DEIM Forum 2023 2b-5-4

不揮発メモリを対象とする空間索引構造の実装方式の検討と予備実験

吉岡　弘隆†　　早水　悠登††　　合田　和生††　　喜連川　優††

† 東京大学　大学院情報理工学系研究科
〒 153–8505 東京都目黒区駒場 4–6–1
†† 東京大学　生産技術研究所
〒 153–8505 東京都目黒区駒場 4–6–1
E-mail: †{hyoshiok,haya,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし
近年，不揮発メモリと呼ばれるバイト単位アクセス可能なメモリが出荷されている．主記憶メモリに利用される DRAM のような高速性とストレージのような永続性を備え持つデバイスとして注目を浴びている．不揮発メモリの基本的な動作特性（レイテンシー，スループット等）やデータベースへの適応事例などが紹介されてきている．また，B+木やハッシュ索引への実装についての報告も徐々にされてきている．しかしながら不揮発メモリを対象とする空間検索構造の実装方式の報告については FBR [5]，PMR [14] などに見られる程度で，十分とは言えない．そこで本研究では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を行った．
キーワード　データベース技術，不揮発メモリ，NVM，空間索引構造

1　はじめに

不揮発メモリと呼ばれるバイト単位アクセス可能なメモリが近年出荷されている [26], [28].

その特徴をまとめると，

- 性能（スループット，レイテンシなど）は SSD (NAND Flash)，HDD より高い，DRAM より低い，
- DRAM と異なり永続性がある（電源が遮断されても情報は喪失しない）
- 記憶容量は DRAM より大きい
- バイト単位のアクセスが可能である（DRAM と同様）
- CPU cache coherent である（DRAM と同様）
- DMA （Direct Memory Access）と RDMA （Remote Direct Memory Access）をサポートする（DRAM と同様），
- ユーザ空間でアクセスできる．SSD や HDD のようにアクセスするために system call は必要がない．アクセスするためにカーネルコード，ページキャッシュ，割込などは必要ないなどがある．

このような優れた特性を持つため，今後広く利用されることが期待される．

一方で，従来の揮発性メモリと違い，不揮発であるという特徴はプログラミングモデルの変更を余儀なくする．そのため，その動作特性の違いは単にスループットやレイテンシーだけでなく，アルゴリズムやデータ構造のあり方にも影響を与える．

不揮発メモリは従来のメモリ同様にバイト単位でアクセス可能であるが，CPU から見て不揮発メモリへのアクセスは通常キャッシュ経由でなされる．図 1 に示すようにキャッシュは揮発性なので永続性を必要とする場合は適宜キャッシュ内容を flush する必要がある．Intel x86 の場合，CLFLUSH 命令などによって明示的に行う．flush 関連の命令は CLFLUSH の他，CLWB，CLFLUSHOPT があり，キャッシュをバイパスする Non-temporal 命令もある．また，メモリへの書き込みを同期するために fence が必要になる．

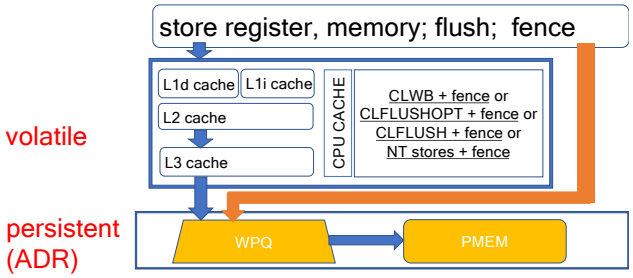

図 1　CPU とキャッシュ，メモリの関係

また記憶の永続性を担保する動作についてはファイルシステムでは本質的な問題だが，メインメモリに於いては永続性は存在しなかったので一般的な問題ではなかった．例えばクラッシュコンシステンシーという概念は従来は主にハードディスクなど永続性を持つデバイスにおいて議論されてきたが，揮発性メモリではクラッシュした時点でそもそも永続性を保証しないので一般的には問題とならなかった．しかし不揮発メモリに於いてはストア命令によって記憶が永続化されるので，ディスクへの書き込みと同様な問題が生じる．ここで言うクラッシュコンシステンシーとは，情報の変更を首尾一貫した状態で行うことを指す．

