不揮発メモリを含む階層型ストレージを対象とする効率的R木管理手法の検討

吉岡　弘隆†　　合田　和生††　　喜連川　優††

† 東京大学 大学院情報理工学系研究科 情報理工学専攻
†† 東京大学 生産技術研究所
E-mail: †{hyoshiok,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし　本論文は、DRAM、PMEM、SSD からなる階層型ストレージ上で空間インデックスである R 木を効率的に運用するための適応的マルチレイヤ R 木（以下、AMR-Tree）を提案する。AMR-Tree は、R 木ノードの属性（内部／リーフ）とアクセス頻度（ホット度）に基づいてノード単位で配置先を決定する。内部ノードを DRAM にピン留めし、リーフノードを PMEM と SSD の間で動的に再配置する三層配置設計と、再配置時の参照更新を Leaf Table により局所化する機構を示す。三層 tiering 機構の budget enforcement（容量予算に基づくバックグラウンド降格）を実装し、PMEM 予算の 20%〜unlimited 設定において tier 分布が設計通りに制御されること、およびバックグラウンド移動のオーバーヘッドが挿入スループットの約 4.0%以内に留まることを確認した。search/insert パスの Leaf Table 統合を完了し、リーフの参照解決が Leaf Table 経由で透過的に行われる構成を実現した。

キーワード　空間索引, R 木, 不揮発メモリ (PMEM), 階層型ストレージ, 適応的配置

1　はじめに

近年のサーバでは、DRAM に加えて永続メモリ（PMEM）や SSD を併用する構成が一般化しつつある。こうした階層型ストレージは容量とコストの面で有利である一方で、アクセス階層の違いに起因する性能変動や尾部遅延（p95、p99）の悪化、さらに永続化操作（キャッシュ書き戻しとフェンス）に起因する更新性能の低下が顕在化しやすい。

空間データ管理では、位置情報や GIS を支える索引として R 木が広く用いられる [1]。しかし、R 木の更新では挿入・削除に伴い探索経路上のノード更新が繰り返され、さらに分割やマージといった構造変更操作（SMO）が発生する。階層型ストレージ上でこれらをそのまま実行すると、PMEM では永続化操作の頻発が尾部遅延を押し上げ、SSD ではランダム I/O が律速となりやすい。

本研究の提案は、R 木のノード属性（内部ノード／リーフノード）に基づき、配置場所を DRAM ／ PMEM ／ SSD のいずれかに動的に変更可能とする点にある。すなわち、変更が頻繁で遅延の影響が大きい箇所は DRAM に、永続性が必要な箇所は PMEM ないし SSD に配置する。具体的には、内部ノードは DRAM にピン留め（pin）し、リーフノードはアクセス傾向に応じて PMEM と SSD の間で再配置（tiering）する。図 1 に示すように、内部ノードの経路支配性とリーフノードの I/O 支配性という役割差を利用することで、性能と一貫性の両立を目指す。

1.1　過去の発表との関係と差分

本稿の位置付けを明確にするため、過去の我々の研究成果との関係を述べる。DEIM Forum 2023 では、PMEM を対象とする空間検索構造の実装方式に関する検討と予備実験を行い、PMEM における永続化操作や実装上の論点を整理した [2]。また、DEIM Forum 2024 では、PMEM を対象とする空間索引について SSD との性能比較を実施し、更新処理における性能差を定量化した [3]。

これらに対し本稿は、DRAM ／ PMEM ／ SSD **の三層を同時に対象**とし、内部ノードとリーフノードの役割差に基づく pin と tiering の統合設計、ならびに再配置時の参照更新を局所化する機構（Leaf Table）と更新手順を示す。さらに、実機環境において budget enforcement 機構による tier 分布の制御性と移動オーバーヘッドを定量的に評価する。

1.2　本稿の構成

本稿の構成は以下の通りである。第 2 章で背景を整理し、第 3 章で提案と実装（階層配置、データ構造、移動手順、更新手順）を述べる。第 4 章で三層 tiering 機構の評価を行い、第 5 章で関連研究を整理した上で、第 6 章で結論と今後の計画を述べる。

2　背　　景

近年のサーバでは、DRAM に加えて PMEM や SSD を併用する階層型ストレージが一般化しつつある。DRAM および PMEM は CPU のメモリコントローラを介して DDR 系インタフェースにより接続され、ロード／ストア命令で直接アクセ

