DEIM Forum 2017 E5-5

# ARkR クエリ処理のクラスタリングによる効率化

山下 雅弘† 董 于洋† 陳 漢雄†† 古瀬 一隆††

† 筑波大学院システム情報工学研究科コンピュータサイエンス専攻 〒 305–8573 茨城県つくば市天王台 1–1–1

†† 筑波大学システム情報系 〒 305–8573 茨城県つくば市天王台 1–1–1

E-mail: †{yamasy,tou}@dblab.is.tsukuba.ac.jp, ††{chx,furuse}@cs.tsukuba.ac.jp

**あらまし** 本論文では、集約逆 k ランククエリ (以下,ARkR) と呼ばれる検索手法をクラスタリングを用いて効率化することを目的とする。ARkR は、複数の商品データをクエリとして受け取り、クエリ商品と全てのユーザ及び、他の商品と全てのユーザについてスコアを計算し、クエリ商品が他の商品よりも良いスコアとなっている上位 k 人のユーザを解として返す検索手法である。ARkR は、クエリとして受け取る商品の分布が広い場合に、既存の手法では検索効率が著しく低下する。これに対応するため、クエリとして与えられたデータを複数のクラスタに分割し、それぞれのクラスタに対して既存の手法を適用する。これにより、商品の分布に関わらず効率よく ARkR を処理できるようになる。

**キーワード** ARkR, クラスタリング

## 1. はじめに

Top-k クエリ [1] や逆 k ランククエリ [2] と呼ばれる検索手法は、クエリとして一つのデータを渡すような検索手法である。前者は、ユーザの好みをクエリとして、ユーザの好みと商品一つ一つとのスコアを計算してスコアの低い順にランク付けを行い、上位 k 個の商品を解として返す。これは、ユーザ視点の検索モデルであり、ユーザの好みに応じた商品を検索するという使い方ができる。逆に後者は、ある商品をクエリとして与えると、その商品とすべてのユーザ及び他の商品とすべてのユーザについてスコアを計算し、クエリとして与えられた商品が他の商品よりも良いスコアとなっている上位 k 人のユーザを解として返す。こちらの手法は、メーカー視点の検索モデルで、潜在的な消費者の発見や商品のマーケティングに応用することができる。

表 1,2 は、逆 k ランククエリで k=1 のときの例を示している。表 1 は二人のユーザの携帯電話に関する好みを示しており、また表 2 に示されている 5 つの異なる携帯電話 ($p1 - p5$) がそれぞれのユーザについて”Ranks”の欄で順位付けされている。表 2 ではそれぞれの携帯電話に関する情報とユーザごとのスコアとユーザがつけた順位、そして逆 k ランククエリの結果を表している。逆 k ランククエリでは、どのユーザが最も高い評価を下すかを求めるので、この例の場合、k=1 なのでユーザを一人探し出す。その結果は、表 2 の最も右の欄に示されている。

逆 k ランククエリは、一つの商品に対しての検索手法であるため、複数の商品を一つの商品として売り出すバンドリングという商法を取るようなメーカーに対しては用いることができなかった。ここで、集約逆 k ランククエリ (Aggregate Reverse k-Ranks Query: ARkR) と呼ばれる検索手法が提案された [3]。ARkR クエリは、逆 k ランククエリと同様の検索手法であるが、クエリとして複数の商品を与えるという点が異なる。

表 3 は ARkR において k=1 の時の簡単な例である。バンドルされた商品 $\{p1, p2\}$ がクエリとして与えられたとする。$\{p1, p2\}$ の集約ランクは、各ユーザについて各商品のランクの和をとったものになる。Jack の場合は 5, Dave の場合は 6 となるので、AR1R は Jack がユーザの中で最も好むとして Jack を解として求める。

| user | u[smart] | u[rating] | Ranks |
|---|---|---|---|
| Jack | 0.8 | 0.2 | p3,p2,p1,p4,p5 |
| Dave | 0.3 | 0.7 | p2,p5,p3,p4,p1 |

