重み付きインターバルデータに対するTop-k範囲探索

Lee Jimin[1]　天方大地[2]　原隆浩[3]

本稿では，クエリインターバルに重なるインターバルの集合において，最も重みの大きいk個のインターバルを返すTop-k範囲クエリに注目する．これは，イベント追跡やスケジューリング等において，与えられた範囲内で最も有意味な情報を取り出したい場合に有用である．既存アルゴリズムでは，クエリインターバルに重なる全てのインターバルを計算し，それらの中から重みが最も大きいk個のものを取り出す必要があるため，非効率的である．本稿では，この非効率性を解消するための2つのアルゴリズムWIT（Weighted Interval Tree）およびSAIT（Segmented AIT）を提案する．実データを用いた実験により，提案アルゴリズムが既存アルゴリズムよりも高速であることを示す．

1　序論

動機. インターバルデータの処理は，時間データベース [2]や計算幾何学 [3]等の様々な分野で研究されている．特に，インターバルデータにおける範囲探索は重要な問題であり，従業員の稼働状況の追跡 [11]からタクシーの軌跡の分析 [1]まで広く適用されている．しかし，範囲検索は結果のサイズをコントロールできないため，膨大な数のデータが結果に含まれる可能性がある．そこで，範囲検索結果に含まれるデータをランク付けし，重みやタイプ等の特定の基準に基づいて最も関連性の高いk個のインターバルを抽出することを考える．つまり，本稿ではTop-k範囲クエリに着目する．これは，重み付きインターバルの集合から，与えられたクエリ範囲に重なる最も重みの大きいk個のインターバルを出力する問題であり，以下の応用例が考えられる．

- **交通分析**：2023年10月20日の12時から14時の間で最も高額だったタクシー運行を検索する．
- **イベント管理**：2022年夏に世界各地で開催されたイベントのうち，最も参加者の多かった3つのイベントを検索する．
- **仮想通貨分析**：過去4時間以内に発生したビットコインの最大の値下がり幅を10件検索する．

交通分析の例では，運輸会社はクエリ結果を活用して，運賃のピーク時間を特定し，価格戦略を最適化することができる．イベント管理では，クエリ結果から将来のイベント計画や資源の効果的な配分が可能になる．また，インターバルデータを利用した通貨の分析により，投資家やアナリストは市場の変動を研究し，戦略を立てることができる．

課題. インターバルデータを扱うデータ構造として，インターバル木 [9]，セグメント木 [3], ピリオドインデックス [10]，およびHINT [1]等が提案されているが，Top-k範囲クエリに対しては効率的でない．範囲検索に最もよく使われるデータ構造であるインターバル木は，$O(N)$の空間を使用し，範囲クエリを$\Omega(M)$時間で応答することができる．ここで，Nはインターバルの数，Mはクエリ範囲と重なるインターバルの数である．インターバル木を用いてTop-k範囲検索を処理する単純な方法として，範囲検索結果を全て取り出し，それを重み順にソートした後，上位k個のインターバルを返す方法がある．この方法の時間計算量は$\Omega(M \log M)$で，Mが大きい場合に非効率的である．他のデータ構造も同じ問題を抱えている．文献 [14]では重みを考慮するデータ構造を提案しているが，スタビングクエリに特化しており，範囲探索の場合はインターバル木と同じ処理が必要となる．そのため，Top-k範囲クエリを効率的に処理するためには，新しい技術（データ構造）が必要である．

貢献.　上記の課題に対応するため，2つのアルゴリズムWITとSAITを提案する [13]．WITは，簡易化したインターバル木を拡張して各ノードに重み付けをすることにより，クエリに重ならない，または重みが小さいインターバルを枝刈りするよう設計されている．WITは実践的には高速であるが，最悪の時間計算量が$O(N \log k)$であるため，理論的には非効率である．そこで，理論的に高速なクエリ処理を実現するため，SAITを提案する．SAITはAIT [12]を拡張し，クエリ範囲と重なるインターバルにおいて重みの大きいものへの高速なアクセスを可能にすることにより，時間計算量を$O(\log^2 N + k \log N \log(\log N) + k \log N \log k)$に削減している．本研究では，理論的な分析と実データを用いた実験を通して，提案アルゴリズムが既存アルゴリズムを上回る性能を持つことを示す．

2　問題定義

インターバルxを$[x.start, x.end, x.weight]$と定義し，$x.start$は始点，$x.end$は終点，および$x.weight$は重みとする．インターバルの集合$X$とクエリ範囲$q = [q.start, q.end]$が与えられたとき，$q \cap X$を$q$と重なる$X$内のインターバルの集合と定義する．つまり，$q \cap X = \{x \in X \mid x.start \leq q.end \ \wedge \ x.end \geq q.start\}$である．本論文で取り組む問題を以下で定義する．

定義 1（Top-k範囲クエリ）　インターバルの集合X，クエリインターバルq，および結果のサイズkが与えられたとき，このクエリは$q \cap X$の中から重みが最も大きいk個のインターバルを出力する．

3　WIT

本章では，Top-k範囲クエリに効率的に応答するために設計したWITを紹介する．

[1] 非会員　大阪大学
lee.jongmin@ist.osaka-u.ac.jp
[2] 正会員　大阪大学
amagata.daichi@ist.osaka-u.ac.jp
[3] 正会員　大阪大学
hara@ist.osaka-u.ac.jp