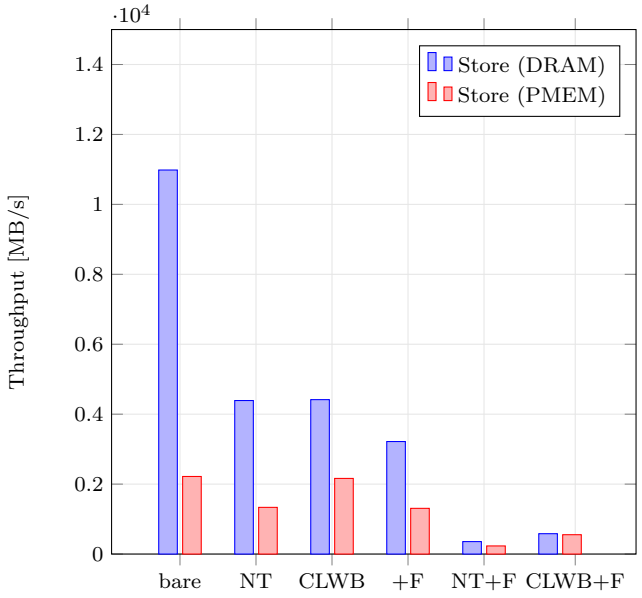

図 2　64 byte sequenial store, bare, NT, CLWB はデフォルトの store 命令および Non-temporal 命令，CLWB 命令を追加したもの，＋ F はさらに FENCE 命令を追加したものを示す [33]

不揮発メモリへのストア命令は必ずしも同期的にメディアに書き込むことを保証しない，通常はキャッシュにのみ書き込み，非同期にメモリ (メディア) に書き込む．そのため，プログラムのストアの順序とメモリへの書き込み順は必ずしも等しくならず，書き込み順の一貫性を担保する仕組みが必要になる．

メモリレベルでの永続性を担保する命令（FLUSH 命令）や書き込み順の同期を取る命令（FENCE 命令）などをプログラマが意識しないと不揮発メモリを利用した正しいプログラム（クラッシュした時点での整合性を持つ）は構築できない．

またこのような各種の flush や fence が性能に与える影響は必ずしも自明ではない．そこでわれわれは，先行研究として pmmeter というマイクロベンチマークを作成し，flush 命令と同期命令が与えるオーバーヘッドを評価した ([33])．64 バイトのシーケンシャルアクセスのスループットは DRAM で 10.9GB/s，PMEM で 2.2GB/s だが flush 命令を追加すると 4.4GB/s と 2.1GB/s にそれぞれ低下する．さらに fence 命令を追加すると 0.58GB/s と 0.55GB となる (図 2 参照)．flush と fence は大きなコストが発生する．

そのような特性を踏まえ，従来の広く利用されている索引技術を不揮発メモリへ適用する研究が近年活発に行われている．

例えば B+木索引はメモリアクセスとディスクアクセスのレイテンシの大きな違いを前提にファイルをディスクに格納した場合に最適化された構造で，検索時にディスクアクセス回数を削減するようなアルゴリズムになっている．

不揮発メモリ向けの B+木索引 [10], [12], [16] ，ハッシュ索引 [11]，ファイルシステム [25] ，データベースエンジン，基本的な動作特性 [24], [30] などの報告はあるが，空間索引構造についてはまだ十分に検討がされていない．そこで本研究では，不揮発メモリを対象とする空間検索構造の実装方式を検討し予備実験をおこなった．

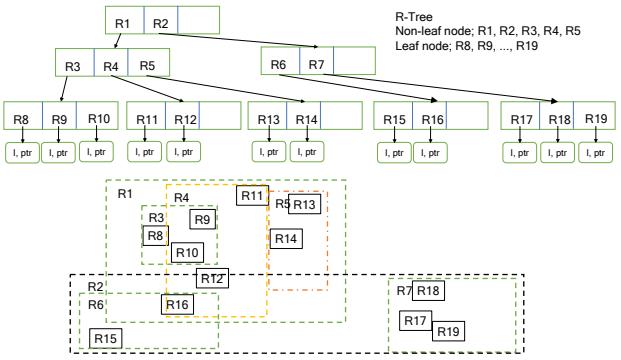

図 3　R-Tree の例

以下に論文の構成をしめす．第 2 章でまず，空間索引構造を紹介し，それを不揮発メモリに適応するときの課題を整理する．そして，第 3 章で今回行った実装方式の検討についてふれ，予備実験を紹介する．その実験で明らかになった問題を議論する．第 4 章で関連研究を紹介し，最後にまとめと今後の課題を述べる．