表 1: ユーザの好みと携帯のランク付けの例

| | p[smart] | p[rating] | Score on Jack | Score on Dave | Rank in Jack | Rank in Dave | R-1Rank |
|---|---|---|---|---|---|---|---|
| p1 | 6 | 7 | 6.2 | 6.7 | 3rd | 5th | Jack |
| p2 | 2 | 3 | 2.2 | 2.7 | 2nd | 1st | Dave |
| p3 | 1 | 6 | 2.0 | 4.5 | 1st | 3rd | Jack |
| p4 | 7 | 5 | 6.6 | 5.6 | 4th | 4th | Jack |
| p5 | 8 | 2 | 6.8 | 3.8 | 5th | 2nd | Dave |

表 2: ユーザによる個々の携帯電話のランク付けと逆 k ランククエリ (k=1) の例

| $\|Q\|$=2 | sum Rank in Jack | sum Rank in Dave | AR-1Rank |
|---|---|---|---|
| p1,p2 | 5 (3 + 2) | 6 (5 + 1) | Jack |
| p4,p5 | 9 (4 + 5) | 6 (4 + 2) | Dave |

表 3: ARkR の例 (k=1)

先行研究では、ARkR クエリを提案するとともに、TPM と呼ばれる R 木を用いた効率化手法も提案した。しかし、TPM はクエリが空間上に広く分散しているようなクエリ分布の場合では性能が悪くなることがわかっている。そこで、本研究では ARkR クエリに対するクエリをクラスタリングし、それぞれのクラスタに対して TPM を適用することによってクエリが広く

分散していても計算を効率化する方法を提案する。

本論文では、まず ARkR に関わる既存の研究や手法についてまとめ、既存の手法での問題点を指摘する。続いて、その問題点を解決するために提案した手法とその根拠を示す。最後に、まとめと今後の課題を示し、今後の研究の方向性を示す。

## 2. 関 連 研 究

ランク付けは、商品の評価指標として重要な役割を果たす。ランクを用いたクエリ処理はこれまでも広く研究されてきた。その中でも本研究と関わりの深い、逆ランククエリについて述べる。

**逆ランククエリ (RkR)**: 逆ランククエリは、本研究の対象である ARkR を含め、逆 top-k クエリ、逆 k ランククエリなどで見られる、商品についてユーザをランク付けし評価を行うクエリ処理の総称である。以下に、逆 top-k クエリ及び逆 k ランククエリについて簡潔に説明する。

**逆 top-k クエリ** [1,4]: クエリとして与えられた商品を他の商品と比べた際に、上位 k 位以内に評価するようなユーザを探す手法である。 逆 top-k クエリを効率化する手法として Vlachou ら [5] は、木構造及び、境界に基づいた位置決めによる分枝限定法を用いたアルゴリズムを提案した。また、Vlachou らは、[6,7] 逆 top-k クエリにおける応用例を多数挙げている。

**逆 k ランククエリ** [2]: 強い影響力のある商品のみがユーザを獲得するような逆 top-k クエリに対して、いかなる商品に対してもその商品を他のユーザよりも相対的に高く評価する上位 k 人のユーザを探すことで、商品の影響力に関わらずユーザを探せるようにした手法である。

## 3. 集約逆 k ランククエリ (ARkR)

ARkR は、前章でも触れた逆 k ランククエリを拡張したものである。逆 k ランククエリでは、一つのクエリに対してのみランクを計算し解を求めていたものを、ARkR では、複数のクエリに対してのランクを計算し、解を求めれるようにしたものである。これによって、例えば複数の商品などを一つのセットとして販売するような販売者に対して、どのような消費者がその商品を購入するのかターゲットを絞り込むことができ、そのような消費者に対するアプローチを考えれるようになるという応用例が考えられる。

### 3.1 ARkR の定義

**定義 1.** ($rank(u,q)$): データ集合 $P$, 重み付きベクトル $u$, クエリ $q$ が与えられた時、$u$ によるクエリ $q$ のランクは $rank(u,q) = |S|$ となり、 $S \subsetneqq P$ かつ $\forall p_i \in S$, $f(u,p_i) < f(u,q) \wedge \forall p_j \in (P-S)$, $f(u,p_j) \leqq f(u,q)$ を満たす。

**定義 2.** ($Aggregate\ Reverse\ k\ Rank\ query$, $ARkR$): データ集合 $P$, 重み付きベクトル集合 $U$, 正の整数 $k$, クエリ集合 $Q$ が与えられた時、ARkR クエリは $S \subsetneqq U, |S| = k$ かつ $\forall u_i \in S, \forall u_j \in (U-S)$, $ARank(u_i,Q) \leqq ARank(u_j,Q)$ を満たすような $S$ を返す。