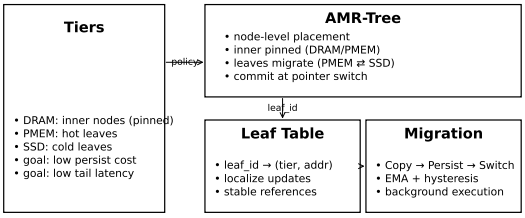

図 1　階層型ストレージ上の R 木配置概観 (例)。内部ノードを DRAM に固定し、リーフを PMEM ／ SSD 間で移動する。

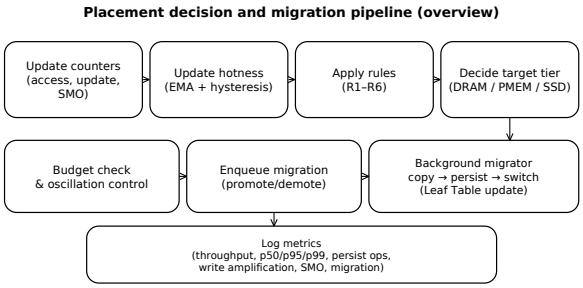

図 2　配置決定と移動のフロー（概要)。

ス可能である。一方、SSD は PCIe ＋ NVMe や SATA ／ SAS 経由のブロックデバイスとしてアクセスされる。PMEM はバイトアドレス可能な不揮発メモリであるが、CPU キャッシュを介するため、更新を永続化するにはキャッシュライン書き戻し（flush）と順序保証（fence）が必要となる。これらの永続化操作は、更新頻度が高い場合にスループットや尾部遅延を支配し得る [4,5]。R 木は更新に伴い、ノード分割（split）やマージ等の構造変更操作（SMO）が発生し、書き込み増幅（WA）を誘発する。したがって、階層型ストレージ上では、永続化回数と I/O 量の双方を意識した配置設計が重要である。

内部ノードは探索経路を支配し、わずかな遅延増が多くの操作に波及する。一方、リーフノードはデータ本体に近く、移動や I/O 量の支配因となる。本研究はこの役割差を利用し、内部ノードとリーフノードで異なる配置方針を採用する。

階層型ストレージ上で R 木ノードを動的に再配置する場合、移動中の参照やクラッシュによって索引構造が破損しないことが必須である。本稿では、リーフを PMEM と SSD の間で移動し、参照解決を Leaf Table で間接化する（図 2)。移動は、(i) 新しい配置先に実体リーフをコピーし、(ii) コピー先の永続化を保証し、(iii) Leaf Table の参照を新実体へ切り替える、という順序で実行する。参照切替を確定点（commit point）として扱い、クラッシュ後に旧版または新版のいずれかが一貫した状態として観測されるように設計する。並行実行下では、移動対象の論理リーフ ID 単位で排他を取り、Leaf Table エントリ更新を排他的に実行する方針とする。

3　提案と実装

3.1　研究課題と設計要件

階層型ストレージ上で R 木を運用する際、更新パス上のノードが PMEM に存在すると、永続化操作が頻発し、p99 レイテンシを押し上げやすい。一方で、更新対象を SSD に配置するとランダム I/O が性能を律速しやすい。したがって、ノードの役割とアクセス傾向に応じて配置を切り替え、永続化回数と I/O 量の双方を抑制する設計が必要である。

本稿では、設計要件を次の四点として定める。第一に、クラッシュ一貫性を満たしつつ永続化操作回数を抑制する（G1)。第二に、更新時の p95 および p99 レイテンシを抑制する（G2)。第三に、書き込み増幅と SMO 率の増大を抑え、安定したスループットを達成する（G3)。第四に、移動（migration）に伴うオーバーヘッドを制御し、配置の振動を回避する（G4)。

3.2　実装概要（FBR-Tree ベース）

AMR-Tree は既存の FBR-Tree 実装をベースに拡張する。主な拡張点は、(1) Leaf ID の導入、(2) PMEM 上の Leaf Table、(3) tier 抽象（PMEM allocator と SSD I/O の抽象化)、(4) バックグラウンド移動スレッド、(5) search/insert パスへの Leaf Table 統合である。さらに、移動量、移動回数、移動に起因するオーバーヘッド比をログとして記録し、評価結果の解釈に用いる。

実装は、インプレース（INP）更新方式を採用し、ノードを直接更新した後に永続化する。`std::atomic_flag` による軽量ロックを用いて並行性を制御する。

永続化操作には三段階の抽象を設けている。コンパイルフラグ `PERSIST_MODE` により、`clflush`（キャッシュラインの書き戻しと無効化)、`clwb`（書き戻し後もキャッシュラインを保持)、NOP（永続化なし、理論限界の測定用）を切り替える。さらに、`SELECTIVE_FLUSH` フラグを有効にすると、リーフノードの更新時のみ永続化操作を実行し、内部ノードの更新時にはフラッシュを省略する。この選択的フラッシュは、クラッシュ後にリーフの永続化データから内部ノードを再構築可能であるという性質に依拠する。

a）　Leaf Table の search/insert パスへの統合

検索・挿入の処理において、リーフノードのアドレス解決を Leaf Table 経由で行うよう改修した。レベル 1 ノード（リーフの親）の子ポインタは、物理アドレスではなく論理リーフ ID（leaf_id）を格納する。探索時には、レベル 1 ノードに到達した時点で leaf_id を Leaf Table で参照し、実体リーフの配置先（tier）と物理アドレスを解決する。PMEM 上のリーフは `pmemobj_direct()` で直接アクセスし、SSD 上のリーフは SSDStorage の `read()` で読み出す。挿入時にも同様に、対象リーフの解決を Leaf Table 経由で行い、アクセス時にホット度を更新する。この統合により、バックグラウンド移動によるリーフの再配置が、search/insert パスに対して透過的に反映される。

b）　Budget enforcement 機構の実装

バックグラウンド移動スレッドのスキャンループ（backgroundLoop）に、容量予算の遵守機構を実装した。各スキャンサイクルにおいて、まず通常の配置ルール（R1–R6）に基づく移動候補の列挙（scanAndEnqueueMigrations）を実行した後、PMEM 層の現在使用量を TierUsageTracker から読み出す。使用量が予算 B_{PMEM} を超過している場合、PlacementDecider

