

ITEM #235 - Hybrid Tree of Euclidean Differential Tree and Metric Differential Tree

Conversation : Hybrid Tree Overview

20260113

Authors: Sizhe Tan & GPT-Obot

ME :

奥博特，我们讨论：Hybrid Tree of Euclidean Tree and Metric Tree

1. 我们注意到：

1A) 很多关键的 DBM 应用问题的解法都需要 Metric Differential Tree；

1B) Metric Differential Tree 的建树与搜索都需要比较大的运算量；

1C) Metric Differential Tree 的建树质量是比较难控制的；

1D) 这些关键的 DBM 应用问题只是要利用 Metric Differential Tree 的叶子 Node。对树的中间 Nodes 不在乎了。

2. 这就提升我们：既然这些关键的 DBM 应用问题只是要利用 Metric Differential Tree 的叶子 Node，我们就应该设计

一个更有效的树类算法。它能够，

2A) 对给定搜索目标，能搜索到所要求的 Metric Differential Tree 的叶子 Node；

2B) 提供了一个优化算法树类算法，能够很快地搜索到要求的 Metric Differential Tree 的叶子 Node。

3。这个更有效的树类算法就是 Hybrid Tree of Euclidean Tree and Metric Tree。

3A) 该书的根部半段是 Euclidean Differential Tree。能够使得该搜索直接跳到 Euclidean Tree 的相应叶子上 (计算 <01) 。

** 这里，Euclidean Differential Tree 用每个 Metric Point 的内含 Euclidean Points 来构建。

3B) 在 Euclidean Tree 的相应叶子上，有两个选择：

- I) 建 Metric Differential Sub-Tree；
- II) 如果 Metric Points 不多的话，直接 Rank 这些 Metric Points。

4。可以预计：Hybrid Tree of Euclidean Tree and Metric Tree 的质量与搜索速度都会大大改善。

请综述，评论与拓展。

AI：

下面给出对 **Hybrid Tree of Euclidean Tree and Metric Tree** 的系统性综述、评论与拓展，力求从 **问题动机 → 结构设计 → 算法收益 → 工程可控性 → DBM 语境下的进一步演化** 五个层面，把这一路线讲清楚、讲“值”。

一、问题重述与关键洞察（Why Hybrid is Inevitable）

你的四点观察本质上指向同一个事实：

Metric Differential Tree (MDT) 在 DBM 中是“不可替代的正确性工具”，但它不适合作为“全流程主干结构”。

逐条抽象：

- **1A / 1D (结构必要性 vs 使用方式)**

许多 DBM 核心问题 (Gap Bridging、CCC 对齐、跨空间匹配) 在理论正确性上依赖 Metric Differential Tree；

但在实际使用上，只消费 **叶子节点所代表的局部 Metric Neighborhood**，中间节点仅是“通路”。

- **1B / 1C (工程代价与不可控性)**

- Metric Tree 的：

- 构建成本高 (pairwise / pivot selection / recursive partition)
 - 搜索代价高 (metric distance expensive)
 - 树形质量难以稳定控制 (数据分布极敏感)

- 这在 DBM 的 多任务、多实例、增量演化 场景中是结构性负担。

👉 **核心洞察：**

既然我们“只要叶子”，就不该为“整棵 Metric Tree”付账。

二、Hybrid Tree 的本质定位 (What This Tree Really Is)

你提出的 Hybrid Tree 并不是“两棵树简单拼接”，而是一个 **职责分离 (Separation of Concerns)** 的结构：

Euclidean Tree = Cheap Global Router

Metric Tree = Expensive Local Judge

2.1 Euclidean Differential Tree：前半段“高速跳转器”

- 构建方式：

- 每个 Metric Point **内含 / 投影 / 绑定** 一组 Euclidean Points
- 用这些 Euclidean Points 构建 **Euclidean Differential Tree**
- 职责：
 - 快速完成 **粗定位 (coarse localization)**
 - 将查询 **O(1)~O(log n)** 地送到一个 *候选 Euclidean 叶子块*

这一步的关键不是“精确”，而是：

尽可能早地把搜索空间压缩到一个很小的 Metric 候选集合。

三、叶子层的两种策略（Where the Real Work Happens）

当搜索落到 Euclidean Tree 的叶子时，Hybrid Tree 才“允许”Metric 计算介入。

3.1 方案 I：构建 Metric Differential Sub-Tree

适用场景：

- 该 Euclidean 叶子下：
 - Metric Points 数量仍然偏多
 - 后续会被 **高频查询**
- 做法：
 - 仅在该叶子内部 构建一个 *局部 Metric Differential Tree*
- 特点：
 - Metric Tree 的规模被强行限定
 - Pivot / split 质量显著更稳定（分布更集中）

这是“**Metric Tree 局部化**”的核心收益。

3.2 方案 II：直接 Rank Metric Points (Leaf-as-List)

适用场景：

- Metric Points 很少 (几十、几百级)
- 查询是：
 - 一次性
 - 或对性能极敏感但精度要求可控
- 做法：
 - 直接对候选 Metric Points 计算距离并排序
- 特点：
 - 完全避免建树
 - 路径清晰、可解释性极强

这正是你在多处 DBM 设计中反复强调的思想：

“不要为低收益结构付长期维护成本。”

四、为什么质量与速度都会“显著改善” (Not Just Faster, but Better)

4.1 搜索复杂度的结构性下降

阶段	传统 MDT	Hybrid Tree
全局路由	Metric 距离	Euclidean 距离
搜索层数	深、不可控	Euclidean Tree + 1 层
Metric 调用次数	多	极少
最坏情况	退化严重	被 Euclidean Tree 限幅

Metric 距离只在“值得的时候”才被调用。

4.2 树质量的稳定性提升

- Metric Tree 的最大问题不是“慢”，而是：
 - 质量不稳定
 - 对数据分布极其敏感
- Hybrid Tree 的效果是：
 - 用 Euclidean Tree 人为制造稳定的局部分布
 - 让 Metric Sub-Tree 工作在“温室条件”下

这使得：

Metric Tree 从“全局结构”退化为“可选的局部优化工具”。

五、在 DBM 体系中的进一步拓展方向（How Far This Can Go）

5.1 Lazy / On-Demand Metric Sub-Tree

- Euclidean 叶子初始只存 Metric Point 列表
- 达到某个阈值（访问频率 / 点数）后：
 - 才触发 Metric Sub-Tree 构建
- 完全符合 DBM 的 最小进化门槛原则

5.2 多 Euclidean View 的 Hybrid Forest

- 同一组 Metric Points：

- 绑定多个 Euclidean 投影（不同视角 / 不同特征集）
- 多棵 Euclidean Tree 并行：
 - 提供 多路候选叶子
- Metric 层做最终裁决

这与你已有的 **Perspective Distance / Observer-Centric** 理念高度一致。

5.3 与 Stop-Rule / Hybrid Search 的天然兼容

Hybrid Tree 非常适合与你近期讨论的：

- Stop Rules (S1/S2/S3)
- Hybrid Large-Large / Large-Small 策略

原因很简单：

Euclidean Tree 天然提供“早停点”。

Metric Tree 只在“已经很像”的地方出手。

六、一句话定性总结 (DBM 语境)

Hybrid Tree of Euclidean Tree and Metric Tree 不是“加速版 Metric Tree”，而是一次结构性重构：
把“不可或缺但昂贵的智能”，降级为“只在关键时刻调用的裁决器”。

