# Adaptive Radix Tree の多次元索引への拡張

鈴木　駿也† 　杉浦　健人† 　石川　佳治† 　陸　可鏡†

† 東海国立大学機構名古屋大学大学院情報学研究科　〒 464–8603 愛知県名古屋市千種区不老町

Email: {ssuzuki, lu}@db.is.i.nagoya-u.ac.jp, {sugiura, ishikawa}@i.nagoya-u.ac.jp

**あらまし**　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複数列キーでの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 階数曲線を用いて多次元索引へと拡張した研究として universal B 木（UB 木）が存在するが，各ノードに割り当てられるキー範囲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上位ビットから階層的に分割するため，Z 階数曲線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した universal adaptive radix tree (UART) を提案する．UART はトライ木の挿入性能を活用するとともに，Z 階数曲線の性質を利用することで範囲検索にも最適化し，二次索引として利用可能な構成として実装した．実験において提案する UART の性能を UB 木や $R^*$ 木といった既存の多次元索引と比較し評価した結果，読み書き両方において優れた性能であることを確認した．

**キーワード**　索引構造，多次元索引，Z 階数曲線．

## 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 [1] 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，逐次的な更新に弱いという課題がある．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 [2,3] が，これらも逐次的な更新を重視しておらず，読み書き両方において安定した性能を持つ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 [4] を多次元索引へ拡張した研究として universal B 木（UB 木） [5] が存在する．UB 木は多次元データを Z 階数曲線に基づき 1 次元の Z 値 [6] に変換し，$B^+$ 木のキーとして使用する．UB 木は多次元データを扱うがその構造は $B^+$ 木のままであるため，書き込みに対する性質なども $B^+$ 木のそれを受け継ぎ，空間索引としては逐次的な更新に強いという特徴を持つ．

しかし，UB 木は多次元値を Z 階数曲線により 1 次元化し管理するが，ノード分割も 1 次元値として行うため，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 木の各ノードの空間分割は定まっておらず，非連続な領域やバラつきが存在し，あるノードを包含する最小矩形（バウンディングボックス）が過剰に広がってしまうことがある．例えば，最下段の $\{19, 32\}$ を持つノードのバウンディングボックスは図中の下半分以上の空間を占めるものとなるが，ノード内には左下の空間に対応するレコードは存在しないため無駄な領域を多く含む．このような無駄な領域の存在により索引層での枝刈りなども困難となり，実用上では範囲検索の性能が劣化しやすいという課題がある．

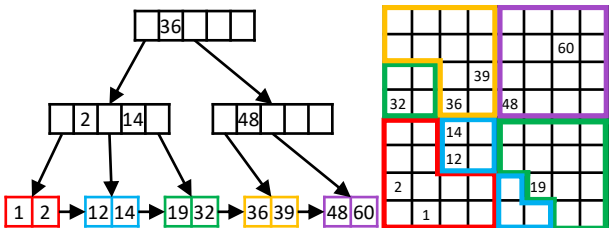



**図 1:** UB 木における空間分割の例

一方で，主な索引構造の 1 つとしてトライ木が存在する．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） [7] 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 はインメモリデータベース管理システムである HyPer [8] 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．例えば，ART を空間結合の際に使用できるよう拡張した研究 [9] 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．

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 UB 木同様，与えられた多次元値を Z 階数曲線によって 1 次元に変換した Z 値を挿入キーとして利用する．ただし，UART は変換した Z 値を上位から 1 バイトずつ分割し，階層的に空間を分割する．例として，Z 値の上位 2 ビットを用いて空間を分割したものを図 2 に示す[1]．図に示す通り UART における空間

1：分割の単位は 1 バイトであるが，概念を直感的に説明するためにここでは 2 ビットのみの例を示す．

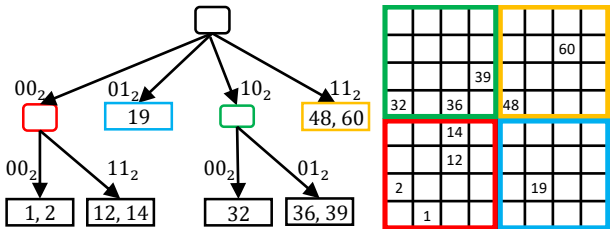



**図 2:** UART における空間分割の例

分割は四分木のような理想的なものとなり，各ノードの対応するキーの範囲がそのバウンディングボックスと一致する．この性質を利用することで，UART では範囲検索時に索引層での積極的な枝刈りや検索範囲への包含判定の効率化が行える．更に，ART の構造を利用しノードサイズや木の深さを適応的に変化させることで，メモリ利用効率や検索効率を向上させる．また，葉ノードには単一キーに対して複数の値を保存可能な構造を採用し，二次索引としても利用可能なよう実装する．

## 2 準　　備

本章では，UART のベースとなる索引構造である UB 木と ART について説明する．

### 2.1 Universal B 木

UB 木は多次元データを Z 階数曲線により Z 値に変換し，$B^+$ 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 値と UB 木の範囲検索処理を説明する．