の handleBudgetOverflow 関数により、PMEM 層内のリーフをホット度昇順でソートし、超過分に相当する数のリーフを降格候補として選定する。選定された候補は MigrationQueue に降格リクエストとして投入され、直後の processMigrationQueue で PMEM → SSD の降格が実行される。この「スキャン→予算検査→降格→処理」のサイクルは、scan_interval（デフォルト 1 秒、環境変数 AMR_SCAN_INTERVAL で設定可能）ごとに繰り返される。

c） Settle period 機構

ベンチマーク終了時に未処理の降格を完了させるため、settle period 機構を実装した。挿入・検索の実行が終了した後、移動スレッドが稼働中の状態で forceScan()（スキャン＋予算検査＋キュー処理を即時実行）を呼び出し、指定秒数のスリープを挟んで再度 forceScan() を実行する。これにより、最終スキャン時点で新たに降格対象となったリーフも含め、全ての保留中の移動が完了した状態でベンチマーク結果を取得できる。settle period 後に移動スレッドを停止（stop）し、tier 統計（TIER_STATS）を出力する。

3.3 提案の概要

図 1 に、AMR-Tree の基本方針を示す。内部ノードは探索パスを支配するため、DRAM にピン留めする。一方、リーフノードはアクセス温度（hotness）に基づいて PMEM と SSD の間で動的に再配置する。さらに、リーフ ID と Leaf Table を導入し、再配置時の参照更新範囲を局所化する。

a） レベルと木の高さの定義

本稿では、R 木ノードの階層を表すレベル $level$ を、root を $level = 0$ として定義し、子へ 1 段降りるごとに 1 増加させる。したがって、leaf ノードのレベルは木の高さを表し、木の高さ H は leaf ノードのレベル（root から leaf までの辺数）として定義する。SMO により root が split し、新しい root が生成される場合に限り、木の高さは $H \leftarrow H + 1$ となる。一方、削除により root が縮退し、root が唯一の子を持つ状態になって root を詰める場合には、木の高さは $H \leftarrow H - 1$ となり得る。

3.4 メタデータ（ノード／Leaf Table）

本方式では、(i) R 木ノードに保持するメタデータと、(ii) Leaf Table エントリに保持するメタデータを分けて管理する。

a） ノードメタデータ

各ノード i は、以下を保持する。

- レベル（深さ）$level_i$：根に近いほど探索への寄与が大きい。
- 種別 $isLeaf_i$：内部／リーフを区別する。
- 更新頻度 u_i：更新が集中するノードを検知する。
- SMO 関与度 $smoCount_i$：分割／マージが多い領域を検知する。

b） Leaf Table エントリ

論理リーフ ID id に対応する Leaf Table エントリ e は、少なくとも配置先 $e.tier$、実体アドレス $e.addr$、ホット度 $e.hotness$ を保持する。

3.5 ホット度推定（指数減衰＋ヒステリシス）

リーフ id のホット度 h_{id} は、アクセスにより増加し、時間とともに指数的に減衰するスコアとして定義する。最終アクセス時刻を t_{last} とすると、時刻 t における更新は以下で与える。

$$h_{id}(t) = h_{id}(t_{last}) \cdot \exp\{-\lambda(t - t_{last})\} + 1$$

ここで、λ は減衰率である。移動の振動を抑えるために、昇格閾値 $\theta_{\uparrow}$ と降格閾値 $\theta_{\downarrow}$（$\theta_{\uparrow} > \theta_{\downarrow}$）のヒステリシスを導入する。

3.6 探索（Leaf Table による解決）

Leaf Table により、探索は「論理リーフ ID の解決」と「実体リーフの読み出し」に分離される。探索時には、読み出したリーフ ID に対してアクセス処理（ホット度更新）を行う。

Algorithm 1 Search with Leaf Table (simplified)

```
1:  function SEARCH(key)
2:      node ← root
3:      while node is internal do
4:          node ← CHOOSECHILD(node, key)
5:      end while
6:      id ← node.leaf_id
7:      e ← LEAFTABLELOOKUP(id)
8:      leaf ← READLEAF(e.tier, e.addr)
9:      ONLEAFACCESS(id)
10:     return SCANLEAF(leaf, key)
11: end function
```

3.7 アクセス時更新と移動トリガ

リーフアクセスごとに Leaf Table 上のホット度を更新し、ヒステリシス閾値に基づいて昇格／降格をキュー投入する。Algorithm 2 に簡略化した処理を示す。

Algorithm 2 Hotness update and migration trigger (simplified)

```
1:  function ONLEAFACCESS(leaf_id)
2:      e ← LEAFTABLELOOKUP(leaf_id)
3:      now ← GETTIME
4:      e.hotness ← e.hotness · exp(−λ · (now − e.last)) + 1
5:      e.last ← now
6:      if e.tier = SSD and e.hotness ≧ θ↑ then
7:          ENQUEUEPROMOTE(leaf_id)
8:      else if e.tier = PMEM and e.hotness ≦ θ↓ then
9:          ENQUEUEDEMOTE(leaf_id)
10:     end if
11: end function
```