3.1 事前知識

3.1.1 インターバル木

インターバル木 [9]は，範囲クエリを効率的にサポートする二分探索木であり，各ノードu_iは以下の要素を持つ．

- c_i：インターバルの中点に基づいて計算される中心点
- L_i^l：c_i と重なるインターバルのリスト（始点順に整列）
- L_i^r：L_i^l と同じインターバルのリスト（終点順に整列）
- u_i^l：u_i の左の子ノード
- u_i^r：u_i の右の子ノード

インターバル木は，各ノードでインターバルの中点の中央値を基準に中心点を選択し，領域を再帰的に分割する．各ノードu_iにおいて，c_iよりも終点が小さいインターバルは左の部分木に，始点が大きいインターバルは右の部分木に配置される． c_i と重なるインターバルはノードu_iに格納される．この処理は，左および右の部分木に対して再帰的に続き，各部分木は自身の中心点を選択する．図1はインターバル木の例を示している．

3.1.2 簡易版インターバル木

WITの基盤として簡易版インターバル木をまず設計する．これは，インターバル木に似た二分探索木であるが，各ノードに1つのインターバルしか含まない構造である．簡易版インターバル木の各ノードu_iは以下の要素を持つ．

- x_i: 中心インターバルであり，インターバルの端点または中点の中央値に基づいて計算される．
- u_i^l: u_i の左の子ノード．
- u_i^r: u_i の右の子ノード．

中心インターバルはインターバルの中点（または端点）の中央値に基づいて計算される．中点が中央値より小さいインターバルは左の部分木に配置され，中点が中央値より大きいインターバルは右の部分木に配置される．中心インターバルは現在の根に格納され，各ノードで再帰的にインターバルを分割して格納する．この構造の空間計算量は$O(N)$である．図2は，始点を基準で中心インターバルを選択した場合の例を示す．

簡易版インターバル木自体は効率的な範囲探索をサポートしない．範囲クエリを処理する際，中心インターバルが下位レベルのノードの配置を保証しないため，全ての場合において両方の部分木を探索する必要がある．そのため，この木では範囲クエリに$O(N)$時間かかる．しかし，この制約にもかかわらず，WITは

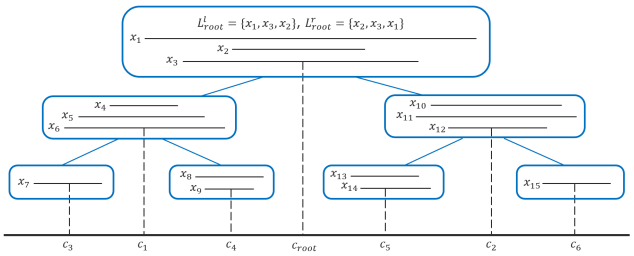

図1 インターバル木の例

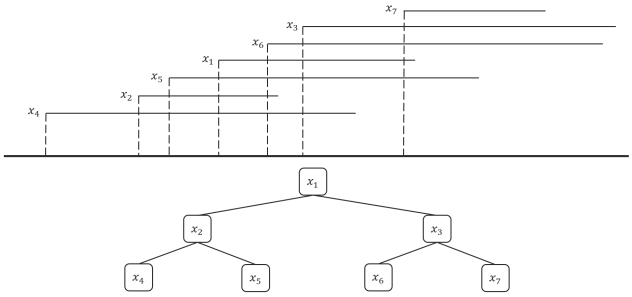

図2 始点を基準で構築された，簡易版インターバル木の例

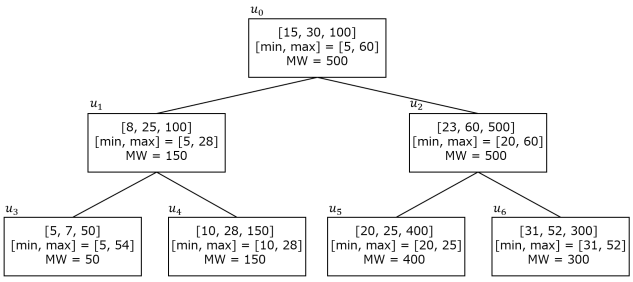

図3 WITの例

この簡易版インターバル木を基盤としている．1つのノードに1つのインターバルしか持たない設計により，各ノードに重みを追加することが簡単になり，Top-kクエリにおける枝刈りが単純化されるからである．

3.2 概要

WITは簡易版インターバル木の派生型である．主なアイデアは，各ノードにその部分木内の最大重みを付加することである．これにより，Top-kに含まれないインターバルしか管理されていない部分木の枝刈りが可能となる．

構造. WIT の各ノードu_iには，簡易版インターバル木の要素に加えて以下の要素が含まれる．

- MW_i：u_iを根とする部分木に存在する全てのインターバルの中での最大重み．
- $[\min_i, \max_i]$：u_iを根とする部分木に存在するインターバルの端点の最小値と最大値（部分木の区間）

構築. Xが与えられた場合，まずインターバルを始点でソートする．その後，この順における中央のインターバルを選び，残りのインターバルを2つのリストに分割する．始点が中央値より小さいインターバルは左部分木に，大きいインターバルは右部分木に割り当てられる．中央のインターバルに対応するノードu_{root}が作成され，そのmin, max, MWは上述の定義に従って初期化される．この処理を再帰的に繰り返すことにより，WITが構築される．図3はWITの例を示している．

WITの構築には，ソートのコストとインターバル木の構築コストにより$O(N \log N)$時間がかかる．空間計算量は$O(N)$である．

3.3 クエリ処理

WITは，Top-k範囲クエリの結果に寄与しない部分木を枝刈りすることにより効率的なクエリ処理を実現している．サイズkのヒープHを使用して，Top-kインターバルの候補を記録する．クエリ範囲と重なるインターバルxが見つかった場合，$x.weight$が現在のH内のインターバルにおける最小の重みより大きい，または$|H| < k$である場合にHにxが追加される．

