中。图 2 和算法 2 中，这三个步骤几乎一一对应，但在处理基本情况时存在一些细微差别。对于删除，还需要一个额外步骤：对于所有受影响的叶节点，如果其祖先节点的子树总大小小于叶节点包裹阈值，就将这些祖先节点扁平化。我们的更新算法仍然简单，因为 Orth 树无需重新平衡。

3.3 理论分析

由于篇幅限制，我们将分析推迟到第 A 节，这里仅列出定理。

定理 3.1。算法 1 构建一个大小为 n 的 P-Orth 树，所需工作量为 $O(n \log \Delta)$，跨度为 $O(\log n \log \Delta)$，缓存复杂度为 $O(n/B \log_M \Delta)$。对大小为 n 的 P-Orth 树执行大小为 $m = O(n)$ 的批量更新，所需工作量为 $O(m \log \Delta)$，跨度为 $O(\log m \log \Delta)$，缓存复杂度为$O(m/B \log_M \Delta)$。

这里，Δ表示纵横比，定义为 $\max d(x,y) \min d(x,y)$，其中所有点均为 x 和 y。请注意，当点集在 $\mathbb{R}^D$ 中不含重复项时，$\log \Delta \geq \log \Theta(n^{1/D}) = \Omega(\log n)$。

在更强的假设下——例如，有有界长宽比（$\Delta \leq n^c$ 对于某个常数 $c > 0$）和常数扩张率（完整定义见附录 A）——我们可以得到更紧的界。在有界长宽比的条件下，我们可以证明，构建所需工作量为 $O(n \log n)$，跨度为 $O(\log^2 n)$，缓存复杂度为 $O(n/B \log_M n) = O(\text{Sort}(n))$。更新所需工作量为 $O(m \log n)$，跨度为 $O(\log m \log n)$，缓存复杂度为 $O(m/B \log_M n)$。在同时满足这两个假设时，a k-NN 查询可在 $O(k \log n)$ 工作量内回答 [16]。

4 Spatial PaC-Tree（SPaC 树）

本节介绍 Spatial PaC 树（SPaC 树）的设计。这是一种高度并行的 R 树，能够极快地构建和更新，同时保持良好的查询速度。**现有的 R 树**。如第 2 节所述，R 树是对象划分树，因此构建时采用的启发式方法具有灵活性。早期的原始设计 [10, 13, 32, 45, 55]是增量式的：逐个插入点；贪心策略会迭代地为该点选择一个子树。当某个子树远重于其兄弟子树时，会通过启发式方法（例如“线性” [5, 32], “二次” [32], 或“R*” [10,32]）进行划分。虽然这种方法简单且动态性很强，但难以推广到并行批量更新。因此，关于并行 R 树的既有研究主要集中于并行查询 [36, 41, 48, 63] 或静态构建（批量加载） [1, 6, 28, 40, 47, 52, 56]。然而，对于纯静态场景， kd 树和 Orth 树通常是更好的选择。

一种很有前景的并行化 R 树的方法是使用空间填充曲线（SFC）。空间填充曲线将高维空间中的点映射到一维空间（见图 1），从而可以按照这一维顺序，使用二叉搜索树（BST）或 B 树组织所有点——等价地，如果每个节点都维护其包围盒，就能得到一棵 R 树。Tropf 和 Herzog 最早注意到了这一思想 [58], , 后来这一思想在 Hilbert R 树中得以实现 [34, 37],, 该树构建于 B 树之上。不幸的是，在 B 树上执行并行批量更新颇具挑战。Qi 等人 [50] 指出，对数方法可以避开 B 树上的并行更新，但会显著增加查询开销 [43]。

PaC 树。 PaC 树 [23] 是一种采用叶节点包装技术的并行二叉搜索树（BST），可提升空间和 I/O 效率：将大小低于阈值 ϕ（通常为 32）的子树扁平化为压缩叶节点，并以数组形式存储。它以分治方式采用“基于 Join 的框架”，实现高并行性，并支持完整的二叉搜索树接口，包括构建、单次更新和批量更新，以及各种一维查询。

我们的 SPaC 树。 乍看之下，PaC 树似乎提供了一种直接的解决方案，可将 R 树并行化：它们可以直接用于支持基于空间填充曲线的方法，从而兼具高效率和高并行性。我们实现了这一直接方案，却有些出乎意料地发现，它的速度远低于P-Orth 树和Pkd 树（见CPAM-H 和CPAM-Z，图 3）。瓶颈在于，PaC 树会根据空间填充曲线对点强制施加全序，代价过高。相比之下，P-Orth 树和

算法 3： 并行 SPaC 树构建

输入： 点序列 P。
输出： 位于 P 中的点构成的 SPaC 树 T。

```
1  函数BuildSPaCTree(P)
2      A ← 大小为 |P| 的空对辅助序列 code, id
3      A′ ← HybridSort(P, A)
4      返回 BuildSorted(P, A′)
   // 修改样本排序，以便在排序时计算空间填充曲线代码
5  函数 HybridSort(P, A)
6      从 P 中抽取点样本，并计算其空间填充曲线编码
7      对样本排序并进行二次抽样，以获得枢轴
8      将 P 划分为块，并计算块的偏移量，记为 F[]
9      并行循环 i第 - 块 B 执行
10         并行循环 j第 - 个点 p 位于 B 执行
11             k ← p的空间填充曲线代码
12             id ← p的标识符
13             A[F[i] + j] ← k, id      // 将代码和标识符存储在 A
14         对切片 A[F[i], A[F[i + 1])进行排序
15         与样本合并，获取每个块的计数
16     使用矩阵将 A 重新分配到桶 A′
         转置 [9, 19], 其中 i第 F个桶的偏移量为[i]
17     并行循环 第 i个桶执行                    // 递归排序
18         对切片 A′[F[i], A′[F[i + 1]) 进行排序
19     返回 已排序序列 A
   // 递归构建树。
20 函数 BuildSorted(P, A)
21     n ← |P|
22     如果 n ≤ φ 则          // 输入规模小于叶节点封装
23         使用 A中的标识符检索点 S ⊆ P
24         返回一个包含点 S 及其包围盒的叶节点
25     else
26         m ← n/2
27         并行执行：
28             L ←BuildSorted(P[0, m) A[0, m))
29             R ←BuildSorted(P[m + 1, n) A[m + 1, n))
30     k ← 在 P 中标识符为 A[m]的点          // 枢轴点
31     返回 一个内部节点，其左子节点为 L，右子节点为 R，枢轴
         k，并根据子节点计算包围盒
```

Pkd 树中的叶节点内的点不排序。

为降低更新开销，我们引入了空间 ***PaC*** 树（***SPaC*** 树）。其主要目标是让叶节点中的点保持未排序状态，这需要重新设计并拆分底层 PaC 树算法的部分内容。总体而言，与 PaC 树相比，SPaC 树在两个方面有所改进：1 ）将整个构造算法整合到排序算法中，把空间填充曲线的计算延迟至排序算法的第一轮分发，从而提升整体性能；2）在批量更新中允许叶节点保持未排序状态，从而减少批量更新的工作量，同时几乎不会对查询产生负面影响。

本节其余部分将介绍这种新设计及其分析。

4.1 SPaC 树构建

我们首先介绍SPaC 树的构造算法，见算法 3。要使用 PaC 树 进行构建，一个简单的思路是先计算每个点的 SFC 代码，按此排序，然后在排序后的点上构建一棵平衡二叉搜索树。尽管理论效率很高，直接以这种方式调用 PaC 树 在CPAM中进行构建，速度最多会慢 3× 倍于 Pkd 树的构建。为提升性能，我们主要通过重新设计排序算法来避免不必要的内存读写操作。该算法如函数“HybridSort”（算法 3）所示，包含两项主要改进。首先，我们不再预先计算 SFC 值，而是在排序过程中首次访问各点时计算这些值，从而省去对相关数组的一轮读写操作。其次，我们只对$\langle code,\ id\rangle$ 对进行排序（第 13 行），不对坐标排序。这缩小了递归排序过程的内存占用（因此速度更快），代价是在将点取至叶节点时会产生更多缓存未命中。总体而言，这降低了运行时间。结合这两项技术，算法1相较于朴素实现可获得稳定的加速比（在二维数据上为 3.1–3.5×；见图 3）。

4.2 SPaC 树上的批量更新

我们的 SPaC 树构建在 PaC 树 [23], 之上，后者是一种采用基于 Join 的算法框架 [17]的并行二叉搜索树。其核心思想是仅使用 Join 操作进行树的重新平衡：该操作接收两棵子树 L、 R以及中间的一个键 k，并返回一棵新的平衡树，其中包含 $L \cup \{k\} \cup R$。这里的关键观察是，作为一种空间索引，叶节点中点的顺序（本例中基于希尔伯特编码或 Z 编码）并不能帮助进行空间查询——查询叶节点时，无论如何都必须扫描其中所有点。因此，我们的目标是谨慎地重新设计基于 Join 的算法，在保持理论效率的同时，使其更好地适用于空间索引场景，方法是放宽叶节点中的键顺序限制。实验表明，这项改进显著加快了更新速度，同时不牺牲查询性能。

我们在算法 4 中展示了详细的批量插入算法。该算法首先计算空间填充曲线代码并对输入进行排序。在将点筛分到叶节点后，如果叶节点的大小未超过阈值，算法会将点追加到叶节点并将其标记为未排序；如果超过阈值，则重新构建该叶节点（第 11 行和第 12 行）。接下来，标准的 Join 操作连接两个子树 L 和 R，并执行重新平衡（第 19 行）。不失一般性地，假设 L 比 R 更重，并调用 RightJoin 操作（第 21 行）。RightJoin 递归地划分 L 的右子树，直到可以使用 R 返回一棵平衡树（第 25 行）。当划分到达叶节点时，我们使用 Expose 操作将叶节点展开为一棵树，与 PaC 树中的做法相同（第 32 行）。不同之处在于，如果叶节点被标记为未排序，我们会先对点进行排序（第 43 行）。当划分后的子树与 R 达到平衡时（第 25 行），我们创建一个新的树节点 R'，其子节点为两个平衡子树（第 28 行），并将 R' 连接到 L （第 29 行）。需要注意的是，之前的叶节点展开可能会破坏受影响叶节点的叶节点封装。在这种情况下，我们通过检查树大小来恢复叶节点封装：如果大小适合，则直接将其扁平化为一个叶节点（第 46 行）；如有必要，则将点重新分配到两个叶节点中（第 42 行）。如果叶节点被标记为未排序，我们会先对点进行排序（第 43 行）。