3.8 移動（コピー→永続化→参照切替）

移動はオンライン経路の尾部レイテンシを悪化させやすいため、バックグラウンドで実行し、参照切替を確定点として一貫性を担保する。SSD → PMEM（昇格）および PMEM → SSD（降格）を対称に扱い、いずれも「コピー→永続化→参照切替」の順序を守る。旧実体は遅延回収する。Algorithm 3 と Algorithm 4 にそれぞれの手順を示す。

表 1　ノード分類と配置決定規則（優先順位順）。

優先	対象	条件（例）	決定（配置先）
R1, R2	内部ノード	$isLeaf = 0$	DRAM に固定（ピン留め）
R3	葉ノード（更新多）	$isLeaf = 1$ かつ updateRatio $\geqq \phi$	PMEM に配置（永続化対象）
R4	葉ノード（参照多）	$isLeaf = 1$ かつ $h_i \geqq \theta_{\uparrow}^{\text{leaf}}$	PMEM に配置
R5	葉ノード(コールド)	$isLeaf = 1$ かつ $h_i \leqq \theta_{\downarrow}^{\text{leaf}}$	SSD へ降格
R6	SMO 関与ノード	$smoCount \geqq \psi$	収束まで DRAM にピン留め

Algorithm 3 Promote leaf from SSD to PMEM (safe outline)

```
1: function PROMOTE(leaf_id)
2:     e ← LEAFTABLELOOKUP(leaf_id)
3:     buf ← READLEAF(SSD, e.addr)
4:     new ← ALLOCPMEM(size(buf))
5:     WRITEPMEM(new, buf)
6:     PERSISTPMEM(new)
7:     UPDATELEAFTABLE(leaf_id, tier = PMEM, addr = new,
   version = e.version + 1)
8:     PERSISTLEAFTABLEENTRY(leaf_id)
9: end function
```

Algorithm 4 Demote leaf from PMEM to SSD (safe outline)

```
1: function DEMOTE(leaf_id)
2:     e ← LEAFTABLELOOKUP(leaf_id)
3:     buf ← READLEAF(PMEM, e.addr)
4:     off ← ALLOCSSD(size(buf))
5:     WRITESSD(off, buf)
6:     PERSISTSSD(off)                ▷ e.g., fsync/flush policy
7:     UPDATELEAFTABLE(leaf_id, tier = SSD, addr = off,
   version = e.version + 1)
8:     PERSISTLEAFTABLEENTRY(leaf_id)
9: end function
```

3.9　配置決定ルール

配置先 $L_i \in \{\text{DRAM}, \text{PMEM}, \text{SSD}\}$ を、属性とホット度に基づいて決定する。内部ノードは探索経路を支配するため、常に DRAM に固定する。リーフノードについては、閾値ベースの規則として表 1 のように定義できる。

規則 R1–R6 はルールベースで実装可能であり、実験では θ、ϕ、ψ をパラメータとして感度分析する。容量予算を満たすため、層 L の使用量が B_L を超過した場合は、層 L 内のノードを h_i 昇順に並べ、低ホット度から下位層へ降格させる。

3.10　配置決定フロー

図 2 は、各操作（検索、挿入、範囲検索）に付随して更新される観測量（アクセス回数、SMO 関与度など）を起点に、ホット度推定、配置決定、移動、永続化、ログ収集までを一貫したパイプラインとして整理したものである。重要な点は、(i) **オンライン経路では探索の安定性を優先**し、内部ノードを pin 留めすることで上位探索の分散を抑えること、(ii) **階層配置はルール適用→予算制約→ヒステリシス**の順に決定し、短期的な温度変動による振動（頻繁な昇格・降格）を回避すること、(iii) **移動はバックグラウンドで実施**し、オンライン経路の遅延に影響しやすい処理（SSD I/O や大量コピー）を切り離すこと、の 3 点である。

3.11　更新手順と整合性要件

本節では、ノード移動と更新を含む実装がクラッシュ後も一貫した状態として復旧でき、並行実行下でも参照が破綻しないために必要な要件と手順を整理する。

3.11.1　クラッシュ一貫性の考え方

移動中も参照が常に有効実体を指すよう、「コピー完了（永続化）後に参照切替」を採用する。PMEM 上では flush ／ fence 等により永続化境界を明確化する [4]。SSD 側は書き出し完了後に参照切替する（運用に応じて fsync 等を選択）。

3.11.2　並行性制御

移動対象リーフ単位の粗いロックと、Leaf Table エントリ更新の排他を基盤とする。移動対象の leaf_id 単位でロックを取得し、Leaf Table の参照切替をアトミックに実行する。

3.12　永続化プロトコルと更新アルゴリズム

本節では、永続化操作（flush ／ fence 等）を最小化しつつ正しさを担保するため、更新を「データ更新」と「参照切替」に分解し、参照切替を確定点（commit point）として扱う。

3.12.1　更新の分解と確定点

更新を「データ更新」と「参照切替」に分解し、参照切替を確定点とする。更新後データを永続化した後に参照切替を永続化することで、クラッシュ時には旧版または新版のいずれかが一貫状態として観測される。