この時、ARank は Sum, Max, Min などの集約計算が考えられる。この論文においては、Sum について考え、次のように定義する。

**定義 3.** $ARank(u,Q) = \sum_{q_i \in Q} rank(u,q_i)$

### 3.2 TPM (Tree Pruning Method)

TPM は、R 木 [8] によるデータ集合 $P$ のインデックス化によって作られる R 木のノード (エントリ) 及び、各 $u \in U$ によるクエリ集合 $Q$ の上界 ($Q.up$) と下界 ($Q.low$) を利用することにより ARkR の効率化を図るアルゴリズムである。図 1 は TPM の実行過程について二次元空間で示したものである。まず、$Q$ についての MBR を計算する。そして、ある重み付きベクトル $u_i$ に対して垂直な直線により $Q$ の上界と下界を計算し、それぞれ $Q.up$, $Q.low$ と記す。$Q$ を上界と下界に分ける垂線を図中では破線で表している。この破線により空間は 3 つの領域に分けられ、それぞれ $BelowQ$, $AboveQ$, それ以外とする。この例の場合、ノード $e2$ は $BelowQ$ に含まれており、ノード $e5$ は $AboveQ$ に含まれている。$BelowQ$ と $AboveQ$ に含まれるノードはそれぞれの上界と下界を調べることによってフィルタリングすることができる。そのルールを以下に示す。

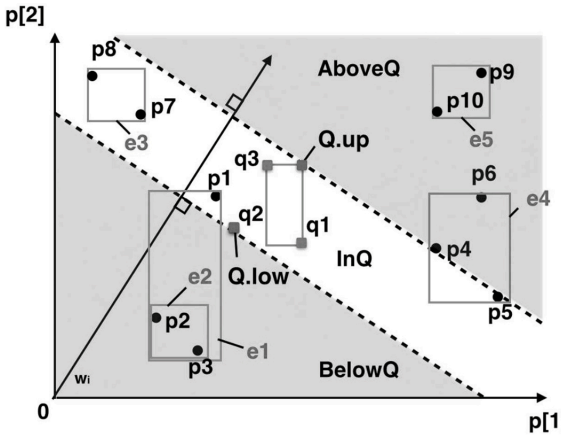



図 1: 二次元データ集合 $P$ の空間における $u_i$ と $Q$ の上界と下界による空間分割の例

（1） ($e_p$ が $BelowQ$ に含まれる)
$f(u,e_p.up) < f(u,Q.low)$ が成立するならば、$\forall p \in e_p, \forall q \in Q, f(u,q) > f(u,p)$ も成立するため、$e_p$ に含まれるデータを数え上げる。

（2） ($e_p$ が $AboveQ$ に含まれる)
$f(u,e_p.low) > f(u,Q.up)$ が成立するならば、$\forall p \in e_p, \forall q \in Q, f(u,q) < f(u,p)$ も成立するため、$e_p$ に含まれるデータを切り捨てる。

（3） (1,2 以外の場合)
$f(u,e_p.low) > f(u,Q.low)$ かつ $f(u,e_p.up) < f(u,Q.up)$ が成立するならば、$e_p$ は解の候補として保持し、精査する必要がある。

また、TPM は各 $u \in U$ についてのランクを計算する際に、$minRank$ と呼ばれる閾値をもちいて、R 木の探索を短縮する方法をとっている。このアルゴリズムは、ARank-P [3] と呼ばれる。

### 3.3 TPM における問題点

3.2 で述べたフィルタリングの第 3 のルールにおける精査とは、ノード $e_p$ に含まれるすべてのデータについて各 $q \in Q$ とのスコアを計算することである。そのため、検証するべきノードが増えれば増えるほど計算量が増えることとなる。検証するべきノードが増える原因として、クエリが図 2a のように、空間上に大きく分散している場合が考えられる。既存の手法では、このような場合に計算量が大幅に増えてしまうという問題点を抱えている。そこで、本研究ではクエリをクラスタリングすることによって既存の手法を用いても、計算量を削減することのできる手法を提案する。

## 4. 提 案 手 法

ここでは、クエリをクラスタリングすることによる TPM の処理能力の向上の原理を説明する。4.1 では、クエリのクラスタリングによるフィルタリングの改善の原理を例を交えて説明し、4.2 では、提案手法のアルゴリズムの紹介及びそれぞれの動作について詳しい説明を行う。