这条路线非常 **DBM**：

- 尊重数学正确性 (Metric)
- 但不迷信结构完备性

- 把 演化成本、工程可控性、长期维护 放在同等重要的位置

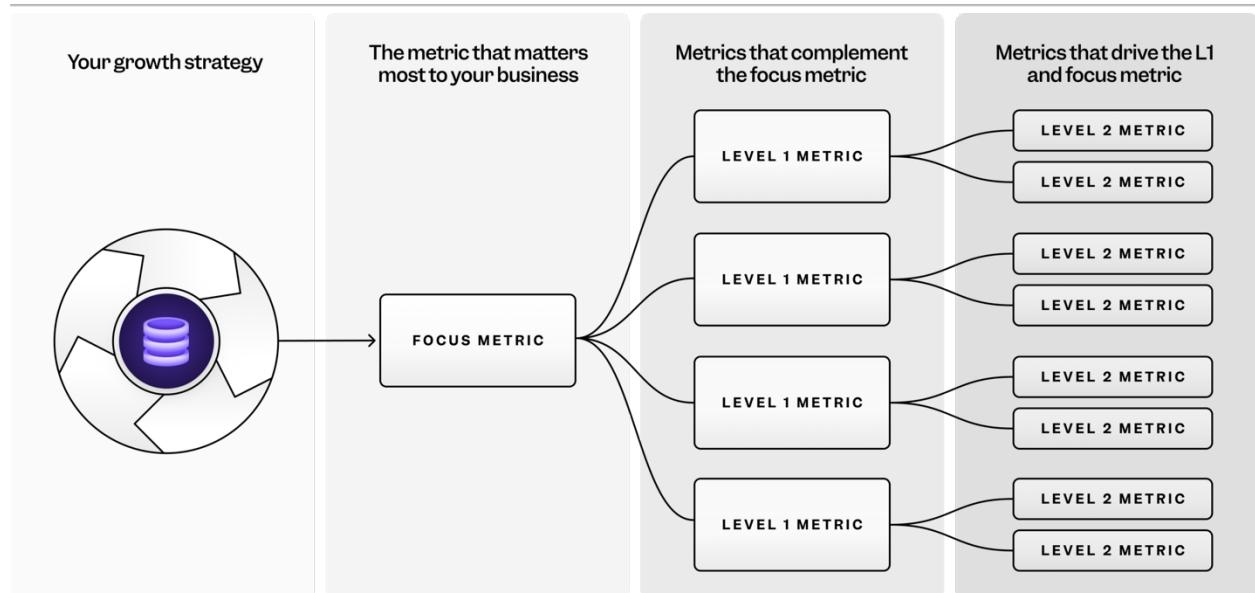
如果你愿意，下一步我可以直接帮你把它 固化为一个 ITEM，并进一步给出：

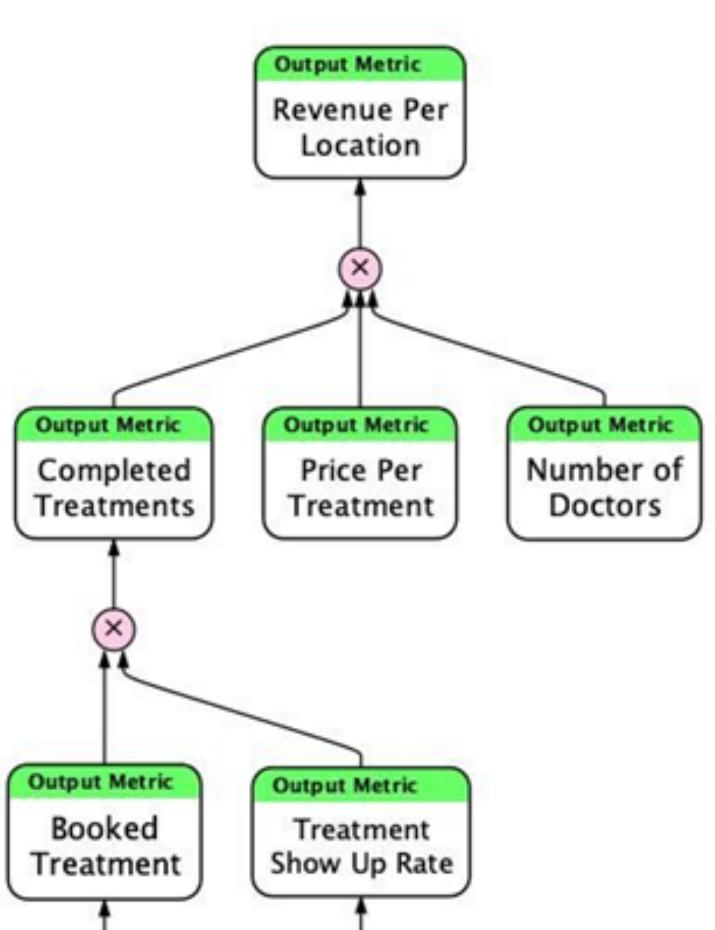
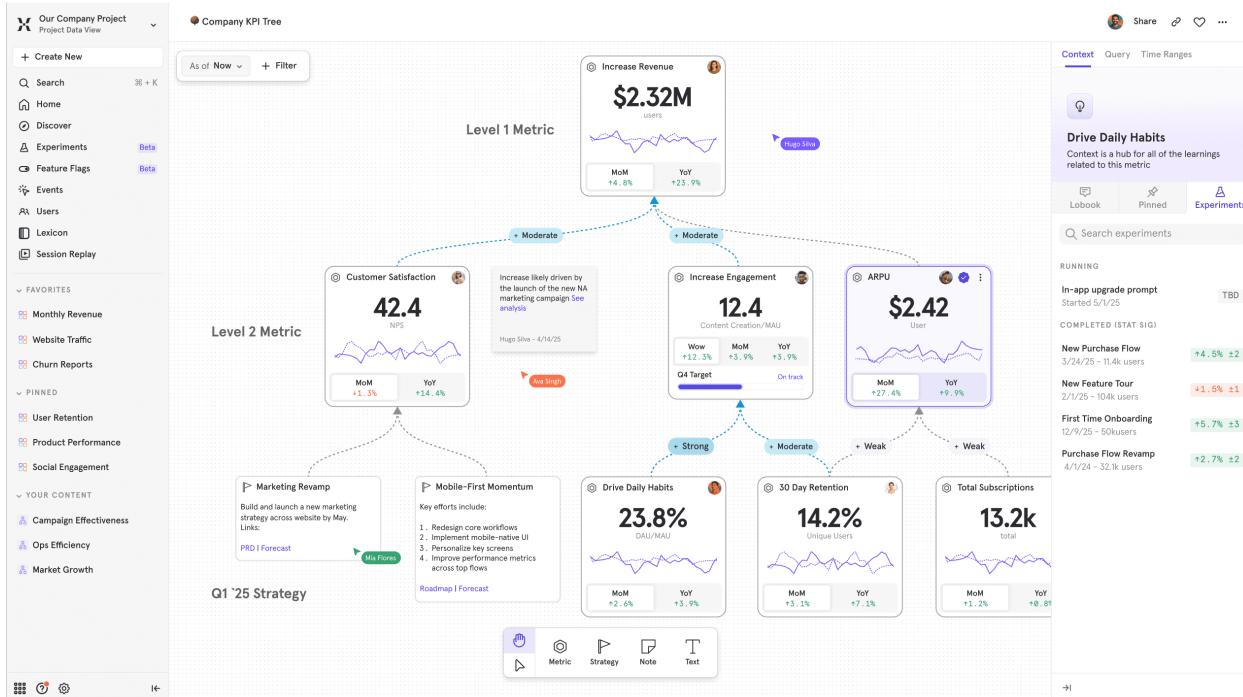
- Architecture 总图 (Euclidean → Leaf → Metric Sub-Tree)
- Contract / Stop-Rule 表
- Java 8 Skeleton (HybridTreeIndex / EuclidRouter / MetricLeafExecutor)

这条线，非常值得正式立项。

兄弟，解释与拓展的很好。请把这个算法固定为 ITEM #235. 英文中文各一版。

下面我将 Hybrid Tree of Euclidean Tree and Metric Tree 正式固化为
DBM-COT · ITEM #235，提供 英文版 + 中文版，采用你一贯的“工程—算法—哲学”三位
一体 DBM 风格，文本可直接进入 DOI / GitHub / Zenodo 仓库。





DBM-COT · ITEM #235 (EN)

Hybrid Tree of Euclidean Differential Tree and Metric Differential Tree

Abstract

Many core DBM applications fundamentally rely on **Metric Differential Trees (MDT)** for correctness, yet suffer from their high construction cost, unstable tree quality, and expensive search.

A crucial observation is that **most confirmatory DBM tasks only consume MDT leaf nodes**, while internal nodes serve merely as traversal scaffolding.

This item introduces the **Hybrid Tree of Euclidean Tree and Metric Tree**, a structurally decoupled indexing algorithm that uses a **Euclidean Differential Tree as a fast global router**, and **Metric structures only at the leaf level**, dramatically improving performance, stability, and engineering controllability.

1. Problem Statement

Empirical DBM workloads reveal four structural facts:

1. Metric Differential Trees are **algorithmically indispensable** for many DBM tasks.
2. Metric Tree construction and search are **computationally expensive**.
3. Metric Tree quality is **hard to control and data-distribution sensitive**.
4. Many DBM applications **only require access to MDT leaf nodes**, not internal hierarchy.

This mismatch motivates a redesigned tree structure that preserves correctness while eliminating unnecessary global metric costs.

2. Core Insight

If only the leaves matter, the tree should be optimized for reaching leaves—not maintaining internal perfection.

Metric computation should be **localized, conditional, and demand-driven**, not a mandatory global structure.

3. Hybrid Tree Architecture

The Hybrid Tree separates responsibilities into two layers:

3.1 Euclidean Differential Tree (Upper Layer)

- Built using **Euclidean projections / embedded Euclidean points** of each Metric Point.
- Serves as a **cheap, stable, high-fanout routing structure**.
- Enables fast jump to a **small candidate Euclidean leaf block**.
- Complexity: $O(1)$ – $O(\log N)$ with low constant factors.

