名古屋大学情報学部
コンピュータ科学科
卒業論文

Universal Adaptive Radix Tree における空間分割の改善に関する研究

2024 年 2 月

102030229　杉江 祐介

Universal Adaptive Radix Tree における空間分割の改善に関する研究

杉江 祐介

内容梗概

概要文.

キーワード: キーワード

目次

図目次

表目次

第1章

はじめに

多次元索引は複数の次元で表されたキーによる検索を補助するデータ構造である．地理情報システムにおける空間オブジェクトの検索や，データベース管理システムにおける多次元データの選択演算効率化などに利用される．多次元データは複数の次元から成るが，メモリやストレージ上では 1 次元空間上に配置されるため，多次元空間上での局所性を適切に反映した索引構造が求められる．

多次元索引は多次元空間を直接扱うものと 1 次元空間へ射影して扱うものの大きく 2 つに分けられ，本稿では後者を主に扱う．多次元空間を直接扱う索引の代表例は R 木 [1] であり，多次元の包囲矩形などを用いて多次元空間を階層的に分割していく．1 次元空間へ射影するものは主に空間充填曲線に基づいており，Universal B 木（UB 木）[2] が代表である．後者の利点は，既存の効率的な 1 次元索引を流用でき，挿入・削除に伴う木の構造変更などが容易な点である．一方で欠点として，多次元データを 1 次元化するため，隣接するデータが多次元空間上では近接していない可能性がある点が挙げられる．

本稿では UB 木および Adaptive Radix Tree（ART）[3] を組み合せた索引構造である Universal ART（UART）[4] の改善について述べる．UART は汎用的に使用可能な多次元索引だが，データに偏りがある場合に挿入と範囲検索の性能が低下するという課題を持つ．性能低下の原因は，不適切な空間分割による疎な部分空間の生成である．UART ではノードの空きスペースがなくなった際に対応する多次元空間を分割することでノードを分割するが，分割後の部分空間が十分な数のレコードを持つという保証がない．そのため挿入されるレコードに空間的な偏りがある際に，レコードを少数しか持たない疎なノードが多数生成され性能を低下させている．

本研究では分割後の各空間が十分な数のレコードを持つよう空間分割の手法を改善する．まず，UART の基となる ART と UB 木の概要について述べ，続けて UART の概要について述べる．最後に，既存手法における空間分割の問題点と空間使用量に基づく空間分割の改善について説明し，論文全体のまとめを述べる．

第2章

関連研究

UART の構成要素として利用する UB 木と ART について説明する．

2.1 Universal B木（UB木）

UB 木 [5] は，多次元空間を Z 階数曲線に基づき 1 次元空間に変換し，変換後の Z 値を B^+ 木のキーとして使用する索引構造である．構造自体と，検索・挿入・削除といった各操作は B^+ 木と変わらない．そのため，UB 木の特徴である Z 階数曲線と B^+ 木について述べる．

Z 階数曲線 [6] は空間充填曲線の一種であり，2 次元空間であれば “Z” の記号を描くような曲線となる．Z 値は Z 階数曲線上の開始点からの距離であり，距離が近いものは多次元空間上でも近いことが多い．ただし，空間充填曲線の性質上，Z 値としては隣接するものが多次元空間上では遠く離れる場合もある．Z 値への変換は，多次元座標の各座標を 2 進符号化し上位ビットから順に交互配置することで行う．

B^+ 木 [7] は，図 2.1 に示すように索引部とデータ部からなるバランス木である．索引部はレコードのキー値とデータ部へのポインタが格納された中間ノードの集まりである．一方，データ部はキー値に対応するレコードと，隣の葉ノードをつなぐポインタが格納された葉ノードの集まりである．そのため，木の高さが葉ノードの広がりに比べ低くなりノードアクセス数が減少する．

以上の構造から，B^+ 木は読み取り操作（点検索および範囲検索）・書き込み操作両面においてバランスの取れた性能を持つ．索引部がレコードそのものではなくキー値のみになっていることから，点検索におけるキー値の比較や書き込み操作による索引部の変更