尽管算法 4 看起来很复杂，但我们可以证明它的

算法 4：*SPaC* 树上的并行批量插入 输入：点序列 P 和一棵 *SPaC* 树 T。输出：插入了 P 的 *SPaC* 树。*1* 函数 *PtreeBatchInsert(T, P) 2* 计算 P 中各点的空间填充曲线编码，并据此对 P 排序。// 实际上，我们使用算法 *3* 中的 *HybridSort() 3* 返回 *InsertSorted(T, P) 4* 函数 *InsertSorted(T, P)*5 $n \leftarrow |P|$6 如果 $n = 0$，则返回 T *7* 如果 T 是叶节点，则 *8* 如果 $|T| + n \le \phi$，则 *9* 将 P 追加到 T，并将 T 标记为未排序 *10* 更新 T 的包围盒 *11* 返回 T *12* 否则返回 *BuildSPaCTree(*$P \cup T$*)*13 $k \leftarrow$（T 的根节点中）枢轴对应的空间填充曲线编码14 $t \leftarrow$（基于编码）在 P 中进行二分搜索 k *15* 并行执行：16 $L \leftarrow$ *InsertSorted(*T_ℓ, $P[0, t)$*)*17 $R \leftarrow$ *InsertSorted(*T_r, $P[t, n)$ *) 18* 根据 L 和 R 的包围盒更新 T 的包围盒 *19* 返回 *Join(L,* T_p*, R)* // 返回一棵平衡树，将 L 和 R 与枢轴 k 连接起来。*20* 函数 *Join(L, k, R)* // 此函数与 [17, 23] 中相同 *21* 如果 L 较重，则返回 *RightJoin(L, k, R) 22* 如果 R 较重，则返回 *LeftJoin(L, k, R) 23* 返回 *Node(L, k, R)* // 递归检查 L 的右脊，直到子树大小与 R 达到平衡。创建一个树节点 R'，其子节点为两棵平衡子树，将 R' 附加到 L，并重新平衡 L。*24* 函数 *RightJoin(L, k, R)* // *LeftJoin* 与之对称 *25* 如果 L 且 R 平衡，则 // 划分在此终止 *26* 返回 *Node(L, k, R)* // 返回一棵平衡树 *27* $\langle L_\ell, k', L_r \rangle \leftarrow$ *Expose(L)* // 如果 L 是叶节点，则将其展开为一棵树 28 $R' \leftarrow$ *RightJoin(*L_r, k, R*)*// 递归划分 L 的右子树29 $L' \leftarrow$ *Node(*L_ℓ, k', R'*)* // 将新平衡的树 R' 附加到 L *30* 通过旋转重新平衡 L' *31* 返回 L'// 如果 T 是叶节点，则将其展开为一棵树，并在必要时重新排列各点。*32* 函数 *Expose(T) 33* 如果 T 是叶节点，则 *34* 如果 T 被标记为未排序，则重新排列各点 *35* 根据 T 中已排序的点构建一棵完美平衡树 T' *36* 返回 $\{T'_\ell, T'_P, T'_r\}$ *37* 否则返回 $\{T_\ell, T_p, T_r\}$ // 原样返回该树 *38* 函数 *Node(*T_ℓ, k, T_r*)* // 维护叶节点封装不变式。*39* 创建一个节点 T，其枢轴为 k，左子树为 T_ℓ，右子树为 T_r。 40 $n \leftarrow |T|$41 如果 $n > 2\phi$，则返回 T// 不适用叶节点封装 42 $n > \phi$ T_ℓ T_r 43 T_ℓ T_r 否则如果 则 // 重新分配叶节点中的点，并对 和 中标记为未排序的点进行排序。*44* 将 T 的子树重新分配到两个大小为 $n/2$ 的叶节点中。*45* 返回 T *46* 否则 // 树大小低于叶节点封装阈值，将其嵌入一个叶节点中 *47* 将 T 扁平化，并创建一个封装该树的叶节点。*48* 返回这个新的叶节点

正确性，证明其等价于 PaC 树。关于页限制，我们将分析推迟到第B节。

批量删除算法与插入类似。唯一的区别是，到达叶节点时，它会移除其中的点，必要时将该叶节点标记为未排序，并更新包围盒。叶节点封装的不变量以与插入时相同的方式保持不变，即第 23 和29行。

4.3 理论分析

由于篇幅限制，完整分析留待第 B 节，本文仅呈现结果。

定理 4.1。对于具有整数坐标的 n 个点，可以在 $O(n \log n)$ 工作量、$O(\log n)$ 跨度和 $O(Sort(n))$ 缓存复杂度下，构建带有希尔伯特曲线或 Z 曲线的 *SPaC* 树。对大小为 m 的批量更新（插入或删除）在大小为 n 的 *SPaC* 树上执行时，需要 $O(m \log n)$ 工作量和 $O(\log^2 n)$ 跨度。

5 实验

我们通过深入实验，研究 Ψ-Lib 以及其他空间索引在合成数据集和真实世界数据集上的性能。结果表明，P-Orth 树和 SPaC 树在构建和更新性能方面均表现优异，在大多数情况下优于 Pkd 树，并且比现有的 Orth 树和 R 树基线方法快得多。P-Orth 树和 SPaC 树的查询性能也与先前工作中的对应方法相当，或表现得更好。除了展示我们新算法的有效性之外，我们相信，实验还首次系统地研究了各种并行空间索引，包括 kd 树、Orth 树和 R 树。

配置。我们使用一台配备 112 个核心（224 个硬件线程）、四颗 Intel Xeon Platinum 8176 处理器和 1.47 TB 内存的机器。Ψ-Lib isinC++，并使用 GCC 14.2.1 和 -O3进行编译。我们使用 ParLaylib [15] 来实现分叉-汇合并行。我们的代码可在以下位置获取： [42]。报告的数值为预热运行后 3 次运行的平均值。关于参数选择的更多细节见第 C 节。

基线方法。我们将以下基线方法作为比较对象。

- **Pkd 树** [43]：最先进的并行 k树。
- **Zd 树** [16]：最先进的并行 Orth 树。Zd 树使用 Morton 编码对数据进行预排序，以帮助标准 Orth 树的构建和更新算法。来自 [16] 的原始代码在更新算法中存在已确认的错误（作者已确认）。我们根据他们的论文自行实现了该算法。我们仔细验证过，我们的构建时间与他们的代码相近。
- **CPAM** [23]：作为基线，我们使用 CPAM 库中的 PaC 树（将其视为黑盒）来存储每个点的 SFC 代码作为键。它基于莫顿曲线（CPAM-Z）或希尔伯特曲线（CPAM-H）为所有点保留全序。该基线展示了我们的新设计如何通过仅维护偏序来提升性能。
- **BoostR 树** [51]：Boost 库中的 R 树。Boost R 树是顺序式的，只支持点更新（不支持批量更新）。我们主要将其用作基线，以验证 SPaC 树的查询性能。因此，在所有变体中，我们采用二次版本，因为它在动态场景下具有最佳的树质量。

在 Ψ-Lib 中，我们测试了第 3 节介绍的并行 Orth 树——P-Orth 树，以及第 4 节详述的两种 R 树——SPaC-H 树和 SPaC-Z 树，它们分别使用希尔伯特曲线和莫顿曲线，对

		构建 time	构建后查询（50%） 10NN InD	10NN 分布外	范围计数	范围列表	增量插入 批次大小 10%	1%	0.1%	0.01%	增量插入后查询（50%） 10NN 分布内	10NN 分布外	范围计数	范围列表	增量删除 批次大小 10%	1%	0.1%	0.01%	增量删除后查询（50%） 10NN 分布内	10NN 分布外	范围计数	范围列表
mo_nu	★P-Orth	3.23	**.362**	**.363**	**.078**	1.15	4.27	10.3	19.7	29.7	**.381**	**.382**	**.080**	**1.04**	**4.32**	**10.2**	**19.6**	**29.9**	**.394**	**.397**	**.089**	1.18
	Zd 树	4.83	.490	.485	.142	1.59	7.80	13.2	24.9	48	.482	.486	.145	1.53	9.30	18.3	32.5	52	.510	.508	.137	1.50
	★SPaC-H	3.34	1.93	1.93	.157	1.17	3.90	6.00	13.0	26.8	1.96	1.96	.163	1.25	4.69	14.4	27.7	42.2	1.95	1.96	.161	1.24
	★SPaC-Z	**3.10**	9.13	9.20	.229	1.22	**3.68**	**5.75**	**12.6**	**26.2**	9.21	9.29	.237	1.31	4.78	14.2	27.4	42.0	9.12	9.20	.237	1.30
	CPAM-H	11.3	2.52	2.50	.216	1.39	15.3	38.2	108	159	2.44	2.44	.225	1.46	17.2	39.0	107	157	2.43	2.43	.225	1.46
	CPAM-Z	10.8	11.9	11.9	.311	1.46	14.7	37.6	108	158	11.6	11.7	.326	1.54	16.7	38.4	106	157	11.5	11.5	.325	1.52
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	11.1	11.1	1.49	7.81	N/A	N/A	N/A	N/A	.775	.783	.653	4.60
	Pkd 树	3.66	.398	.397	.097	**1.08**	4.52	10.64	20.9	48.9	.433	.416	.103	1.05	4.37	11.2	23.0	59.8	.411	.412	.104	**1.07**
en_pews	★P-Orth	4.63	**.220**	**.373**	.098	1.22	5.29	5.67	5.72	9.40	**.227**	**.333**	**.074**	**1.01**	**1.92**	3.08	4.25	8.74	**.222**	**.322**	**.073**	**1.02**
	Zd 树	5.37	.284	.388	.140	1.64	6.00	4.08	6.1	12.5	.285	.408	.127	1.47	6.00	4.73	11.9	29.0	.293	.415	.121	1.45
	★SPaC-H	3.32	.855	.661	.156	1.19	2.99	3.09	4.12	8.85	1.05	.797	.162	1.29	2.32	2.50	3.23	**8.37**	.941	.766	.159	1.26
	★SPaC-Z	**3.04**	1.94	.874	.214	1.23	**2.77**	**2.83**	**3.83**	**8.74**	1.74	1.16	.217	1.32	2.16	**2.27**	**3.01**	9.09	2.00	1.05	.215	1.30
	CPAM-H	10.3	1.16	.765	.216	1.43	13.5	10.2	8.29	19.4	1.19	.812	.227	1.48	15.0	11.1	8.67	21.2	1.19	.853	.222	1.48
	CPAM-Z	9.81	2.62	1.04	.292	1.47	13.1	9.96	8.04	20.4	2.50	1.21	.309	1.53	14.4	10.7	8.52	22.7	2.71	1.18	.300	1.54
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	.921	.931	3.18	6.10	N/A	N/A	N/A	N/A	.653	.651	.455	4.14
	Pkd 树	5.16	.243	.435	**.093**	**1.11**	13.7	24.0	35.8	39.9	.331	1.61	.108	1.09	10.0	24.3	36.8	62.8	.263	.337	.098	1.07
ned_av	★P-Orth	12.2	.155	.279	**.054**	1.12	13.7	11.5	12.6	26.2	.160	.247	**.050**	**1.01**	6.79	8.48	12.5	28.3	.160	.239	**.050**	**1.01**
	Zd 树	5.6	.192	**.156**	.086	1.57	5.95	4.24	6.1	14.6	.196	**.158**	.073	1.40	6.08	5.25	15.9	36.1	.198	**.157**	.072	1.39
	★SPaC-H	3.09	2.24	.565	.103	1.15	2.71	2.95	4.18	9.23	2.26	.494	.107	1.23	2.22	2.54	3.54	8.21	2.25	.578	.106	1.22
	★SPaC-Z	**2.94**	3.92	1.02	.165	1.19	**2.52**	**2.68**	**3.77**	**8.74**	3.61	.855	.171	1.27	**2.03**	**2.27**	**3.26**	**7.80**	4.30	2.29	.168	1.26
	CPAM-H	10.0	2.52	.663	.146	1.38	13.3	10.2	8.59	19.0	2.48	.592	.152	1.45	14.9	11.2	9.12	19.7	2.55	.680	.152	1.43
	CPAM-Z	9.66	4.62	1.22	.229	1.42	13.0	9.94	8.24	18.5	3.32	1.08	.239	1.49	14.0	10.8	8.75	20.2	5.08	2.86	.237	1.49
	Boost-R†	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	.922	.924	.521	4.45	N/A	N/A	N/A	N/A	.617	.624	.429	4.20
	Pkd 树	6.10	**.110**	.632	.060	**1.07**	12.8	25.2	32.9	53.6	**.109**	.725	.064	1.03	9.36	18.9	28.8	51.6	**.112**	.804	.063	1.04