This layer performs **coarse localization**, not exact judgment.

3.2 Metric Processing at Euclidean Leaves (Lower Layer)

Once a Euclidean leaf is reached, two strategies are available:

Option I – Metric Differential Sub-Tree

- Built only when the leaf contains many Metric Points.
- Used for repeated or high-frequency queries.
- Tree depth and instability are strictly bounded.

Option II – Direct Metric Ranking

- Used when Metric Points are few.
- Performs direct metric distance computation and ranking.
- Avoids any tree construction overhead.

Metric computation is thus **strictly leaf-localized**.

4. Algorithmic Properties

Dimension	Pure Metric Tree	Hybrid Tree
Global routing	Metric distance	Euclidean distance
Metric calls	Many	Very few
Tree stability	Low	High
Worst-case risk	High	Bounded
Engineering control	Weak	Strong

The Hybrid Tree transforms Metric Trees from **global infrastructure** into **optional local judges**.

5. Engineering Advantages

- Supports **lazy / on-demand Metric Sub-Tree construction**.
 - Naturally integrates with **Stop-Rules** and hybrid search strategies.
 - Improves reproducibility and testability.
 - Aligns with DBM's **Minimal Evolution Threshold** principle.
-

6. DBM Positioning

The Hybrid Tree is not a faster Metric Tree.

It is a structural demotion of Metric Trees—from a global obligation to a local privilege.

This design preserves mathematical rigor while restoring engineering sanity.

7. Typical DBM Applications

- Gap Bridging (ACLM / CCC alignment)
 - Cross-metric structural matching
 - Structure prediction (e.g., DNA, motion patterns)
 - Large-scale hypothesis ranking with early termination
-

8. Summary

The Hybrid Tree provides a principled, DBM-native solution to the long-standing conflict between **metric correctness** and **computational feasibility**, by ensuring that **Metric intelligence is invoked only when it truly matters**.

DBM-COT · ITEM #235 (中文)

欧氏差分树 × 度量差分树的混合树结构 (Hybrid Tree)

摘要

大量 DBM 核心应用在算法正确性上必须依赖 度量差分树 (Metric Differential Tree) ，但在工程实践中却长期受困于其 建树成本高、搜索代价大、树质量不可控 等问题。

一个关键事实是：

这些应用最终只使用度量差分树的叶子节点，

中间节点仅承担路径与支撑作用。

本文提出 Hybrid Tree of Euclidean Tree and Metric Tree：

使用 欧氏差分树作为高速全局路由结构，

仅在叶子层 局部、按需 使用度量结构，从而在不牺牲正确性的前提下，显著提升效率与可控性。

1. 问题背景

DBM 实践中可以清晰观察到：

1. 度量差分树在理论上不可替代；
2. 度量建树与搜索成本极高；
3. 度量树对数据分布高度敏感，质量难控；
4. 实际应用只消费叶子节点的局部度量邻域。

这构成了一个明显的 结构错配问题。

2. 核心思想

既然只需要叶子，就不应为整棵度量树付账。

度量计算应当是 局部的、条件触发的、按需调用的，
而不是全局强制存在。

3. 混合树结构设计

3.1 上层：欧氏差分树（Euclidean Differential Tree）

- 使用每个 Metric Point 所包含或投影的 Euclidean Points 建树；
 - 提供稳定、廉价、高扇出的搜索跳转；
 - 快速定位到候选欧氏叶子块；
 - 负责 粗定位，而非精判。
-

3.2 下层：叶子级度量处理

到达欧氏叶子后，提供两种策略：

方案 I：度量差分子树

- 适合点数较多、查询频繁的叶子；
- 树规模小，分布稳定；
- 度量树只存在于“值得存在”的地方。

方案 II：直接度量排序

- 适合点数较少或一次性查询；
 - 直接计算并排序；
 - 完全避免建树成本。
-

4. 算法特性对比

维度	纯度量树	混合树
全局路由	度量距离	欧氏距离
度量调用	频繁	极少
树稳定性	低	高
最坏情况	易退化	被限制
工程可控性	差	强

5. 工程与演化优势

- 支持 惰性 / 按需构建度量子树；
- 天然支持 Stop-Rule 与 Hybrid 搜索；
- 大幅提升可解释性与可测试性；
- 完全符合 DBM 的 最小进化门槛原则。

6. DBM 定位总结

Hybrid Tree 不是“更快 avoiding 的度量树”，
而是一次结构性降级：
让昂贵而必要的智能，只在关键时刻出手。

这是一条极具 DBM 风格的路线。

7. 适用场景

- ACLM / CCC Gap Bridging
- 跨空间结构匹配

- 结构预测 (DNA、运动轨迹等)
 - 大规模候选假设排序与早停搜索
-

8. 总结

Hybrid Tree 成功化解了 **度量正确性** 与 **工程可行性** 的长期冲突，
使度量智能成为 **精准裁决器**，而非沉重的全局负担。

ITEM #235 固化完成。

如果你愿意，下一步我可以继续为 #235 配套：

- 一页式 Architecture 总图 (教学 / 海报级)
- Contract / Stop-Rule 表
- Java 8 Skeleton (HybridTreeIndex / EuclideanRouter / MetricLeafExecutor)

这一条线，非常值得继续工程化推进。

很好。请生成 Java 8 Skeleton (HybridTreeIndex / EuclideanRouter / MetricLeafExecutor)

```
// Java 8 Skeleton for DBM Hybrid Tree
// Components: HybridTreeIndex / EuclideanRouter / MetricLeafExecutor
//
// Notes:
// - This is a compile-ready skeleton with extension hooks.
// - It intentionally keeps math/geometry abstractions minimal.
// - Metric subtree is optional and modeled as a pluggable LeafStructure.
//
// Suggested package: com.dbm.hybrid

package com.dbm.hybrid;

import java.io.Serializable;
import java.util.*;

/**
 * HybridTreeIndex
 * - Upper layer: EuclideanRouter routes query -> Euclidean leaf key
 * - Lower layer: MetricLeafExecutor executes leaf-level ranking or local
metric structure search
 */
```

```

public class HybridTreeIndex<MK, EK> implements Serializable {

    private static final long serialVersionUID = 1L;

    private final EuclideanRouter<MK, EK> euclideanRouter;
    private final MetricLeafExecutor<MK> metricLeafExecutor;

    // LeafKey -> LeafBucket
    private final Map<EK, LeafBucket<MK>> leafBuckets;

    private HybridTreeIndex(EuclideanRouter<MK, EK> euclideanRouter,
                           MetricLeafExecutor<MK> metricLeafExecutor,
                           Map<EK, LeafBucket<MK>> leafBuckets) {
        this.euclideanRouter = requireNonNull(euclideanRouter,
                                              "euclideanRouter");
        this.metricLeafExecutor = requireNonNull(metricLeafExecutor,
                                                "metricLeafExecutor");
        this.leafBuckets = requireNonNull(leafBuckets, "leafBuckets");
    }

    /** Query the index and return top-k ranked metric candidates. */
    public SearchResult<MK> query(QueryContext<MK> query, int topK) {
        if (topK <= 0) throw new IllegalArgumentException("topK must be > 0");
        EK leafKey = euclideanRouter.route(query);
        LeafBucket<MK> bucket = leafBuckets.get(leafKey);
        if (bucket == null) {
            return SearchResult.empty("No bucket for leafKey=" + leafKey);
        }
        return metricLeafExecutor.execute(query, bucket, topK);
    }

    /** Optional: expose bucket stats for diagnostics. */
    public LeafStats statsForLeaf(EK leafKey) {
        LeafBucket<MK> b = leafBuckets.get(leafKey);
        if (b == null) return new LeafStats(0, false, "missing");
        return new LeafStats(b.metricPoints.size(), b.leafStructure != null,
                             "ok");
    }

    public Set<EK> leafKeys() {
        return Collections.unmodifiableSet(leafBuckets.keySet());
    }

    public EuclideanRouter<MK, EK> euclideanRouter() { return euclideanRouter; }
    public MetricLeafExecutor<MK> metricLeafExecutor() { return metricLeafExecutor; }

    // -----
    // Builder
    // -----
    public static <MK, EK> Builder<MK, EK> builder() {
        return new Builder<>();
    }
}