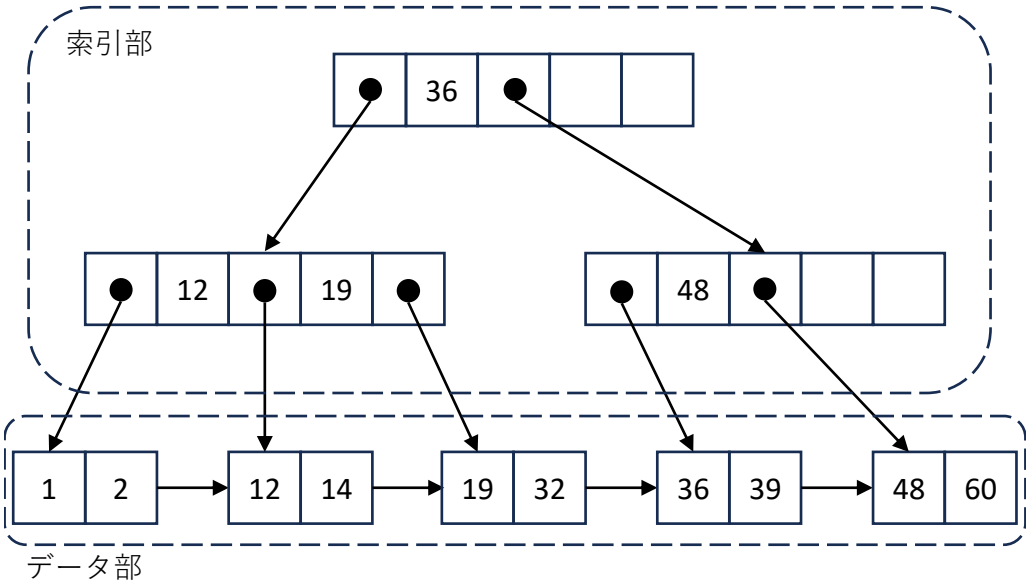

図 2.1　B^+ 木の構造

が効率よく行える．更に，葉ノードのポインタをたどることで全検索や範囲検索も高速に行うことができる．また，各ノードはが全て同じサイズの領域として管理されるのでメモリ管理が容易となり利用効率も向上する．

UB 木は B^+ 木の一種であるためそのデータ構造は効率的だが，空間分割が非効率的であるという問題がある．UB 木はノードの空き容量がなくなった際にノードを分割するが，分割点は分割後のサイズによって決定され，空間的な面は基本的に考慮されない．つまり，多次元空間上で非連続な領域が同じノードに割り当てられる可能性があり，各ノードに対応する多次元包囲矩形が過剰に拡大しうる．

2.2　Adaptive Radix Tree（ART）

ART [3] は基数木（Radix Tree）の拡張であり，レコード数に応じてノードサイズを動的に選択する索引構造である．また，基数木はトライ木の空間利用効率を最適化した索引構造である．そのため，本節ではトライ木と基数木の概要について述べてから，動的選択の概要について述べる．

トライ木は，文字列を表すための索引構造である．根ノード以外のノード一つ一つが 1 文字を表し，文字の種類数だけ子ノードへのポインタを持つ．文字列は，根ノードか

ら葉ノードまでに通ったノードの順で文字をつないだものである．

トライ木は，文字列の検索速度に優れるが，空間利用効率が非常に大きい．全ての子ノードが文字の種類数だけポインタを持つため，文字の種類数が多かったり，文字列が長かったりすると非常に大量のメモリを消費する．加えて，ポインタの多くがヌルポインタとなり空間利用効率も悪い．そこで，トライ木の空間利用効率を抑えた索引構造が基数木である．

基数木は，基本的にトライ木と同様の索引構造である．基数木とトライ木の違いは，子ノードが一つしかないノードをひとまとめにすることである．つまり，ある文字の後に続く文字が 1 種類しかない場合は，ノードに文字列を対応させる．まとめた文字が多いほど，ノードの数が減り空間使用量を抑えられる．しかし，まだ多くのノードがヌルポインタを持つため空間利用効率はよくない．そこで，大量のヌルポインタを減らした索引構造が ART である．

ART では与えられたキーを 1 バイトの部分キーに分割し索引を構築するため，各ノードで最大 256 個のレコードを保持する．そこで，ヌルポインタの数を減らすために，子ノードの数に応じて使用するノードを変える．選択されるノードは Node4，Node16，Node48，Node256 の 4 種類であり，それぞれ対応する数のレコードを格納できる．各ノードは保持するレコードの数だけでなく，保持する方法も違う．