最快时间以粗体和下划线标示　　：在最快速度的 1.1 倍以内　　：在最快时间的 2 倍以内　　：在最快时间的 5 倍以内　　：最快时间的 >5 倍

图 3. 合成数据上的运行时间（秒）。越低越好。 每项测试中的最快时间以粗体和下划线标示。我们使用颜色标记结果在 1 倍以内的情况。1×， 2×， 5×，以及 > 5× 最快时间。构建、查询和增量插入/删除的详细设置见第 5.1 节开头。 分布内/分布外：分布内/分布外。 †: Boost R 树是顺序式的，且仅支持点更新。因此，我们省略构建/更新时间，并报告逐个增量插入/删除点后的查询时间。

SPaC 树进行构建。我们为所有测试的索引维护包围盒。关于更多实现细节，请参阅第 C 节。

5.1 合成数据集上的总体评估

配置。 我们测试合成数据中点、查询和更新模式的不同分布。 $[0, 10^9]$中的所有坐标均为 64 位整数。我们使用三种工作负载：均匀分布、Sweepline 和 Varden。均匀分布会从空间中均匀随机地抽取每个点。Sweepline 同样使用均匀分布的数据，但会沿第一维对所有点进行排序。这样可以模拟一种偏斜的更新模式，其中更新点呈现空间局部性。Varden [27] 通过在空间中进行随机游走生成，并以较低概率重新从随机位置开始。点会形成聚类，且不同聚类之间相距较远，从而模拟偏斜的点分布。

我们测试静态和动态情形。除了直接测量树构建时间外，我们还使用具有不同批大小的增量插入/删除工作负载来模拟高度动态的场景。对于批次大小 b，增量插入工作负载意味着通过 n/b 次批量插入逐步构建索引；删除则相反（分 n/b 个批次删除索引）。我们报告所有操作的总运行时间。这反映了更新效率如何受到不断演化的数据集的影响。在此工作负载下，我们还在完成一半批次后测量查询时间。查询性能反映了大规模更新后各索引的质量受到的影响。对于静态设置，我们还提供查询时间

在使用一半数据构建树后，以便与动态场景进行比较。我们还测试单个批次的更新时间，并在第D节中展示结果。

我们测试了 k-近邻和范围查询（见第 2节介绍）。我们对分布内（ 10^7 10-近邻查询和分布外（InD）OOD）查询均运行了。对于范围查询，我们测试了5×10^4 范围计数和范围列表查询，范围大小为10^4–10^6。不同查询并行运行。除图 3 外，我们还进一步研究了 k-近邻和范围列表性能如何随其输出大小变化，并在图 4 和 5中展示结果。

我们在图 3 中总结了所有受测索引在合成数据集上的性能，数据集包含 10^9 2维点。第 E节提供了三维点上的结果。接下来，我们将详细分析性能。

5.1.1 构建。 对于树构建，我们的SPaC 树在所有工作负载下都是所有索引中最快的。这一优势源于将二维数据嵌入一维，从而简化了计算，以及第 Ψ节介绍的4中的各种优化。SPaC-Z 树 略快于 SPaC-H 树，因为 Morton 编码的计算比希尔伯特码更简单。基线方法 CPAM-H 和CPAM-Z 比我们的3× 慢SPaC 树，这是由于维护叶节点中排序的开销。在批量更新和查询中，这种影响更加显著。这证明了我们放宽叶节点排序这一技术的必要性。

对于 Orth 树，在均匀分布和 Sweepline 数据集上，P-Orth 树同样表现良好（比最快的 SPaC-Z 树慢不超过 52%），并且快于所有其他基线方法。P-Orth 树优于 Pkd 树的优势来自两个

方面：1）作为四叉树，P-Orth 树比二叉 *k*d 树的树高更低、局部性更好；2）在 P-Orth 树的每个节点上确定划分点（计算坐标范围的中点），比 *k*d 树更简单（后者需要估算所有点的中位数）。

在 Varden 上，P-Orth 树的速度比其他方法慢。由于 Orth 树使用坐标中位数来划分空间，因此天生不适合处理偏斜数据，并且受偏斜分布的影响最大。尽管 Zd 树同样是一种 Orth 树，但其性能仍较为合理——Zd 树构建的主要开销是按 Morton 顺序对所有点进行排序，而在我们的实现中，这一步采用比较排序完成。所有其他索引都是基于比较的，因此偏斜数据对其构建时间的影响很小。

总而言之，在构建方面，SPaC 树的性能始终优于所有其他基线方法。P-Orth 树在非偏斜数据上同样具有竞争力，但在偏斜数据上表现出劣势。

5.1.2 增量批量更新。 批量更新的结论与构建的结论非常相似。SPaC- 树的总体性能最佳，且 SPaC-Z 树略优于 SPaC-H 树。SPaC-Z 树 在所有增量插入中速度最快，并且在大多数增量删除情况下也是如此。基于第 5.1.1节中分析的相同原因，P-Orth 树对于 Varden 数据而言并不理想。在其他所有情况下，P-Orth 树要么表现最佳，要么接近最佳。

对于所有索引，批次大小越小，增量更新所需时间就越长。一方面，批次越小，并行性提升的空间就越有限。另一方面，批次数量越多，也意味着对树的修改越多，需要投入更多精力重新平衡树，使树离完全平衡的状态更远。唯一无需重新平衡的索引是 Orth 树，因此其连续更新性能受批次大小的影响最小。

对于高度动态的数据，与 P-Orth 树和 SPaC 树相比，Pkd 树在更新时间方面竞争力较弱。一个关键原因是，Pkd 树每个更新点的摊销成本为 $O(\log^2 n)$，而 P-Orth 树和 SPaC 树的成本分别为 $O(\log \Delta)$ 和 $O(\log n)$，其中 Δ 是纵横比。因此，P-Orth 树和 SPaC 树的更新速度都快于 Pkd 树。具体而言，P-Orth 树的增量更新速度最高可比 Pkd 树快 7.18× ，SPaC 树最高可快 7.5× 。即使对于 Δ 相对较大的 Varden，P-Orth 树的增量更新速度也几乎总是快于 Pkd 树。

5.1.3 查询。 我们在三种情况下运行查询：1）构建大小为 5×10^8 的树之后；2）应用 50% 的插入批次之后；3）应用 50% 的删除批次之后。大多数索引在构建后几乎完全平衡，因此第一种情况反映了它们在最佳情况下的（静态）查询性能。后两种情况则反映了更新对索引质量的影响。在图 3中，我们仅选取了 10 近邻查询和相对较大的范围查询结果

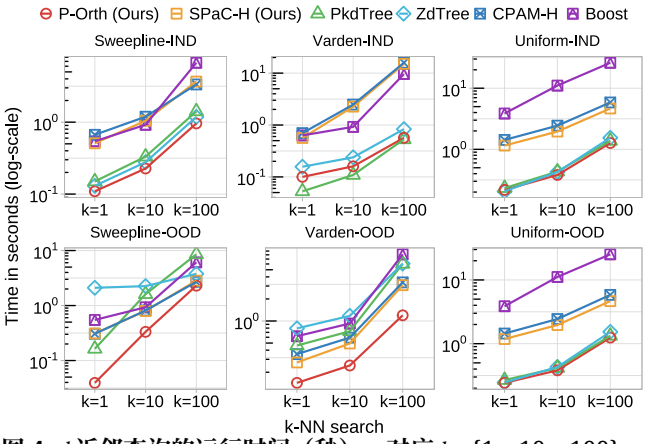


图 4. *k*近邻查询的运行时间（秒），对应 $k \in\{1、10、100\}$。越低越好。数据集包含 5 亿个二维点。树通过增量插入构建，批次比例为 0.01%。测试包含来自分布内和域外分布的 10^7 个点所发起的 k 近邻查询。图表采用双对数刻度。