根ノードから探索を開始し，次のようにノードを辿る．現在のノードu_iにおいて，x_iがクエリ範囲と重なり，Top-kの条件を満たす場合，Hに挿入する．その後，各子ノードを根とする部分木がTop-kインターバルを含む可能性を確認する．その子ノードのMWがH内のインターバルにおける最小の重み以下の場合，その部分木は枝刈りされる．また，その部分木がクエリ範囲と重ならない場合も，その部分木を訪問する必要はない．この処理により，簡易版インターバル木で必要だった全ての部分木の探索を避けることができる． Algorithm 1は，WITがTop-k範囲クエリを処理する擬似コードを示している．

例1. 図3において，クエリ範囲$q = [29, 40]$，$k = 2$のTop-k範囲クエリを想定する． x_0はqと重なり，Hは空であるため，x_0がHに追加される．次に左の子ノードu_1を確認する．左部分木の領域$[5, 28]$はqと重ならないため，左部分木は枝刈りされる．次に，右の子ノードu_2を評価する．右部分木の領域$[20, 60]$もqと重なるため，u_2を探索する． x_2はqと重なり，Hに追加される．これにより，Hは2つのインターバル$[15, 30, 100]$および$[23, 60, 500]$を含む．その後，u_5が訪問される． MW_5 が H の最大要素（$[15, 30, 100]$）の重みを超えているため，$[15, 30, 100]$ は H から取り除かれ，x_5がHに追加される．最後に，u_6においてMW_6はH内のインターバルの最小の重みより小さいため，Hは更新されず，操作を終える．

最悪の場合，WITの全てのノードを探索し，各ノードでHが更新される． Hの更新には$O(\log k)$時間がかかるため，全体の時間計算量は$O(N \log k)$である．

議論. インターバル木のように各ノードで複数のインターバルを含む木では，その全てのインターバルが枝刈りの条件を満たしている場合のみに枝刈り可能であるため，枝刈りしづらい．一方，WITは個々のインターバルと部分木の最大重みに注目するため，より高い階層で多くのインターバルを枝刈りできる可能性が高まる．

バケッティング. WITの性能をさらに向上させるために，バケッティング [14]を導入する．バケッティングを導入したWITをWITB（WIT with Bucketing）と呼び，インターバルを重みが大きい順でソートし，b個の互いに素なバケットに分割する．各バケットB_iは独立したWIT $\mathcal{W}_i$を構築する．クエリが与えられたとき，$\mathcal{W}_1$からAlgorithm 1を実行し，終了時に$|H| = k$であれば探索を終了する．そうでない場合，$\mathcal{W}_2$でAlgorithm 1を実行する．これを$|H| = k$となるまで繰り返す．

ここで，最初のいくつかのバケット内でTop-kインターバルが見つかれば，残りのバケットをスキップできる．この最適化により，個々の木のサイズが小さくなり，クエリ処理の高速化が得られる場合がある．

Algorithm 1: WITを用いたTop-k範囲クエリ

Input: WIT of X, query interval $q = [q_{\text{start}}, q_{\text{end}}]$, k
Output: Min-heap H

$H \leftarrow \emptyset$
TopK(u_{root}, q, H)
return H
Function TopK(u_i, q, H)**:**
  Let *left* and *right* be the index of u_i^l and u_i^r
  if x_i *overlaps* q **then**
    if $|H| < k$ **then**
      Insert x_i into H
    else if $x_i.weight > Top(H).weight$ **then**
      Remove the Top element from H
      Insert x_i into H
  if u_i^l *exists* ***and*** $[\min_{\text{left}}, \max_{\text{left}}]$ *overlaps* q **then**
    if $|H| < k$ ***or*** $MW_{\text{left}} > Top(H).weight$ **then**
      TopK(u_i^l, q, H)
  if u_i^r *exists* ***and*** $[\min_{\text{right}}, \max_{\text{right}}]$ *overlaps* q **then**
    if $|H| < k$ ***or*** $MW_{\text{right}} > Top(H).weight$ **then**
      TopK(u_i^r, q, H)

4 SAIT

WITおよびWITBは実践的に優れた性能を発揮するものの，最悪時間が$O(n)$である，つまり理論的には総当たりに相当し，理論的な効率性に欠ける．そのため，$o(n)$時間を達成するSAITを提案する．

4.1 予備知識

4.1.1 AIT

AIT [12]は，インターバル木を拡張したデータ構造であり，範囲検索の派生問題（Independent range sampling）を効率的に解決するために提案された． AITの各ノードu_iは，インターバル木の要素に加えて以下の情報を管理する．

- AL_i^l：u_iを根とする部分木に存在する全てのインターバルを含むリスト（始点順に整列）
- AL_i^r：AL_i^lと同じインターバルを含むリスト（終点順に整列）

つまり，各ノードは中心点とそれに重なるインターバル（L^lおよびL^r）だけでなく，その部分木に存在するインターバルのリスト（AL^lおよびAL^r）も格納している．このリストにより，部分木に存在する全てのノードを訪問することなく，クエリ範囲と重なるインターバルにアクセスできる． AITの空間計算量

