Vol.2022-DBS-175 No.14
Vol.2022-IFAT-148 No.14
2022/9/9

Adaptive Radix Treeの多次元索引への拡張

鈴木 駿也[†1,a)]　杉浦 健人[†1,b)]　石川 佳治[†1,c)]

概要：多次元索引は地理情報を始めとする空間データへの索引付けや類似検索などに利用されており，近年では機械学習を利用した索引構造も提案されその性能改善が図られている．多くのデータベースで用いられている B^+ 木を Z 階数を用いて多次元索引へと拡張した研究として universal B 木が存在するが，ノードのデータ分割が空間的に適したものにならない．一方で，トライ木を元とした索引構造はキーを数ビットずつ分割するため，Z 階数を適用したときに四分木のような理想的な分割となる．そこで，本研究ではトライ木ベースの索引構造として有名な adaptive radix tree を多次元データへ拡張する．また，二次索引や範囲問合せという多次元索引として重要な性質に最適化する．

1. はじめに

多次元索引は地理情報アプリケーションにおける空間データの索引付けや，二次索引による類似検索など，その利用は多岐にわたる．多次元索引として広く利用されている R^* 木 [2] は範囲検索や近傍検索に特化しているが，挿入時のノード分割計算量が高く，逐次的な更新に適していないと考えられる．近年では機械学習を利用した索引構造も提案されその性能改善が図られている [3, 7] が，これらも逐次的な更新を重視しておらず，汎用的に使用可能な多次元索引は少ない．

多次元索引の中でも，多くのデータベースで用いられている B^+ 木 [11] を多次元索引へ拡張した研究として universal B 木（UB 木） [1] が存在する．UB 木は多次元データを 1 次元の Z 階数 [8] に変換し，B^+ 木のキーとして使用する．UB 木の構造は B^+ 木であるため逐次的な更新に特化している．しかし，ノード分割時には分割するノードのみで空間分割を考慮するため，全体の空間分割は最適なものとならない．図 1 に UB 木の空間分割を示す．図 1 左の UB 木の色付きノードの対象とする範囲を，図 1 右の対応する色で表す．このように，UB 木の空間分割は非連続な領域やバラつきが存在する．また，UB 木のように比較演算を多用する索引は，多次元データのようなサイズの大きいキーを扱うことに適していない [6].

主な索引構造の 1 つとして，トライ木が挙げられる．トライ木はキーを数ビットずつ分割し，葉ノードへのパスとして保存する．トライ木は検索にキー全体の比較演算を使用しないため，キー長の長いデータの索引付けに特化している．近年提案された adaptive radix tree（ART）[6] は，トライ木のノードとパスの圧縮により CPU キャッシュの利用効率を最適化した構造である．ART はインメモリ DBMS の HyPer [4] の索引構造として使用され，近年でも多くの研究例が存在する．ART を空間結合に拡張した研究 [5] が存在するが，これは一般的な空間問合せや更新に対応しておらず，汎用的な多次元索引として最適化されていない．

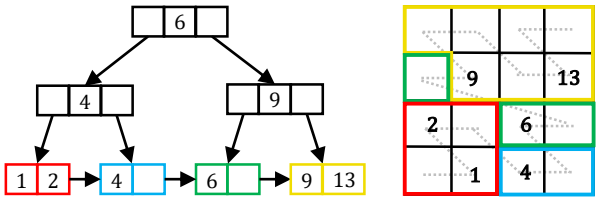

図 1　UB 木の空間分割

そこで，本研究では ART を汎用的な多次元索引として最適化した，universal adaptive radix tree（UART）を提案する．UART は ART のキーに Z 階数を適用する．UART は Z 階数を数ビットずつ分割するため，空間分割は図 2 に示す通り四分木のような理想的なものとなる．更に葉ノードを連結し問合せ処理に空間分割の特性を利用することで，範囲検索に最適化する．また，葉ノードは単一キーに対して複数の値を保存可能な構造であり，二次索引として利用可能にする．

本稿の構成は以下の通りである．2 章では，関連研究として UART のベースとなる UB 木，ART について概説する．次に，3 章では UART の構造とその空間的特性，そして問合せ処理について説明する．最後に，本稿のまとめと