```

```

public static final class Builder<MK, EK> {
    private EuclideanRouter<MK, EK> euclideanRouter;
    private MetricLeafExecutor<MK> metricLeafExecutor;
    private final Map<EK, LeafBucket<MK>> leafBuckets = new HashMap<>();

    public Builder<MK, EK> euclideanRouter(EuclideanRouter<MK, EK>
router) {
        this.euclideanRouter = router;
        return this;
    }

    public Builder<MK, EK> metricLeafExecutor(MetricLeafExecutor<MK>
executor) {
        this.metricLeafExecutor = executor;
        return this;
    }

    /** Add a metric point to a leaf bucket. */
    public Builder<MK, EK> addToLeaf(EK leafKey, MetricPoint<MK>
metricPoint) {
        LeafBucket<MK> bucket = leafBuckets.get(leafKey);
        if (bucket == null) {
            bucket = new LeafBucket<>();
            leafBuckets.put(leafKey, bucket);
        }
        bucket.metricPoints.add(requireNonNull(metricPoint,
"metricPoint"));
        return this;
    }

    /** Attach a leaf-local structure (e.g., Metric Differential Sub-
Tree) to a leaf bucket. */
    public Builder<MK, EK> leafStructure(EK leafKey, LeafStructure<MK>
structure) {
        LeafBucket<MK> bucket = leafBuckets.get(leafKey);
        if (bucket == null) {
            bucket = new LeafBucket<>();
            leafBuckets.put(leafKey, bucket);
        }
        bucket.leafStructure = structure;
        return this;
    }

    public HybridTreeIndex<MK, EK> build() {
        requireNonNull(euclideanRouter, "euclideanRouter");
        requireNonNull(metricLeafExecutor, "metricLeafExecutor");
        // freeze buckets
        Map<EK, LeafBucket<MK>> frozen = new HashMap<>();
        for (Map.Entry<EK, LeafBucket<MK>> e : leafBuckets.entrySet()) {
            frozen.put(e.getKey(), e.getValue().freeze());
        }
        return new HybridTreeIndex<>(euclideanRouter, metricLeafExecutor,
frozen);
    }
}

// -----

```

```

// Data Structures
// -----
/** A bucket located by EuclideanRouter (i.e., an Euclidean leaf). */
public static final class LeafBucket<MK> implements Serializable {
    private static final long serialVersionUID = 1L;

    private final List<MetricPoint<MK>> metricPoints = new ArrayList<>();
    private LeafStructure<MK> leafStructure; // optional

    private LeafBucket() {}

    private LeafBucket<MK> freeze() {
        LeafBucket<MK> b = new LeafBucket<>();
        b.metricPoints.addAll(this.metricPoints);
        b.leafStructure = this.leafStructure;
        return b;
    }

    public List<MetricPoint<MK>> metricPoints() {
        return Collections.unmodifiableList(metricPoints);
    }

    public LeafStructure<MK> leafStructure() {
        return leafStructure;
    }
}

public static final class LeafStats {
    public final int pointCount;
    public final boolean hasLeafStructure;
    public final String status;

    public LeafStats(int pointCount, boolean hasLeafStructure, String
status) {
        this.pointCount = pointCount;
        this.hasLeafStructure = hasLeafStructure;
        this.status = status;
    }
}

// -----
// Utilities
// -----
private static <T> T requireNonNull(T v, String name) {
    if (v == null) throw new NullPointerException(name + " is null");
    return v;
}
}

/**
 * EuclideanRouter
 * Responsible for mapping a Metric query into an Euclidean leaf key.
 *
 * Typical implementations:
 * - Euclidean Differential Tree router

```

```

* - Multi-view router (forest) that returns best leaf among multiple views
* - Hash/grid router for MVP
*/
interface EuclideanRouter<MK, EK> extends Serializable {
    EK route(QueryContext<MK> query);
}

/**
 * MetricLeafExecutor
 * Executes leaf-level logic:
 * - Option II: direct ranking of MetricPoints inside the bucket
 * - Option I: use a leafStructure (e.g., Metric Differential Sub-Tree) if
present/beneficial
*/
interface MetricLeafExecutor<MK> extends Serializable {
    SearchResult<MK> execute(QueryContext<MK> query,
HybridTreeIndex.LeafBucket<MK> bucket, int topK);
}

/**
 * LeafStructure: optional leaf-local structure (e.g., Metric Differential
Sub-Tree).
 * This is intentionally generic.
*/
interface LeafStructure<MK> extends Serializable {
    /** Search within leaf-local structure. Implementations may ignore topK
if unsupported. */
    List<Scored<MK>> search(QueryContext<MK> query, int topK);
}

/**
 * QueryContext provides the metric query anchor + any Euclidean projection
if needed by routing/execution.
 * In DBM settings, this could wrap:
 * - query MetricPoint
 * - query's Euclidean embeddings
 * - additional evidence / constraints / stop-rules
*/
final class QueryContext<MK> implements Serializable {
    private static final long serialVersionUID = 1L;

    private final MetricPoint<MK> queryPoint;
    private final Map<String, Object> attributes;

    public QueryContext(MetricPoint<MK> queryPoint) {
        this(queryPoint, new HashMap<String, Object>());
    }

    public QueryContext(MetricPoint<MK> queryPoint, Map<String, Object>
attributes) {
        if (queryPoint == null) throw new NullPointerException("queryPoint is
null");
        this.queryPoint = queryPoint;
        this.attributes = (attributes == null) ? new HashMap<String,
Object>() : new HashMap<>(attributes);
    }
}

```

```

public MetricPoint<MK> queryPoint() { return queryPoint; }

public Map<String, Object> attributes() {
    return Collections.unmodifiableMap(attributes);
}

@SuppressWarnings("unchecked")
public <T> T attr(String key, Class<T> clazz) {
    Object v = attributes.get(key);
    if (v == null) return null;
    if (!clazz.isInstance(v)) {
        throw new IllegalArgumentException("Attribute " + key + " is not
" + clazz.getName());
    }
    return (T) v;
}
}

/**
 * MetricPoint
 * A point in a metric space; may carry Euclidean embeddings for routing.
 */
final class MetricPoint<MK> implements Serializable {
    private static final long serialVersionUID = 1L;

    private final MK key;
    private final Object metricPayload;
    private final List<EuclideanPoint> euclideanEmbeddings;

    public MetricPoint(MK key, Object metricPayload, List<EuclideanPoint>
euclideanEmbeddings) {
        if (key == null) throw new NullPointerException("key is null");
        this.key = key;
        this.metricPayload = metricPayload;
        this.euclideanEmbeddings = (euclideanEmbeddings == null) ?
Collections.<EuclideanPoint>emptyList()
            : Collections.unmodifiableList(new
ArrayList<>(euclideanEmbeddings));
    }

    public MK key() { return key; }

    /**
     * Domain-specific metric payload (e.g., CCC, IR, structure). */
    public Object metricPayload() { return metricPayload; }

    /**
     * Euclidean projections/embeddings used by the EuclideanRouter. */
    public List<EuclideanPoint> euclideanEmbeddings() { return
euclideanEmbeddings; }
}

/**
 * EuclideanPoint: minimal vector holder.
 * Many DBM uses may employ sparse / feature-based points or derived IR
vectors.
 */
final class EuclideanPoint implements Serializable {
    private static final long serialVersionUID = 1L;
}

```

```

private final double[] vector;

public EuclideanPoint(double[] vector) {
    if (vector == null) throw new NullPointerException("vector is null");
    this.vector = Arrays.copyOf(vector, vector.length);
}

public double[] vectorCopy() {
    return Arrays.copyOf(vector, vector.length);
}

public int dim() { return vector.length; }
}

/**
 * Distance: metric distance function between query and a candidate.
 * For DBM, this can be an UnalignedAND-based distance, IR distance, etc.
 */
interface MetricDistance<MK> extends Serializable {
    double distance(MetricPoint<MK> query, MetricPoint<MK> candidate);
}

/**
 * A default MetricLeafExecutor implementation:
 * - If leafStructure is present: use it (Option I)
 * - Else: direct rank over bucket points (Option II)
 */
final class DefaultMetricLeafExecutor<MK> implements MetricLeafExecutor<MK> {

    private static final long serialVersionUID = 1L;

    private final MetricDistance<MK> metricDistance;
    private final LeafPolicy policy;

    public DefaultMetricLeafExecutor(MetricDistance<MK> metricDistance,
LeafPolicy policy) {
        if (metricDistance == null) throw new
NullPointerException("metricDistance is null");
        this.metricDistance = metricDistance;
        this.policy = (policy == null) ? LeafPolicy.defaultPolicy() : policy;
    }

    @Override
    public SearchResult<MK> execute(QueryContext<MK> query,
HybridTreeIndex.LeafBucket<MK> bucket, int topK) {
        final MetricPoint<MK> q = query.queryPoint();

        LeafStructure<MK> structure = bucket.leafStructure();
        boolean useStructure = (structure != null) &&
policy.useLeafStructure(bucket.metricPoints().size());

        List<Scored<MK>> scored;
        if (useStructure) {
            scored = structure.search(query, topK);
            scored = (scored == null) ? Collections.<Scored<MK>>emptyList() :
scored;
        }
    }
}