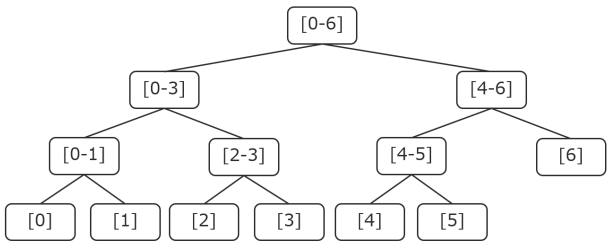

図4　セグメント木の例

は$O(N \log N)$である．

クエリ$q = [q_{\text{start}}, q_{\text{end}}]$を処理する際，AITは次のように動作する．根ノードから探索を開始し，各ノードu_iで以下の場合に応じて処理を行う．

- **ケース 1**（$q_{\text{end}} < c_i$）：L_i^lを用いて，qと重なるインターバルを見つける．右部分木に存在するの全てのインターバルはc_i以降に開始点があるため，qと重なる可能性がなく，左部分木のみを探索すればよい．
- **ケース 2**（$q_{\text{start}} > c_i$）：L_i^r にアクセスし，qと重なるインターバルを見つける．同様に左部分木に存在する全てのインターバルはqと重ならないため，右部分木のみを探索し，左部分木は枝刈りできる．
- **ケース 3**（$q_{\text{start}} \le c_i \le q_{\text{end}}$）：ノード内の全てのインターバルが$q$と重なる．この場合，アクセスが必要なのは左部分木のAL^rと右部分木のAL^lのみであり，それ以外のノードを探索する必要はない．

範囲クエリの時間計算量は$O(\log N + M)$となり，Mはクエリ結果に含まれるインターバルの数を表す．つまり，Top-kクエリに単純にAITを使うだけは$o(n)$時間を達成できない．

4.1.2　セグメント木

セグメント木 [3]は，配列に対する効率的なクエリ処理を可能にする二分木である．セグメント木は配列をセグメント（区間）に分割し，各ノードu_iは範囲$[l_i, r_i]$を表し，以下の要素を持つ．

- V_i：ノード u_i に格納されているセグメントの集約値（合計，最小値，最大値，またはカウント等）．
- u_i^l：u_i の左の子ノード
- u_i^r：u_i の右の子ノード

根ノードは配列全体の範囲を表し，各葉ノードは1つの要素を表す．各ノードには，2つの子ノードの値をマージして計算された値が格納される．この値は通常，その範囲内の最小値または最大値である．サイズNの配列に対するセグメント木の構築には，$O(N \log N)$時間必要であり，空間計算量は$O(N \log N)$である．

SAITは，セグメント木を利用して最小範囲カバー（MRC: Minimum Range Cover）を効率的に求める． MRCとは，クエリ範囲$[L, R]$を完全にカバーするセグメント木の最小のノード集合を指す．例えば，図4では，$[3, 5]$のMRCは$\{[3], [4-5]\}$となる．ここで，任意の整数$R > 0$が与えられた場合，範囲$[0, R]$のMRCを求めるには$O(\log N)$時間必要である．

4.2　概要

SAITはAITの派生型であり，AITの要素に加えて各ノードにセグメント木を組み込み，重みが大きくクエリに重なるインターバルを優先的にアクセスできるようにしている．これにより，Top-kインターバルを理論的に高速に取得することが可能となる．

構造. まず，SAITで使用されるセグメント木を定義する．整列済みのインターバルリストLが与えられたとき，$ST(L)$をLのセグメント木と定義する． $ST(L)$は，Lのインターバルを階層的に分割する構造である．各ノードはリスト内の特定のインデックス範囲を表し，その範囲に含まれる全てのインターバルを格納する連結リストを持つ．この連結リストは重み順に整列されており，重みが大きいインターバルに早くアクセスできるようになっている．リストLと対応する$ST(L)$ の例を図5に示す．

SAIT の各ノードu_iには以下の要素が含まれる．

- $c_i, L_i^l, L_i^r, AL_i^l, AL_i^r$：AITと同様
- $ST(L_i^l)$, $ST(L_i^r)$, $ST(AL_i^l)$, $ST(AL_i^r)$
- L_i^w：L_i^lと同じインターバルを含むリスト（重み順に整列）
- u_i^l：u_iの左の子ノード
- u_i^r：u_iの右の子ノード

構築. インターバル集合が与えられた場合，全てのインターバルはまず根ノードのAL_i^lおよびAL_i^rに挿入され，それぞれ始点順と終点順に整列される．次に，インターバルの中点の中央値が計算され，それがc_iとなる． $AL_i^l(AL_i^r)$の中からc_iと重なるインターバルが選ばれ，$L_i^l(L_i^r)$に追加される．その後，$ST(L_i^l)$, $ST(L_i^r)$, $ST(AL_i^l)$, および$ST(AL_i^r)$の4つのセグメント木が構築される．AL_i^lおよびAL_i^rにおいて，c_iよりも終点が小さいインターバルは左の子ノードに送られ，c_iよりも始点が大きいインターバルは右の子ノードに送られる．部分木は再帰的に構築される．