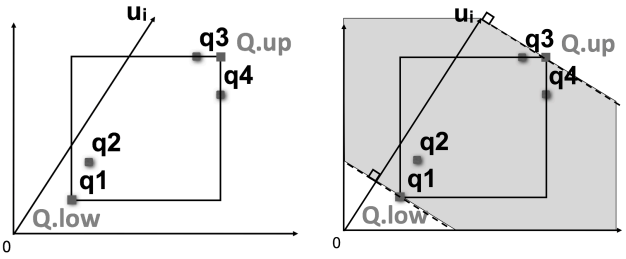
### 4.1 Q のクラスタリングによるフィルタリングの改善

本手法では、クエリ集合 $Q$ を複数のクラスタに分割し、それぞれのクラスタについて、既存の効率化を行うアルゴリズムである TPM を改良し、それを適用することでさらなる効率化を図る。この手法を CTPM(Clustered TPM) と呼ぶこととする。例として図 2a のようにクエリ {$q1$-$q4$} が与えられた場合、図 3a に示すように 2 つのクラスタ $\{Q_{C1}, Q_{C2}\}$ にクラスタリングができたとする。そして各クラスタについてそれぞれの上界と下界を計算し、各々のクラスタに対して TPM を用いて ARkR を計算する。これが、本提案手法のおおまかな流れである。

本来、図 2a のようにクエリの MBR が大きく広がっている状態では、この MBR の上界 (Q.up) と下界 (Q.low) の点からユーザ $ui$ に垂線を引いた場合の間の空間 (図 2b) に存在するデータ p については逐次計算を行わなければならなかった。しかし、図 3a のようにクエリをクラスタリングすることによって、図 3b に示すように、各クラスタにおいての上界及び下界の点で処理を分けることが可能になるため、計算するべき間の領域が TPM に比べて小さくなっていることがわかる (図中の灰色の領域)。以上の理由により、TPM を用いた際に計算が必須であった上界と下界の間の空間に存在するデータ p との計算を CTPM では大幅に削減できることが言える。詳しいアルゴリズムの説明は、4.2 節の CTPM アルゴリズムで行う。

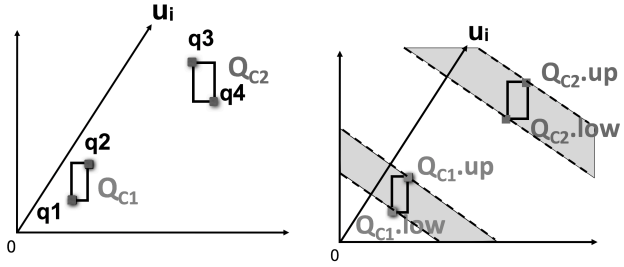
更に、クラスタごとに見ると、上界以上の点、あるいは下界以下の点も増えるため、フィルタリング可能な領域を増やすことができ、計算量を減らすことができる。また、TPM の節で触れた $minRank$ との判定を早めに行うことも可能となる。なぜならば、$minRank$ と比較される計算中のランクがクラスタの下界が広がることによって、より早く大きく見積もることができるからである。例を用いて説明する。

図 4 に示すように、クラスタが 2 つに分けられている場合、各クラスタの下界にあるデータ p の合計値がクエリ集合の最低ランクに関わる (図中の鈍色の領域)。そのため、元の手法ではクエリ集合全体の下界以下のデータのみを用いて最低ランクを計算していたが、クラスタリングを行うことによって、クラスタごとの下界以下のデータを合計したものを最低ランクとして扱うことができ、元の下界のデータよりも大きくなる (図中の薄灰色の領域)。これによって、TPM における $minRank$ との比較判定において元の手法よりも早期に木構造の探索を止め、計算量を削減することができる。詳しいアルゴリズムの説明は、4.2 節の CARank-P アルゴリズムで行う。