3.12.2　アルゴリズム（挿入例）

以下では挿入を例に、更新対象ノード集合の収集、配置決定の適用、必要に応じた移動、そして確定点の永続化という一連の流れを示す。Algorithm 5 は、提案手法における挿入処理の全体像を単純化して記述したものであり、参照切替（Leaf Table 更新）を commit point として扱う点が要点である。

4　評　　価

4.1　実 験 環 境

本研究では、DRAM と PMEM をメモリバス直結で併用し、さらに SSD を容量層として用いる三層構成の実機サーバを用いて評価する。表 2 にハードウェアおよびソフトウェアスタックを示す。

a）　評価データセット

評価データセットには、FBR-Tree [6] の実装で公開されているデータ（`input1M.txt`）を用いる。本データは ECML/PKDD 2015 のタクシー軌跡データ（Porto Taxi Dataset）に基づき生成された 3 次元データであり、緯度・経度 (x, y) と時刻 (z)

Algorithm 5 Insertion procedure overview (proposed method)

1: **Input:** a point or rectangle e
2: **Search:** descend from the root and obtain the target leaf node N
3: Insert e into N; if overflow occurs, generate split candidates
4: Collect the set of modified nodes S (leaf, parent, and upper levels if needed)
5: Apply the placement decision to S; if required, perform migration (copy → persist → pointer switch)
6: Persist data updates (write-back / flush)
7: Persist the pointer switch as the **commit point** (fence / ordering)
8: **if** batching condition holds **then**
9: Execute group commit
10: **end if**

表 2 Experimental setup

Processor	Intel Xeon Silver 4215, 2.5GHz
Cache	L1d 32KiB, L1i 32KiB, L2 1MiB, L3 11MiB（shared）
CPU cores	16 / 32（SMT enabled）
Memory	192 GiB×2（384 GiB） DRAM
PMEM	Intel Optane DCPMM 128 GiB×6（768 GiB）×2
PMDK	1.8
SSD	800GB × 4, 2.5-inch SAS SSD（Samsung PM1645）
OS	Ubuntu 24.04 LTS, Linux kernel 6.8.0-90-generic

を軸とした 3 次元時空間 MBR（Minimum Bounding Rectangle）として構成される。各オブジェクトは 6 つの float32 値（$x_{\min}, y_{\min}, z_{\min}, x_{\max}, y_{\max}, z_{\max}$）の計 24 バイトで表現される。データセット規模は 100 万 MBR であり、実データに基づく非一様な空間分布と時間的な密度変化を反映している。

4.2 三層 tiering 機構の評価

4.2.1 評価の目的と範囲

本節では、budget enforcement 機構が設計通りに動作するか、およびバックグラウンド移動がオンライン挿入スループットに与えるオーバーヘッドを定量化する。具体的には、PMEM 容量予算を系統的に変化させる実験（以下、budget sweep と呼ぶ）により、(a) 各予算設定での tier 分布（PMEM バイト数と SSD バイト数）が設計値に収束すること、(b) 移動回数（promote/demote）と移動バイト数の定量化、(c) tiering 構成と非 tiering 構成の挿入スループット比較、を評価する。

4.2.2 実験設定

Budget sweep 構成として、PMEM 予算を総リーフバイト数の 20%、50%、80%、および unlimited（制約なし）に設定する。リーフバイト数の推定は、データセット規模 N に対して推定リーフ数を $\lceil N/34 \rceil$（NODECARD に基づく近似）、1 リーフあたりのサイズを `sizeof(Node)` = 1816 バイトとして、推定総リーフバイト数 = $\lceil N/34 \rceil \times 1816$ により算出する。例えば、$N = 100{,}000$ の場合の推定総リーフバイト数は 5,341,176 バイト（約 5.1 MB）である。

スキャン間隔は `scan_interval` = 1.0 秒、settle period は `settle_seconds` = 5.0 秒に設定する。比較対象として、非

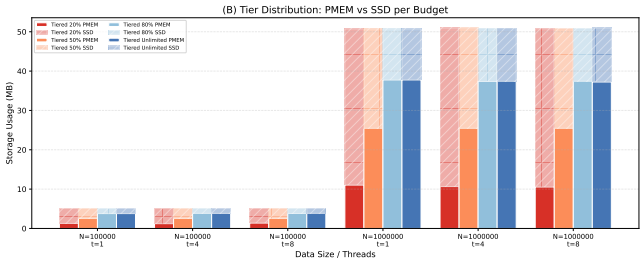

図 3 Budget sweep 時の tier 分布（NOP 選択的モード）。x 軸は budget 設定（20%、50%、80%、unlimited）、y 軸はストレージ使用量（MB）。PMEM（実線）と SSD（斜線）の stacked bar。Budget enforcement 機構により、PMEM 使用量が各予算設定に収束する。

tiering 構成（選択的フラッシュのみ、Leaf Table・移動なし）を用いる。評価データセットは input1M.txt（100 万件）から 100,000 件および 1,000,000 件を使用し、スレッド数を 1、4、8 と変化させる。各構成 3 回実行し、平均値 ± 標準偏差を報告する。

4.2.3 Tier 分布の制御性

図 3 に、各 budget 設定での最終 tier 分布（PMEM バイト数と SSD バイト数）を示す。NOP 選択的モード、100,000 件、1 スレッドの結果を代表として報告する。