†1 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University
a) ssuzuki@db.is.i.nagoya-u.ac.jp
b) sugiura@i.nagoya-u.ac.jp
c) ishikawa@i.nagoya-u.ac.jp

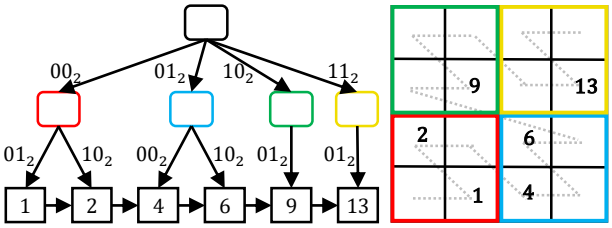

図 2 UART の空間分割

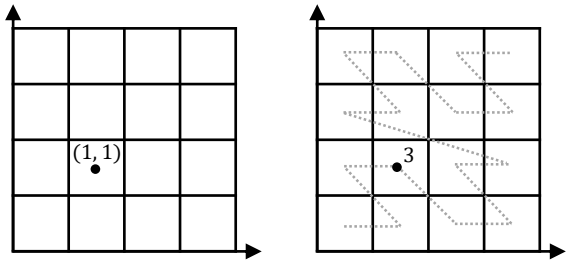

図 3 元空間と Z 階数曲線を適用した空間

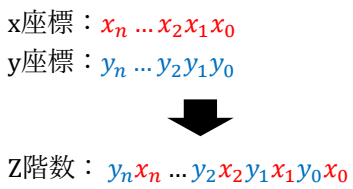

図 4 2 次元データの Z 階数の計算

今後の方針について 4 章で述べる．

2. 関連研究

2.1 Universal B 木

UB 木は多次元データを Z 階数に変換し，B^+ 木に適用したものである．UB 木の問合せ処理は基本的に B^+ 木と同一であるが，範囲検索処理が大きく異なる．本節では Z 階数と範囲検索処理を説明する．

2.1.1 Z 階数

Z 階数は，多次元データをその空間を充填する Z 階数曲線の上の開始点からの距離に変換した値である [8]．図 3 右に示すように，Z 階数曲線は空間を再帰的 “Z” 字で充填するような曲線である．図 3 左で元々 $(1, 1)$ の点は，Z 階数曲線上でその開始点から 3 番目の座標にあるため 3 という 1 次元値に変換される．

多次元値から Z 階数への変換は，実際には各座標値のビットを交互に配置することで計算できる．図 4 に 2 次元データの Z 階数の計算を示す．2 次元データの x 座標値のビット列を $x_n \ldots x_2x_1x_0$，y 座標値のビット列を $y_n \ldots y_2y_1y_0$ と表すとき，その Z 階数のビット列は $y_nx_n \ldots y_2x_2y_1x_1y_0x_0$ と表すことができる．多次元データの各座標値を順番にはさみ合わせることで，任意次元のデータを Z 階数に変換できる．

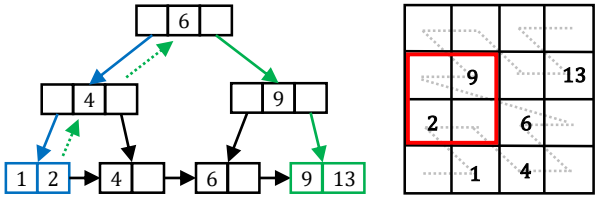

図 5 UB 木の範囲検索

2.1.2 範囲検索

UB 木の範囲検索は，範囲外の値を発見したときに次の範囲内の値を計算し，ジャンプすることで範囲外の走査を避ける [9]．図 5 に UB 木の範囲検索時のノード遷移を示す．図 5 の赤い矩形で範囲検索する場合，葉ノード中の 2 から 9 を走査する必要がある．まず，範囲内最小の値である 2 を持つ葉ノードに青色のパスで遷移し，葉ノードを走査する．しかし中間の 4，6 は範囲外であり，無駄な走査となる．この場合，緑色のパスで表すように，スタックから残りの検索範囲と重複する親ノードへ回帰し，葉ノードへ遷移する．UB 木は各階層での空間分割が定められていないため，根ノードから検索し直す方が簡単な場合でも，親ノードへの回帰を繰り返すような処理が発生しうる．また，親ノードへの回帰毎に，そのノードが残りの検索範囲と重複しているか判定する必要がある．