#### 2.1.1 Z 値

Z 値は，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 [6]．例えば，元々 $(0, 1)$ の点は，Z 階数曲線上でその開始点から 1 番目の座標にあるため 1 という値に変換される．多次元値から Z 値への変換は，各座標値のビットを交互に配置することで計算できる．2 次元データの場合，x 座標値のビット列を

$$\mathbf{x} = x_n \ldots x_2 x_1 x_0$$

y 座標値のビット列を

$$\mathbf{y} = y_n \cdots y_2 y_1 y_0$$

と表すとき，その Z 値のビット列は

$$\mathbf{z} = y_n x_n \ldots y_2 x_2 y_1 x_1 y_0 x_0$$

と表すことができる．同様に，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 値に変換できる．

#### 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 [10]．UB 木はノード分割が事前に定まっていないことやノードの空間分割に非連続な領域があることから，効率的な範囲検索が難しい．各ノードが Z 値の最大値と最小値しか持っていないため，

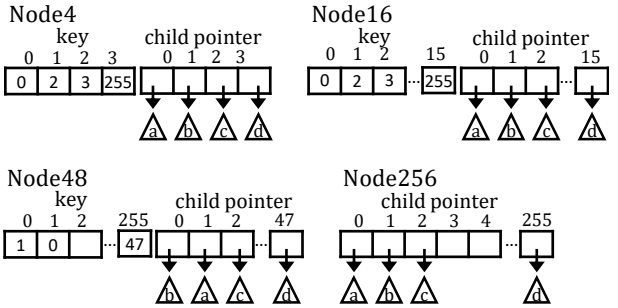



**図 3:** ART の中間ノード

ノードが検索範囲と交差するか判定する場合には逐次的に次の範囲内の値を計算し，ノード範囲と比較する必要がある．次の範囲内値の再検索時，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 [7] 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

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．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 $2^s$ 個の子ノードのポインタを持つ．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 [7]．

#### 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，中間ノードは最大 256 個の子ノードのポインタを持つ．図 3 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，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ノードのポインタを，その $i$ 番目として持つ．例えば，図 3 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，ポインタを 48 個持つ．Node48 は部分キー $i$ に対応する子ノードのポインタを，要素番号 $i$ のキーに格納されている要素番号のポインタに持つ．図 3 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 個のポインタのみ持つ．Node256 はキー $i$ に対応する子ノードのポインタを，その $i$ 番目に持つ．図 3 の

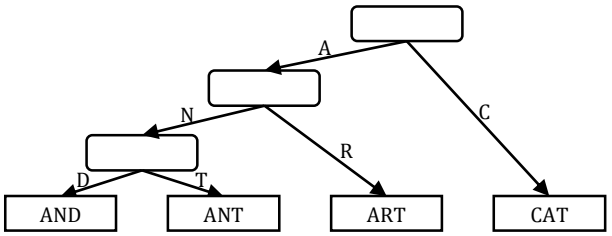


**図 4:** Adaptive radix tree

Node256 では部分キー 255 に対応する部分木 $d$ を，要素番号 255 のポインタに持つ．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 4 に ART の概観を示す．図 4 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．同様に，各中間ノードにおいて子ノードが 1 つしかないノードを省略することで，空間効率や検索効率を最適化する．省略されたパスは，続くノードのヘッダに保存する．

### 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，その要素番号の子ノードへ遷移する．Node48 ではキー配列の $i$ 番目に入っている要素番号の子ノードへ遷移する．Node256 では $i$ 番目の子ノードへ遷移する．各ノードで，部分キーが見つからない場合や子ノードの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，遷移先の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 層と UB 層からなり，キーとして Z 値を使用して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．

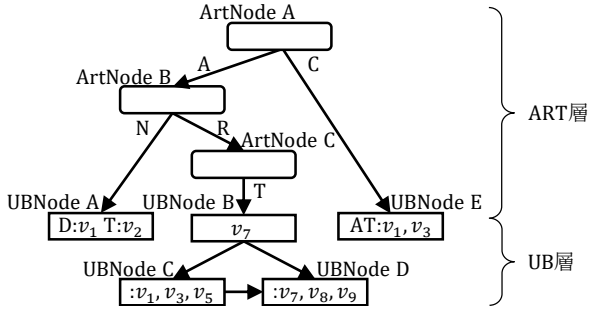


**図 5:** Universal adaptive radix tree

図 5 に UART の概観を示す．なお，図 5 はキーを文字列として表現している．ART 層はキーのプレフィックスを分割して保存することで空間を分割し，UB 層でデータを効率的に保持する．ART 層は ART の中間ノードを拡張した ART ノードからなり，UB 層は UB 木の葉ノードを拡張した UB ノードからなる．UB 層の UB ノードは二次索引として単一のキーに複数のレコードを保存可能にし，単一のキーへの値が集中した場合，ノードを分割して UB 木に置き換えられる．図 5 では UBNode A にキー “AND”，“ANT” とそれぞれに対応するレコード $\{v_1, v_2\}$ を保存している．また，UBNode B，C，D はキー “ART” にレコード $\{v_1, v_3, v_5, v_7, v_8, v_9\}$ を保存する UB 木となっている．

### 3.1 ART 層

ART 層は，キーとして与えられた Z 値を上位バイトから分割することで，多次元空間を階層的に分割する索引層である．1 階層は 1 バイト（256 区画）単位で空間がグリッドに分割され，トライ木の性質により各グリッドへの効率的なアクセスが可能となる．以下では，ART 層における空間分割とノードレイアウトについて詳細を述べる．