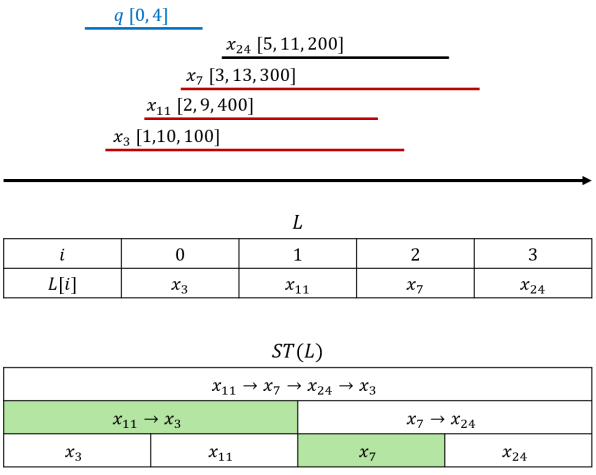

図5　Lと$ST(L)$の例及びST(L)に対する$[0, 2]$のMRC

定理 1（SAITの空間計算量） SAITの空間計算量は$O(N \log^2 N)$である．

証明. SAITの高さはAITに準ずるため，その高さは$O(\log N)$である．各ノードu_iにおいて，その部分木に含まれるインターバルの数をk_iとすると，全てのインターバルは$O(\log N)$個のノードに格納されるため，$\sum_i k_i = N \times O(\log N) = O(N \log N)$が成り立つ．各ノード$u_i$には，その部分木に含まれる全てのインターバルに基づいたセグメント木が格納されており，このセグメント木の空間計算量は$O(k_i \log k_i)$である．したがって，全体の空間計算量は$\sum_i O(k_i \log k_i) = O(N \log N \log N) = O(N \log^2 N)$となる． □

定理 2（SAIT構築の時間計算量） SAITの構築にかかる時間計算量は$O(N \log^2 N)$である．

証明. 全体の時間計算量に最も影響を与える要因は，各ノードでのセグメント木の構築である．各ノードu_iにおいて，その部分木に含まれる全てのインターバルを基にセグメント木を構築するには$O(k_i \log k_i)$の時間がかかる．そのため，定理1の証明と同様に本定理が導かれる． □

4.3 クエリ処理

クエリ範囲qおよびkが与えられた時，根ノードから探索を開始して次のようにノードを探索する．各ノードu_iにおいて，以下の場合に応じた処理が実行される． Algorithm 2は提案アルゴリズムの擬似コードを示している．

Algorithm 2: Top-k Range Query in SAIT

Input: a SAIT of X, query interval $q = [q_{\text{start}}, q_{\text{end}}]$, integer k
Output: Min-heap H

```
H ← ∅
TopK(u_root, q, H)
return H
Function TopK(u_i, q, H):
    Let left and right be the index of u_i^l and u_i^r
    if u_i = ∅ then
        return
    if q_end < c_i then
        j ← BinarySearch(L_i^l, q_end)
        SegTree(ST(L_i^l), j, H)
        TopK(u_left, q, H)
    else if c_i < q_start then
        j ← BinarySearch(L_i^r, q_start)
        SegTree(ST(L_i^r), j, H)
        TopK(u_right, q, H)
    else
        j ← 0
        while |H| < k and L_i^w[j].weight > Top(H).weight do
            Add L_i^w[j] into H
            j ← j + 1
        l ← BinarySearch(AL_left^r, q_start)
        SegTree(ST(AL_left^r), l, H)
        m ← BinarySearch(AL_right^l, q_end)
        SegTree(ST(AL_right^r), m, H)
```

• **場合1**：$q_{\text{end}} < c_i$．まず，L_i^lを使用して，クエリ範囲qと重なるインターバルのインデックスを見つける．このインデックスは，L_i^lが始点順に整列されているため，範囲$[0, j]$を形成するこの処理は，L_i^lのサイズをN_iとした場合，二分探索を使用して$O(\log N_i)$時間で完了する．

そして，$ST(L_i^l)$上で$[0, j]$のMRCを見つける．このMRCを$\mathcal{M}$とし，それが連結リストの集合であるとする．その後，$\mathcal{M}$に含まれる各リストの最初の要素を取り出し，最大ヒープH_Nに挿入する．H_Nのサイズは$O(\log N_i)$であり，各レベルにつき最大1つのノードしか選択されない．このステップの処理の時間計算量も$O(\log N_i)$となる．

次に，「ヒープ間の処理」を行う． H_Nのトップ要素x_{top}（H_N内で最も重みが大きいインターバル）がHのトップ要素（H内で最も重みが小さいインターバル）よりも重みが大きい場合，Hのトップ要素をポップしてx_{top}をHに追加する．そうでない場合，ヒープ間の処理を終了する．その後，H_Nのトップ要素をポップし，x_{top}を含む連結リストにまだインターバルが残っている場合，その次のインターバルをH_Nに追加する．この処理をH_Nが空になるまで繰り返す．この方法では，クエリに重なるインターバルのみを重みの大きい順にアクセスするため，最大でk個のインターバルにしかアクセスしない．そのため，ヒープ間の処理は$O(k \log(\log N_i) + k \log k)$時間で完了する．これは，両方のヒープで$O(k)$回の更新が行われるためである．

最後に，左部分木のみを探索する．右部分木にはqと重なるインターバルが存在しないため，探索の必要がない．

例2. 図5において，LをL^lの例とする．クエリ範囲qが場合1に対応する場合，重なるインターバル$L[i]$ $(i = 0, 1, 2)$が選択される．その後，$ST(L)$上で$[0, 2]$のMRCが抽出され，結果として$x_{11} \to x_3, x_7$となる．次に，x_{11}とx_7を用いて最大ヒープH_Nが形成される．ここで$k = 2$，Hが空であると仮定すると，まずx_{11}がH_Nからポップされ，x_3がH_Nに挿入される．その後，x_7がポップされる．このようにして，ポップされるインターバルは最大でk個のみとなることがわかる．

• **場合2**：$q_{\text{start}} > c_i$．この場合の処理は，場合1の処理を逆方向にしたものに相当する．左部分木は枝刈りされ，右部分木のみが探索される．

Algorithm 3: SegTree(ST, j, H)