2.2 Adaptive Radix Tree

ART [6] はトライ木の中間ノードとパスを動的に変化させ，キー分布に最適化したものである．本節ではトライ木と，ART のトライ木からの変更点である中間ノードとパス圧縮，そして問合せ処理を説明する．

2.2.1 トライ木

図 6 に示すように，トライ木は葉ノードまでのパスにキー，葉ノードにレコードを保存する木構造である．図 6 のように，トライ木はパスに部分キーを保存するが，これはキーのデータ列を s ビットずつ分割した値だと言える．そのため，各中間ノードは 2^s 個の子ポインタを持つ．トライ木の中間ノードは，子の数が 1 つの場合でも 2^s 個の子ポインタを持つ必要があるため，その多くがヌルポインタとなりうる [6]．

トライ木は B^+ 木のような比較演算を多用する索引と比べ，探索の計算量が小さい．トライ木はキーの分割した値が子ポインタの配列上で参照すべき要素番号となるため，各ノードでの遷移に探索を必要としない．したがって，キー長が k のとき，探索の時間計算量は $O(k)$ となる．対して B^+ 木のような索引は，長さ k のキーの比較による計算量は $O(k)$ であるため，データ数が n のとき探索全体の時間計算量は $O(k \log n)$ となる [6]．

2.2.2 中間ノード

ART の中間ノードは子ノードの数によって種類を変更する．ART はキーの分割ビット数が 8 であり，中間ノードは

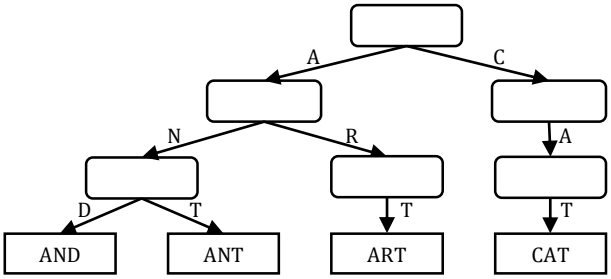

図 6 トライ木

最大 256 個の子ポインタを持つ．図 7 に ART の中間ノードを示す．ART には Node4, Node16, Node48, Node256 の 4 種類の中間ノードがあり，それぞれの名前が最大の子の数を表す．

Node4, Node16 はキー, 子ポインタをそれぞれ 4 個, 16 個ずつ持つ．Node4，Node16 は i 番目のキーに対応する子ポインタを，その i 番目として持つ．図 7 の Node4 は部分キー 0, 2, 3, 255 のみ保持し，その要素番号 3 となる 255 には部分木 d が対応する．

Node48 はキーを 256 個, 子ポインタを 48 個持つ．Node48 は部分キー i に対応する子ポインタを，要素番号 i のキーに格納されている要素番号の子ポインタに持つ．図 7 の Node48 では，部分キー 255 に対応する部分木 d の要素番号 47 を，キー配列の要素番号 255 に持つ．ART は分割ビット数が 8 であるため，部分キーの値は 0 以上 255 以下となる．したがって，Node48 では探索を必要とせずにキー要素へ直接アクセスできる．

Node256 は 256 個の子ポインタのみ持つ．Node256 はキー i に対応する子ポインタを，その i 番目に持つ．図 7 の Node256 では部分キー 255 に対応する部分木 d を，要素番号 255 の子ポインタに持つ．Node256 も Node48 と同様に，キーの探索を必要としない．Node256 は元々のトライ木の中間ノードと同じ構造である．

2.2.3 パス圧縮

ART は子の数が 2 以上の中間ノードのみ作成し，パスを圧縮する．図 8 に ART の概観を示す．図 8 では，木全体で “C” から始まるキーは “CAT” のみである．そのため，根ノードからのパス “C” には直接葉ノードのポインタを持ち，パス “AT” は省略される．省略されたパスは，続くノードのヘッダーに保存する．