**3.1.1** 空間分割

UART は，図 2 で示したように，Z 値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 は次元数を 2，分割ビット数を 2 とした場合の UART の空間分割であり，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．図 2 で示したように次元数，分割ビット数がともに 2 の場合，根ノードから深さ 1 のノードは四分木と同じ空間分割となり，同様に中間ノードの子ノードは親ノードの空間を 4 つのグリッドに分割していく．

実際には，UART の分割ビット数は 8（1 バイト）であるため，各階層で 256 個のグリッドに分割する．2.1.1 項で説明したように，Z 値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，続くビットはそれぞれのドメインにおいて同様である．したがって，Z 値の上位 $s$ ビット（プレフィックス）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．同様に，後続の $s'$ ビット（サフィックス）はそのグリッド内において $2^{s'}$ 個のサブグリッド

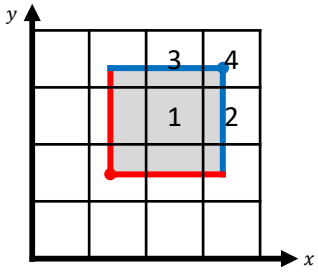



**図 6:** UART の分割範囲と検索範囲の例

のどれに属するかを示す．UART においてはトライ木による検索性能を活用するために，ビット数として 8 を使用し，対象空間を階層的に分割する．

UART ではこの階層的な空間分割を範囲検索の効率化に利用する．上述したように，あるグリッドに含まれるサブグリッドはそのグリッドに完全に包含される．つまり ART 層において，上位のノードで行った検索範囲との交差判定の結果は，下位ノードでの交差判定の効率化に利用できる．図 6 に，UART の空間分割と検索範囲の例を示す．検索範囲は赤と青の辺からなる矩形であり，赤色の辺が始点側，青色の辺が終点側を示す．空間 1 は，x 軸および y 軸ともに始点・終点の内側にあるため，空間 1 を持つノードは検索範囲に完全に包含される．したがって，空間 1 以下のノードにおいて追加の包含判定は不要であり，全てを検索対象としてフルスキャンできる．一方，空間 3 は x 軸に関して検索範囲に包含されているが，y 軸において終点側と交差している．そのため空間 3 以下のノードでは，y 軸の終点側のみで包含しているかを確認すればよい．同様に空間 2 では x 軸の終点側，空間 4 では x 軸および y 軸で終点側と比較することで包含判定を効率化できる．

また，下位ノードでの包含判定において，上位層におけるキーの情報が不要である点も利点となる．UART では上述したように上位ノードで包含されなかった次元のみを確認すればよいが，各次元の値が包含されないのは上位ノードのキーのプレフィックスが始点側ないし終点側のキーのプレフィックスと一致したときのみである．つまり，下位ノードでは自身の階層に対応するサフィックスのみを用いて包含判定を行えばよい．このように，階層的な空間分割を持たない UB 木では各次元での包含判定にキーの全バイトが必要であるのに対し，UART では階層構造を利用することで効率的な包含判定が可能となる．

ART ノードが Node48 または Node256 である場合，現在の値から次の検索範囲内の値（NextJumpIn 点）を計算し [11]，範囲検索での子ノードの走査を効率化する．UART は空間のグリッド分割とプレフィックスの共有により，各階層 1 バイトで NextJumpIn 点を計算できる．更に，トライ木では 1 バイトで表されるキーへ $O(1)$ でアクセスできるため，NextJumpIn 点の計算およびその遷移先へのアクセス共に効率よく行える．

#### 3.1.2 ART ノード

ART ノードは ART の中間ノードと概ね同様の構造である．図 7 に ART ノードのレイアウトを示す．

ART ノードは ART の中間ノード同様，2.2.2 項に示した

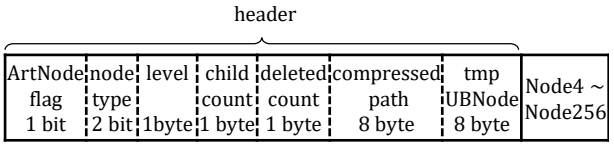



**図 7:** ART ノード

Node4，Node16，Node48，Node256 の 4 種類があるが，それぞれ共通のヘッダを持つ．ArtNode flag は ART ノードと UB ノードの識別，node type は ART ノードの Node4 から Node256 の識別に用いる．level は現在のノードの共通プレフィックスの長さである．child count にはノードの持つ子の数，deleted count にはノードから削除された子の数を保存する．compressed path には圧縮されたパスを保存する．compressed path は 8 バイトであるが，圧縮したパス長に 1 バイト使用するため，1 つのノードで最大 7 バイトのパスを圧縮する．UART ではパス中に完全にキーを保存する必要があるため，圧縮できるパスが 7 バイト以上の場合，複数ノードにわたってパスを圧縮する．

また，UART では対象の ART ノードまでのプレフィックスのみに対応するレコードを保持するためのノードとして，一時保存用の UB ノード（tmp UBNode）を各 ART ノードに用意する．つまり，ART ノード内の各レコードがそれぞれ 1 バイトのキーに対応した子ノードを持つのに対し，tmp UBNode は NULL キーに対応する UB ノードである．UART では前述したように各階層 1 バイトを用いて空間を 256 分割するが，あるキーに偏りが発生している際に不用意に空間分割してしまうと，分割した子ノードのほとんどにレコードが含まれない可能性がある．特に，UART はトライ木を基にした非平衡木であり，そのような疎な空間分割を繰り返すことで木全体のバランスが大きく崩れてしまう可能性がある．そのため，キーに対応するレコードが少数しか存在しないものを一時的にヘッダで保持することで，キーに偏りがある際の平衡の崩れを抑えている．