(a) 与えられたクエリとその MBR　(b) 計算するべき間の領域

図 2: TPM におけるクエリの MBR とフィルタ不可能な領域




(a) クエリのクラスタリングを行った例　(b) 計算するべき間の領域

図 3: CTPM におけるクエリの MBR とフィルタ不可能な領域

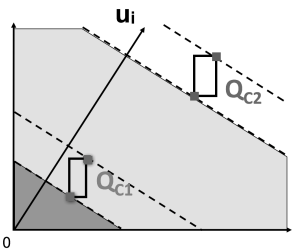



図 4: クラスタリングによる minRank の判定に用いられる最低ランクの幅の広がり

### 4.2 CTPM アルゴリズム

**CARank-P アルゴリズム**: アルゴリズム 1 に示すように、入力として $P, u, Q_C, minRank$ が与えられた時、CARank-P アルゴリズムは、全クラスタ $Q_C$ に含まれるクエリに対する集約ランク CARank が $minRank$ よりも小さいかを確認する。そして、CARank$(u, Q_C) < minRank$ が成立する場合、$Q_C$ の集約ランクを返す。また、入力として与えられる $P$ は事前に R 木によってインデクシングされているものとする。このアルゴリズムでは R 木のエントリー ($e_p$) を用いて、各 MBR についてフィルタリングを行う。

**Algorithm 1** CARank-P

**Input:** $P, u, Q_C, minRank$
**Output:** $rnk$; If $u$ is included ($-1$ otherwise);

```
1: rnk ⇐ 0, Cand ⇐ ∅
2: heapP.enqueue(RtreeP.Root())
3: while heapP.isNotEmpty() do
4:     e_p ⇐ heapP.dequeue()
5:     if e_p is non-leaf node
6:         foreach q_C ∈ Q_C do
7:             if e_p is under q_C
8:                 rnk ⇐ rnk + e_p.size() × |q_C|
9:                 if rnk ≧ minRank
10:                    return -1
11:                end if
12:            else if (e_p overlaps q_C) and (e_p.children() ∉ heapP)
13:                heapP.enqueue(e_p.children())
14:            end if
15:        end foreach
16:    end if
17:    if e_p is a leaf node
18:        foreach q_C ∈ Q_C do
19:            if e_p is under q_C
20:                rnk ⇐ rnk + |q_C|
21:                if rnk ≧ minRank
22:                    return -1
23:                end if
24:            else
25:                foreach q_i ∈ q_C do
26:                    if f(q_i, u) > f(e_p, u)
27:                        rnk ⇐ rnk + 1
28:                    end if
29:                end foreach
30:            end if
31:        end foreach
32:    end if
33: end while
34: if rnk ≦ minRank
35:    return rnk
36: else
37:    return -1
38: end if
```

1 行目に示す $rnk$ は、全クラスタ $Q_C$ の集約ランク CARank の一時カウンターとして用いる変数である。次に、アルゴリズムは R 木に格納されている $P$ の root から逐次的にエントリー $e_p$ を $heapP$ に挿入し、$e_p$ が葉ノードか否かで処理を分ける (2,3,4,5,18 行目)。 ここから各クエリクラスタ $q_C \in Q_C$ について、$e_p$ が葉ノードでない場合、$e_p$ の上界が $q_C$ の下界よりも小さくなるかを調べ、小さい場合は $rnk$ に $e_p$ に含まれるデータ数を加算し、そうでない場合は、$e_p$ の子ノードを $heapP$ に挿入する (7,8,13,14 行目)。$e_p$ が葉ノードの場合、$e_p$ の上界が $q_C$ の下界よりも小さくなる場合は $rnk$ にクラスタ $q_C$ に含まれるクラスタ数を加算し、そうでない場合は、各クエリ $q_i \in q_C$ について $q_i$と $u$、$e_p$と $u$ のスコアを逐次比較し、$rnk$ を更新する (18-33 行目)。いずれの場合においても $rnk$ を更新する際は、常に $minRank$ との比較を行い、$rnk \geqq minRank$ となった場合、$-1$ を返しその時点で計算を終了する。これは、$u$ が ARkR の解でないことを示す。逆に、$rnk \leqq minRank$ の場合は、$rnk$ を集約ランク値として返す。

**CTPM アルゴリズム**: CTPM アルゴリズムは、アルゴリズム 2 に示す通り、$P, U, Q$ を入力として受け取り、クエリ Q に対する ARkR の解を出力する。まず、解候補の $heap$ を $k$ 個の $u$ とそれらの $Q$ に対する集約ランクで初期化し、$k$ 番目のランク値を $minRank$ として保持する。(1,2 行目) 次に、k-means 法を用いて $Q$ をクラスタリングし、各 $u \in (U -$ 最初に用いた $k$ 個の要素) に対して集約ランク $rnk$ を先ほどの CARank-P アルゴリズムを用いて計算する。(3,4,5 行目) もし、$rnk$ が $-1$ でなければ計算対象の $u$ は ARkR クエリの解候補であるので $heap$ を更新する。(7 行目) 更新された $heap$ の $k$ 番目のランク値を $minRank$ に代入し、以降各 $u$ に対し繰り返し計算を行う。(4,8 行目) すべての計算が終了した時点で、ARkR クエリの解集合となっている $heap$ を返す。(9 行目)