2.3 問合せ処理

検索は, 根ノードから葉ノードに到達するまで, 与えられたキーの部分キー i を用いてノード遷移して行う．Node4, Node16 では，ノード中のキー配列から i と一致する値を持つ要素を探索し，その要素番号の子ポインタへ遷移する．Node48 ではキー配列の i 番目に入っている要素に入っている番号の子ポインタへ遷移する．Node256 では i 番目の子ポインタへと遷移する．各ノードで，i が見つからない

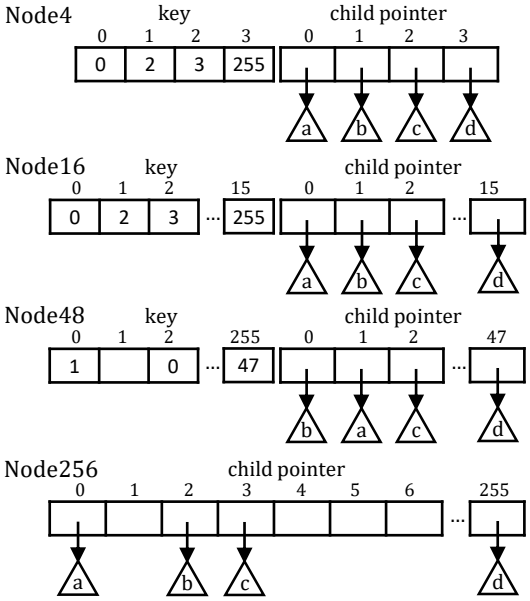

図 7 ART の中間ノード

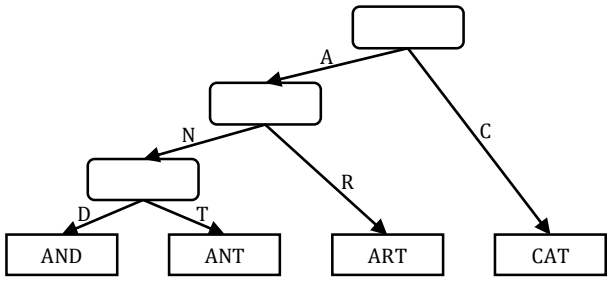

図 8 Adaptive radix tree

場合や子ポインタがヌルポインタである場合，そこで検索を打ち切る．また，遷移ノードに圧縮パスが格納されている場合，一致する限り部分キーを次に進める．圧縮パス全てが後続の部分キーと一致しない場合は検索を打ち切る．

挿入は，検索の手順でノード遷移して行う．遷移先がなくなった時，後続の部分キーをキーとして，レコードを持つ葉ノードをそのノードに挿入する．遷移先のノードの圧縮パスが続く部分キーと一致しなければ，新たな中間ノードを作成し，現在のノードと葉ノードを挿入する．挿入時にノードの持つ子の数が許容量の上限に達している場合は，ノードを 1 段階大きいものに拡張して挿入する．

削除は挿入処理と同様である．該当する葉ノードをその親ノードから削除し，要素数が閾値を下回れば一段階小さいノードに縮小する．子の数が 1 つになる場合，子ノードと親ノードの入れ替えを行い，パスを圧縮する．

3. Universal Adaptive Radix Tree

UART は ART のキーとして Z 階数を使用し，多次元索引に拡張したものである．特に，本稿では UART の空間索引や OLAP での利用を念頭に置く．そのため，キーの重複を許した二次索引としての実装や，空間問合せおよび解析