### 3.2 UB 層

UB 層は UB ノードと UB 木からなる．UB 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レコードの配列のペアを複数持つ．図 5 の leaf A は，共通プレフィックスが “AN” であり，“D” と “T” のサフィックスを持つ．したがって UBNode A はキー “AND”，値 $v_1$ のレコードとキー “ANT”，値 $v_2$ のレコードを持つ．

UB ノードは UB 木の葉ノードを二次索引に対応させたものである．図 8 に UB ノードのレイアウトを示す．UB ノードはヘッダとレコード領域，メタデータ配列から構成される．

ヘッダの ArtNode flag は ART ノードと同様，ART ノードと UB ノードの識別に用いる．prefix length は UB ノードの共通プレフィックスの長さである．block size にはレコード領域の大きさ，deleted size はレコード領域中の削除された領域の大きさを保存する．metadata array は対応するレコードへのアクセスに用いるメタデータの配列である．meta count には保持して

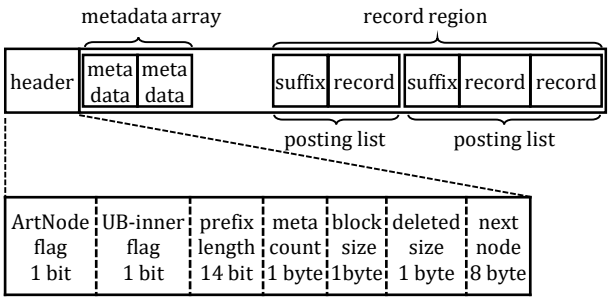



**図 8:** UB ノード

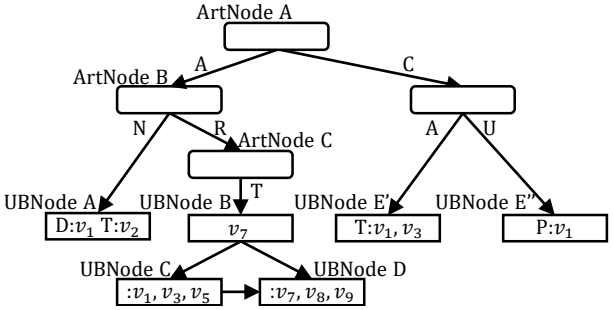



**図 9:** 部分木の存在するキー（“CUP”）を挿入した状態

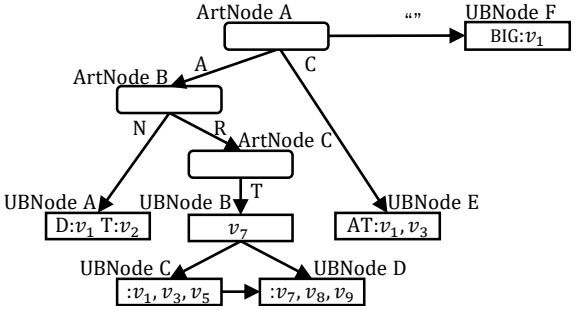



**図 10:** 部分木の存在しないキー（“BIG”）を挿入した状態

いるメタデータの数を保存する．UB-inner flag と next node は UB ノードが UB 木へ置き換わった際に使用する UB 木内の中間ノードの識別フラグと，兄弟ノードのポインタである．

レコード領域中には，キーのサフィックスと対応する複数のレコードである転置索引（図中 posting list）を持つ．転置索引には単一のサフィックスに複数のレコードを保存可能であり，二次索引として利用できる．転置索引中，レコードはその値で整列する．

メタデータ配列はレコード領域中の転置索引を参照するためのメタデータの配列である．各メタデータはその参照するレコードのサフィックスで整列している．同一のサフィックスを持つ転置索引が存在する場合，それらを指すメタデータは参照する最大のレコードで整列する．

### 3.3 問合せ処理

UART は挿入と削除，範囲検索をサポートする．キーは多次元座標で与えられ，Z 値に変換して処理する．

#### 3.3.1 更　　新

本項では，挿入および削除の問合せ処理を説明する．説明の簡略化のためキーを文字列として表現しているが，実際は Z 値で同様の処理を行う．

挿入は，2.3 節の検索処理と同様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 UB ノードを挿入する．挿入処理は主に対象キーとの共通のプレフィックスを持つ UB 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 UB ノードが存在している場合，到達した UB ノードに対象のキーのサフィックスとレコードを挿入する．UB ノードに空きがない場合，レコードを複数の新たな UB ノードに分割する．分割時，分割後の UB ノードを持つ新たな ART ノードを作成し，分割前の UB ノードが存在したパスに挿入する．図 9 に，図 5 にキー “CUP”，レコード $v_1$ を挿入した状態を示す．図 9 では，図 5 の UBNode E が UBNode E′ と UBNode E″ に分割している．新たに生成した ArtNode D に UBNode E′ と UBNode E″ を挿入し，ArtNode D は ArtNode 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 ART ノードの圧縮パスとして保存する．また，UB ノードはレコード数の多い順に最大で 4 つ作成し，残ったレコードは新たな ART ノードのヘッダ中の UB ノードに保存する．UB ノードの分割時，キーを 1 つしか持たない場合はその UB ノードをレコード内の値をキーとする UB 木に置き換える．