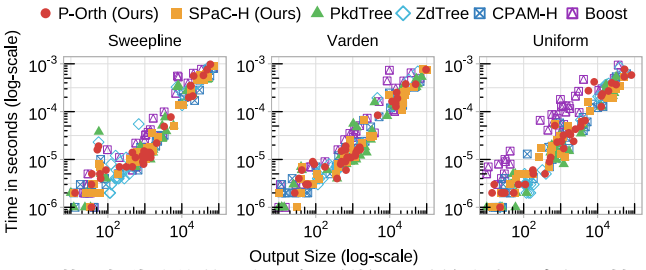


图 5. 范围报告查询的运行时间（秒），随输出大小变化。越低越好。数据集包含 5 亿个二维点。树通过增量插入构建，批次比例为 0.01%。图表采用双对数刻度。

。为提供更多细节，在图 4 和图 5中，我们进一步展示查询性能如何随输出大小（即 k 个 k近邻）以及范围列表查询中的范围大小而变化。

***k*近邻查询。**如图4所示，在 k近邻查询中，空间划分树明显快于 R 树。这是因为 R 树的包围盒存在重叠。对于SPaC 树，尽管SPaC-H 树在构建与更新方面略慢于 SPaC-Z 树，但查询效率高得多。这是因为希尔伯特曲线比莫顿曲线具有更好的局部性（相邻编码在几何位置上总是彼此接近）。在 R 树中，SPaC 树在所有查询中的性能均与 Boost R 树相当或更好，SPaC-H 树的速度慢 3.7×倍至快 5.66× 倍，几何平均数为快 2.5× 倍。

在空间划分树中，Orth 树的整体性能最佳。这是因为访问子树时，P-Orth 树可以从 4 个象限中选择 1 个，这比 Pkd 树和 Zd 树从 2 个半空间中选择 1 个更有效。因此，Pkd 树和 Zd 树具有竞争力，但通常比 P-Orth 树慢。唯一的例外是 Varden 数据。对于 InD 查询，由于 Varden 的分布偏斜，Orth 树可能不平衡，因此基于比较的 Pkd 树表现

	构建	更新 Ins.	更新 删除	查询 10NN	查询 RG	构建	更新 Ins.	更新 删除	查询 10NN	查询 RG
	Cosmo（3D），n=317M					OSM（二维），n=776M				
★P-Orth	1.90	16.2	17.9	.120	**.566**	4.96	14.5	14.9	.083	.050
Zd 树	1.65	13.8	20.7	.146	.862	5.88	16.5	23.9	.182	.055
★SPaC-H	1.02	6.59	9.40	.393	.764	2.26	8.19	7.98	.981	.085
★SPaC-Z	**.837**	**5.63**	**8.46**	2.58	.980	**2.12**	**7.91**	**7.37**	2.91	.132
CPAM-H	5.48	19.6	19.8	.509	1.04	7.26	15.6	16.7	1.10	.118
CPAM-Z	5.38	19.0	19.2	4.39	1.30	7.01	15.4	16.6	3.47	.182
Boost-R	N/A	N/A	N/A	.274	.977	N/A	N/A	N/A	.484	.435
Pkd 树	1.89	101	800	**.107**	.602	4.32	29.3	26.3	**.071**	**.049**

：<1。快 1 倍 **最快时间**：2 倍最快 <5 倍 快：>5 倍最快

图6. 真实世界数据集上的运行时间（秒）。越低越好。插入/删除：增量插入/删除，批次大小为 0.01%。“RG”：范围列表查询。

越好。有趣的是，与此相反，两种 Orth 树在 Varden 的分布外查询上都表现出优势。原因仍是不平衡——对于 Varden，点高度聚集，使得树中的这些区域很深，而其他区域较浅。由于分布外查询的分布与输入不同，它们很可能命中较浅的区域，因此速度快得多。

范围查询。如图3和5所示，Pkd 树在范围查询上表现出微小但稳定的优势。这是因为范围查询会访问所有与查询框重叠的子树。在这种情况下，P-Orth 树必须显式检查四个子树的边界框，而对Pkd 树节点进行的每次不重叠检查，都可以剪去该子树中一半的点。对于其他索引，范围查询上的相对性能与 k-NN 查询类似。有趣的是，尽管SPaC 树在范围列表查询上仍慢于 kd 树和P-Orth 树，但差距小得多，尤其是在范围较大时——此时查询时间主要花在输出结果列表上，掩盖了不同索引之间剪枝效果的差异。因此，与 k-NN 查询相比，范围查询对索引类型的敏感度较低。

更新对查询的影响。在动态场景下，Orth 树（P-Orth 树和 Zd 树）与历史无关（叶节点包装除外），也就是说，树的最终状态不受操作顺序影响。因此，它们的查询性能受批量更新的影响最小，在动态场景下表现最佳。

对于所有其他索引，树在更新后可能会变得不那么平衡。事实上，与静态设置相比，它们的速度都有所下降。这种影响对大多数索引来说都较为有限（通常在 20% 以内）。例外情况均出现在分布外 k-NN 查询中：在 Sweepline 上进行增量插入后，Pkd 树的速度下降了 3.7×；在 Varden 上进行增量删除后，CPAM-Z 和 SPaC-Z 树的速度下降了约 2.5×。

总而言之，对于查询而言，Orth 树和 kd 树天然优于 R 树。kd 树在非均匀数据上处理 InD 查询时表现更好，但在分布外查询中可能较差。P-Orth 树在几乎所有查询和工作负载中的查询性能都是最佳或接近最佳。

5.2 真实世界数据集上的操作

对于真实世界数据集，我们测试高度聚类的数据集 COSMO [53] 以及北美洲的 OpenStreetMap（OSM [33]）。我们测试 10^7 分布内 10-NN 查询，以及 5×10^4 范围

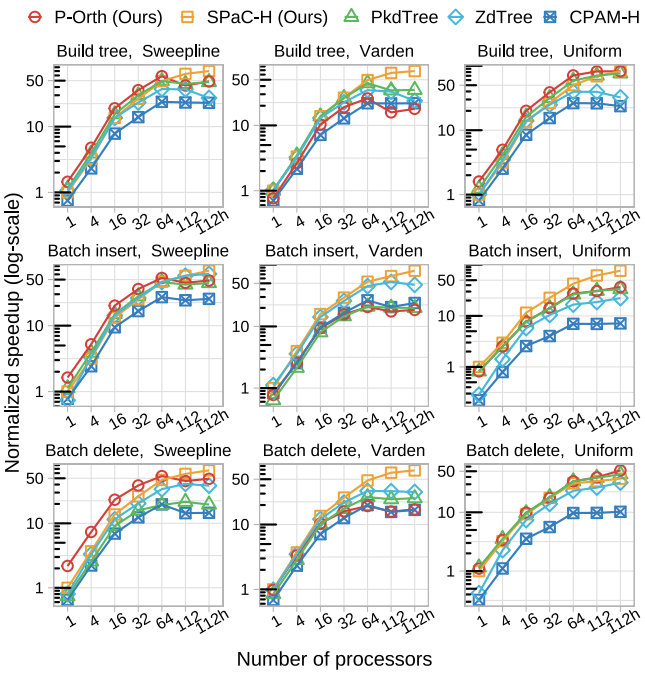


图 7. 在不同处理器数量下，树构建、批量插入和批量删除的归一化并行加速比。越高越好。加速比分别以单线程下 SPaC 树的运行时间为基准进行归一化。数据集包含 1000M 个二维点。“批量插入”是指向包含 1000M 个点的树中再插入 1% 的点，“批量删除”是指从树中删除 1% 的点。两者均在单个批次中完成。“112h”表示使用全部 112 个核心（224 个超线程）。由于 Boost 是串行的，因此没有其结果。

列表查询，范围大小为 10^4–10^6。坐标向下取整为 64 位整数。我们移除重复项，并将所有点平移至正坐标。为确保空间填充曲线在三维空间中正常工作，我们将坐标缩放至 $[0, 10^6]$。我们评估构建、批次比例为 0.01% 的增量更新以及构建后的查询，并在图 6中展示结果。

SPaC 树在构建和更新方面远快于其他方法。在真实世界数据上，这一优势比在合成数据上更为显著。具体而言，SPaC 树的构建速度约为 Pkd 树的2× 倍，更新速度则快 3.5–94× 倍。P-Orth 树的构建时间与 Pkd 树相近，但更新速度快得多（1.8–44.7× 倍）。

在查询方面，R 树的表现仍逊于空间划分树。在大多数情况下，Pkd 树的查询性能最佳，但 P-Orth 树始终具有竞争力——在所有情况下，差距都在 20% 以内。考虑到 P-Orth 树的更新速度快 1.8–44.7×，与 Pkd 树相比，P-Orth 树在查询/更新权衡方面表现出色得多。

5.3 可扩展性

我们评估了所测试索引在构建、插入和删除方面的可扩展性，如图 7 所示。我们使用 10^9 2 个点。批量插入使用一个大小为 10^7 的批次。图 7 中的数据点展示了相对于 SPaC-H 树 单核心性能的加速比，因此图 7

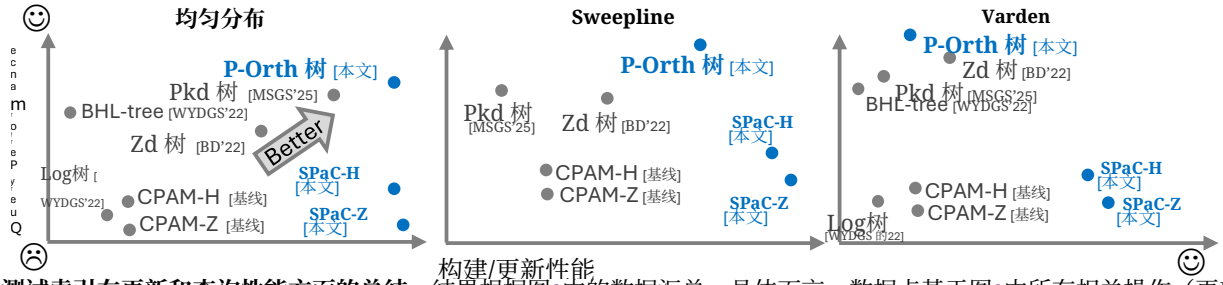


图 8. 测试索引在更新和查询性能方面的总结。结果根据图3中的数据汇总。具体而言，数据点基于图3中所有相关操作（更新或查询）的几何平均数。Log树 和 BHL-tree [62] 的数据点根据Pkd 树 论文 [43]估算得出。我们提出的新算法以蓝色标示。请注意，该图仅展示了本文所测试基准测试的平均结果。更全面的结论见表2。