Node4 は，キーと子ノードへのポインタを 4 個ずつ持つ．キーとキーに対応する子ノードへのポインタは，どちらも i 番目に保持される．ここで，キーが保存される配列を key_array，ポインタが保存される配列を $pointer\_array$ とすると，$key\_array[i] = key$，$pointer\_array[i] = child\_pointer$ が成立する．つまり，i 番目のキーは i 番目のポインタに対応している．Node16 もキーとと子ノードへのポインタを持つ数が 16 になっただけで，Node4 と同じ構造である．

Node48 は，キーと子ノードへのポインタを 48 個ずつ持つ．しかし，key_array のサイズは 256 である．そして，キーの値がキーが格納されているインデックスの値と一致し，キーの値と対応する子ノードへのポインタが格納されているインデックスの値と一致する．つまり，$key\_array[key] = i$，$pointer\_array[i] = child\_pointer$ が成立する．

Node256 では，キーと子ノードへのポインタを 256 個ずつ持つ．key_array はなく，キーが $pointer\_array$ のインデックスに一致する．つまり，$pointer\_array[key] = child\_pointer$ が成立する．

ART は部分キーに偏りのある箇所では容量の少ない Node4 や Node16 が使用され，基数木の欠点である空間利用効率の悪化を克服している．UART でもキーの管理にノー

ド選択を用いることで空間利用効率を向上させている.

第3章

Universal Adaptive Radix Tree (UART)

UART は UB 木を拡張し，主に空間分割を担う ART 層と挿入されたレコードの保持を担う UB 層に分けた多次元索引である．Z 階数曲線は多次元空間を 1 次元上の値として表現しており，上位ビットから下位ビットに進むにつれ粒度の細かい空間分割が行われる．そこで，UART では変換後の Z 値を 1 バイトずつの部分キーに区切り，部分キーによって表される空間分割を ART により管理する．つまり，ART 層では多次元空間上でレコードが密な領域ほど空間が分割され，深い階層が生成される．UB 層は挿入されたレコードを保持するデータ層であり，該当領域に存在する疎なレコード群への効率的な範囲走査を実現する．

図 3.1 に UART の概念図を示す．本来の UART では Z 値を 1 バイトずつに区切るが，図 3.1 では Z 値を 4 ビットごとに区切っている．また，ART 層の丸はレコードを，UB 層の丸は UB 木のノードを表している．そして，区切られた Z 値のうち上位 4 ビットと中位 4 ビットは各層のグレー色で塗りつぶされた区画に，下位の 4 ビットは最下層にある灰色のレコードに対応している．以下では，ART 層と UB 層の構造について概略を述べる．

■ART 層　ART 層は Z 値をキーとして ART に格納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．ART ではキーを 1 バイトの部分キーに分割するため，各階層で空間は 256 分割される．分割された空間のうちレコードが密に存在する箇所は後述する手続きによって下の階層が生成され，再帰的により細かい空間分割が行われる．一