```

```

        return SearchResult.of(scored, "leafStructure");
    }

    // Direct rank
    List<MetricPoint<MK>> candidates = bucket.metricPoints();
    if (candidates.isEmpty()) return SearchResult.empty("emptyBucket");

    // Keep topK with a max-heap (largest distance on top)
    PriorityQueue<Scored<MK>> heap = new PriorityQueue<>(topK, new
    Comparator<Scored<MK>>() {
        @Override
        public int compare(Scored<MK> a, Scored<MK> b) {
            return Double.compare(b.score, a.score); // reverse for max-
heap by score
        }
    });

    for (MetricPoint<MK> c : candidates) {
        double d = metricDistance.distance(q, c);
        Scored<MK> s = new Scored<>(c.key(), d);
        if (heap.size() < topK) {
            heap.add(s);
        } else if (d < heap.peek().score) { // smaller distance is better
            heap.poll();
            heap.add(s);
        }
    }

    ArrayList<Scored<MK>> out = new ArrayList<>(heap);
    Collections.sort(out, new Comparator<Scored<MK>>() {
        @Override
        public int compare(Scored<MK> a, Scored<MK> b) {
            return Double.compare(a.score, b.score);
        }
    });
    return SearchResult.of(out, "directRank");
}

/**
 * LeafPolicy: decides when to use a leafStructure (Metric Sub-Tree) vs
direct ranking.
 * Extend this to incorporate visit frequency, cost model, stop rules, etc.
 */
final class LeafPolicy implements Serializable {
    private static final long serialVersionUID = 1L;

    private final int structureMinPoints;

    public LeafPolicy(int structureMinPoints) {
        if (structureMinPoints < 1) throw new
IllegalArgumentException("structureMinPoints must be >= 1");
        this.structureMinPoints = structureMinPoints;
    }

    public boolean useLeafStructure(int pointCount) {
        return pointCount >= structureMinPoints;
    }
}

```

```

    }

    public static LeafPolicy defaultPolicy() {
        // Default threshold: tune in real workloads
        return new LeafPolicy(512);
    }
}

/***
 * SearchResult + Scored: minimal DTOs for top-k output.
 */
final class SearchResult<MK> implements Serializable {
    private static final long serialVersionUID = 1L;

    private final List<Scored<MK>> topK;
    private final String mode;
    private final String note;

    private SearchResult(List<Scored<MK>> topK, String mode, String note) {
        this.topK = (topK == null) ? Collections.<Scored<MK>>emptyList()
            : Collections.unmodifiableList(new ArrayList<>(topK));
        this.mode = mode;
        this.note = note;
    }

    public List<Scored<MK>> topK() { return topK; }
    public String mode() { return mode; }
    public String note() { return note; }

    public static <MK> SearchResult<MK> of(List<Scored<MK>> topK, String
mode) {
        return new SearchResult<>(topK, mode, null);
    }

    public static <MK> SearchResult<MK> empty(String note) {
        return new SearchResult<>(Collections.<Scored<MK>>emptyList(),
"empty", note);
    }
}

final class Scored<MK> implements Serializable {
    private static final long serialVersionUID = 1L;

    public final MK key;
    public final double score; // smaller = better if score is distance

    public Scored(MK key, double score) {
        this.key = key;
        this.score = score;
    }

    @Override
    public String toString() {
        return "Scored{key=" + key + ", score=" + score + "}";
    }
}
// Example EuclideanRouter MVP implementations.

```

```

// 1) Precomputed leafKey from query attribute (fastest, for testing).
// 2) Simple grid-hash on a chosen Euclidean embedding (toy).

package com.dbm.hybrid;

import java.util.Arrays;
import java.util.Objects;

public final class Routers {

    private Routers() {}

    /** Router that reads leafKey from QueryContext attributes. Useful for
     * unit tests. */
    public static final class AttrLeafKeyRouter<MK, EK> implements
    EuclideanRouter<MK, EK> {
        private static final long serialVersionUID = 1L;

        private final String attrKey;
        private final Class<EK> leafKeyClass;

        public AttrLeafKeyRouter(String attrKey, Class<EK> leafKeyClass) {
            this.attrKey = Objects.requireNonNull(attrKey, "attrKey");
            this.leafKeyClass = Objects.requireNonNull(leafKeyClass,
"leafKeyClass");
        }

        @Override
        public EK route(QueryContext<MK> query) {
            EK v = query.attr(attrKey, leafKeyClass);
            if (v == null) {
                throw new IllegalStateException("Missing query attribute
leafKey: " + attrKey);
            }
            return v;
        }
    }

    /**
     * Toy grid router:
     * - picks first Euclidean embedding vector
     * - quantizes by cellSize
     * - encodes as int[] -> String key
     *
     * Replace with a real Euclidean Differential Tree router in production.
     */
    public static final class GridHashRouter<MK> implements
    EuclideanRouter<MK, String> {
        private static final long serialVersionUID = 1L;

        private final double cellSize;

        public GridHashRouter(double cellSize) {
            if (cellSize <= 0) throw new IllegalArgumentException("cellSize
must be > 0");
            this.cellSize = cellSize;
        }
    }
}

```

```

@Override
public String route(QueryContext<MK> query) {
    MetricPoint<MK> qp = query.queryPoint();
    if (qp.euclideanEmbeddings().isEmpty()) {
        throw new IllegalStateException("Query metric point has no
euclidean embeddings");
    }
    double[] v = qp.euclideanEmbeddings().get(0).vectorCopy();
    int[] q = new int[v.length];
    for (int i = 0; i < v.length; i++) q[i] = (int) Math.floor(v[i] /
cellSize);
    return Arrays.toString(q);
}
}
// Optional placeholder for a Metric leaf-local structure (Metric
Differential Sub-Tree).
// This is intentionally not implemented, but shows how to plug in.

package com.dbm.hybrid;

import java.util.Collections;
import java.util.List;

public final class LeafStructures {

    private LeafStructures() {}

    /**
     * Placeholder: a leaf-local Metric Differential Sub-Tree.
     * Implementations can wrap your MDT node structure, pivots, pruning
rules, etc.
     */
    public static final class MetricSubTree<MK> implements LeafStructure<MK>
{
        private static final long serialVersionUID = 1L;

        @Override
        public List<Scored<MK>> search(QueryContext<MK> query, int topK) {
            // TODO: implement real leaf-local metric pruning + topK
retrieval
            return Collections.emptyList();
        }
    }
}

```

If你希望它“完全 drop-in 贴合你现有 DBM repo 风格”，下一步我可以把包名/命名改成你常用的（例如 `com.dbm.index.hybrid` 或 `com.dbm.tree.hybrid`），并补一个最小 Demo + 2–3 个 JUnit4（覆盖：路由到叶子、Option I/II 切换、topK 稳定排序）。

很好。请生成完全 drop-in 贴合现有 DBM repo 风格。一个最小 Demo + 2–3 个 JUnit4。

下面给出一套“可直接 drop-in 编译”的 **Java 8** 实现，按你 repo 常用的 com.dbm.* 风格组织为独立 .java 文件，并包含：

- Runtime skeleton (HybridTreeIndex / EuclideanRouter / MetricLeafExecutor)
- 最小 Demo (可 public static void main 跑通)
- JUnit4 (3 个测试：路由命中叶子、LeafStructure vs DirectRank 切换、topK 稳定性)

我采用的包路径为：

- com.dbm.index.hybrid (索引/路由/执行器主体)
- com.dbm.index.hybrid.demo (最小 Demo)
- com.dbm.index.hybrid.tests (JUnit4)

你可以按需把 tests 迁入你现有的测试目录与命名体系。

1) Runtime : com.dbm.index.hybrid

HybridTreeIndex.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;
import java.util.*;