Input: Segment tree ST, integer j, min-heap H
Output: Updated min-heap H

```
M ← a set of linked lists in the MRC of [0, j] on ST
H_N ← max-heap of first intervals from each list in M
while H_N ≠ ∅ do
    x ← H_N.pop()
    if |H| < k then
        Insert x into H
    else if x.weight > Top(H).weight then
        Remove the top element from H
        Insert x into H
    else
        break
    Add x's next interval to H_N if exist
```

• **場合3**：$q_{\text{start}} \le c_i \le q_{\text{end}}$．このノードで管理されている全てのインターバルが$q$と重なるため，$L_i^w$の中から最大で$k$個のインターバルを順番に確認する．その後，左部分木のAL^rと右部分木のAL^lを探索する必要がある．したがって，場合2の処理を，L_i^rの代わりに左の子ノードのAL^rで実行する．同様に，場合1の処理をL_i^lの代わりに右の子ノードのAL^lで実行する．上記の3つのリストにクエリと重なる残りの全てのインターバルが含まれているため，これ以上ノードを確認する必要はない [12].

定理 3 Algorithm 2の時間計算量は$O(\log^2 N + k \log N \log(\log N) + k \log N \log k)$である．

証明. 場合1および場合2では，1つのノードにおける処理全体の時間計算量は$O(\log N_i + k \log(\log N_i) + k \log k)$であり，$O(\log N)$ノードを探索する [12]．一方，場合3では，クエリ処理が終了し，インターバルが処理全体で再訪されることはない．したがって，Top-k範囲クエリの全体的な時間計算量は$O(\log^2 N + k \log N \log(\log N) + k \log N \log k)$となる． □

5 実験

本章では，本研究の実験結果を示す．本実験ではWIT，WITB，およびSAITを既存アルゴリズムと比較した．そのうちの1つは，全てのインターバルを重み順に整列し，線形スキャンによってTop-kインターバルを出力する方法であり，このアルゴリズムをBF（BruteForce）と呼ぶ．また，インターバル木およびHINT^{m}を比較アルゴリズムとして用いた．全てのアルゴリズムはC++で実装され，g++ 11.4.0 を使用し，-O3 フラグを付けてコンパイルした． HINT^{m}は著者の実装を用いた[*1]．実験は全て，Intel Xeon Platinum 8268 CPU（2.90 GHz）を搭載し，768 GBのRAMを備えたUbuntu 22.04 LTSの計算機で実施した．

表1 データセットの統計

Dataset	BOOKS	BTC	TAXIS
Cardinality	2,312,602	3,766,762	6,053,995
Domain size	31,507,200	6,876,400	6,208,602
Min length	0	0	1
Max length	31,406,400	547,077	580,217
Avg length(%)	6.98	0.022	0.017

データセット及びクエリ．本実験では，3つの実データBOOKS [1]，BTC[*2]，およびTAXIS[*3]を使用した．各データセットの特性を表1に示す．（インターバルxのlengthは，$x.end - x.start$である．） BOOKSは，2013年にAarhusの図書館で本が借りられていた期間のデータ集合である． BTCはビットコインの価格区間の集合であり，始点と終点としてそれぞれ最低価格と最高価格を使用した． TAXISは，2024年7月から8月の間にNYCで利用されたタクシーの乗車時間のデータである．

これらのデータセットのインターバルには重みが割り当てられていないため，ポアソン分布を使用して各インターバルにランダムな重みを割り当てた．各実験では，10,000個のランダムなクエリを生成した．各クエリ範囲の長さは，デフォルトでドメインサイズの0.1%とした．始点は一様ランダムに選択され，終点はその長さに基づいて決定し．さらに，kの値はデフォルトで5に設定した [8].

前処理時間およびメモリ使用量. 表2に前処理時間およびメモリ使用量を示す． WITとWITBは，比較アルゴリズムよりもやや長い時間前処理時間を要するものの，十分に高速である． SAITはやや長い前処理時間を必要とする． SAITでは各ノードでセグメント木を構築する必要があるため前処理時間が長くなる．

次に，メモリ使用量について注目する．BFは，ソートしたインターバル配列のみを必要とするため，割愛した． WITとWITBのメモリ効率が良いが，SAITは多くのメモリを必要とする．メモリリソースが制約となる場合，WITまたはWITBが妥当な選択肢となる．

クエリ時間. 次に，各アルゴリズムのクエリ時間を比較する．表3は，Top-k範囲クエリに応答する平均時間を示している．BFは長いインターバルを持つデータセット（BOOKS）で最も優れた性能を発揮する．一方，WITはBOOKSにおいて性能が低く，WITBとSAITは比較的高速にクエリに応答する．これは，WITBのバケッティング技術とSAITの枝刈り戦略によるものである．また，短いインターバルを持つデータセット（BTCおよびTAXIS）においては，WITBとSAITが良好な性能を示す．ここで，インターバル木やHINT^{m}は，BTCおよびTAXISにおいてWITよりも優れた性能を発揮する．これは，これらのアルゴリ

[*1] https://github.com/pbour/hint

[*2] https://www.kaggle.com/datasets/swaptr/bitcoin-historical-data

[*3] https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

表2　前処理時間およびメモリ使用量

	前処理時間 [s]			メモリ使用量 [GB]		
Dataset	BOOKS	BTC	TAXIS	BOOKS	BTC	TAXIS
BF	0.10	0.17	0.28	-	-	-
Int. Tree	0.46	1.27	2.19	0.15	0.33	0.45
HINT^m	0.63	0.49	1.01	0.12	0.08	0.11
WIT	0.39	0.59	1.03	0.28	0.45	0.72
WITB	0.58	0.95	1.61	0.47	0.75	1.37
SAIT	16.34	72.58	112.44	17.84	79.97	124.72