共通プレフィックスを持つ UB ノードが存在していない場合，ART ノードのヘッダ中の UB ノードに挿入する．つまり，ART ノードからキー長 0 の部分キーで，そのノードと同じ共通プレフィックスを持つ UB ノードに挿入する．図 10 に，図 5 にキー “BIG”，レコード $v_1$ を挿入した状態を示す．図 5 には “B” から始まるキーが存在しないため，ArtNode A からパス “B” は存在しない．そのため，図 10 では UBNode F を新たに作成し，ArtNode A のヘッダ（部分キー “” のパス）に挿入する．ヘッダ中の UB ノードに空き領域がなくなった場合，UB ノードを分割し，ART ノードの子ノードとして挿入する．上述の分割と同様に，最大で現段階の ART ノードの残りの子ノード数だけ新たに作成することで，段階的に ART ノードおよび空間の分割数を拡大する．

UB ノードの検索中，ART ノード中の圧縮パスと対象キーのサフィックスが一致しない場合は，一致しなかったサフィックスでの分岐を表す ART ノード生成する．つまり，検索キーと圧縮パスとで一致する部分を自身の圧縮パスとして持つ新たな ART ノードを生成し，元々の ART ノードを子ノードとして挿入する．その後，挿入対象のレコードについてはヘッダ中の UB ノードへサフィックスを用いて挿入する．図 11 に圧縮パスとサフィックスの不一致時の例を示す．元々の ART ノードは圧縮パス “AAAAA” を持つが，挿入キーのサフィックスは “ABCDE…” であり，一致しない．このとき，共通する部分キー “A” を持つ新たな ART ノードを作成し，元々のノードの圧縮パスを縮小して挿入する．新たな ART ノードは挿入キーとプレフィックスが一致するため，そのヘッダに UB ノードを作成して挿入する．

削除は挿入処理と同様に，2.3 節の検索処理で UB ノードを検索し，対象レコードを削除する．UB ノード内のレコードが全

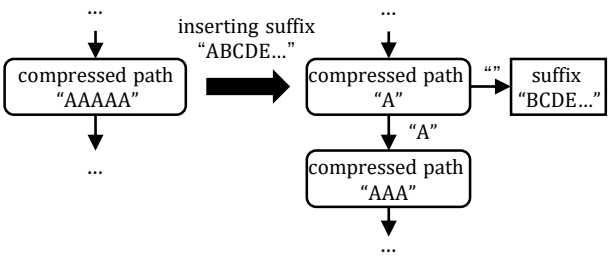



**図 11:** 圧縮パスの縮小と新規ノードの挿入

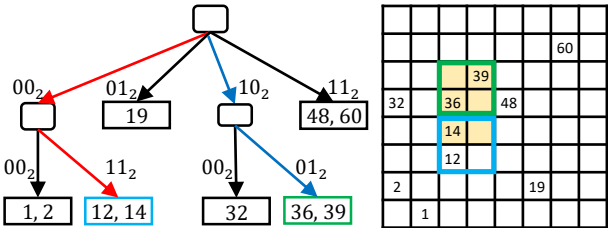



**図 12:** UART の範囲検索

て削除された場合，そのノードを親ノードから削除する．ART ノードの持つ子ノードが少なくなった場合，1 段階小さい ART ノードに縮退する．検索に失敗した場合は処理を打ち切る．

### 3.3.2 範 囲 検 索

範囲検索は，深さ優先探索によって検索範囲と重複する範囲を持つノードを探索する．入力として検索範囲の始点および終点の座標が与えられ，それらを Z 値に変換して以下のように処理する．

最初に，範囲内最小のキーを持つ UB ノードを，2.3 節の検索処理と同様に検索する．最小キーの検索処理に失敗した場合，そのキーより次に大きい範囲内のキーを持つ UB ノードに遷移する．UB ノードではノード中のレコードを走査し，範囲内のキーを持つレコードのみ返す．UB ノードの走査終了後，その親の ART ノードから次に検索範囲と重複する UB ノードに遷移する．ART ノードの走査終了後，同様にその親ノードから次に検索範囲と重複する子ノードに遷移する．ART ノードのヘッダ中の UB ノードにレコードが存在する場合，子ノードと同様にその UB ノード中のレコードを走査する．

図 12 右の色付き空間での範囲検索時の UART のノード遷移を図 12 左に示す．最初に，範囲内最小のキーである 14 を持つ UB ノードに赤色矢印のパスで遷移する．キー 14 を持つ UB ノードは，その範囲が完全に検索範囲内と重複していないため，UB ノード中の範囲内のレコードのみ収集する．その後根ノードに回帰し，根ノードは次に部分キー “$01_2$” の子ノードを持つが，その範囲は検索範囲と重複しないためスキップする．そして，青色矢印のパスで次に範囲と重複する UB ノードに遷移する．該当 UB ノードの範囲は検索範囲に完全に包含しているため，包含判定なしで UB ノード内全てのレコードを収集する．