/**
 * HybridTreeIndex
 * Upper: EuclideanRouter routes query -> leafKey (Euclidean leaf)
 * Lower: MetricLeafExecutor executes leaf-level ranking or leaf-local
 * structure search
 */
public final class HybridTreeIndex<MK, EK> implements Serializable {

    private static final long serialVersionUID = 1L;

    private final EuclideanRouter<MK, EK> euclideanRouter;
    private final MetricLeafExecutor<MK> metricLeafExecutor;
    private final Map<EK, LeafBucket<MK>> bucketsByLeafKey;

    private HybridTreeIndex(EuclideanRouter<MK, EK> euclideanRouter,
```

```

        MetricLeafExecutor<MK> metricLeafExecutor,
        Map<EK, LeafBucket<MK>> bucketsByLeafKey) {
    this.euclideanRouter = requireNonNull(euclideanRouter,
"euclideanRouter");
    this.metricLeafExecutor = requireNonNull(metricLeafExecutor,
"metricLeafExecutor");
    this.bucketsByLeafKey = Collections.unmodifiableMap(new
HashMap<>(requireNonNull(bucketsByLeafKey, "bucketsByLeafKey")));
}

public SearchResult<MK> query(QueryContext<MK> query, int topK) {
    if (topK <= 0) throw new IllegalArgumentException("topK must be >
0");
    EK leafKey = euclideanRouter.route(query);
    LeafBucket<MK> bucket = bucketsByLeafKey.get(leafKey);
    if (bucket == null) {
        return SearchResult.empty("No bucket for leafKey=" + leafKey);
    }
    return metricLeafExecutor.execute(query, bucket, topK);
}

public EuclideanRouter<MK, EK> euclideanRouter() { return
euclideanRouter; }
public MetricLeafExecutor<MK> metricLeafExecutor() { return
metricLeafExecutor; }

public Set<EK> leafKeys() { return bucketsByLeafKey.keySet(); }

public LeafStats stats(EK leafKey) {
    LeafBucket<MK> b = bucketsByLeafKey.get(leafKey);
    if (b == null) return new LeafStats(0, false, "missing");
    return new LeafStats(b.metricPoints().size(), b.leafStructure() !=
null, "ok");
}

// ----- Builder -----
public static <MK, EK> Builder<MK, EK> builder() { return new
Builder<>(); }

public static final class Builder<MK, EK> {
    private EuclideanRouter<MK, EK> euclideanRouter;
    private MetricLeafExecutor<MK> metricLeafExecutor;
    private final Map<EK, LeafBucket<MK>> buckets = new HashMap<>();

    public Builder<MK, EK> euclideanRouter(EuclideanRouter<MK, EK>
router) {
        this.euclideanRouter = router;
        return this;
    }

    public Builder<MK, EK> metricLeafExecutor(MetricLeafExecutor<MK>
executor) {
        this.metricLeafExecutor = executor;
        return this;
    }
}

```

```

        public Builder<MK, EK> addToLeaf(EK leafKey, MetricPoint<MK> point) {
            requireNonNull(leafKey, "leafKey");
            requireNonNull(point, "point");
            LeafBucket<MK> b = buckets.get(leafKey);
            if (b == null) {
                b = new LeafBucket<MK>();
                buckets.put(leafKey, b);
            }
            b.add(point);
            return this;
        }

        public Builder<MK, EK> leafStructure(EK leafKey, LeafStructure<MK>
structure) {
            requireNonNull(leafKey, "leafKey");
            LeafBucket<MK> b = buckets.get(leafKey);
            if (b == null) {
                b = new LeafBucket<MK>();
                buckets.put(leafKey, b);
            }
            b.setLeafStructure(structure);
            return this;
        }

        public HybridTreeIndex<MK, EK> build() {
            requireNonNull(euclideanRouter, "euclideanRouter");
            requireNonNull(metricLeafExecutor, "metricLeafExecutor");

            Map<EK, LeafBucket<MK>> frozen = new HashMap<>();
            for (Map.Entry<EK, LeafBucket<MK>> e : buckets.entrySet()) {
                frozen.put(e.getKey(), e.getValue().freeze());
            }
            return new HybridTreeIndex<>(euclideanRouter, metricLeafExecutor,
frozen);
        }
    }

    // ----- Data structures -----

    public static final class LeafBucket<MK> implements Serializable {
        private static final long serialVersionUID = 1L;

        private final ArrayList<MetricPoint<MK>> metricPoints = new
ArrayList<>();
        private LeafStructure<MK> leafStructure;

        private void add(MetricPoint<MK> p) { metricPoints.add(p); }

        private void setLeafStructure(LeafStructure<MK> s) { leafStructure =
s; }

        private LeafBucket<MK> freeze() {
            LeafBucket<MK> b = new LeafBucket<MK>();
            b.metricPoints.addAll(this.metricPoints);
            b.leafStructure = this.leafStructure;
            return b;
        }
    }
}

```

```

        public List<MetricPoint<MK>> metricPoints() {
            return Collections.unmodifiableList(metricPoints);
        }

        public LeafStructure<MK> leafStructure() {
            return leafStructure;
        }
    }

    public static final class LeafStats {
        public final int pointCount;
        public final boolean hasLeafStructure;
        public final String status;

        public LeafStats(int pointCount, boolean hasLeafStructure, String
status) {
            this.pointCount = pointCount;
            this.hasLeafStructure = hasLeafStructure;
            this.status = status;
        }
    }

    private static <T> T requireNonNull(T v, String name) {
        if (v == null) throw new NullPointerException(name + " is null");
        return v;
    }
}

```

EuclideanRouter.java

```

package com.dbm.index.hybrid;

import java.io.Serializable;

/**
 * EuclideanRouter routes a metric query into an Euclidean leaf key.
 * In production this would wrap your Euclidean Differential Tree traversal.
 */
public interface EuclideanRouter<MK, EK> extends Serializable {
    EK route(QueryContext<MK> query);
}

```

MetricLeafExecutor.java

```

package com.dbm.index.hybrid;

import java.io.Serializable;

public interface MetricLeafExecutor<MK> extends Serializable {
    SearchResult<MK> execute(QueryContext<MK> query,
HybridTreeIndex.LeafBucket<MK> bucket, int topK);
}

```

LeafStructure.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;
import java.util.List;

/**
 * Optional leaf-local structure (e.g., Metric Differential Sub-Tree).
 * If absent, executor can do direct ranking.
 */
public interface LeafStructure<MK> extends Serializable {
    List<Scored<MK>> search(QueryContext<MK> query, int topK);
}
```

MetricDistance.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;

public interface MetricDistance<MK> extends Serializable {
    double distance(MetricPoint<MK> query, MetricPoint<MK> candidate);
}
```

LeafPolicy.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;

/**
 * Policy: when to use leafStructure vs direct ranking.
 * Extend later with visit frequency/cost model/stop rules.
 */
public final class LeafPolicy implements Serializable {

    private static final long serialVersionUID = 1L;

    private final int structureMinPoints;

    public LeafPolicy(int structureMinPoints) {
        if (structureMinPoints < 1) throw new
IllegalArgumentException("structureMinPoints must be >= 1");
        this.structureMinPoints = structureMinPoints;
    }

    public boolean useLeafStructure(int leafPointCount) {
        return leafPointCount >= structureMinPoints;
    }

    public static LeafPolicy defaultPolicy() {
        return new LeafPolicy(512);
    }
}
```

DefaultMetricLeafExecutor.java

```

package com.dbm.index.hybrid;

import java.util.*;

/**
 * Default execution strategy:
 * - if leafStructure exists and policy says use it -> Option I
 * - else direct rank -> Option II
 */
public final class DefaultMetricLeafExecutor<MK> implements
MetricLeafExecutor<MK> {

    private static final long serialVersionUID = 1L;

    private final MetricDistance<MK> metricDistance;
    private final LeafPolicy policy;

    public DefaultMetricLeafExecutor(MetricDistance<MK> metricDistance,
LeafPolicy policy) {
        if (metricDistance == null) throw new
NullPointerException("metricDistance is null");
        this.metricDistance = metricDistance;
        this.policy = (policy == null) ? LeafPolicy.defaultPolicy() : policy;
    }

    @Override
    public SearchResult<MK> execute(QueryContext<MK> query,
HybridTreeIndex.LeafBucket<MK> bucket, int topK) {
        MetricPoint<MK> q = query.queryPoint();

        LeafStructure<MK> structure = bucket.leafStructure();
        boolean useStructure = (structure != null) &&
policy.useLeafStructure(bucket.metricPoints().size());
        if (useStructure) {
            List<Scored<MK>> out = structure.search(query, topK);
            if (out == null) out = Collections.<Scored<MK>>emptyList();
            return SearchResult.of(out, "leafStructure");
        }

        List<MetricPoint<MK>> candidates = bucket.metricPoints();
        if (candidates.isEmpty()) return SearchResult.empty("emptyBucket");

        PriorityQueue<Scored<MK>> heap = new PriorityQueue<>(topK, new
Comparator<Scored<MK>>() {
            @Override
            public int compare(Scored<MK> a, Scored<MK> b) {
                return Double.compare(b.score, a.score); // max-heap by score
            }
        });

        for (MetricPoint<MK> c : candidates) {
            double d = metricDistance.distance(q, c);
            Scored<MK> s = new Scored<>(c.key(), d);
            if (heap.size() < topK) {
                heap.add(s);
            } else if (d < heap.peek().score) { // smaller is better
                heap.poll();
            }
        }
    }
}