表3　平均クエリ時間 [μs]

Dataset	BOOKS	BTC	TAXIS
BF	0.2	53.5	1061.6
Interval Tree	250.0	24.1	21.5
HINT^m	543.4	28.4	38.9
WIT	108.0	29.5	40.0
WITB	1.7	1.8	8.4
SAIT	11.1	7.2	7.1

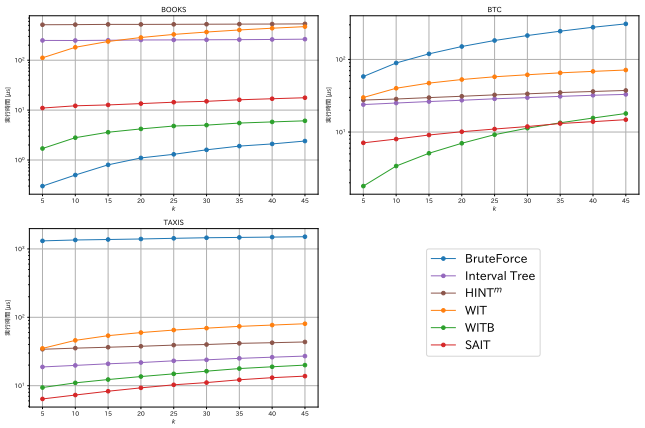

図6　kの影響

ズムが範囲クエリに最適化されており，重なるインターバルの数がkに匹敵するためである．この結果は，短いインターバルを持つデータセットにおいても，WITBのバケッティング技術が効果的であることを明確に示している．

kの影響. 次に，kの影響を評価した結果を図6に示す．インターバル木やHINTのように，クエリに重なる全てのインターバルを抽出してからそれらを整列し，Top-k結果を返すアルゴリズムは，クエリに応答する処理時間はkにほとんど影響しない．これに対し，WIT，WITB，およびSAITはこの問題に対応しており，クエリ時間がkに比例する．

クエリの長さの影響. 次に，クエリ範囲の長さの影響を評価した．図7にその結果を示す．クエリの平均の長さは，ドメインサイズの割合としてランダムに選択した．クエリ範囲の長さが増加するにつれて，クエリ範囲と重なるインターバルの数が増加する．しかし，提案アルゴリズムはクエリに重なる全てのインターバルにアクセスする必要を排除しており，クエリ範囲の長さに関係なく一貫した性能を実現している．

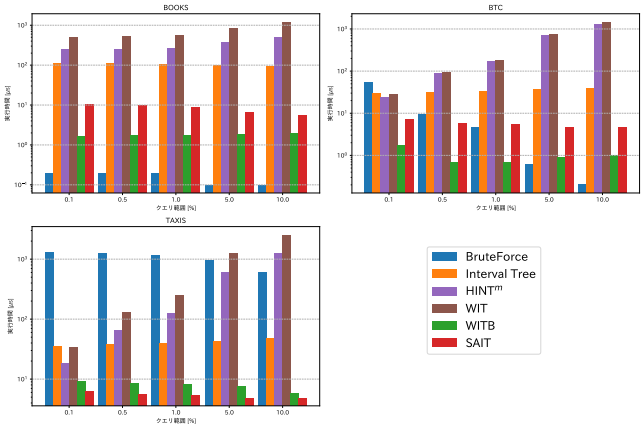

図7　クエリの長さの影響

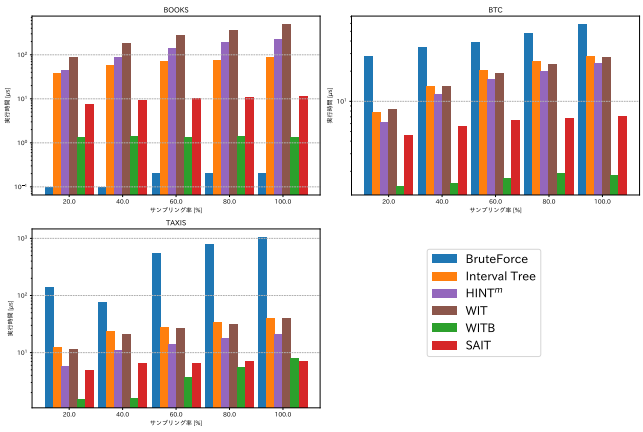

図8　データサイズの影響

データセットサイズの影響. データセットをランダムにサンプリングしてデータサイズに対する各アルゴリズムのクエリ性能を調査した．その結果を図8に示している．この結果から，SAITがデータセットのサイズに対して頑健であることが確認できる．WITとWITBの性能はデータセットに依存して異なるものの，クエリ時間はデータセットのサイズに対して基本的に準線形である．

6　関連研究

インターバルデータに関する操作は，Allenのインターバル代数 [24] に基づいて研究されてきた．インターバルデータで広く使用されている操作の1つにJoin [15] がある．インターバルJoinは，2つのデータセットから重なるインターバルのペアを見つける結合操作である．この目的のために，インデックスベースのアルゴリズム [16] [17]やパーティショニングベースのアルゴリズム [18]が提案されている．