2　空間索引構造

2.1　R-Tree

多次元データを高速に検索する方法として空間索引構造が知られている．その例としてここでは R-Tree を取り上げる (図 3 参照)．R-Tree [8] は Guttman によって提案された空間索引構造であり，次のような特徴を持つ．

- R-tree の leaf ノードは (I, tuple 識別子) という形式で，I は多次元のオブジェクト，tuple 識別子は当該オブジェクトを一意に識別する．
- leaf ノードでない場合 (non-leaf ノード) は，(I, 子ノードのポインタ) という形式である．
- 子ノードは leaf ノードか non-leaf ノードである．
- 各 leaf ノードはルートノードでなければ m から M 個の索引レコードを持つ．
- leaf ノードの各索引レコードは n 次元の最小矩形である．
- ルートノードは leaf ノードでないかぎり 2 個の子ノードを持つ．
- 全ての leaf は等しいレベルである（leaf ノードまでの高さは等しい）

図 3 では，ルートノードが R1 と R2 の子ノードを持ち，R1 は R3,R4,R5 の子ノードを持つ，R3 は同様に R8,R9,R10 を持ち，それらは leaf ノードになる．leaf ノードは多次元オブジェクトデータと tuple 識別子を持つ．B+tree と同様に，leaf ノードにデータを格納していくと，leaf までの高さを一定に保つために適宜分割などが発生する．

Rtree そのものについては, [21] という参考書があり，Guttaman の提案以降の主要な提案が網羅されている．ただし出版年が 2006 年なので，近年のハードウェア動向については考慮されていない．

表 1　実験環境

CPU model	Intel Xeon Silver, 2.5GHz
	8 core, 2 socket
No. of nodes	2
Cache	L1d 32KiB, L1i 32 KiB,
	L2 1 MiB, L3 11 MiB (shared)
DRAM	32 GiB * 12 (384 GiB)
DCPMM	128 GiB *12 (1536 GiB)
OS	CentOS 7.7.1908, linux kernel 3.10
PMDK	1.8

2.2　不揮発メモリのプログラミングモデル

不揮発メモリをストレージデバイスとして利用するときにはいくつかの注意すべき点がある．従来のプログラミングで永続性を確保したい場合は通常ファイルシステムないしデータベースなどを利用しておこなっているが，不揮発メモリの場合，ファイルシステムなどの利用は不必要となるが下記のようなプログラミングモデルとなるので従来型プログラムを変更する必要がある．

2.2.1　明示的な情報の永続化

不揮発メモリは情報をメモリにストアすると永続化されるが，そのタイミングは一般には非同期であり，通常はキャッシュに書き込まれるだけでメディアへの書き込みタイミングはプログラマは意識しない (図 1 参照)．また書き込む順も不定である．そのため，明示的にキャッシュからフラッシュし，書き込み順を同期する必要がある．書き込み順を指定しないと，一貫性を保てない状態が発生する場合がある．例えば，あるページへのポインタがあった場合，あるページへの書き込みが終了する前に，そのページへのポインタを設定した場合，まだ情報が書き込まれていない場所への参照が発生してしまい，そのタイミングでシステムがクラッシュすると一貫性が保持されないままになる．最近のプロセッサは out of order で実行するために，プログラムを書いた順番に実行されるとは限らないので注意が必要である．Intel Optane DC Persistent Memory Module (以下 Intel DCPMM と称する) の場合，命令ごとの同期をとるために，明示的な FENCE 命令が必要になる．

2.2.2　アトミック書き込みは最大 8 バイト

Intel DCPMM がアトミックに書き込めるのは最大 8 バイトである．それ以上大きな塊での書き込みはアトミックに行われない．そのため書き込み途中でシステムがクラッシュすると一貫性が保持されない．R-Tree の木構造の変更はアトミックに行う必要がある．

上記のような問題に対処するため，不揮発メモリを対象とした B+Tree や Hash などで様々な提案がなされている．(主な s サーベイとして B+木索引では [10], [12], [16] ，ハッシュ索引 [11] などがある)．

空間索引については, [5] (以下 FBR と称する) および [14] (以下 PMR と称する) などが上記の問題に対応している．

多次元空間索引はデータベース分野のみならずコンピュータグラフィックス，位置情報システム，高性能システム (High Performance Computing) など様々な分野での応用が期待される．科学技術計算分野でのデータ量の爆発は不揮発メモリの応用分野としても期待されている．