也反映了各索引之间真实的效率对比（越高越好）。总体而言，所有索引都能很好地扩展至 224 个超线程，其差异主要来自工作量（即单核心性能）。其中，SPaC-H 树的相对加速比最高，构建时最高可达 82.9×，插入时可达 80×。这可能是因为它作为一维搜索树，结构简单。结合较低的工作量和良好的可扩展性，SPaC-H 树在构建和更新性能方面总体最佳。批量删除的可扩展性与批量插入相似；在 112 个核心上，SPaC-H 树的可扩展性最佳，在 Sweepline 上为 67×，在 Varden 上为 37.4×，在均匀分布上为 68.4×。P-Orth 树在均匀分布数据上的可扩展性良好，但在 Sweepline 和 Varden 上稍差，这是由于树不平衡。

5.4 总结

综合所有实验结果，我们在图8 中直观展示了所有受测索引在更新性能与查询性能之间的权衡，并在此简要总结。更详细的说明见表2。

Pkd 树。 Pkd 树在查询方面表现稳定，但处理分布外查询时性能可能下降。其更新性能尚可，但不如 P-Orth 树和 SPaC- 树有竞争力。我们的结果表明，它们最适用于更新频率较低至中等、对查询吞吐量要求较高且查询主要来自分布内的场景。

P-Orth 树（本文）。 P-Orth 树通常在整体性能以及查询与更新之间的权衡方面表现最佳，尤其适用于非偏斜数据。它最适合偏斜程度较低的数据场景，且适用于任意更新查询比。它也适合在大量更新后进行查询，因为树质量不会因频繁更新而下降。

SPaC 树（本文）。 SPaC-H 树的更新性能略逊于 SPaC-Z 树，但查询性能显著更好。我们建议将 SPaC-H 树设为 SPaC 树的默认配置。与 Pkd 树和 P-Orth 树相比，SPaC-H 树的查询性能较弱，但构建和更新速度显著更快。总体而言，SPaC-H 树最适用于高度动态的场景：要么更新需要极高的吞吐量和极低的延迟，要么更新的频率远高于查询。

6 结论

本文系统地研究了并行空间索引，特别关注如何在高度动态的工作负载下实现高性能更新。我们提出了两种新的数据结构：并行 Orth 树，即P-Orth 树；以及并行 R 树，即SPaC 树系列。与现有并行空间索引相比，这两种结构的更新性能都更出色，同时在查询方面仍具有竞争力，或优于文献中的同类结构。我们还介绍了全面的实验，以了解现有并行空间索引和我们新提出的并行空间索引的性能，并在第5.4 节和图8中分享了研究发现。

7 致谢

本研究得到美国国家科学基金会（NSF）资助，项目编号为 CCF-2103483、TI-2346223 和 IIS-2227669；同时得到美国国家科学基金会 CAREER 奖资助，项目编号为 CCF-2238358 和 CCF-2339310，以及 Google Research 学者计划的支持。

参考文献

[1] Daniar Achakeev、Marc Seidemann、Markus Schmidt 和 Bernhard Seeger。2012 年。基于排序的 R 树并行加载。发表于 *BigSpatial@ACM 空间信息特别兴趣小组（SIGSPATIAL）*。62–70 页。[2] Stephen Adams。1992 年。在函数式语言中高效实现集合。技术报告 CSTR 92-10。南安普顿大学。[3] Stephen Adams。1993 年。高效集合——一种平衡之道。《函数式编程》3，04（1993），553–561 页。[4] Tomas Akenine-Möller、Eric Haines 和 Naty Hoffman。2019 年。实时渲染。CRC 出版社。[5] Chuan-Heng Ang 和 Tuck-Choy Tan。1997 年。用于 R 树的新型线性节点分裂算法。发表于国际空间数据库研讨会（*SSD*）。施普林格，337–349 页。[6] Lars Arge、Mark De Berg、Herman Haverkort 和 Ke Yi。2008 年。优先级 R 树：一种实用高效且最坏情况最优的 R 树。*ACM 算法汇刊（TALG）*4，1（2008），1–30 页。[7] Lars Arge、Klaus H Hinrichs、Jan Vahrenhold 和 Jeffrey Scott Vitter。2002 年。动态 R 树的高效批量操作。*Algorithmica* 33，1（2002），104–128 页。[8] Nimar S Arora、Robert D Blumofe 和 C Greg Plaxton。2001 年。多道程序多处理器的线程调度。《计算系统理论》（*TOCS*）34，2（2001），115–144 页。[9] Michael Axtmann、Sascha Witt、Daniel Ferizovic 和 Peter Sanders。2017 年。原地并行超标量样本排序（ipsssso）。发表于欧洲算法研讨会（*ESA*）。[10] Norbert Beckmann、Hans-Peter Kriegel、Ralf Schneider 和 Bernhard Seeger。1990 年。R* 树：一种面向点和矩形的高效稳健访问方法。发表于*ACM SIGMOD* 国际数据管理会议（*SIGMOD*）。322–331 页。

[11] Jon Louis Bentley。1975 年。用于关联搜索的多维二叉搜索树。ACM 通讯 18，9（1975），509–517 页。[12] Jon Louis Bentley。1978 年。可分解搜索问题。技术报告。卡内基梅隆大学。[13] Stefan Berchtold、Daniel A. Keim 和 Hans-Peter Kriegel。1996 年。X 树：一种高维数据索引结构。发表于 VLDB 基金会会议录（PVLDB）。Morgan Kaufmann，28–39 页。[14] Jiři Bittner、Michal Hapala 和 Vlastimil Havran。2015 年。用于光线追踪的增量式包围体层次结构构建。《计算机与图形学》47（2015），135–144 页。[15] Guy E. Blelloch、Daniel Anderson 和 Laxman Dhulipala。2020 年。ParlayLib——用于共享内存多核机器上并行算法的工具包。发表于 ACM 算法与体系结构并行性研讨会（SPAA）。507–509 页。[16] Guy E Blelloch 和 Magdalen Dobson。2022 年。低维空间中的并行最近邻与批量更新。发表于算法工程与实验（ALENEX）。工业与应用数学学会，195–208 页。[17] Guy E. Blelloch、Daniel Ferizovic 和 Yihan Sun。2016 年。用于并行有序集合的 Just Join。发表于 ACM 算法与体系结构并行性研讨会（SPAA）。[18] Guy E. Blelloch、Jeremy T. Fineman、Yan Gu 和 Yihan Sun。2020 年。二叉分叉模型中的最优并行算法。发表于 ACM 算法与体系结构并行性研讨会（SPAA）。89–102 页。[19] Guy E. Blelloch、Phillip B. Gibbons 和 Harsha Vardhan Simhadri。2010 年。低深度缓存无关算法。发表于 ACM 算法与体系结构并行性研讨会（SPAA）。[20] Guy E. Blelloch 和 Yan Gu。2020 年。用于动态规划的改进型并行缓存无关算法。发表于工业与应用数学学会计算机系统算法原理研讨会（APOCS）。[21] Robert D. Blumofe 和 Charles E. Leiserson。1998 年。多线程计算的空间高效调度。《工业与应用数学学会计算杂志》27，1，1998（）。[22] J. K. Chen、Yin-Fu Huang 和 Yeh-Hao Chin。1997 年。R 树并发操作研究。《信息科学》98，1-4（1997），263–300 页。[23] Laxman Dhulipala、Guy E. Blelloch、Yan Gu 和 Yihan Sun。2022 年。PaC 树：支持并行压缩纯函数式集合。发表于 ACM 编程语言设计与实现大会（PLDI）。[24] Andreas Fabri 和 Sylvain Pion。2009 年。CGAL：计算几何算法库。发表于 ACM SIGSPATIAL 地理信息系统进展国际会议。538–539 页。[25] Raphael A Finkel 和 J on Louis Bentley。1974 年。四叉树：一种用于复合键检索的数据结构。《信息学学报》4（1974），1–9 页。[26] Matteo Frigo、Charles E. Leiserson、Harald Prokop 和 Sridhar Ramachandran。1999 年。缓存无关算法。发表于 IEEE 计算机科学基础研讨会（FOCS）。[27] Junhao Gan 和 Yufei Tao。2017 年。欧几里得 DBSCAN 的困难性与近似。《ACM 数据库系统汇刊》（TODS）42，3（2017），1–45 页。[28] Yván J García R、Mario A López 和 Scott T Leutenegger。1998 年。用于批量加载 R 树的贪心算法。发表于 ACM 地理信息系统进展国际研讨会（SIGSPATIAL GIS）。163–164 页。[29] Yan Gu、Yong He 和 Guy E Blelloch。2015 年。包围体层次结构上的光线专用收缩。发表于《计算机图形学论坛》，第 34 卷。309–318 页。[30] Yan Gu、Yong He、Kayvon Fatahalian 和 Guy Blelloch。2013 年。通过近似凝聚聚类高效构建包围体层次结构。发表于高性能计算机图形学（HPG）。[31] Yan Gu、Zachary Napier 和 Yihan Sun。2022 年。工作窃取与并行缓存复杂度分析。发表于工业与应用数学学会计算机系统算法原理研讨会（APOCS）。工业与应用数学学会，46–60 页。[32] Antonin Guttman。1984 年。R 树：用于空间搜索的动态索引结构。发表于 ACM SIGMOD 国际数据管理会议