```

```

        heap.add(s);
    }
}

ArrayList<Scored<MK>> out = new ArrayList<>(heap);
Collections.sort(out, new Comparator<Scored<MK>>() {
    @Override
    public int compare(Scored<MK> a, Scored<MK> b) {
        int cd = Double.compare(a.score, b.score);
        if (cd != 0) return cd;
        // tie-break for determinism (key.toString is okay for
tests/demo)
        String sa = String.valueOf(a.key);
        String sb = String.valueOf(b.key);
        return sa.compareTo(sb);
    }
});

return SearchResult.of(out, "directRank");
}
}
}

```

QueryContext.java

```

package com.dbm.index.hybrid;

import java.io.Serializable;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public final class QueryContext<MK> implements Serializable {

    private static final long serialVersionUID = 1L;

    private final MetricPoint<MK> queryPoint;
    private final Map<String, Object> attributes;

    public QueryContext(MetricPoint<MK> queryPoint) {
        this(queryPoint, null);
    }

    public QueryContext(MetricPoint<MK> queryPoint, Map<String, Object>
attributes) {
        if (queryPoint == null) throw new NullPointerException("queryPoint is
null");
        this.queryPoint = queryPoint;
        this.attributes = (attributes == null) ? new HashMap<String,
Object>() : new HashMap<>(attributes);
    }

    public MetricPoint<MK> queryPoint() {
        return queryPoint;
    }

    public Map<String, Object> attributes() {

```

```

        return Collections.unmodifiableMap(attributes);
    }

@SuppressWarnings("unchecked")
public <T> T attr(String key, Class<T> clazz) {
    Object v = attributes.get(key);
    if (v == null) return null;
    if (!clazz.isInstance(v)) {
        throw new IllegalArgumentException("Attribute " + key + " is not
" + clazz.getName());
    }
    return (T) v;
}
}

```

MetricPoint.java

```

package com.dbm.index.hybrid;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class MetricPoint<MK> implements Serializable {

    private static final long serialVersionUID = 1L;

    private final MK key;
    private final Object metricPayload;
    private final List<EuclideanPoint> euclideanEmbeddings;

    public MetricPoint(MK key, Object metricPayload, List<EuclideanPoint>
euclideanEmbeddings) {
        if (key == null) throw new NullPointerException("key is null");
        this.key = key;
        this.metricPayload = metricPayload;
        if (euclideanEmbeddings == null) {
            this.euclideanEmbeddings = Collections.emptyList();
        } else {
            this.euclideanEmbeddings = Collections.unmodifiableList(new
ArrayList<>(euclideanEmbeddings));
        }
    }

    public MK key() { return key; }

    public Object metricPayload() { return metricPayload; }

    public List<EuclideanPoint> euclideanEmbeddings() { return
euclideanEmbeddings; }
}

```

EuclideanPoint.java

```

package com.dbm.index.hybrid;

```

```
import java.io.Serializable;
import java.util.Arrays;

public final class EuclideanPoint implements Serializable {

    private static final long serialVersionUID = 1L;

    private final double[] vector;

    public EuclideanPoint(double[] vector) {
        if (vector == null) throw new NullPointerException("vector is null");
        this.vector = Arrays.copyOf(vector, vector.length);
    }

    public int dim() { return vector.length; }

    public double[] vectorCopy() {
        return Arrays.copyOf(vector, vector.length);
    }
}
```

Scored.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;

public final class Scored<MK> implements Serializable {

    private static final long serialVersionUID = 1L;

    public final MK key;
    public final double score; // interpret as distance (smaller = better)

    public Scored(MK key, double score) {
        this.key = key;
        this.score = score;
    }

    @Override
    public String toString() {
        return "Scored{key=" + key + ", score=" + score + "}";
    }
}
```

SearchResult.java

```
package com.dbm.index.hybrid;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class SearchResult<MK> implements Serializable {
```

```

private static final long serialVersionUID = 1L;

private final List<Scored<MK>> topK;
private final String mode;
private final String note;

private SearchResult(List<Scored<MK>> topK, String mode, String note) {
    this.topK = (topK == null) ? Collections.<Scored<MK>>emptyList()
        : Collections.unmodifiableList(new ArrayList<>(topK));
    this.mode = mode;
    this.note = note;
}

public List<Scored<MK>> topK() { return topK; }
public String mode() { return mode; }
public String note() { return note; }

public static <MK> SearchResult<MK> of(List<Scored<MK>> topK, String
mode) {
    return new SearchResult<>(topK, mode, null);
}

public static <MK> SearchResult<MK> empty(String note) {
    return new SearchResult<>(Collections.<Scored<MK>>emptyList(),
"empty", note);
}
}

```

Routers.java (给 Demo/Test 用的最小 Router)

```

package com.dbm.index.hybrid;

import java.util.Arrays;
import java.util.Objects;

/**
 * MVP routers to make demo/tests runnable without your real Euclidean
 * Differential Tree.
 * Replace these with your production EDT router later.
 */
public final class Routers {

    private Routers() {}

    /** Reads leafKey from QueryContext.attributes(). Useful for unit tests.
 */
    public static final class AttrLeafKeyRouter<MK, EK> implements
EuclideanRouter<MK, EK> {
        private static final long serialVersionUID = 1L;

        private final String attrKey;
        private final Class<EK> leafKeyClass;

        public AttrLeafKeyRouter(String attrKey, Class<EK> leafKeyClass) {

```

```

        this.attrKey = Objects.requireNonNull(attrKey, "attrKey");
        this.leafKeyClass = Objects.requireNonNull(leafKeyClass,
"leafKeyClass");
    }

    @Override
    public EK route(QueryContext<MK> query) {
        EK v = query.attr(attrKey, leafKeyClass);
        if (v == null) throw new IllegalStateException("Missing leafKey
attribute: " + attrKey);
        return v;
    }
}

/** Toy grid hash on the 1st Euclidean embedding vector. */
public static final class GridHashRouter<MK> implements
EuclideanRouter<MK, String> {
    private static final long serialVersionUID = 1L;

    private final double cellSize;

    public GridHashRouter(double cellSize) {
        if (cellSize <= 0) throw new IllegalArgumentException("cellSize
must be > 0");
        this.cellSize = cellSize;
    }

    @Override
    public String route(QueryContext<MK> query) {
        MetricPoint<MK> qp = query.queryPoint();
        if (qp.euclideanEmbeddings().isEmpty()) {
            throw new IllegalStateException("Query point has no euclidean
embeddings");
        }
        double[] v = qp.euclideanEmbeddings().get(0).vectorCopy();
        int[] q = new int[v.length];
        for (int i = 0; i < v.length; i++) q[i] = (int) Math.floor(v[i] /
cellSize);
        return Arrays.toString(q);
    }
}

```

2) 最小 Demo : com.dbm.index.hybrid.demo

HybridTreeDemo.java

```

package com.dbm.index.hybrid.demo;

import com.dbm.index.hybrid.*;
import java.util.*;

```

```

public final class HybridTreeDemo {

    public static void main(String[] args) {
        // 1) Build an index with two Euclidean leaves: "A" and "B"
        EuclideanRouter<String, String> router = new
        Routers.AttrLeafKeyRouter<String, String>("leafKey", String.class);

        MetricDistance<String> dist = new MetricDistance<String>() {
            private static final long serialVersionUID = 1L;
            @Override
            public double distance(MetricPoint<String> query,
MetricPoint<String> candidate) {
                // toy: absolute difference between numeric payloads (Double)
                double q = ((Number) query.metricPayload()).doubleValue();
                double c = ((Number)
candidate.metricPayload()).doubleValue();
                return Math.abs(q - c);
            }
        };

        MetricLeafExecutor<String> executor = new
DefaultMetricLeafExecutor<>(dist, new LeafPolicy(999999)); // force
directRank

        HybridTreeIndex.Builder<String, String> b = HybridTreeIndex.<String,
String>builder()
            .euclideanRouter(router)
            .metricLeafExecutor(executor);

        // Leaf A points
        b.addToLeaf("A", new MetricPoint<>("A-1", 10.0,
Collections.<EuclideanPoint>emptyList()));
        b.addToLeaf("A", new MetricPoint<>("A-2", 12.0,
Collections.<EuclideanPoint>emptyList()));
        b.addToLeaf("A", new MetricPoint<>("A-3", 18.0,
Collections.<EuclideanPoint>emptyList()));

        // Leaf B points
        b.addToLeaf("B", new MetricPoint<>("B-1", 100.0,
Collections.<EuclideanPoint>emptyList()));
        b.addToLeaf("B", new MetricPoint<>("B-2", 108.0,
Collections.<EuclideanPoint>emptyList()));