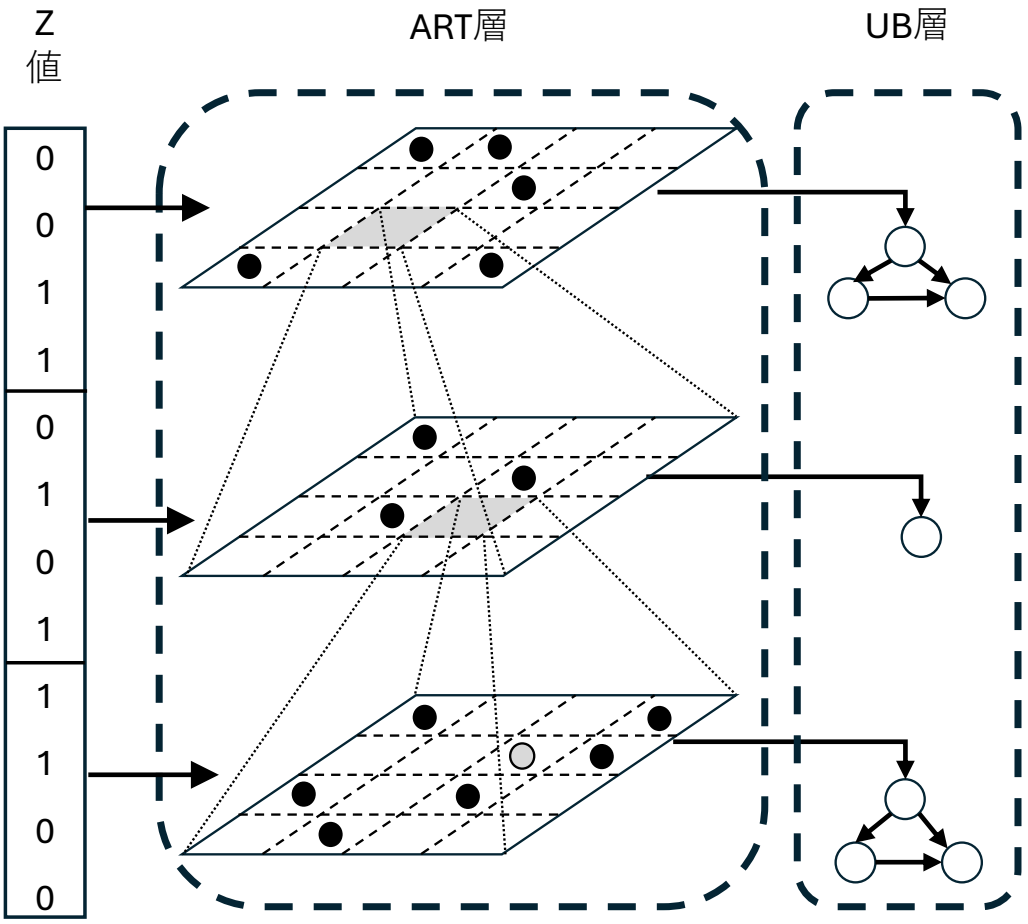

図 3.1　UART の構造

方，ART 中の各ノードは UB 木の根ノード（i.e., UB 層へのポインタ）をヘッダ領域に持ち，下の階層を持たない空間のレコードはそちらで保持される．

■UB 層　UB 層は ART 層の各階層各空間で保持される UB 木の集合であり，レコードの実体はこちらで管理される．各 UB 木はその階層における Z 値の部分キーを検索キー，挿入された空間オブジェクトとペイロードの組をレコードとして持つ．なお最下層，つまり Z 値の全長を用いた空間分割後の UB 木は，部分キーとなる Z 値を持たないため一般的な B^+ 木として生成される．UB 木の構築方法は既存のものと同様であり，B^+ 木に由来する効率的な範囲走査を可能とする．ただし，既存研究において各 UB 木は根ノードのみしか持たない点に注意する [4].

第4章

UARTにおける空間分割の効率化

既存研究の課題として，オブジェクトが疎に分布する空間の管理が不十分であり，偏った分布における性能低下が挙げられる [4]．例えば全オブジェクトが 1 つの部分空間に偏る場合は，最下層まで ART が展開され，B^+ 木によって全オブジェクトが管理されるため空間効率に関する問題は発生しない．同様に，オブジェクトが空間全域で一様に分布する場合は，各部分空間に対応する UB 木が一定数のオブジェクトを管理するためこちらも問題ない．しかし，分布が適度に偏っているとき，既存の UART では少数のレコードしか持たない UB 木が多数生成され性能が低下してしまう．

分布の偏りによる性能低下は，既存研究における UB 木が根ノード 1 つしか持たず，オブジェクトが疎な空間のレコードを十分に保持できないためである．既存研究では UB 木の根ノードを一種の書込み用バッファとして利用しており，空き容量がなくなった際は即座にオブジェクト数が最も多い部分空間を選び ART 層で下層を生成する．この密な空間を選ぶという方針自体は誤っていないが，部分空間における空間使用量を考慮せず，単純にレコード数の多い空間を選ぶという手続きに問題がある．つまり，各部分空間のオブジェクト数の差が少ない（i.e., 分布がロングテールを持つ）とき，この手続きではオブジェクト数が少ない部分空間においても下層が生成されてしまう．現実的なワークロードやデータセットにおいてロングテールは頻出するため，この問題への対応は必須である．

そこで本研究では，下層生成の条件として一定の空間使用量を持つことを使用し，UB 層において根ノード単体ではなく UB 木としてレコードを保持する．UB 木は Z 値を使用した二次索引であり，同じ Z 値を持つレコードは内部で転置リスト（posting list）として保持される．既存研究では各転置リスト内のレコード数のみを見たが，本研究で

は転置リストのサイズも考慮し一定のサイズを超えるまで下層を生成しない．つまり，ノードの空き容量がなく下層も生成しない場合はノードを分割し，UB 木を拡張する．これにより，生成された下層には一定以上のオブジェクトが必ず含まれるため，各ノードの空間利用効率およびそれに伴う範囲走査性能の向上が見込める．