数据（SIGMOD）。47–57。[33] Mordechai Haklay 和 Patrick Weber。2008。OpenStreetMap：用户生成的街道地图。IEEE《普适计算》7, 4（2008），12–18。[34] Herman Haverkort 和 Freek V Walderveen。2008。用于 R 树的四维希尔伯特曲线。《实验算法学期刊》（JEA）16（2008），3–1。[35] Chris L Jackins 和 Steven L Tanimoto。1980。八叉树及其在表示三维对象中的应用。《计算机图形学与图像处理》（CGIP）14, 3（1980），249–270。[36] Ibrahim Kamel 和 Christos Faloutsos。1992。并行R树。ACM SIGMOD 国际数据管理会议（SIGMOD）21, 2（1992），195–204。[37] Ibrahim Kamel 和 Christos Faloutsos。1993。关于R树打包。载于信息与知识管理国际会议（CIKM）。490–499。[38] Jinha Kim、Seung-Keol Kim 和 Hwanjo Yu。2013。大型社交网络中影响力最大化的可扩展并行处理？载于2013年IEEE第29届数据工程国际会议（ICDE）。IEEE，266–277。[39] Christian Lauterbach、Michael Garland、Shubhabrata Sengupta、David Luebke 和 Dinesh Manocha。2009。在图形处理器上快速构建包围体层次结构。载于《计算机图形学论坛》，第28卷。Wiley Online Library，375–384。[40] Scott T Leutenegger、Mario A Lopez 和 Jeffrey Edgington。1997。STR：一种简单高效的R树打包算法。载于IEEE数据工程国际会议（ICDE）。IEEE，497–506。[41] Lijuan Luo、Martin DF Wong 和 Lance Leong。2012。GPU上的R树并行实现。载于第17届亚洲及南太平洋设计自动化会议。IEEE，353–358。[42] Ziyang Men、Bo Huang、Yan Gu 和 Yihan Sun。2025。并行动态空间索引。https://github.com/ucrparlay/SpaceTreeLib。[43] Ziyang Men、Zheqi Shen、Yan Gu 和 Yihan Sun。2025。带批量更新的并行kd树。ACM SIGMOD 国际数据管理会议（SIGMOD）3, 1（2025），1–26。[44] Vincent Ng 和 Tiko Kameda。1994。R-Link树：一种可恢复的空间数据索引结构。载于数据库与专家系统应用（DEXA），第856卷。施普林格，163–172。[45] Yutaka Ohsawa 和 Masao Sakauchi。1990。一种新型树形数据结构，具有适用于超大型空间数据库的同质节点。载于IEEE数据工程国际会议（ICDE）。IEEE计算机学会，296–297。[46] Jacopo Pantaleoni 和 David Luebke。2010。分层LBVH：用于动态几何体实时光线追踪的分层线性包围体层次结构构建。载于高性能图形学（HPG）。87–95。[47] Apostolos Papadopoulos 和 Yannis Manolopoulos。2003。空间数据的并行批量加载。《并行计算》29, 10（2003），1419–1444。[48] Sushil K Prasad、Michael McDermott、Xi He 和 Satish Puri。2015。基于GPU的并行R树构建与查询。载于IEEE并行与分布式处理国际研讨会（IPDPS）研讨会。IEEE，618–627。[49] Jianzhong Qi、Yufei Tao、Yanchuan Chang 和 Rui Zhang。2018。理论最优且经验高效、具有强并行性的R树。《VLDB基金会会刊》（PVLDB）11, 5（2018），621–634。[50] Jianzhong Qi、Yufei Tao、Yanchuan Chang 和 Rui Zhang。2020。使用空间填充曲线打包R树：理论最优性、经验效率与批量加载并行性。ACM 数据库系统汇刊（TODS）45, 3（2020），1–47。[51] Boris Schäling。2011。Boost C++ 库。Boris Schäling。[52] Bernd Schnitzer 和 Scott T. Leutenegger。1999。主从式R树：一种新型并行R树架构。载于SSDBM。IEEE计算机学会，68–77。[53] Nick Scoville、H Aussel、Marcella Brusa、Peter Capak、C Marcella Car-ollo、M Elvis、M Giavalisco、L Guzzo、G Hasinger、C Impey 等。2007。宇宙演化巡天（COSMOS）：概述。《天体物理学杂志增刊》172, 1（2007），1。

[54] Timos K. Sellis、NickRoussopoulos 和 Christos Faloutsos。1987。R+-树：一种用于多维对象的动态索引。载于*VLDB*基金会会刊（*PVLDB*）。Morgan Kaufmann，507–518。[55] Timos K. Sellis、NickRoussopoulos 和 Christos Faloutsos。1987。R+-树：一种用于多维对象的动态索引。载于*VLDB*基金会会刊（*PVLDB*）。Morgan Kaufmann，507–518。[56] 赖淑华、朱凤华和孙永强。2000。工作站集群上的并行R树设计。载于网络信息系统数据库国际研讨会。施普林格，119–133。[57] Yihan Sun、Daniel Ferizovic 和 Guy E Blelloch。2018。PAM：并行增强映射。载于*ACM*并行编程原理与实践研讨会（*PPOPP*）。[58] Herbert Tropf 和 Helmut Herzog。1981。动态平衡树中的多维范围搜索。*Angewandte Info.* 2（1981），71–77。[59] Timo Viitanen、Matias Koskela、Pekka Jääskeläinen、Aleksi Tervo 和 Jarmo Takala。2018。PLOCTree：一种快速、高质量的硬件BVH构建器。*ACM*计算机图形学与交互技术1，2（2018），1–19。[60] Ingo Wald、Thiago Ize 和 Steven G Parker。2008。用于光线追踪动画场景的快速、并行且异步的包围体层次结构构建。《计算机与图形学》32，1（2008），3–13。[61] Michael S Warren 和 John K Salmon。1993。一种并行哈希八叉树N体算法。载于*1993*年*ACM/IEEE*超级计算会议论文集。12–21。[62] Rahul Yesantharao、Yiqiu Wang、Laxman Dhulipala 和 Julian Shun。2021。并行批处理动态k-d树。*arXiv*预印本 *arXiv:2112.06188*（2021）。[63] Simin You、Jianting Zhang 和 Le Gruenwald。2013。使用R树在图形处理器上进行并行空间查询处理。载于*ACM SIGSPATIAL*大规模地理空间数据分析国际研讨会。23–31。

	一般特性	现有解决方案及其特性
***k*d-tree**	+ 线性空间 + 适用于大多数查询（例如 k近邻、范围查询） + 不重叠边界框（因此 可在查询中有效剪枝） + 适用于各种分布的查询通常都很快 + 基于比较，能够抵抗偏斜数据的影响 + 易于推广到三维以上 维度 – 更新缓慢且复杂	**Pkd 树 [43]** + 用于构建与更新的 I/O 优化 + 快速构建： $O(n\log n)$ 工作量和多对数跨度 + 在大多数测试中，查询速度名列前茅，但分布外查询除外，尤其是 偏斜分布 – O批次大小为 m 的批量更新需要 $(m\log^2 n)$ 工作量，不适合 更新频繁的工作负载 **BHL-tree 和 Log树 [62]** + 可利用 vEB布局进行查询优化（由于其静态特性） + 构建的工作量为 $O(n\log n)$，跨度为多对数级 – 批量更新成本高： $O(m\log^2 n)$（Log树）或 $O((n+m)\log(n+m))$ （BHL-tree，由于需要完全重建） – Log树采用对数方法，导致查询效率低下
Orth-tree	+ 线性空间 + 适用于大多数查询（例如 k近邻、范围） + 不重叠边界框（因此 能在查询中有效剪枝） + 查询速度快，尤其适用于非偏斜数据 + 与历史无关（叶包装除外） + 简单/快速的构建与更新， 尤其适用于非偏斜数据 – 对偏斜数据敏感 – 通常无法推广到三维以上 维度	**P-Orth 树（本文）** + 用于构建与更新的 I/O 优化 ⋆ 在非偏斜数据上具有最快的查询性能 ⋆ 更新通常比 Pkd 树更快，即使数据存在相当程度的偏斜； 比 SPaC 树慢 + 快速构建： $O(n\log\Delta)$ 工作量和多对数跨度 ⋆ 快速批量更新： $O(m\log\Delta)$ 工作量和多对数跨度 – 在构建、更新和查询中最容易受到偏斜数据的影响；效率较低 在偏斜数据上执行分布内查询的效率较低 **Zd 树 [16]** + 由于采用比较排序，相对不易受偏斜影响 + 快速构建： $O(n\log n)$ 工作量，多对数跨度 + O批量更新需要 $m(\Delta\log\Delta)$ 工作量，其中 Δ 是纵横比 – 更新/构建通常比 P-Orth 树慢 – 仅支持整数坐标和莫顿曲线
R-tree/BVH	+ 线性空间 + 由于采用对象划分，规则灵活 + 构建与更新简单快速 + 适用于常见查询（例如，k近邻， 范围） + 很容易推广到三个以上 维度 – 包围盒重叠（因此 查询中的剪枝效果不佳）；查询通常 比空间划分树更慢	**SPaC 树（本文）** ⋆ 兼容希尔伯特曲线、莫顿曲线及其他空间填充曲线 ⋆ 将多维数据嵌入一维，从而简化算法设计并实现 高度并行（在测试的索引中自身加速比最高） + 快速构建： $O(n\log n)$ 工作量和多对数跨度 ⋆ 超快速批量更新： $O(m\log n)$ 工作量和多对数跨度 + 基于比较；对偏斜数据具有鲁棒性 ⋆ 在所有基线方法中，构建和更新速度最快；具有显著优势 在更新方面 – 仅支持整数坐标 – 由于包围盒重叠，查询速度比空间划分树慢 框彩在线 **Boost R 树 [51]** + 支持多种启发式方法 – 不支持并行构建或批量更新 – 由于包围盒重叠，查询速度比空间划分树慢 盒
Range tree	+ 范围查询的最坏情况工作量界 – $O(n\log n)$ 空间 – 仅支持范围查询 – 在超过两个维度时效率低下	**CPAM/PAM 范围树 [23, 57]** + 并行构建，工作量为 $O(n\log n)$，跨度为多对数 – 不支持简单的并行批量更新

表 2. 不同空间树及现有并行构建、更新和查询解决方案的主要特征汇总。符号“⋆”标示我们的关键技术贡献。在界限中， m 表示批次大小， n 表示索引大小， Δ 表示纵横比。

关于 P-OrthTrees 的分析

现在，我们分析 P-Orth 树构建和批量更新算法的理论保证。设 $S \subseteq M$ 为有界欧几里得空间 M 中的有限点集，并将 $B_p(r) \subseteq S$ 表示为以 p 为中心、半径为 r 的球所包含的点集。那么，当且仅当 $\forall p \in M$ 且 $r > 0$ 时，S 具有（ρ，c）–扩展率：

$$|B_p(r)| \geq \rho \implies |B_p(2r)| \leq c \cdot |B_p(r)| \tag{1}$$

常数 c 称为扩展率，而 ρ 通常设为 $O(\log |S|)$。若 $c = O(1)$，则称扩展率较低。直观而言，低扩展性质确保点在空间中均匀分布。

类似地，纵横比 Δ 定义为：

$$\Delta = \frac{\max d(x,y)}{\min d(x,y)} \quad \forall x, y \in S \tag{2}$$

若对于某个常数 $c > 0$，$\Delta < n^c$ 成立，则称其有界。

现在我们将证明，在纵横比有界的假设下，P-Orth 树。若不作这些假设，树高将变为 $O(\log \Delta)$。我们可以在以下分析中替换树高，从而得到定理3.1。