**Algorithm 2** Clustered Tree-Pruning Method (**CTPM**)

**Input:** $P, U, Q$
**Output:** result set $heap$

```
1: initialize heap with first k weighting vectors and aggregate ranks of Q
2: minRank ⇐ heap.lastRank
3: Q_C ⇐ KMeans(Q)
4: foreach u ∈ U− {first k element in U} do
5:     rnk ⇐ ARank-P(P, u, Q_C, minRank)
6:     if rnk ≠ −1
7:         heap.insert(u, rnk)
8:         minRank ⇐ heap.lastRank
9:     end if
10: end foreach
11: return heap
```

## 5. 実験・考察

実際に、与えられたクエリをクラスタリングすることによって ARkR の検索効率が向上しているかを検証するため、上記のアルゴリズムをもとに実験を行った。5.1 節では、実験に用いたマシンの性能とデータについて説明し、5.2 節では、実験結果を提示し、それぞれパラメータを変更した場合について考察を述べる。

### 5.1 実験環境・設定

本実験は、多様なデータセットに対して、TPM と CTPM アルゴリズムを用いた際の処理時間を比較する。用いたマシンは、CPU: 2.6GHz Intel Core i7、 RAM:16GB の Mac で、アルゴリズムの実装言語は双方とも C++である。

合成データのクエリデータ $Q$ は、クラスタ (CL) に従って生成され、クエリデータのクラスタリングには、k-means 法を用い、クラスタ数は $\sqrt[3]{|Q|}$ に従うものとする。合成データの商品データ $P$ およびユーザの好みのデータ $U$ は、人口データを

用い一様分布 (UN) 及びクラスタ (CL) に従って生成されるものとする。既定値は、 $|Q| = 20, |P| = |U| = 100\text{k}$, パラメータ $k = 10$, 次元数 $d = 3$ とする。

実データでの実験は、1949 年から 2009 年までの 20960 の NBA 選手のデータを用いて、それらの選手の中でクエリで与えられた選手の組み合わせについて、他のユーザよりもこの組み合わせを好む上位 10 人のユーザは誰かという問い合わせを行う。選手のデータは、それぞれの選手の得点数、リバウンド数、アシスト数、ブロック数、そしてスティール数の 5 次元からなる。クエリで与えられる選手の組み合わせは選手データの中から無作為で選ばれるものとする。ユーザ $U$ は合成データと同様の方法で生成する。既定値は、$|P| = 20960, |U| = 100\text{k}$, パラメータ $k = 10$, 次元数 $d = 5$ とする。

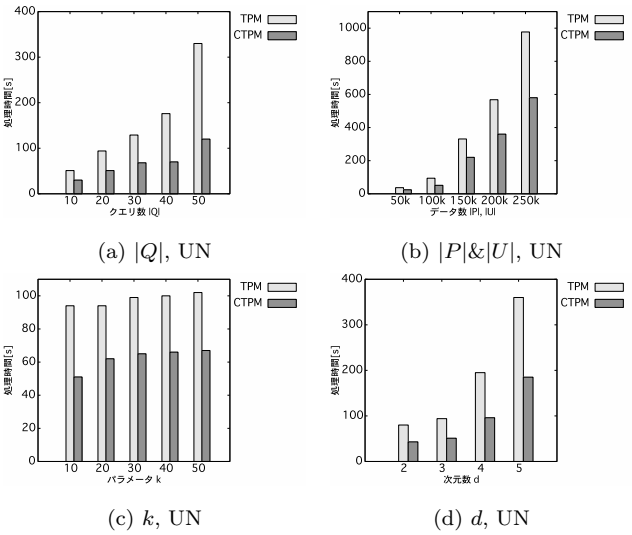
実験では、同じデータセットに対して 100 回アルゴリズムを実行した際の処理時間の平均値を、最終的な処理時間として扱う。一様分布の合成データについてパラメータ $Q, P$ と $U, k, d$ の値を変えた場合における、TPM と CTPM アルゴリズムの処理時間、クラスタの合成データについては $d$ を変えた場合を一様分布のデータについて比較を行う。実データについては、クエリ数 $|Q|$ を変えた場合における TPM と CTPM アルゴリズムの処理時間について比較を行う。