第5章

評価実験

本章では，偏りなどデータのパラメータに対する頑健性と，実データを用いた有用性を評価する．そのために，各データに対して読み込みと書き込みを行い，それぞれのスループットを測定する．以下ではより詳細な実験方法について述べる．

本実験では，頑健性・有用性のどちらの評価においても，用意したデータに挿入，構築・破棄，範囲検索処理を行いスループットを測定する．比較対象は既存手法の UART と，多次元索引としてよく使われる UB 木と R^* 木である．R^* 木は空間利用効率が高く範囲検索に優れており，データの次元数に合わせて boost と libspatialindex の 2 つの実装を用いる．boost の実装が，libspatialindex の実装に比べ高速な一方で，4 次元以上に対応していないためである．どの実験も全て単一スレッドで行い，命令を逐次的に発行する．表 5.1 に本実験で使用する環境を示す．

そして，有用性の評価には，日本，アジア，地球の規模の異なる 3 つの地理データセットを用いる．また，パラメータに対する頑健性の評価には，各次元の値が Zipf 関数

表 5.1　実験用サーバの構成

Item	Value
CPU	Intel(R) Xeon(R) Gold 6262V (two sockets)
RAM	DIMM DDR4 (Registered) 2933 MHz (16GB × 14)
OS	Ubuntu 22.04.2 LTS
Compiler	GNU C++ ver. 9.4.0

に従うシミュレーションデータを用いる．

Zipf 関数は各次元の値が Zipf の法則に従うように多次元値を生成する関数である．Zipf の法則は，ある要素の出現頻度とその大きさに双曲線の関係が成り立つという経験則である．そして，Zipf データセットは各次元独立で近似的に式 (5.1) に示す Zipf 分布に従う[*1]．

$$f(k;\alpha,|W|) = \frac{1/k^{\alpha}}{\sum_{n=1}^{|W|} 1/n^{\alpha}} \tag{5.1}$$

式 (5.1) 中のαはスキューパラメータであり，$\alpha = 0$ の時に Zipf データセットは一様分布となる．

5.1　ページサイズ

UART のハイパラメータに葉ノードのページサイズがある．ページサイズはレコードの容量として捉えられるため，挿入操作や範囲検索の結果を左右することが容易に考えられる．そこで，UART とその他の索引構造との比較する前に，UART の改善後と改善前においてページサイズが各性能に与える影響を評価する．

UART の改善前後で挿入と範囲検索のスループットを測定した．以降では，UART の改善後を UART（改善後），UART の改善前を UART と表記する．UART（改善後）は全てのページサイズにおいて UART よりよい結果が得られた．挿入では，UART（改善後）は UART の 1.3 倍から 1.8 倍ほどのスループットになった．UART（改善後）ではページサイズの影響が小さく，UART ではページサイズが大きくなるとスループットが小さくなった．また範囲検索では，UART（改善後）は UART の 1.5 倍から 1.9 倍ほどのスループットになった．挿入のときとは反対に，UART（改善後）ではページサイズが大きくなるとスループットも大きくなり，UART ではページサイズの影響が小さかった．結果を図 5.1 に示す．また，実験に用いた UART のパラメータを表 5.2 に示す．

UART（改善後）が全ての項目で UART を上回った理由は，空間利用効率が上昇したためだと考えられる．UART（改善後）は UART に比べ空間を分割しないため，下層が比較的生成されない．階層が低いと，各操作におけるページアクセス数が減少する．そのため，挿入や範囲検索時にキーを見つける時間が減少し，性能が向上したと考えられる．