引理A.1。 </style}点集上大小为 的 P-Orth 树的高度为 ($\log n$)，假设 P具有低扩展率和有界长宽比。

证明。 根据低扩展率，H 的边长至多为 d_{max} 的某个常数比例；当距离为 d_{min} 的两个点被分开时，递归停止。由于有界长宽比，$d_{max}/d_{min} = n^c$，且划分器在空间中位数处对 H进行划分，因此需要 $O(\log n)$ 层划分器，才能将 H 的边长缩减至 d_{min}。证毕。□

利用上述引理，我们现在证明，我们的 Orth 树构造算法在 n 个点上具有 $O(n \log n)$ 工作量、多对数跨度，以及 $O(\text{Sort}(n))$ 缓存复杂度。这里假设缓存大小为 $M = \Omega(\text{polylog}(n))$，如 [19, 20], 所述，令骨架高度 $\lambda = \epsilon \log(M)$，其中 $\epsilon < 1/(2D)$，并在筛分算法中令块大小为 $l = 2^{D \cdot \lambda}$。记$O(\text{Sort}(n)) = O(n/B \cdot \log_M n)$为对 [19]进行排序的最优缓存复杂度。

定理A.2。 在上述参数设置下，假设 P具有低扩展率和有界纵横比，则可以在点集 P上构建一棵大小为 n 的 *P-Orth* 树，其工作量为 $O(n \log n)$，跨度为$O(\log^2 n)$，缓存复杂度为 $O(Sort(n))$。

证明。 除筛分过程中的点外，每个点在每轮中最多处理一次；对于筛分过程中的点，查找一个点所属的桶需要 $O(\lambda \cdot D)$的工作量。算法经过 $O(\log n)/\lambda$ 轮递归后终止，因此每个点的总工作量为 $O(\lambda \cdot D) \cdot O(\log n)/\lambda = O(\log n)$。因此，总工作量为 $O(n \log n)$。

对于跨度而言，实际上树骨架的构建以及每个块的处理都是顺序地完成的。然而从理论上讲，它们可以分别在 $O(\lambda \log n)$ 和 $O(\log n)$ 中实现并行化跨度。所有其他操作需要 $O(\log n)$ 的跨度。总的来说，每轮的跨度为 $O(\lambda \log n)$。该算法具有 $O(\log n)/\lambda$ 轮递归，因此总体跨度为 $O(\log^2 n)$。

现在考虑缓存复杂度。构建树骨架和计算子区域都完全可以放入缓存。块大小为 $l = 2^{\lambda \cdot D} = \underline{M}^{\epsilon \cdot D} \leq \sqrt{M}$，这意味着每个块都能完全放入缓存。因此，筛分算法需要 $O(n/B)$次块传输。所有其他操作总共需要 $O(n/B)$次块传输，即 $O(n/B \cdot \log n/\lambda) = O(n/B \cdot \log_M n)$次输入输出操作。□

对于批量更新，我们假设批次大小为 $m = O(n)$；如果 $m = \omega(n)$，则在下文的插入界中，只需将 n 替换为 $m + n$，而删除的界不变。

定理 A.3. 对大小为 n的 P-Orth 树进行大小为 $m = O(n)$的更新（插入或删除），在假设更新点具有低膨胀率和有界长宽比的情况下，可在最优的 $O(m \log n)$ 工作量、$O(\log^2 n)$ 跨度以及 $O(m(\log(n/m) + (1 + \log_M m)/B))$ 缓存复杂度内完成。证明。以插入为例，删除的情况类似。对于工作量，注意更新后的树与从头开始在所有点上构建的树相同，其高度根据引理 A.1 为$O(\log(m+n))$。高度差为 $O(\log(m+n)) - O(\log n) = O(1)$。由于每个叶包装有 $O(1)$ 个点，且每个点都需要 $O(\log n)$ 的工作量才能到达叶节点，因此总工作量为 $O(m \log n)$。

跨度的分析与定理中的构建分析相同。A.2。

更新的缓存界分为两部分。第一部分是在批次内进行排序。这部分的缓存复杂度为 $O(m(1 + \log_M m)/B)$。第二部分是访问原始 P-Orth 树中的树节点。在大小为n 的树中查找 m 个叶节点会触及 $O(m \log(n/m))$ 个树节点 [17]。将两部分的开销相加，即可得到所述缓存复杂度。□

将树高 $O(\log n)$ 替换为 $O(\log \Delta)$即可得到定理 3.1，且不需要有界长宽比这一假设。

B SPaC 树上的分析

B.1 正确性

我们通过证明 SPaC 树的更新算法与 PaC 树的更新算法等价，来证明前者的正确性。这里讨论插入算法，删除算法也可用类似方法证明。首先，算法 3 构造出的树与 PaC 树相同，因此第 6 行和第 12 行返回的树保持不变。第 13 行的分裂键对于两棵树相同，因此 SPaC 树会像 PaC 树在第 9 行那样，将相同的点插入叶节点，但会保留这些点的未排序状态。两棵树的连接和右连接操作（第 19 行和第 21 行）完全相同。在这种情况下，两棵树的划分都会到达相同的叶节点，而第 34 行和第 43 行确保在继续执行之前，SPaC 树中的点顺序与 PaC 树中的一致。展开和节点中的其他操作保持不变，因此正确性得以成立。

B.2 成本分析

定理 B.1。 一个包含 n 个点的 *SPaC* 树可以在 $O(n$对数$n)$工作量、 $O($对数$n)$跨度以及 $O($排序$(n))$缓存复杂度下构建。

证明。 HybridSort （算法3 ）是对样本排序算法的简单修改 [9, 19]——所有额外操作（即计算 SFC 代码并存储点 ID）都不会带来额外的渐近开销。算法 BuildSorted （算法 3 ）是一种并行分治算法，其工作量为 $O(n)$，跨度为$O(\log n)$，缓存复杂度为 $O(n/B)$。证毕。□

在以下证明中，我们假设批次大小为 $m = O(n)$。请注意，以下证明依赖于 PaC 树的分析，相关内容可见于 [23]。

定理 B.2。 对大小为 m、规模为 n 的 *SPaC* 树进行一次批量更新（插入或删除），需要 $O(m \log n)$工作量，跨度为 $O(\log^2 n)$。

证明。 我们首先证明跨度界。排序需要 $O(\log m)$ 的跨度，随后插入或删除点需要经过 $O(\log\ n)$ 轮才能到达叶节点。扩展叶节点并恢复点的顺序需要常数时间。每一轮中，连接和右连接的跨度均为 $O(\log m)$ [23]。整个过程的跨度总计为 $O(\log m \log\ n)$。

现在给出工作量界。对 m 点进行排序需要$O(m \log m)$工作量，每个点在每一轮需要执行 $O(1)$ 次操作。叶节点扩展需要常数时间。Join 的工作量渐近上受 RightJoin 的工作量 [23],约束，而 RightJoin 的总工作量为$O(m \log\ \frac{n}{m})$ 。因此，总工作量为 $O(m \log n)$。□

结合两个引理可得定理4.1。

C 实现细节

P-Orth 树。 我们选择在 P-Orth 树骨架中为二维点构建 $\lambda = 3$ 层，为三维点构建 $\lambda = 2$ 层，这在我们的机器上通常能提供良好的性能。对于每个包围盒，为节省内存，我们只存储左下角和右上角的点坐标（不含额外元数据）。单个 k-NN 查询会按照子树到查询点的最小距离递增的顺序遍历子树；该距离通过比较查询点与每个子树对应的包围盒计算得出。

SPaC 树。 SPaC 树的实现基于 PaC 树的代码 [23], ，但经过了精心重新设计，以优化性能。简单地将 PaC 树视为黑盒，会引入将输入点转换为键值对的额外开销——按照其用户手册中的建议，使用空间填充曲线代码作为键，并将整个点作为值。我们通过重新设计接口避免了这一开销，使 SPaC 树能够自动解析每个点中的空间填充曲线代码，将其作为键，并将其余属性作为值。这样，它就可以直接在输入序列上运行，从而减少预处理时间和内存使用量。

我们还引入了一种启发式方法，用于优化叶节点溢出时的批量更新。算法 4 中的原始方法会无条件地通过调用算法 3。当许多点受到影响时，这种方法的效率可能较低，因为在递归节点分配之前，必须先将插入批次与叶节点中的点合并，即使该批次已经排好序。因此，这会产生显著的开销。另一种方法是显式地将叶节点展开为带有空叶节点的平衡树，然后在该子树上执行批量插入。我们通过阈值在这两种策略之间进行选择。如果溢出的叶节点与新批次的总大小低于某个阈值（在我们的例子中为 4ϕ），我们就执行标准的局部重建。否则，我们展开该叶节点，并在展开后的子树上执行批量插入。

参数选择。 我们通过实证方式设置参数，以便在我们的机器上使所有实现达到最佳性能。对于 SPaC 树和 CPAM，我们将叶包装设置为 40；对于所有其他基线方法，则设置为 32。SPaC 树和 CPAM 均采用权重平衡方案，平衡参数为 $\alpha = 0.2$，即左右子树的权重最多相差 20%。对于 Pkd 树，我们采用其论文中建议的 $\alpha = 0.3$。

D 批量更新

现在，我们进一步提供单批次更新的实验结果。我们通过将批次大小从 10^5 调整到 10^9 个点，评估批量更新的性能，结果如图 10所示。实验设置包括一棵由 10^9 个点构建的初始树。随后，我们执行两种独立操作：批量插入，即添加从相同分布中抽取的新点；以及批量删除，即从树中移除数量相同的现有点。分析中省略了较小的批次大小，因为它们的计算开销较低，并行性的实际收益也因此有限。

所有基线方法在单批次插入和删除中都表现良好。在所有基准测试中，SPaC-H 树都比其他方法快，唯一例外是均匀分布数据上的批量删除，此时由于需要处理不平衡，其速度略慢于 P-Orth 树。在 Varden 数据集上的批量插入中，P-Orth 树比 SPaC-H 树慢，因为面对高度聚类的数据时，其结构会出现偏斜，导致树遍历时间开销更大。对于 Sweepline 和 Varden 等偏斜数据集，Pkd 树通常比 SPaC-H 树慢，因为当重建的子树较大时，其基于重建的平衡方案开销更高，而这在偏斜数据集上很常见。