## 4 評 価 実 験

本章では，UART の挿入と範囲検索処理に注目して性能を評価する．本実験は単一スレッドで行い，逐次的に命令を発行してそのスループットを測定する．表 1 に本実験で使用する環境を示す．

**表 1:** 実験用サーバの構成

| | |
|---|---|
| CPU | Intel(R) Xeon(R) Gold 6258R (two sockets) |
| RAM | DIMM DDR4 (Registered) 2933 MHz (16GB × 12) |
| OS | Ubuntu 20.04.5 LTS |
| Compiler | GNU C++ ver. 9.4.0 |

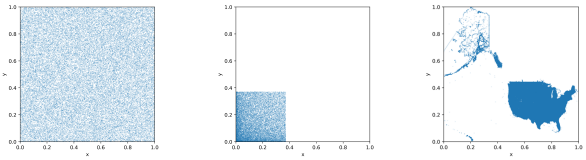
評価実験の比較対象として，UB 木と R* 木を用いる．UB 木は 2.1 節で紹介したように，キーとして Z 値を使用して多次元化した $B^+$ 木である．UB 木は C++を用い，二次索引として実装したものを用いる．R* 木は挿入アルゴリズムにより各ノードの対応空間を最適化し，範囲検索に特化した R 木 [12] である．R* 木は C++での boost[2] と libspatialindex[3] の実装を用いる．boost の実装は高速であるが 4 次元以上に対応していないため，高次元での性能特性のために libspatialindex の実装を用いる．

データセットとして，各次元の値が Zipf の法則に従うよう多次元値を生成したシミュレーションデータ（Zipf）と，アメリカの地形データ中のポイントオブジェクトを表す実データ（USA）を用いる．Zipf データセットは各次元独立に近似的に式 (1) に示す Zipf 分布に従う[4]．

$$f(k;\alpha,|W|) = \frac{1/k^{\alpha}}{\sum_{n=1}^{|W|} 1/n^{\alpha}} \tag{1}$$

Zipf データセットは Skew パラメータ（$\alpha$）を 0 から 1 まで変化させ，$\alpha$ が 0 のとき一様分布となる．USA データセットは OpenStreetMap[5] から取得したアメリカの地形データ中の 2 次元点のデータセットである．Zipf データセット，USA データセットともに各次元 32 ビット整数として用いる．図 13 に 2 次元で $\alpha = 0$，$\alpha = 0.3$ としたときの Zipf データセットと USA データセットの分布を示す．

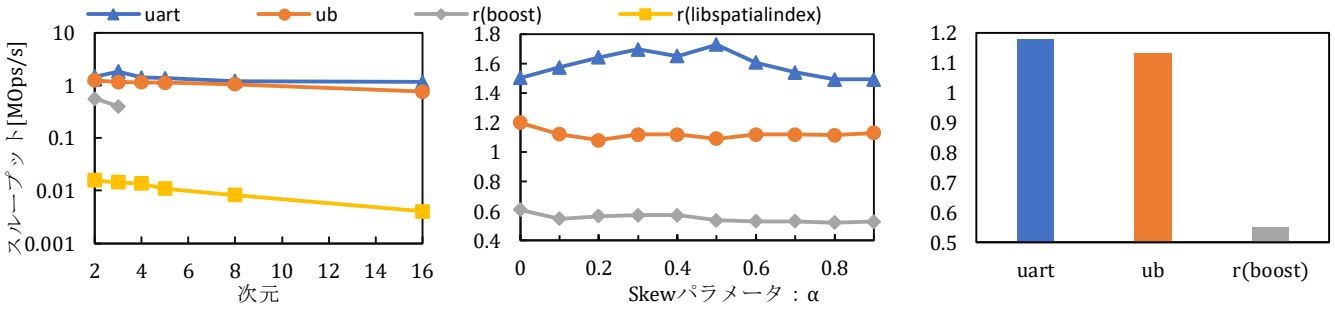
UART のハイパーパラメータとして UB ノードのページサイズが挙げられるが，本実験ではページサイズとして 8,192 バイトを使用する．挿入・範囲検索ともに，8,192 バイトで最良の結果を得られたため，以降の実験はページサイズを 8,192 バイトとする．同様に，UB 木でも最良の結果を得られたためページサイズを 8,192 バイトとする．