[*1] https://github.com/dbgroup-nagoya-u/cpp-utility

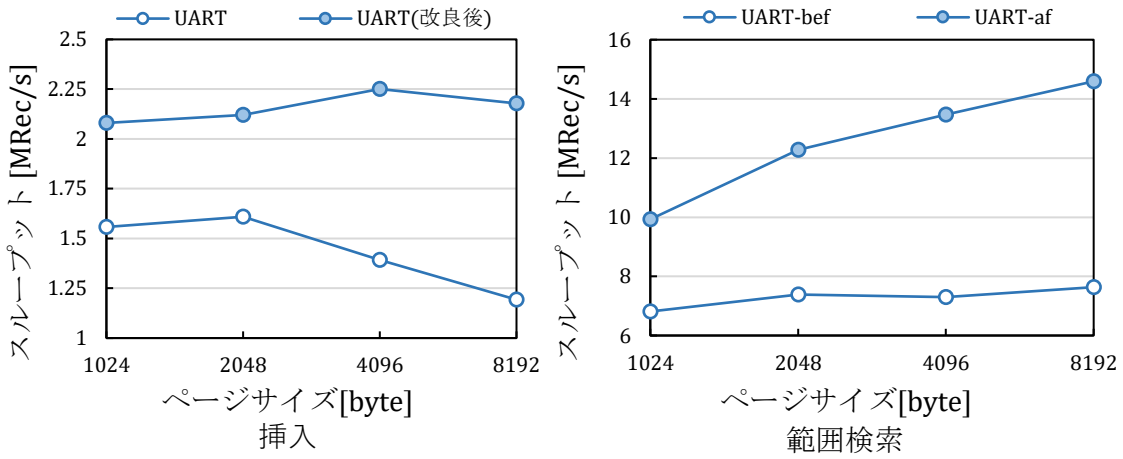

図 5.1　UART の改善における挿入・範囲検索の性能比較

表 5.2　UART のパラメータ

比較対象	操作	型	セット	次元	レコード数	スキュー	選択率
UART	挿入	UInt4	Zipf	2	1,000,000	0.3	NONE
UART（改善後）	挿入	UInt4	Zipf	2	1,000,000	0.3	NONE
UART	範囲検索	UInt4	Zipf	2	1,000,000	0.3	0.0001
UART（改善後）	範囲検索	UInt4	Zipf	2	1,000,000	0.3	0.0001

挿入において，UART（改善後）ではページサイズの影響が小さかった．その理由は，ページアクセス数の減少による時間短縮と挿入に伴う木構造の変化による時間延長がトレードオフであるためだと考えられる．葉ノードにレコードが挿入できなくなった場合，ART 層で下層生成するか UB 層でノードを分割する．ページサイズが小さい場合，下層生成が多く発生し木は高くなるその一方で，ノードの分割回数と 1 分割当たりにかかる時間は小さくなる．ページサイズが大きい場合，下層生成が発生しなくなり木は低くなる．その一方で，ノードの分割回数と 1 分割当たりにかかる時間は大きくなる．つまり，空間利用効率を最適化すると，下層生成とノード分割にかかる時間の増減が打ち消しあうと考えられる．また，UART ではページサイズが大きくなるとスループットが小さくなった．その理由は，ページアクセス数の減少による時間短縮と挿入に伴う木構造の変更による時間延長がトレードオフでないためだと考えられる．空間分割が非効率だと，ページサイズが大きくても木が低くならない．そのため，ページアクセス数が増加し，更にノードの分割回数と 1 分割当たりにかかる時間が大きくなる．

範囲検索において，UART（改善後）ではページサイズが大きいときにスループットが良くなった．その理由は，空間利用効率が最適化されたことによりページアクセス数が減少したためだと考えられる．一方で UART ではページサイズの影響が小さかった理由は，ページサイズを変えても木の高さはほとんど変わらないことが原因だと考えられる．空間分割戦略が非効率だと必要以上に高い木となりページアクセス数が増えてしまう．範囲検索では特にページアクセス数がボトルネックになるので木の高さが性能に強く影響すると考えられる．

以上から，UART（改善後）はページサイズを大きくするとページアクセス数が減少し，読み込み性能が向上すると考えられる．そのためこれ以降の実験では，UART（改善後）と UART のページサイズ数を 8,192 バイトに設定する．

5.2　構築と破棄

第6章

おわりに

本研究では，UART における空間分割の効率化によるメモリ利用効率の向上について提案した．空間使用量を考慮していないという既存研究の課題について説明し，UART における空間使用量を考慮した空間分割について述べた．今後は提案手法の実装，および既存研究との比較による有効性の検証を行う予定である．

謝辞

本研究の一部は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594 の助成，日本電信電話株式会社との共同研究，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．

謝辞

謝辞.

付録 A

付録

参考文献

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

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

[4] 駿. 鈴木, 健. 杉浦, 佳. 石川, and 可. 陸, “Adaptive radix tree の多次元索引への拡張.” 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（DEIM2023），2A-2, 2023.

[5] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, “Integrating the UB-tree into a database system kernel,” in *Proc. VLDB*, pp. 263–272.

[6] V. Gaede and O. GÃ¼nther, “Multidimensional access methods,” *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.

[7] 博. 北川, **データベースシステム**. 昭晃堂, 1996.

関連発表論文

学術雑誌論文

1.

国際会議論文（査読付）

1.

国内発表

1.