E 三维合成数据集上的性能

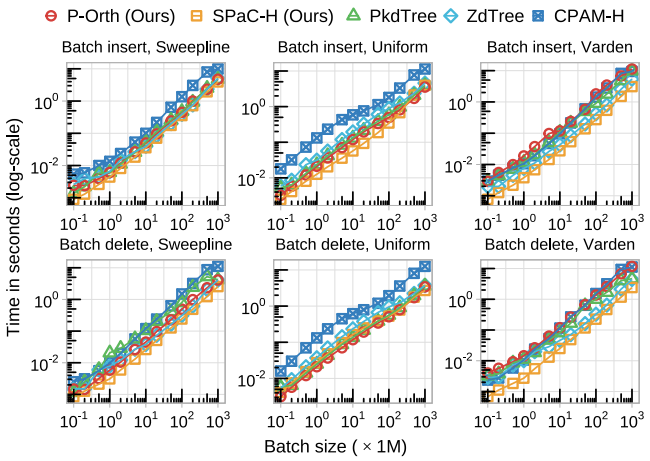
现在，我们提供更多关于三维合成数据集的实验结果。我们使用与二维情况相同的方法生成三维合成数据集，但将坐标范围限制在$[0, 10^6]$ 以内，以使其兼容SPaC-H 树。实验设置与二维第5节相同。结果如图9所示。我们省略了其他基线方法，因为如二维情况图3 所示，它们的速度更慢。

在树构建方面，SPaC-H 树仍然是最快的

		构建time	构建后查询（100%） 10NN InD	10NN OOD	范围 Cnt	范围 List	增量插入 批次大小 10%	1%	0.1%	.01%	增量插入后查询（50%） 10NN 分布内	10NN OOD	范围 Cnt	范围 List	增量删除 批次大小 10%	1%	0.1%	.01%	增量删除后查询（50%） 10NN 分布内	10NN OOD	范围 Cnt	范围 List
mo-nU	★P-OrthTree	5.18	.924	.904	**.701**	1.93	5.69	12.6	24.1	35.2	.988	.967	**.715**	1.80	5.93	**12.5**	**23.8**	**35.4**	1.07	1.06	.964	2.25
	★SPaC-H	**4.08**	7.34	7.29	.968	1.93	**4.70**	**7.09**	**14.7**	**28.8**	7.02	7.04	1.00	2.08	5.78	17.6	33.4	49.9	7.06	7.09	.993	2.05
	PkdTree	4.70	**.916**	**.894**	.758	**1.80**	5.74	13.41	24.5	36.9	**.983**	**.935**	.772	**1.77**	**5.41**	13.0	25.9	61.3	**.916**	**.893**	**.772**	**1.79**
enpeews	★P-OrthTree	6.41	.637	1.373	.877	2.03	6.44	6.33	6.58	19.62	**.666**	1.97	**.830**	**1.80**	4.30	5.15	**5.93**	21.1	.666	1.98	**.831**	**1.80**
	★SPaC-H	**3.90**	3.84	**1.04**	1.13	2.02	**3.76**	**4.03**	**5.81**	**13.0**	4.66	**1.57**	1.15	2.18	**3.01**	**3.30**	6.13	**20.5**	3.86	1.26	1.14	2.16
	PkdTree	5.57	**.603**	1.83	**.854**	**1.87**	16.4	21.1	28.5	69.0	1.010	6.71	1.039	1.98	14.6	28.5	38.0	65.8	**.647**	**.346**	.867	1.83
nedav	★P-OrthTree	7.5	.257	.214	**.552**	1.83	6.7	8.9	11.5	19.0	.263	.224	**.593**	1.71	6.04	7.78	9.7	17.3	.264	.238	.640	1.82
	★SPaC-H	**3.37**	2.33	1.59	.833	1.84	**3.81**	**4.40**	**5.66**	**10.8**	2.52	1.59	.847	1.98	**4.05**	**5.32**	**6.66**	**11.3**	1.87	1.30	.836	1.98
	PkdTree	7.04	**.152**	**.159**	.599	**1.72**	11.8	16.3	19.6	33.8	**.157**	**.167**	.628	**1.70**	6.20	8.9	12.0	23.7	**.160**	**.164**	**.613**	**1.69**

：在最快速度的1.1倍以内 The **最快时间**以粗体和下划线显示：在最快速度的2倍以内：在最快速度的5倍以内：>5倍于最快速度

图 9. 三维合成数据上的运行时间（秒）。越低越好。 每项测试中最快的时间以粗体和下划线标出。我们使用颜色标记与最快时间相差不超过 1 的结果。1×， 2×， 5×，以及 > 5× 最快时间。构建、查询和增量插入/删除的详细设置见第5.1节开头。 分布内/分布外：分布内/分布外。



在包含 1000M 个点、维度为 2 的树上，对来自合成数据集的点执行批量更新的运行时间。越低越好。 批量插入是指从相同分布中插入另一批点，批量删除是指从现有点中删除一批点。批次大小是该批次中的点数（× 1M）。图表采用双对数刻度。

索引，其时间相比 P-Orth 树和 Pkd 树更接近二维情况，因为基于空间填充曲线的索引对维度不太敏感。因此，SPaC-H 树比 P-Orth 树快 1.3–2.2× 倍，比 Pkd 树快 1.2–2.1× 倍。

关于批量更新，SPaC-H 树在大多数基准测试中仍然是最快的，只有在均匀分布的数据上，由于需要处理不平衡，速度略慢于 P-Orth 树。P-Orth 树在所有基准测试中都快于 Pkd 树。原因有两点：1）P-Orth 树无需处理不平衡；2）坐标范围限制在 $[0, 10^6]$,，这使得 P-Orth 树的树高远小于二维情况下的树高（数据范围为 $[0, 10^9]$），因此在 Sweepline 和 Varden 等偏斜数据上的树遍历时间大幅缩短。不过，Pkd 树在查询方面仍然具有优势——在大多数基准测试中速度最快，在其余测试中也具有竞争力。

F Artifact Description

F.1 摘要

PSI（并行空间索引），也称为 Ψ-Lib，是一个经过高度优化的 C++ 库，用于在多核架构上进行并行空间索引和查询。它提供了三种空间索引结构的高效实现：*k*d 树、Orth 树和 R 树，均针对并行处理进行了优化。Ψ-Lib 专为处理大规模空间数据集而设计，支持高效的批量操作（构建/插入/删除）和快速空间查询，例如最近邻搜索和范围搜索。该库不仅提供了高效的实现，还具备强有力的理论保证，因此适用于地理信息系统（GIS）、计算机图形学和空间数据分析等领域。

F.2 框架与实现

该库是一个仅包含头文件的 C++ 库，易于集成到现有项目中。所有基本几何图元和树的基本操作均在 psi::BaseTree 中实现，因此可以轻松扩展以支持其他几何类型和操作。

该系统实现了三种主要的空间树类型：psi::KdTree、psi::OrthTree 和 psi::PTree，此外还提供了两种基线实现：ZdTree 和 Boost-R-tree，用于性能比较。

所有树类型都支持通过分叉-汇合并行（ParlayLib）进行并行构建、批量更新（包括 Build()、BatchInsert()、BatchDelete() 和 BatchDiff()），以及 KNN()、RangeQuery() 和 RangeCount() 等空间查询。

通过使用手工构建的框架执行大量单元测试，验证了这些实现的正确性。该库还提供了一个并行合成数据集生成器，以便进行测试和基准测试。

F.3 制品元信息

- **程序**：C++ 库，支持由 Par- layLib [15]提供的并行功能。
- **编译**：g++ 14.2.1，使用 -O3 优化标志。
- **数据集**：实验中使用了内部生成的合成数据集

和真实世界数据集。

• **运行环境**：基于 Linux 的操作系统（例如 Ubuntu 20.04）。
• **硬件**：任意至少配备 512 GB 内存的基于 x86 的多核机器。

• **输出**：性能指标，包括构建时间、批量更新时间和查询时间。
•**工作流程**：下载代码并运行提供的脚本，即可在 Docker 容器中重现实验。

• **公开可用**：是，代码可在此处获取：[42]。

F.4 依赖项

硬件。实验需要一台现代化的（2010+）基于 x86 的多核心机器。该程序依赖 128 位 CM-PXCHG（需要使用 -mcx16 编译器标志），但不需要硬件事务内存（TSX）。机器应至少配备 512GB 内存，以处理实验中使用的大型数据集。我们使用的是一台配备 112 个核心（224 个硬件线程）的机器，运行 Rocky Linux 8.10，搭载四个英特尔至强铂金 8176 处理器和 1.47 TB 内存。

软件。代码使用 C++20 编写，可以使用 g++ 14.2.1 或更高版本，并配合 -O3 优化标志进行编译。外部依赖项以子模块形式包含在代码仓库中。实验使用 numactl 以实现更好的并行性，并使用 jemalloc 作为内存分配器，以提升内存分配性能。建议使用 Docker，通过代码仓库中的脚本快速设置环境。

数据集。该库附带一个并行合成数据集生成器，可生成具有两种分布的大规模数据集：均匀分布 和 Varden（参见第5.1 节，了解有关分布的更多详情），并支持任意维度。实验中使用的真实世界数据集可从以下位置下载：公开来源，链接见文档。

F.5 安装

可以从公共代码库 [42] 克隆代码。我们在代码仓库中提供了 Dockerfile，可快速设置环境。

F.6 实验工作流程

该工件的使用方法如下：
1. 从 [42] 克隆代码仓库。2. 运行 ./docker-run.sh pull 来构建 Docker 镜像。3. 运行 ./docker-run.sh full --data-path [path]，使用提供的数据集磁盘路径启动 Docker 容器，随后论文中的所有实验都会自动运行。4. 在代码仓库的脚本_ae/plots 文件夹中查看结果。

所有依赖项都已预先安装在 Docker 镜像中，数据集将生成或下载到提供的磁盘路径。

如果无法使用 Docker，也可以按照 README.md 中的说明编译代码，并使用脚本_ae/run.sh 运行实验。

F.7 评估与预期结果

所有操作（构建、批量更新和查询）的运行时间都会输出到标准输出，并保存到脚本_ae/logs 文件夹中。论文中的所有图表都会生成并保存到脚本_ae/plots 文件夹中。在我们的机器上，各次运行的执行时间差异很小，偏差为 1% 至 3%。请注意，性能可能会因硬件配置不同而有所变化，例如核心数量。我们的机器有 112 个核心（224 个硬件线程），建议使用至少配备 64 个核心的机器，以观察到类似的性能趋势。