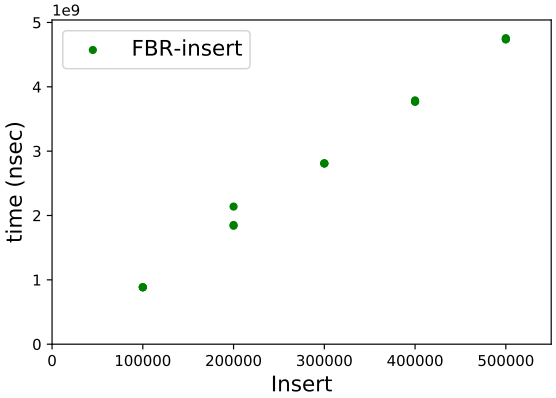

図 4　FBR10 万件から 50 万件データの挿入，縦軸は時間

2.2.3　クラッシュコンシステンシの実装

前述したように不揮発メモリにおいては単なるメモリへの転送だけでは一貫した状態を保持できない．なんらかの方法でクラッシュコンシステンシを確保する必要がある．

R-Tree の場合，B+Tree と同様にページへのレコードの追加時にページ分割が発生する場合があり，それは leaf ノードから root ノードまで波及する場合がある．木構造を変更するためにアトミックに変更する必要があり，その一貫性制御が必要となる．

先行研究 FBR [5] ではクラッシュコンシステンシを確保するために，mutex lock を利用した実装を評価した．一方，PMR [14] では mutex lock ではなく不揮発メモリ向け lock free アルゴリズム (以下 PMwCAS と称する) [29] を利用した実装を評価した．また複数バイトへのアトミックな CAS (以下 MwCAS と称する) [9] を利用した．

FBR は木構造の変更は木構造の root ノードに対するジャイアントロックを mutex lock を排他的に取得することで行っている．PMR はそれを PMwCAS と MwCAS で実装した．

3　実験評価

3.1　ハードウェア，ソフトウェア構成

次のような環境で実験した (表 1)．CPU は Intel Xeon Silver, 2.5Ghz, 8 core, 2 socket, Numa 2 ノード，OS は CentOS 7.7 Linux Kernel 3.10, PMDK 1.8 を利用した．

3.2　実験結果

まず，ランダムに生成した 10 万件から 50 万件の空間データを挿入する時間を計測した．シングルスレッドで挿入した．データ数に比例して挿入時間が増加している．図 4 に示した．データベースは PMDK を利用して不揮発メモリ上に生成して APP ダイレクトモードでメモリ上にマッピングしている．