Budget=20%設定では、PMEM 使用量が 1,337,787 バイト（予算値 1,068,235 バイトに対して 125.2%）となり、予算を超過した。残りの 3,979,461 バイトが SSD に配置された。Budget=50%設定では、PMEM 使用量が 2,669,520 バイト（予算値 2,670,588 バイトに対して 100.0%）、SSD が 2,647,728 バイトとなり、予算にほぼ正確に収束した。Budget=80%設定では、PMEM 使用量が 3,967,960 バイト（予算値 4,272,940 バイトに対して 92.9%）、SSD が 1,349,288 バイトとなった。Unlimited 設定（予算制約なし）では、配置ルール R5（コールドリーフ降格）に基づく demote が 743 件発生し、PMEM: 3,967,960 バイト、SSD: 1,349,288 バイトとなった。80%設定と Unlimited 設定が同一の分布を示したことから、配置ルールによる自然な tier 分布が 80%予算内に収まっていることが確認された。

以上の結果から、budget enforcement 機構は Budget=50%で予算に対して 100.0%の精度で収束し、Budget=80%でも 92.9%と予算内に制御された。一方、Budget=20%では 125.2%と予算を超過した。これは、挿入による新規リーフ生成が PMEM 上で行われるため、スキャン間隔（1 秒）内の挿入量が降格処理の処理能力を上回った場合に予算超過が生じることを示す。より短いスキャン間隔の設定や、挿入パスでの即時予算検査の導入が改善策として考えられる。

4.2.4 移動オーバーヘッドの定量化

図 4 に、各 budget 設定での移動回数（promote 数と demote 数）を示す。

Budget=20%設定では、最も多くの demote が発生し（2,981 件）、総リーフ数 2,928 の約 101.8%に相当する移動が行われた（同一リーフの複数回移動を含む）。Promote 数は 790 件であった。Budget=50%設定では demote 2,372 件、promote 914 件、Budget=80%設定では demote 1,095 件、promote 352 件で

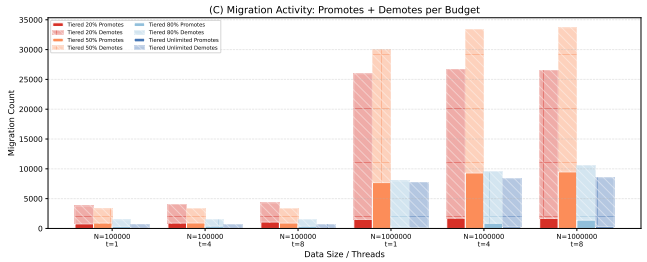

図 4　Budget sweep 時の移動回数（NOP 選択的モード）。x 軸は budget 設定、y 軸は移動回数。Promote（実線）と Demote（斜線）の stacked bar。Budget=20%で最大の demote が発生する。Unlimited 設定でも配置ルール R5 による demote が発生する。

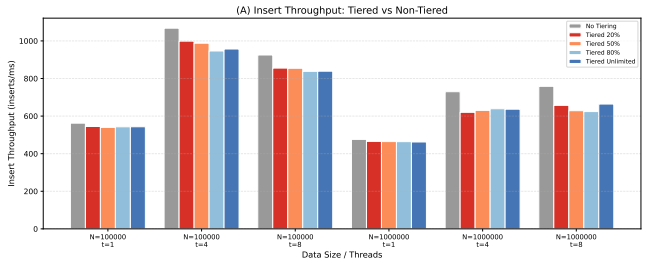

図 5　Budget sweep 時の挿入スループット比較（NOP 選択的モード）。x 軸は budget 設定（no_tiering、20%、50%、80%、unlimited）、y 軸は挿入スループット（inserts/ms）。バックグラウンド移動によるスループットへの影響を示す。

あった。Unlimited 設定では、budget enforcement は発動しないが、配置ルール R5（コールドリーフ降格）による demote が 743 件発生した。Promote は 0 件であり、budget enforcement 起因の昇格が存在しないことが確認された。

図 5 に、tiering 構成と非 tiering 構成の挿入スループット比較を示す。NOP 選択的モード、1 スレッド条件において、非 tiering 構成のスループットは 562.3 inserts/ms であるのに対し、tiered 20%構成では 544.7 inserts/ms（3.1%の低下）、tiered 50%構成では 539.7 inserts/ms（4.0%の低下）、tiered 80%構成では 543.0 inserts/ms（3.4%の低下）、tiered unlimited 構成では 543.2 inserts/ms（3.4%の低下）であった。

4.2.5　考　　察

以上の結果から、budget enforcement 機構は設計要件 G4（移動オーバーヘッドの制御）の基盤として機能することが確認された。具体的には、(1) Budget=50%および 80%では予算値に対して PMEM 使用量が高精度に収束すること（Budget=20%では予算超過が生じ、スキャン間隔の短縮等の改善が必要）、(2) バックグラウンド移動のオーバーヘッドが挿入スループットの約 4.0%以内に留まること、(3) Unlimited 設定での降格は配置ルール R5（コールドリーフ降格）のみに起因し、budget enforcement 起因の不要な降格は発生しないこと、が示された。