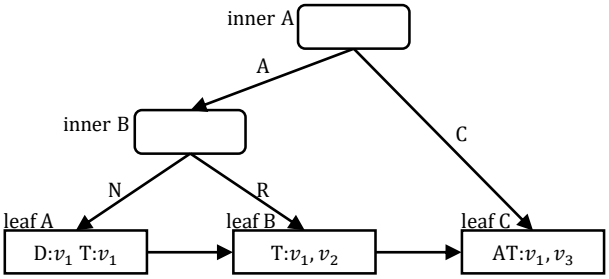

図 9 Universal adaptive radix tree

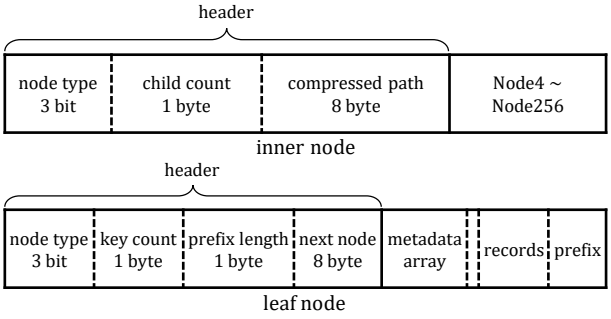

図 10 UART の中間ノード，葉ノードのレイアウト

処理におけるフィルタリング処理のための範囲検索性能を重視する．

UART は ART の葉ノードを連結し，1 つの葉ノードに複数のキーとレコードを保存可能にしたものである．図 9 に UART の概観を示す．葉ノードは，保存するキーの共通プレフィックスをパスとし，キーのサフィックスと対応する値の配列のペアをレコードとして複数持つ．

3.1 ノードのレイアウト

UART の中間ノードは ART の中間ノード，葉ノードは B^+ 木の葉ノードと概ね同様の構造である．図 10 に UART の中間ノードと葉ノードのレイアウトを示す．

中間ノードは 2.2.2 節に示した Node4，Node16，Node48，Node256 の 4 種類があるが，それぞれ共通のヘッダーを持つ．node type は Node4 から Node256 の中間ノードと葉ノードの識別に用いる．child count にはノードの持つ子の数，compressed path には圧縮されたパスを保存する．

葉ノードの node type は中間ノードと同様であり，key count には保持しているキーの数を保存する．prefix length は保持するキーの共通プレフィックスの長さであり，ノードの末尾に共通プレフィックスを持つ．そして next node は隣接する葉ノードのポインタを保管する．metadata array の要素数はキーの数だけあり，それらは対応するレコードへのアクセスに用いる．レコードは，キーから共通プレフィックスを除いたサフィックスと，各キーに対応する値の配列からなる．挿入時，対象となるキーのレコードが既に存在していれば対象値を配列に挿入し，存在していなければ新たにレコードを作成する．

3.2 空間分割

UART は，Z 階数のキーを分割することでデータの空間分割を理想的なものとする．図 2 に次元数を 2，分割ビット数を 2 とした場合の UART の空間分割を示す．図 2 では，図左の色付きノードが分割する領域を，図右の対応する色の領域として表している．実際には UART の分割ビット数は 8 であるため，各階層で 256 個のグリッドに分割する．これにより，UART は同じノード以下に存在するキーはプレフィックスを共有し，範囲検索を効率化できる．

Z 階数の特性により，UART の分割は図 2 のようになる．2.1.1 節で説明したように，Z 階数は多次元データの各次元のビットを交互に配置している．各次元の最上位ビットは，その次元の全ドメインにおいて上位半分以上であるかを示し，続くビットは分割されたドメインにおいて同様である．したがって，Z 階数の上位 s ビットは空間の全ドメインにおいて，2^s 個のグリッドのどれに属するかを示す．UART はキーとなる Z 階数を上位から 8 ビットずつ区切ることで，空間を階層的にグリッド分割する．

3.3 問合せ処理

UART は点検索，範囲検索，挿入，削除をサポートする．点検索，削除は ART の処理と概ね同様である．キーは多次元座標で与えられ，Z 階数に変換して以下のように処理する．