**(a)** Zipf( $\alpha = 0$) **(b)** Zipf( $\alpha = 0.3$) **(c)** USA

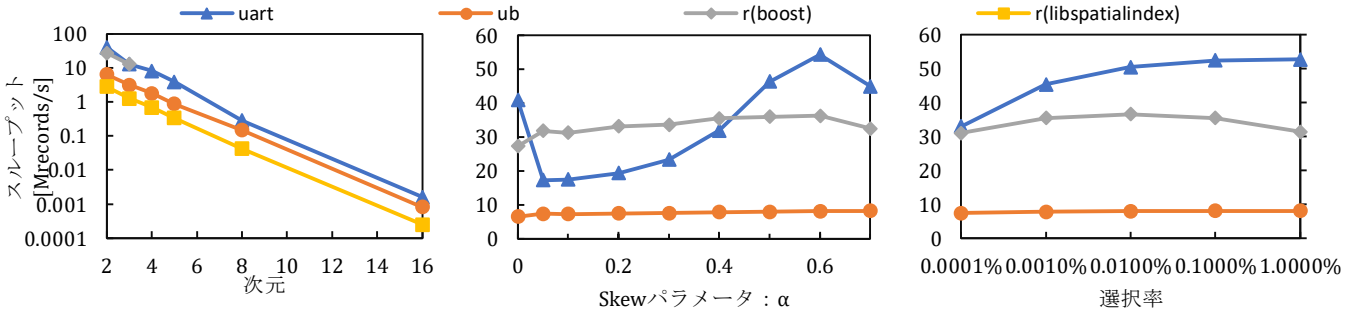
**図 13:** データセットの分布

2 : https://github.com/boostorg/geometry
3 : https://github.com/libspatialindex/libspatialindex
4 : https://github.com/dbgroup-nagoya-u/cpp-utility
5 : https://download.geofabrik.de/




(a) Zipf(α=0) 次元数を変化　(b) 2次元Zipf スキューパラメータを変化　(c) USA

**図 14:** 挿入のスループット




(a) Zipf(α=0) 次元数を変化　(b) 2次元Zipf スキューパラメータを変化　(c) USA 選択率を変化

**図 15:** 範囲検索のスループット

### 4.1 挿　　入

本節では，各索引構造の挿入のスループットの実験結果を紹介する．スループット測定前のデータ数は 1,000 万点であり，挿入回数は 100 万回である．図 14 に，各設定での挿入のスループットを示す．全ての設定において，UART のスループットが最高となった．

図 14(a) に Zipf データセット（Skew パラメータ：$\alpha = 0$）において次元数を変化させた時の挿入のスループットを示す．UART は 16 次元において 2 次元のスループットの 0.8 倍ほどで，高次元での性能劣化に一番頑健であった．UART と UB 木は，Z 値をキーとして扱うため，R* 木のような空間を直接扱う索引よりも高次元での性能劣化が小さくなったと考えられる．

図 14(b) に 2 次元，Zipf データセットで $\alpha$ を変化させた時の挿入のスループットを示す．UART は UB 木や R* 木と比較し，データの分布が性能に大きく影響する結果となった．UART はデータ分布の偏り大きくなるほどプレフィックスを共有しメモリ効率が良くなるため，$\alpha$ が 0.1 から 0.5 の時に性能が向上した．$\alpha$ が 0.5 より大きい時に性能が減少したのは，データの偏りが大きくなり，木の深さが大きくなったためだと考えられる．

図 14(c) に USA データセットでの挿入のスループットを示す．UART の性能は UB 木の 1.04 倍，R* 木の 2.1 倍ほどであった．

### 4.2 範 囲 検 索

本節では，各索引構造の範囲検索のスループットの実験結果を紹介する．スループット測定前のデータ数は 1,000 万点であり，範囲検索の実行回数は 1 万回である．なお，スループットは一秒あたりの範囲検索結果のレコード数を示す．図 15 に各設定での範囲検索のスループットを示す．多くの設定において，UART のスループットが最高であった．また，全ての設定で UART のスループットは UB 木を上回り，空間分割による範囲検索の効率化を確認できた．

図 15(a) に選択率 0.01%，Zipf データセット（Skew パラメータ：$\alpha = 0$）において次元数を変化させた時の範囲検索のスループットを示す．全ての次元において，UART のスループットが最高であった．UART は 16 次元において高次元での性能劣化が一番大きくなった．これは，次元数が大きくなったことで検索範囲との階層的な包含判定の効果が低くなったこと，および 1 レコードあたりのサイズが増えることでノード分割が多発し木全体の構造が非平衡に崩れていったことが原因として考えられる．

図 15(b) に 2 次元，選択率 0.01%，Zipf データセットで Skew パラメータを変化させた時の範囲検索のスループットを示す．$\alpha$ が 0 の時と 0.5 以上の時，UART の性能が最高となった．UART の木の構造はデータの分布に大きく影響し，特に $\alpha$ が 0.05 から 0.4 の時はレコード数の少ない UB ノードが大量に作成されることで性能が低下したと考えられる．

図 15(c) に USA データセットで選択率を変化させた時の範囲検索のスループットを示す．全ての選択率において，UART のスループットが最高であった．R* 木は選択率 1%以降において性能が減少するのに対し，UART は常に上昇傾向であった．UART は検索範囲の境界のノードのみで範囲内判定をする必要があるため，検索範囲が大きくなるほど性能が向上したと考えられる．

図 16 に選択率 0.01%，Zipf データセット（$\alpha = 0$）の各次元数で，検索条件に 2 次元のみを使用した時の範囲検索のスルー

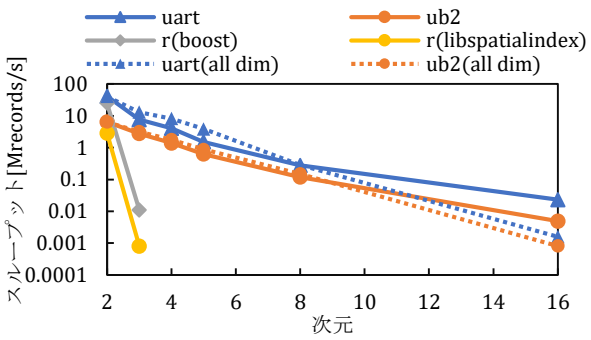



**図 16:** 2 次元の検索条件での次元の変化による範囲検索のスループット

プットを示す．検索条件に使用する次元は一様にランダムに選択し，それ以外の次元は全ドメインを検索範囲とした．なお，図中点線は UART と UB 木の全次元を検索範囲としたときのスループットである．libspatialindex の R* 木は 4 次元以上にも対応しているが，実行時間が長くなりすぎたため省略する．全ての次元において，UART が最高のスループットとなった．UART および UB 木は安定した性能であるのに対し，boost・libspatialindex ともに R* 木は 3 次元において急激に性能が悪化した．