次に，ランダムに生成した 5 万件の空間データをデータベースに挿入するスレッド数を 1, 2, 4, 8, 16, 32 と変化させ，その

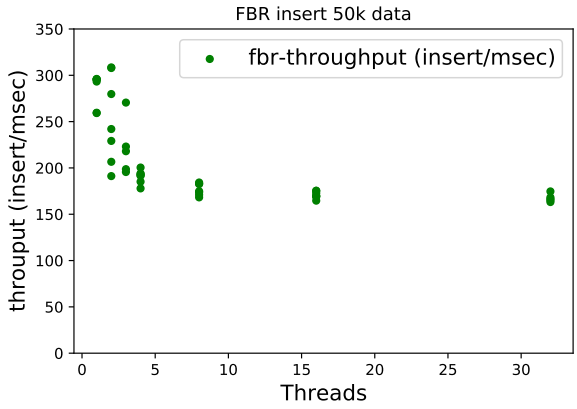

図 5　FBR5 万件データの挿入，スレッド数を変化させる．縦軸はミリ秒あたりの挿入件数

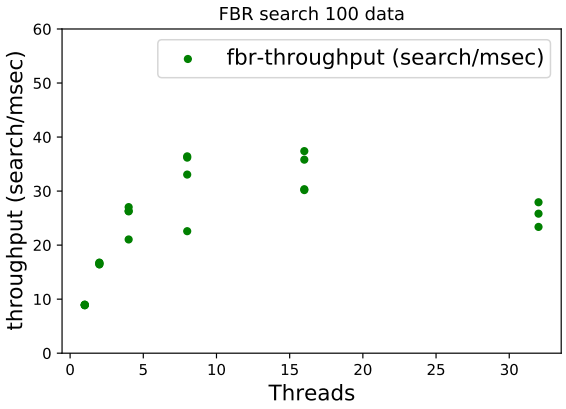

図 6　FBR5 万件のデータから 100 件を検索．スレッド数を変化させる．縦軸はミリ秒あたりの検索件数

スループットを計測した．また同様にデータベースから 100 件検索するスレッド数を変化させ，スループットを計測した．

FBR に挿入した結果を図 5 に示す．横軸にスレッド数，縦軸にミリ秒あたりの挿入件数をとった．スレッド数を増やしても挿入件数は増えず，スケーラビリティがないことを確認した．

スレッド数が 1 の場合，ミリ秒あたりの挿入件数は，259 件/ミリ秒から 296 件/ミリ秒までばらついた．平均 284.9 件/ミリ秒，スレッド数 2 の場合，最小 191.1 件/ミリ秒，最大 308.5 件/ミリ秒，平均 252.1 件/ミリ秒，スレッド数 4 の場合の平均 190.6 件/ミリ秒，スレッド数 8 の場合の平均 176.8 件/ミリ秒，スレッド数 16 の場合の平均 170 件/ミリ秒，スレッド数 32 の場合の平均 167 件/ミリ秒となりスレッド数を増やしてもスループットは向上しなかった．データを挿入する場合 R-tree 全体への mutex lock を使用した並列性制御はコストが高くスケールしないことが確認された．

より粒度の細かい並列性制御の方式が必要とされている．

一方，FBR の検索結果を図 6 に示す．スレッド数を 16 まで増加させた時，ミリ秒あたりの検索数は向上し，スレッド数を更に増やすと，その後はミリ秒あたりの検索数は減少した．

一方で 5 万件のデータから 100 件検索するミリ秒あたりの実行件数は，1 スレッドの場合 8.9 検索件数で，それ以降 2, 4, 8, 16, 32 スレッドでそれぞれ 16.5, 25.1 32.0, 33.4, 25.7 (検索件数/ミリ秒) となり 8 スレッドから 16 スレッドあたりまでスレッド数に比例してスループットが向上した．

これは検索に関してはロックフリー検索を実装しているためと考えられる．FBR-tree のノードはバージョン番号を持っていて，当該ノードが更新された時，バージョン番号を更新する．検索スレッドは検索開始時にそのバージョン番号を記憶しておいて最後にその番号が変化していないかを確認する．途中で変化した場合は再度読み込む（ロールバックする）．ロールバックのコストは高いが，その確率は低いので多くの場合は成功する．

更新と比較して検索の場合はより並列性が高く実行できる．

FBR のスケーラビリティの問題は挿入時に R-Tree のルートノードに対する mutex lock で木全体をロックしてしまうことにあると考えられる．さらにスケーラビリティを向上させるためにはなんらかの方法でボトルネックを解消する必要がある．

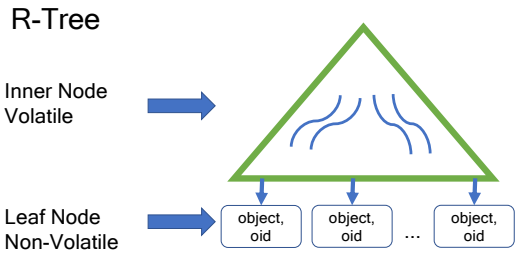

図 7　Inner ノードは PMEM にあるが変更時に flush や同期をとらない．leaf ノードは flush や同期を取る

3.2.1　cache flush 削減

次に cache flush 削減の効果を測定した．我々の先行研究 [33](図 2) によれば，永続性を確保するための明示的な flush および同期命令はコストがたかい．そこで R-Tree の leaf ノードと inner ノードをわけて考え (図 7)，明示的な flush および同期命令は leaf ノードの挿入時にのみ行い，inner ノードの flush は非同期に行う実験を行った．PMEM 上に R-Tree を作成し，シングルスレッドでランダムに生成したデータを挿入する条件で行った．

実験結果は図 8 に示した．縦軸は実行時間 ($nsec * 10^{10}$)，横軸は挿入したノード数 (x1000) である．実行時間は少ないほど（グラフが低いほど）性能が高い．FBR がベースライン (青) で FBR+leaf(赤) が変更した方式である．50 万件から 400 万件を挿入する時間を測定した．いずれの場合も leaf ノードのみを flush する方式のほうが実行時間で減少しており，その比率は 39.4〜48.1%減となった．

leaf ノード以外は明示的に flush しないので，OS が適宜非同期に永続化する．ある瞬間をみるとまだ PMEM に保存されていない状態が発生するが leaf ノードには情報が保存されてい