点検索は，葉ノードまで遷移した後，対象のレコードを全て返す．同一のキーで葉ノードが分割している場合，パスが対象キーとなる中間ノード以下のレコードを全て返す．

3.3.1 範囲検索

範囲検索は，UB 木の範囲検索と概ね同様である．入力として検索範囲の最小，最大となる座標が与えられ，それらを Z 階数に変換して以下のように処理する．

範囲内最小のキーから最大のキーまで葉ノードを順に走査し，現在の葉ノードが検索範囲と重複していなければ，次の範囲内のキー（NextJumpIn）を検索する．まず，範囲内最小のキーを持つ葉ノードへ遷移し，以降レコードを収集しつつ葉ノードを走査する．葉ノードの遷移時，共通プレフィックスから検索範囲と葉ノードの対象範囲の重複を判定する．完全に重複している場合，葉ノード内全てのレコードを収集する．部分的に重複している場合，全てのキーについて範囲内か確認し，範囲内のキーを持つレコードのみ収集する．重複していない場合，NextJumpIn となる現在のキーより大きく最小の範囲内のキーを計算 [10] し，検索する．

NextJumpIn の検索は，前回検索した葉ノードまでのパスのスタックを用いる．UART ではノードの空間分割の特性上，プレフィックスが共通するキーは，根から途中までのパスを共有する．したがって，NextJumpIn と前回検索したキーのプレフィックスが共通している場合，途中までのパ

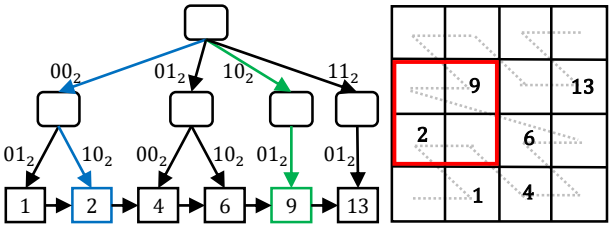

図 11 UART の範囲検索

スはスタックから復元できる．復元可能なパスの深さは共通するプレフィックス長から計算できるため，その深さが十分でない時は根ノードから検索することで無駄なパスの復元を避ける．また，パスを復元する場合，各ノードにおいて検索範囲との重複を判定する必要もない．

図 11 に UART の範囲検索時のノード遷移を示す．図 11 では赤い矩形が検索範囲である．最初に，範囲内最小のキーである 2 を持つ葉ノードに青色のパスで遷移する．2 を持つ葉ノードは，その対象範囲が完全に検索範囲内と重複するため，葉ノード内のレコードを収集する．その後，隣接する 4 を持つ葉ノードに遷移するが，その対象範囲は検索範囲と重複しないため NextJumpIn を計算する．NextJumpIn は 8 となるが，前回検索したキー 2 とプレフィックスが共通しない．そのため，緑色のパスで根ノードから NextJumpIn 以上で最小のキーである 9 を持つ葉ノードへ遷移する．

3.3.2 更新

挿入は，点検索の手順で対象となるキーを検索し，到達したノードにレコードもしくは新たな葉ノードを挿入する．また，対象キーとの共通プレフィックスを持つ葉ノードが存在するかどうかで処理が異なる．

共通プレフィックスを持つ葉ノードが存在している場合，到達した葉ノードに対象のキーや値を挿入する．葉ノードに空きがない場合，保存しているキーのサフィックスの先頭バイトでレコードを分類し，分割する．分割時，分割後の葉ノードを持つ新たな中間ノードを作成し，分割前の葉ノードが存在したパスに挿入する．図 12 に，図 9 にキー “CUP”，値 v_1 を挿入した状態を示す．図 12 では，図 9 の leaf C が leaf C′ と leaf C″ に分割している．新たに生成した中間ノードの inner C に leaf C′ と leaf C″ を挿入し，inner C は inner A に挿入する．なお，キーのサフィックスが更に共通のプレフィックスを持つ場合，新たな中間ノードの圧縮パスとして保存し，後続の部分キーで分類する．また，葉ノードがキーを 1 つしか持たない場合はレコード内の値で同様に分割する．

共通プレフィックスを持つ葉ノードが存在していない場合，新たな葉ノードを作成し，既存の葉ノードと連結する．この場合，葉ノードへ到達する前に中間ノードからの遷移先がなくなる．そこまでの遷移に使用した部分キーと，次の部分キーを共通プレフィックスとする新たな葉ノードを作成し，中間ノードに挿入する．その後，対象キー未満で