## 5 関連研究

本章では，UART と同じくトライ木ベースの多次元索引として PATRICIA hypercube tree（PH 木） [13] と，adaptive cell trie(ACT) [9] を紹介する．

PH 木は，ART と同様に冗長なノードを圧縮したトライ木である PATRICIA 木 [14] を多次元索引に拡張したものである．PH 木は $k$ 次元データの Z 値を $k$ ビットずつ分割し，PATRICIA 木に適用して多次元拡張する．Z 値を $k$ ビットずつ分割することで，部分キーには各次元のビットを 1 ビットずつ保存する．各ノードは最大 $2^k$ 個の子ノードを持つが，その充填率によって 2 種類のノードを使い分ける．範囲検索では，各ノード下の部分空間で，範囲内最小・最大値の計算や検索範囲の包含判定をすることで効率化する．PH 木は次元数が増加すると，各ノードのサイズが指数的に大きくなり，挿入性能が悪化する．

ACT は ART にヒルベルト曲線や Z 曲線によるセル ID を適用することで，空間結合を効率的に処理する索引構造である．UART 同様，キーを 8 ビットずつ分割することで，空間を階層的にグリッド分割する．ACT では多角形の参照を，その領域と重複する空間を持つセル ID に保存する．空間結合処理では問合せ点の集合が与えられ，各点に対応する多角形を索引から検索する．ACT は高速に空間結合を処理できる一方で，挿入や範囲検索などの一般的な空間問合せに対応しない．

## 6 おわりに

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．また，実験によりその性能を測定した．UART はノードの空間分割の特性を範囲検索に利用することで，同じく Z 値をキーとする UB 木よりも範囲検索に特化することを実験により確認した．一方で，範囲検索に最適化した R* 木と同等の範囲検索性能で，挿入性能では上回ることを確認できた．したがって，UART は幅広いデータセットにおいて挿入・範囲検索ともに高性能であり，汎用的に使用可能な多次元索引であると考えられる．

今後の課題として，データセットの偏りによって不安定になる性能の解消が挙げられる．また，UART の元となった ART の同時実行制御の手法 [15] を取り入れ複数スレッドに対応すること，バウンディングボックスを用いることで点データだけでなくポリゴンデータにも対応すること [9] が挙げられる．

## 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．

## 文　献

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-Tree: An efficient and robust access method for points and rectangles,” in *Proc. SIGMOD*, pp. 322–331, 1990.

[2] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, “Effectively learning spatial indices,” *PVLDB*, vol. 13, no. 12, pp. 2341–2354, 2020.

[3] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, “Tsunami: A learned multi-dimensional index for correlated data and skewed workloads,” *PVLDB*, vol. 14, no. 2, pp. 74–86, 2020.

[4] 北川 博之, **データベースシステム**. 昭晃堂, 1996.

[5] R. Bayer, “The universal B-tree for multidimensional indexing: General concepts,” in *Proc. Worldwide Computing and Its Applications (WWCA)*, pp. 198–209, 1997.

[6] P. Rigaux, M. Scholl, and A. Voisard, *Spatial Databases with Application to GIS*. Morgan Kaufmann, 2001.

[7] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: ARTful indexing for main-memory databases,” in *Proc. ICDE*, pp. 38–49, 2013.

[8] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,” in *Proc. ICDE*, pp. 195–206, 2011.

[9] A. Kipf, H. Lang, V. Pandey, R. A. Persa, C. Anneser, E. T. Zacharatou, H. Doraiswamy, P. A. Boncz, T. Neumann, and A. Kemper, “Adaptive main-memory indexing for high-performance point-polygon joins,” in *Proc. EDBT*, 2020.

[10] T. Skopal, M. Krátký, J. Pokorný, and V. Snášel, “A new range query algorithm for universal B-trees,” *Information Systems*, vol. 31, no. 6, pp. 489–511, 2006.

[11] H. Tropf and H. Herzog, “Multidimensional range search in dynamically balanced trees,” *Applied Informatics*, pp. 71–77, 1981.

[12] A. Guttman, “R-Trees: A dynamic index structure for spatial searching,” *SIGMOD Rec.*, vol. 14, no. 2, pp. 47—-57, 1984.

[13] T. Zäschke, C. Zimmerli, and M. C. Norrie, “The PH-tree: a space-efficient storage structure and multi-dimensional index,” in *Proc. SIGMOD*, pp. 397–408, 2014.

[14] D. R. Morrison, “PATRICIA—practical algorithm to retrieve information coded in alphanumeric,” *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[15] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, “The ART of practical synchronization,” in *Proc. of the 12th International Workshop on Data Management on New Hardware*, pp. 1–8, 2016.