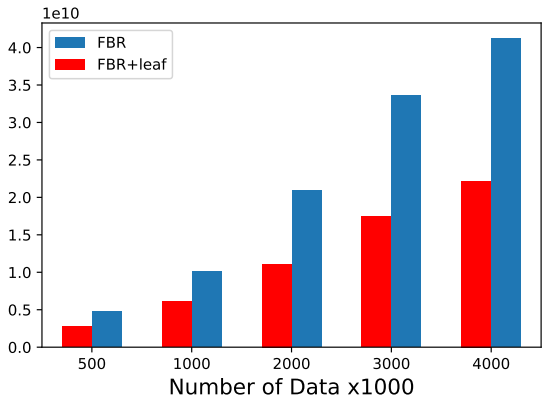

図 8　leaf ノード以外は明示的に flush しない

るので仮にその段階でシステムがクラッシュしたとしても leaf ノードの情報から inner ノードの木構造は再構築できる．今回の実験では再構築のコストについては評価していない．

3.3　実験のまとめ

FBR について追試した．また，FBR の実装について leaf ノード以外，明示的に flush しない方式を提案し，評価したところ，データ挿入の実行時間を 39.4〜48.1%削減した,.

今回追試ができなかった PMR では，mutex lock を使わない方式 (PMwCAS を利用した lock free アップデートを行う) で，スケーラビリティを解決していると言われている [14].

4　関 連 研 究

不揮発メモリ向けの範囲索引について [10], [12], [16] などが様々な提案の比較を行った．ハッシュ索引 [11]，ファイルシステム [25] ，基本的な動作特性 [24], [30] などの報告がある．不揮発メモリ向けの範囲索引について [16] は BzTree [1], FPTree [23], NV-Tree [32], wBTree [2], について評価している. [10] は, [16] で取り上げられなかった以下の範囲索引 LB+-Tree [18], uTree [3], DPTree [35], ROART [20], PACTree [13] をそれぞれ評価した．主に Intel Optane DC Persistent Memory Module 出荷後に提案されたものを実機で評価している．

同様に [11] はハッシュ索引を比較し，Level hashing [36], Clevel hashing [4], CCEH [22], Dash [19], PCLHT [15], SOFT [37] を評価した．ファイルシステム [25] ，データベースエンジン [34]，不揮発メモリ向けのデータ構造 [7]，それぞれのカテゴリで比較検討している．Intel Optane DCPMM の性能評価は [24], [30] にある. [17] は不揮発メモリを大規模メインメモリとして利用する様々なアプリケーションについて調査している．

不揮発メモリ向け R-Tree の研究は近年徐々に増えてきている. [27] は FBR のスケーラビリティの欠如の問題について検討し，many core マシンを対象に，MPR-Tree を提案した．NUMA をサポートし更新のスケーラビリティを向上させた．

SSD(Flash storage) 対応の R-tree の研究として [31] と [6] がある．前者は SSD 向けの最適化について，後者は不揮発メモリと SSD のハイブリッドな構成での最適化について検討している．空間索引構造についてはまだ十分に検討がされていない．そこで本研究では，不揮発メモリを対象とする空間検索構造の実装方式を検討し予備実験をおこなった．

5　まとめと今後の課題

不揮発メモリを対象とする空間索引構造の実装方式の検討と予備実験を行った．FBR の実装を確認し，マルチスレッド環境での挿入および検索のスケーラビリティを確認した．その結果，挿入は R-Tree へのコンテンションが発生し，スループットはスケールしないことが確認され，検索については 16 コア程度であれば，スループットが向上することを確認した．また，R-Tree の leaf ノードのみを明示的に flush する方式を提案し評価した．その結果，FBR と比較して 48.1%ほど性能向上することを確認した．

PMR は FBR のスケーラビリティの問題を解決したと言われているが，今回残念ながら追試することができなかった. [14]

そこで今後は PMR についても追試をし不揮発メモリを対象とする空間索引構造の提案および実装を行う．

6　謝　　辞

本研究の一部は，日本学術振興会科学研究費補助金基盤研究 (B) JP20H04191 の助成を受けたものである．

文　　献

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

[2] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, Vol. 8, No. 7, pp. 786–797, 2015.