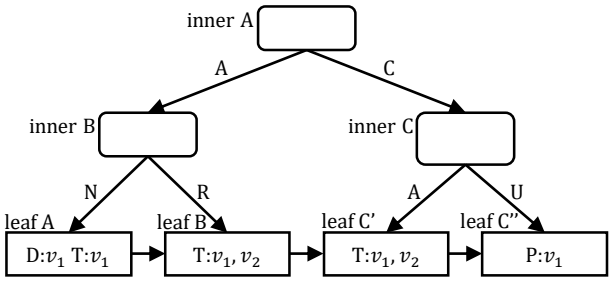

図 12 図 9 に “CUP” を挿入した状態

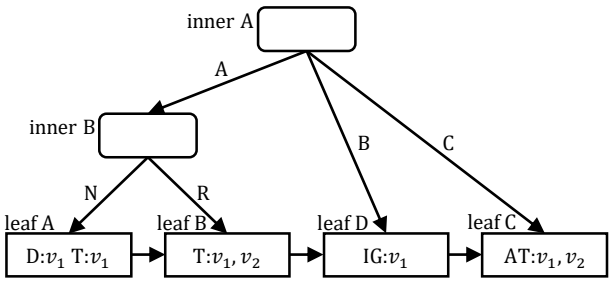

図 13 図 9 に “BIG” を挿入した状態

最大のキーを持つ葉ノードを検索し，新たな葉ノードを連結する．図 13 に，図 9 にキー “BIG”，値 v_1 を挿入した状態を示す．図 9 には “B” から始まるキーが存在しないため，根ノードにパス “B” は存在しない．したがって，図 13 では共通プレフィックス “B” の葉ノード leaf D を新たに作成し，根ノードに挿入する．その後，“BIG” 未満で最大のキーを持つ leaf B を検索し，leaf D を leaf B と leaf C の間に挿入する．

削除は挿入処理と同様である．対象キーを保存している葉ノードに遷移し，そこから対象値を削除する．葉ノード内のレコードが全て削除された場合，その葉ノードを削除する．

4. まとめと今後の方針

本稿では，ART を範囲検索と二次索引向けに最適化し，多次元索引に拡張した UART を提案した．今後の方針として，提案した UART を実装し，性能を評価することが挙げられる．また，統一的な多次元索引のベンチマーカーを用意し，従来の多次元索引である UB 木や R* 木，機械学習による空間索引と比較する．

謝辞 本研究は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594 の助成，および国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）の結果得られたものである．

参考文献

[1] Bayer, R.: The universal B-tree for multidimensional indexing: General concepts, *Proc. Worldwide Computing and Its Applications (WWCA)*, pp. 198–209 (1997).
[2] Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B.: The R*-Tree: An Efficient and Robust Access Method for

Points and Rectangles, *Proc. SIGMOD*, pp. 322–331 (1990).
[3] Ding, J., Nathan, V., Alizadeh, M. and Kraska, T.: Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads, *PVLDB*, Vol. 14, No. 2, pp. 74–86 (2020).
[4] Kemper, A. and Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots, *Proc. ICDE*, pp. 195–206 (2011).
[5] Kipf, A., Lang, H., Pandey, V., Persa, R. A., Anneser, C., Zacharatou, E. T., Doraiswamy, H., Boncz, P. A., Neumann, T. and Kemper, A.: Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins, *Proc. EDBT* (2020).
[6] Leis, V., Kemper, A. and Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases, *Proc. ICDE*, pp. 38–49 (2013).
[7] Qi, J., Liu, G., Jensen, C. S. and Kulik, L.: Effectively Learning Spatial Indices, *PVLDB*, Vol. 13, No. 12, pp. 2341–2354 (2020).
[8] Rigaux, P., Scholl, M. and Voisard, A.: *Spatial Databases with Application to GIS*, Morgan Kaufmann (2001).
[9] Skopal, T., Krátký, M., Pokorný, J. and Snášel, V.: A new range query algorithm for Universal B-trees, *Information Systems*, Vol. 31, No. 6, pp. 489–511 (2006).
[10] Tropf, H. and Herzog, H.: Multidimensional Range Search in Dynamically Balanced Trees, *Applied Informatics*, pp. 71–77 (1981).
[11] 北川 博之：データベースシステム，昭晃堂 (1996).