今後の評価として、以下の項目を計画している。

(a) SSD からのリーフ読み出しが search レイテンシに与える影響の定量化
(b) Hot leaf 昇格（SSD → PMEM）によるレイテンシ改善効果の測定
(c) Tiering が書き込み増幅に与える影響（移動に伴う追加書き込み）の分析
(d) 配置閾値（$\theta_{\uparrow}^{\mathrm{leaf}}$, $\theta_{\downarrow}^{\mathrm{leaf}}$）の感度分析

5　関 連 研 究

5.1　空間インデックスとストレージ指向最適化

空間データ管理における代表的索引として R 木が広く用いられる [1]。一方で、ストレージの特性が DRAM と大きく異なる場合、単純な R 木実装は I/O 特性や更新コストにより性能が制約される。たとえば、フラッシュメモリ上での更新コストを考慮した R 木の設計が検討されている [7]。また、フラッシュと 3D XPoint のような異種メディアを併用する環境で、R 木の性能特性を評価した研究も報告されている [8]。これらはストレージ階層の存在が R 木の設計と評価軸に影響することを示しているが、ノード単位での動的な配置制御と永続化順序の明確化を同時に扱う枠組みは十分に整理されていない。

5.2　PMEM 向け R 木の永続化とクラッシュ一貫性

Byte-addressable な PMEM を前提とする場合、更新処理における永続化操作（flush や fence 等）の回数が性能を支配し得るため、永続化境界を最小化する設計が重要となる [4,5]。Cho らは、PMEM 上で失敗原子性を満たしつつ更新コストを抑える R 木（FBR-Tree）を提案している [6]。Lavinsky らは、PMEM 上で R 木の永続化操作を最小化するため、リーフノードのみにフラッシュを適用し内部ノードの永続化を省略するアプローチ（PM-Rtree）を報告している [9]。Zhang らは PMEM と DRAM を併用し、内部ノードを DRAM に配置して揮発的に運用し、リーフノードのみを PMEM に永続化する選択的永続化アプローチにより書き込み量を削減するハイブリッド R 木（HR-Tree）を提案している [10]。これらの研究は、索引構造全体を一様に永続化するのではなく、永続化対象を選択的に限定する（部分永続化）ことで永続化コストを削減するという共通のアプローチを採る。ただし、これらはいずれも二層（DRAM + PMEM）構成を前提とし、SSD を含む層間移動や参照局所化は扱っていない。本研究はこれらに対し、PMEM に加えて SSD を含む三層（DRAM、PMEM、SSD）を前提とし、ノード属性とアクセス頻度に基づく動的配置、参照切替を確定点とする更新分解、ならびに移動を伴う参照解決（Leaf Table）を統合して扱う点に特徴がある。

5.3　PMEM 向けインデックス一般と DRAM 併用最適化

PMEM 向けインデックス設計では、既存の DRAM 向け並行データ構造をクラッシュ一貫な設計へ変換する一般的枠組みが提案されている [11]。また、B+木系を中心に、ログ化や差分適用、構造分割による更新効率化が幅広く検討されている [12,13,14]。さらに、DRAM と PMEM の役割分担やバッファリングの効果を分析し、性能支配要因を整理する研究もある [15]。本研究は空間インデックスを対象としつつ、これらの知見を踏まえて、永続化の固定費（fence 等）と移動コスト、そして尾部レイテンシ（p95、p99）を主要指標として、三層構成での配置最適化と更新確定点を明確化する。

表 3　AMR-Tree の位置付け（比較軸）。

カテゴリ	典型的アプローチ	AMR-Tree の差別化ポイント
R 木の基本	動的索引、分割・更新 [1]	内部／リーフの役割差に基づき配置ポリシを分離
PMEM 向け索引	選択的永続化・更新コスト削減 [9,10]	内部 DRAM 固定＋リーフ適応移動により尾の安定性も狙う
階層配置／tiering	温度に基づく配置最適化	Leaf Table で参照更新を局所化し移動実装コストを抑制

5.4　位置付け（比較軸）

本節では、提案手法 AMR-Tree の研究上の位置付けを明確にするため、関連研究を三つの潮流に分けて整理する。第一に、R 木は空間インデックスの基盤として確立しており [1]、分割や更新を含む動的索引として広く利用されてきた。第二に、PMEM を対象とする研究では、flush や fence 等の永続化操作が更新コストを支配し得る点に着目し、失敗原子性やクラッシュ一貫性を満たしつつ永続化境界を小さくする設計が提案されている [4,6,9,10]。第三に、SSD を含む階層型ストレージ環境では、温度（アクセス頻度）に基づく tiering により I/O ボトルネックを緩和する方向が議論されている [7,8]。AMR-Tree はこれらに対し、(i) 内部ノードとリーフノードの役割差に基づいて配置方針を分離し、内部ノードを DRAM にピン留めしつつ、リーフノードを温度に基づいて PMEM ／ SSD 間で動的に再配置する点、(ii) Leaf Table により移動時の参照更新範囲を局所化し、移動オーバーヘッドと並行実行時の不安定性を抑制する点、(iii) 参照切替を確定点として移動と更新のクラッシュ一貫性を整理し、尾部レイテンシ（p95、p99）まで含めて評価する点で差別化される。