[3] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, Vol. 13, No. 12, pp. 2634–2648, 2020.

[4] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pp. 799–812, 2020.

[5] Soojeong Cho, Wonbae Kim, Sehyeon Oh, Changdae Kim, Kwangwon Koh, and Beomseok Nam. Failure-atomic byte-addressable r-tree for persistent memory. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 3, pp. 601–614, 2020.

[6] Athanasios Fevgas, Leonidas Akritidis, Miltiadis Alamaniotis, Panagiota Tsompanopoulou, and Panayiotis Bozanis. A study of r-tree performance in hybrid flash/3dxpoint storage. In *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pp. 1–6. IEEE, 2019.

[7] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data structure primitives on persistent memory: an evaluation. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pp. 1–3, 2020.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIG-*

MOD international conference on Management of data, pp. 47–57, 1984.

[9] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pp. 265–279. Springer, 2002.

[10] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. Evaluating persistent memory range indexes: Part two. *arXiv preprint arXiv:2201.13047*, 2022.

[11] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. *Proc. VLDB Endow.*, Vol. 14, No. 5, p. 785–798, mar 2021.

[12] Kaisong Huang, Yuliang He, and Tianzheng Wang. The past, present and future of indexing on persistent memory. *Proc. VLDB Endow.*, Vol. 15, No. 12, p. 3774–3777, sep 2022.

[13] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 424–439, 2021.

[14] Brandon Lavinsky and Xuechen Zhang. Pm-rtree: A highly-efficient crash-consistent r-tree for persistent memory. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*, pp. 1–11, 2022.

[15] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proc' of 27th ACM SOSP*, pp. 462–477, 2019.

[16] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc' of the VLDB Endowment*, Vol. 13, No. 4, pp. 574–587, 2019.

[17] Hai-Kun Liu, Di Chen, Hai Jin, Xiao-Fei Liao, Binsheng He, Kan Hu, and Yu Zhang. A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions. *Journal of Computer Science and Technology*, Vol. 36, No. 1, pp. 4–32, 2021.

[18] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, Vol. 13, No. 7, pp. 1078–1090, 2020.

[19] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[20] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. Roart: Range-query optimized persistent art. In *FAST*, pp. 1–16, 2021.

[21] Yannis Manolopoulos, Apostolos N Papadopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications: Theory and Applications*. Springer Science & Business Media, 2006.

[22] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *FAST*, Vol. 19, pp. 31–44, 2019.

[23] 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' of the 2016 SIGMOD*, pp. 371–386, 2016.

[24] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, p. 304–315, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Gianlucca O. Puglia, Avelino Francisco Zorzo, César A. F. De Rose, Taciano Perez, and Dejan Milojicic. Non-volatile memory file systems: A survey. *IEEE Access*, Vol. 7, pp. 25836–25871, 2019.

[26] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, Vol. 42, No. 2, pp. 34–40, 2017.

[27] Abdul Salam, Safdar Jamil, Sungwon Jung, Sung-Soon Park, and Youngjae Kim. Future-based persistent spatial data structure for nvm-based manycore machines. *IEEE Access*, Vol. 10, pp. 114711–114724, 2022.

[28] Steve Scargall. *Programming Persistent Memory A Comprehensive Guide for Developers*. Apress, Berkeley, CA, 2020.
https://doi.org/10.1007/978-1-4842-4932-1.

[29] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 461–472. IEEE, 2018.

[30] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proc' of SC '19*, pp. 1–19, 2019.

[31] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pp. 17–24, 2003.

[32] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, Vol. 65, No. 7, pp. 2169–2183, 2016.

[33] Hirotaka Yoshioka, Yuto Hayamizu, Kazuo Goda, and Masaru Kitsuregawa. pmmeter: A microbenchmark for understanding synchronization cost on persistent memory. In *2023 IEEE International Conference on Big Data and Smart Computing (BigComp2023)*, pp. 326–327. IEEE, 2023.

[34] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, No. 7, pp. 1920–1948, 2015.

[35] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment*, Vol. 13, No. 4, pp. 421–434, 2019.

[36] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 461–476, 2018.

[37] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, Vol. 3, No. OOPSLA, pp. 1–26, 2019.