さらに，スタビングクエリやインターバルデータの範囲検索を

効率的に処理するために，様々なデータ構造が開発されている．セグメント木 [3]は，スタビングクエリを$O(\log N + M)$時間で処理し，$O(N \log N)$の空間を必要とする．ただし，セグメント木は範囲検索をサポートしないため，本研究のベースラインには適していない．効率的な範囲検索のために，インターバル木 [9]，ピリオドインデックス [10]，タイムラインインデックス [21]，およびHINT [1]が開発されている．しかし，これらのアルゴリズムも5章で示したように本研究の問題を効率的に解決することはできない．

Top-kクエリは多くの分野で研究され，広く応用されている [7]．本研究と密接に関連するいくつかの先行研究が存在する．重み付きインターバルに対する3種類のTop-kクエリが文献 [8]で取り上げられており，本研究では，この研究の境界インターバルのアイデアを採用し，WIT内の各部分木のサブドメインを格納している．また，重み付きインターバルのTop-kスタビングクエリ処理問題は文献 [14]で研究されており，WITBで使用しているバケッティング技術を提案している．文献 [22]では，様々なTop-kクエリに適用可能なTop-k幾何交差クエリ問題に対する一般化された手法を提案している．

7 結論

本研究では，Top-k範囲クエリ問題に取り組んだ．インターバル木やセグメント木のような既存のデータ構造は，クエリに重なる全てのインターバルを取得する必要があるため，Top-k範囲クエリには非効率的である．本研究では，この課題を克服するために，WIT，WITB，およびSAITを提案した．実データセットを用いた実験により，データセット内のインターバルの長さが大きい場合を除き，WIT，WITB，およびSAITが既存アルゴリズムに比べて優れた性能を持つことを示した．

謝辞

本研究の一部は，AIP加速課題（JPMJCR23U2）の支援を受けたものである．

参考文献

[1] G. Christodoulou, P. Bouros, and N. Mamoulis, “HINT: A Hierarchical Index for Intervals in Main Memory,” In *SIGMOD*, pp. 1257–1270, 2022.
[2] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen, “Temporal Data Management - An Overview,” In *eBISS*, Vol. 324, pp. 51–83, 2017.
[3] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, “Computational Geometry: Algorithms and Applications, 3rd Edition,” Springer, 2008.
[4] X. Long and T. Suel, “Optimized Query Execution in Large Search Engines with Global Page Ordering,” In *VLDB*, pp. 129–140, 2003.
[5] P. Cao and Z. Wang, “Efficient Top-k Query Calculation in Distributed Networks,” In *PODC*, pp. 206–215, 2004.
[6] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis, “Automated Ranking of Database Query Results,” In *CIDR*, 2003.
[7] I. F. Ilyas, G. Beskales, and M. A. Soliman, “A Survey of Top-k Query Processing Techniques in Relational Database Systems,” *ACM Computing Surveys*, Vol. 40, No. 4, pp. 11:1–11:58, 2008.
[8] J. Xu and H. Lu, “Efficiently Answer Top-k Queries on Typed Intervals,” *Information Systems*, Vol. 71, pp. 164–181, 2017.
[9] H. Edelsbrunner, “Dynamic Rectangle Intersection Searching,” *Journal of the ACM*, Vol. 31, No. 2, pp. 246–260, 1984.
[10] A. Behrend, A. Dignös, J. Gamper, P. Schmiegelt, H. Voigt, M. Rottmann, and K. Kahl, “Period Index: A Learned 2D Hash Index for Range and Duration Queries,” In *SSTD*, pp. 100–109, 2019.
[11] B. Salzberg and V. J. Tsotras, “Comparison of Access Methods for Time-Evolving Data,” *ACM Computing Surveys*, Vol. 31, No. 2, pp. 158–221, 1999.
[12] D. Amagata, “Independent Range Sampling on Interval Data,” In *ICDE*, pp. 449–461, 2024.
[13] D. Amagata and L. Jimin “Top-k Range Search on Weighted Interval Data,” In *SSTD*, pp. 218–228, 2025.
[14] D. Amagata, J. Yamada, Y. Ji, and T. Hara, “Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data,” In *DEXA*, pp. 146–152, 2024.
[15] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo, “Join Operations in Temporal Databases,” *VLDB Journal*, Vol. 14, No. 1, pp. 2–29, 2005.
[16] J. Enderle, M. Hampel, and T. Seidl, “Joining Interval Data in Relational Databases,” In *SIGMOD*, pp. 683–694, 2004.
[17] D. Zhang, V. J. Tsotras, and B. Seeger, “Efficient Temporal Join Processing Using Indices,” In *ICDE*, pp. 103–113, 2002.
[18] A. Dignös, M. H. Böhlen, and J. Gamper, “Overlap Interval Partition Join,” In *SIGMOD*, pp. 1459–1470, 2014.
[19] F. Cafagna and M. H. Böhlen, “Disjoint Interval Partitioning,” *VLDB Journal*, Vol. 26, No. 3, pp. 447–466, 2017.
[20] I. F. Vega López, R. T. Snodgrass, and B. Moon, “Spatiotemporal Aggregate Computation: A Survey,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 2, pp. 271–286, 2005.
[21] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May, “Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data,” In *SIGMOD*, pp. 1173–1184, 2013.
[22] S. Rahul and R. Janardan, “A General Technique for Top-k Geometric Intersection Query Problems,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 26, No. 12, pp. 2859–2871, 2014.
[23] P. K. Agarwal, L. Arge, and K. Yi, “An Optimal Dynamic Interval Stabbing-Max Data Structure,” In *SODA*, pp. 803–812, 2005.
[24] J. F. Allen, “Maintaining Knowledge about Temporal Intervals,” *Communications of the ACM*, Vol. 26, No. 11, pp. 832–843, 1983.