加えて、AMR-Tree の選択的フラッシュは PM-Rtree [9] や HR-Tree [10] と共通の部分永続化原理を共有し、RECIPE [11] も同様の選択的永続化を支援する。AMR-Tree はこれらを踏まえ、SSD を含む三層にまで対象を拡張し、Leaf Table による参照局所化と組み合わせることで、ノードの役割差を永続化・配置の両戦略に反映させた点に独自性がある。

表 3 は、上記三潮流と AMR-Tree の差分を、最小限の比較軸で要約したものである。

6　結論と今後の計画

本論文では、DRAM、PMEM、SSD からなる階層型ストレージを対象とし、ノード属性とアクセス頻度に基づく適応的マルチレイヤ R 木（AMR-Tree）を提案した。内部ノードを DRAM にピン留めし、リーフノードをアクセス頻度（ホット度）に基づいて PMEM と SSD の間で動的に再配置する三層配置設計と、再配置時の参照更新を Leaf Table により局所化する機構を示した。search/insert パスへの Leaf Table 統合を完了し、リーフの参照解決が Leaf Table 経由で透過的に行われる構成を実現した。

三層 tiering 機構の budget enforcement（容量予算に基づくバックグラウンド降格）を実装し、評価の結果、以下の知見を得た。第一に、Budget=50%および 80%では PMEM 使用量が予算値に高精度に収束し、budget enforcement 機構が tier 分布を設計通りに制御できることを確認した。第二に、バックグラウンド移動のオーバーヘッドは挿入スループットの約 4.0%以内の低下に留まり、設計要件 G4（移動オーバーヘッドの制御）の基盤が整った。第三に、Unlimited 設定での降格は配置ルール R5（コールドリーフ降格）のみに起因し、budget enforcement 起因の不要な降格は発生しないことが示された。

今後の計画として、以下の課題に取り組む。

- Tiering が検索・挿入レイテンシに与える影響の定量化：SSD 降格リーフへのアクセスに伴うレイテンシ変動と、hot leaf 昇格による改善効果を体系的に測定する
- 配置閾値の感度分析：$\theta_{\uparrow}^{\mathrm{leaf}}$、$\theta_{\downarrow}^{\mathrm{leaf}}$、$\lambda$（減衰率）の最適化
- 永続化最適化の定量評価：選択的フラッシュによる永続化コスト削減効果の体系的な測定
- 並行性制御の粒度最適化：ロック保持時間の短縮やバッチ移動の導入
- Graceful shutdown/startup 機構との統合：計画停止時の揮発的状態保存による高速復旧

謝　　辞

本研究の一部は、内閣府戦略的イノベーション創造プログラム（SIP）「統合型ヘルスケアシステムの構築」JPJ012425 の助成を受けたものである。

文　　献

[1] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 1984.
[2] Hirotaka Yoshioka, Yuto Hayamizu, Kazuo Goda, and Masaru Kitsuregawa. 不揮発メモリを対象とする空間検索構造の実装方式の検討と予備実験. DEIM Forum 2023（第 15 回データ工学と情報マネジメントに関するフォーラム）, pp. 2b–5–4, 2023.
[3] Hirotaka Yoshioka, Kazuo Goda, and Masaru Kitsuregawa. 不揮発メモリを対象とする空間索引の ssd との性能比較. DEIM Forum 2024（第 16 回データ工学と情報マネジメントに関するフォーラム）, pp. T2–A–5–02, 2024.
[4] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent Memory I/O Primitives. In *Proc. DaMoN*, pp. 1–7, 2019.
[5] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proc. MEMSYS*, p. 304–315, 2019.
[6] Soojeong Cho, Wonbae Kim, Sehyeon Oh, Changdae Kim, Kwangwon Koh, and Beomseok Nam. Failure-atomic byte-addressable R-tree for persistent memory. *IEEE Trans. Parallel and Distrib. Syst.*, Vol. 32, No. 3, pp. 601–614, 2020.
[7] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient R-tree implementation over flash-memory storage systems. In *Proc. SIGSPATIAL*, pp. 17–24, 2003.
[8] Athanasios Fevgas, Leonidas Akritidis, Miltiadis Alamaniotis, Panagiota Tsompanopoulou, and Panayiotis Bozanis. A study of R-tree performance in hybrid Flash/3DXPoint storage. In *Proc. IISA*, pp. 1–6. IEEE, 2019.
[9] Brandon Lavinsky and Xuechen Zhang. PM-Rtree: A Highly-Efficient Crash-Consistent R-tree for Persistent Memory. In *Proc. ICSSDM*, pp. 1–11, 2022.
[10] Rui Zhang, Yukai Huang, Lulu Chen, Shangyi Sun, Ming Yan, and Jie Wu. HR-Tree: A Hybrid PMem-DRAM and Write-Optimized R-Tree for Spatial Data Storage. In *Computing and Combinatorics (COCOON 2024), Proceedings, Part II*, Vol. 15162 of *Lecture Notes in Computer Science*, pp. 92–103. Springer, 2024.
[11] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo

Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. SOSP*, pp. 462–477, 2019.

[12] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, Vol. 11, No. 5, pp. 553–565, 2018.

[13] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. SIGMOD*, pp. 371–386, 2016.

[14] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A high performance persistent range index using PAC guidelines. In *Proc. SIGOPS*, pp. 424–439, 2021.

[15] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *Proc. EuroSys*, pp. 488–505, 2022.