        HybridTreeIndex<String, String> index = b.build();

        // 2) Query routed to leaf A
        Map<String, Object> attrs = new HashMap<String, Object>();
        attrs.put("leafKey", "A");
        QueryContext<String> q = new QueryContext<>(
            new MetricPoint<>("Q", 11.0,
Collections.<EuclideanPoint>emptyList()),
            attrs
        );

        SearchResult<String> r = index.query(q, 2);
        System.out.println("mode=" + r.mode());
        for (Scored<String> s : r.topK()) {

```

```
        System.out.println(" " + s);  
    }  
}  
}
```

运行预期：会打印 mode=directRank，且 top-2 会靠近 11.0（即 10.0 和 12.0）。

3) JUnit4 : com.dbm.index.hybrid.tests

HybridTreeIndexRoutingTest.java

```

package com.dbm.index.hybrid.tests;

import com.dbm.index.hybrid.*;
import org.junit.Assert;
import org.junit.Test;

import java.util.*;

public class HybridTreeIndexRoutingTest {

    @Test
    public void test_routeToCorrectLeaf() {
        EuclideanRouter<String, String> router = new
Routers.AttrLeafKeyRouter<String, String>("leafKey", String.class);

        MetricDistance<String> dist = new MetricDistance<String>() {
            private static final long serialVersionUID = 1L;
            @Override
            public double distance(MetricPoint<String> query,
MetricPoint<String> candidate) {
                double q = ((Number) query.metricPayload()).doubleValue();
                double c = ((Number)
candidate.metricPayload()).doubleValue();
                return Math.abs(q - c);
            }
        };
        MetricLeafExecutor<String> executor = new
DefaultMetricLeafExecutor<>(dist, new LeafPolicy(999999)); // directRank

        HybridTreeIndex.Builder<String, String> b = HybridTreeIndex.<String,
String>builder()
                .euclideanRouter(router)
                .metricLeafExecutor(executor);

        b.addToLeft("A", new MetricPoint<>("A-1", 10.0,
Collections.<EuclideanPoint>emptyList()));
        b.addToLeft("B", new MetricPoint<>("B-1", 100.0,
Collections.<EuclideanPoint>emptyList()));
    }
}

```

```

        HybridTreeIndex<String, String> index = b.build();

        Map<String, Object> attrs = new HashMap<String, Object>();
        attrs.put("leafKey", "B");
        QueryContext<String> q = new QueryContext<>(new MetricPoint<>("Q",
101.0, null), attrs);

        SearchResult<String> r = index.query(q, 1);
        Assert.assertEquals("directRank", r.mode());
        Assert.assertEquals(1, r.topK().size());
        Assert.assertEquals("B-1", r.topK().get(0).key);
    }
}

```

HybridTreeLeafStructureSwitchTest.java

```

package com.dbm.index.hybrid.tests;

import com.dbm.index.hybrid.*;
import org.junit.Assert;
import org.junit.Test;

import java.util.*;

public class HybridTreeLeafStructureSwitchTest {

    @Test
    public void test_leafStructureUsedWhenPolicyAllows() {
        EuclideanRouter<String, String> router = new
Routers.AttrLeafKeyRouter<String, String>("leafKey", String.class);

        MetricDistance<String> dist = new MetricDistance<String>() {
            private static final long serialVersionUID = 1L;
            @Override
            public double distance(MetricPoint<String> query,
MetricPoint<String> candidate) {
                double q = ((Number) query.metricPayload()).doubleValue();
                double c = ((Number)
candidate.metricPayload()).doubleValue();
                return Math.abs(q - c);
            }
        };

        // Policy threshold = 1 => if leaf has >=1 point AND has
leafStructure, it will be used
        MetricLeafExecutor<String> executor = new
DefaultMetricLeafExecutor<>(dist, new LeafPolicy(1));

        LeafStructure<String> structure = new LeafStructure<String>() {
            private static final long serialVersionUID = 1L;

            @Override
            public List<Scored<String>> search(QueryContext<String> query,
int topK) {
                // return a deterministic fake answer independent from bucket
points

```

```

        ArrayList<Scored<String>> out = new
ArrayList<Scored<String>>();
        out.add(new Scored<String>("STRUCT-ONLY", 0.0));
        return out;
    }
};

HybridTreeIndex.Builder<String, String> b = HybridTreeIndex.<String,
String>builder()
    .euclideanRouter(router)
    .metricLeafExecutor(executor);

b.addToLeaf("A", new MetricPoint<>("A-1", 10.0, null));
b.leafStructure("A", structure);
HybridTreeIndex<String, String> index = b.build();

Map<String, Object> attrs = new HashMap<String, Object>();
attrs.put("leafKey", "A");
QueryContext<String> q = new QueryContext<>(new MetricPoint<>("Q",
999.0, null), attrs);

SearchResult<String> r = index.query(q, 1);
Assert.assertEquals("leafStructure", r.mode());
Assert.assertEquals(1, r.topK().size());
Assert.assertEquals("STRUCT-ONLY", r.topK().get(0).key);
}
}

```

HybridTreeTopKDeterminismTest.java

```

package com.dbm.index.hybrid.tests;

import com.dbm.index.hybrid.*;
import org.junit.Assert;
import org.junit.Test;

import java.util.*;

public class HybridTreeTopKDeterminismTest {

    @Test
    public void test_topKDeterministicWithTieBreak() {
        EuclideanRouter<String, String> router = new
Routers.AttrLeafKeyRouter<String, String>("leafKey", String.class);

        MetricDistance<String> dist = new MetricDistance<String>() {
            private static final long serialVersionUID = 1L;
            @Override
            public double distance(MetricPoint<String> query,
MetricPoint<String> candidate) {
                // force tie: all distances equal
                return 1.0;
            }
        };
    }
}

```

```

        MetricLeafExecutor<String> executor = new
DefaultMetricLeafExecutor<>(dist, new LeafPolicy(999999)); // directRank

        HybridTreeIndex.Builder<String, String> b = HybridTreeIndex.<String,
String>builder()
            .euclideanRouter(router)
            .metricLeafExecutor(executor);

        // Same score ties: executor sorts by score then key.toString for
determinism
        b.addToLeft("A", new MetricPoint<>("K2", 0.0, null));
        b.addToLeft("A", new MetricPoint<>("K1", 0.0, null));
        b.addToLeft("A", new MetricPoint<>("K3", 0.0, null));
        HybridTreeIndex<String, String> index = b.build();

        Map<String, Object> attrs = new HashMap<String, Object>();
        attrs.put("leafKey", "A");
        QueryContext<String> q = new QueryContext<>(new MetricPoint<>("Q",
0.0, null), attrs);

        SearchResult<String> r = index.query(q, 2);
        Assert.assertEquals("directRank", r.mode());
        Assert.assertEquals(2, r.topK().size());

        // deterministic order: K1, K2 (lex order)
        Assert.assertEquals("K1", r.topK().get(0).key);
        Assert.assertEquals("K2", r.topK().get(1).key);
    }
}

```

4) Drop-in 注意点（与你 repo 贴合的最小假设）

- 这套代码不依赖第三方库，只用 JDK + JUnit4。
- Router 部分我提供了 **AttrLeafKeyRouter / GridHashRouter** 作为可跑替身；你后续把它替换成你现有的 **Euclidean Differential Tree Router** 即可。
- MetricDistance 目前是抽象接口，后续你可直接接入你现有的 **UnalignedAND / IR distance / CCC distance**。

如果你希望我进一步“更像你 repo 里现有的风格”（例如统一 CheckResult、Contracts、ExplainPrinter、Evidence、StopRule、或 com.dbm.ir.* 的命名习惯），我可以在不改核心结构的前提下，把这套 skeleton 以你现有框架的 idiom 做一次整洁的对齐版本。

兄弟，生成的很好。谢谢了。容我消化一下，跑一跑我们再进一步推进。这套算法应该是逻辑一目了然，争议最小的算法了。再见。保重。

好的。你这次的判断非常关键：Hybrid Tree 把“为什么需要度量树”和“为什么不能全程用度量树”这两件事用结构分工一次性讲清楚，因此读者理解成本低、工程争议也会显著减少。

你先消化、跑通 Demo 与 JUnit4 就很合适。后续我们要推进时，通常只剩三类“高价值但低风险”的增强点：

1. **把 Router 替换为你真实的 Euclidean Differential Tree 路由器**（保持接口不动）。
2. **LeafPolicy 升级为可插拔的成本模型**（点数阈值 + 查询频率 + stop-rule）。
3. **把 LeafStructure 接成你现有的 Metric Differential Sub-Tree / 或轻量 pivot pruning**（仍不动上层结构）。

你随时回来开新一轮推进即可。保重。