### 5.2 実験結果・考察

**合成データを用いた実験**: 図 5 は、一様分布 (UN) で生成された人口データに対して特定の値を変えて処理時間を計測した結果を示したものである。TPM の処理時間は薄い灰色で、CTPM の処理時間は濃い灰色で示されている。どのような状況に対しても、CTPM が TPM よりも処理時間が早くなっていることがわかる。この中で注目するべきは図 5a において、クエリ数 $|Q|$ が増えた場合においても、CTPM は TPM に比べて大きく処理時間を抑えることができていることである。これは、クラスタリングによってフィルタ領域を増やすことができた結果であると考えることができる。5b は、データ数の変化による処理時間の変化を表している。データ数の増加に対しては、TPM も CTPM も処理時間も増加していることがわかる。図 5c のパラメータ $k$ の変化については、*heap* のサイズが大きくなるだけであるので、TPM も CTPM にも大きな変化は現れていない。図 5d の次元数の増加に対しては、CTPM は TPM と次元数に対する処理の仕方に差がないため次元数の増加に対しては TPM よりも処理時間は抑えることができているが、増え方としては変化がないことがわかる。

図 6 は、一様分布 (図 6a) とクラスタ (図 6b) で生成された合成データについて次元数を変えた場合の結果について示した図である。全体的にはクラスタで生成されたデータの方が一様分布よりも早くなっていることがわかる。これは、クラスタで生成されたデータの方が、R 木によってインデクシングされやすくなっていることが原因であると考えることができる。

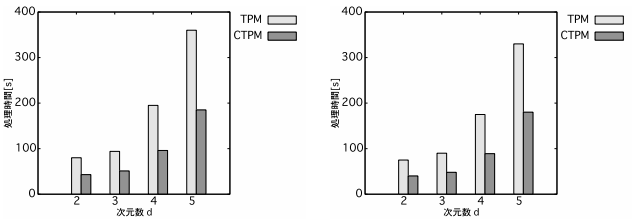
**実データを用いた実験**: 図 7 は、実データについてクエリ数を変えて処理時間を計測した結果を TPM と CTPM について示した図である。実データについても、CTPM は TPM よりも早くなっていることがわかる。クエリ数が増えても CTPM では大きく処理時間が増えていないという結果は、合成データの場合でも見られた傾向であり、きちんと実データにおいても同様の結果が得られていることがわかる。




(a) $|Q|$, UN (b) $|P|\&|U|$, UN

(c) $k$, UN (d) $d$, UN

図 5: 一様分布で生成されたデータにおいて、各パラメータを変更した場合の処理時間の変化。既定値は、$|Q| = 20, |P| = |U| = 100\text{k}, k = 10, d = 3$。




(a) $d$, UN (b) $d$, CL

図 6: 一様分布とクラスタで生成されたデータにおける、処理時間の変化

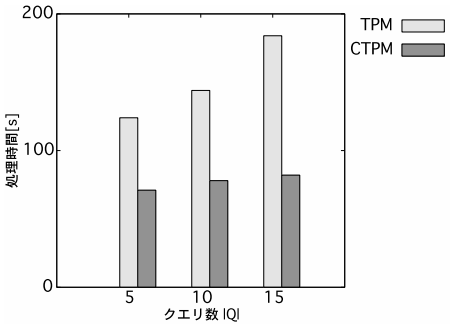



図 7: 実データにおいてクエリ数 $|Q|$ を変えた場合における処理時間の変化

## 6. 結　　論

本研究では、既存の ARkR の効率化を行う手法において特定のクエリの分布において効果を発揮できないことを示し、その問題点を解消するためにクエリをクラスタリングして既存の手法を適用するという提案を行った。この提案をもとに実験を

行い、既存の手法を用いた ARkR をより効率的に計算できるように改良することができた。

今後の課題として、既存のアルゴリズムを適用する際に有効なクラスタリングをどのようにして行うかを検討する必要がある。また、次元数がさらに大きくなった場合に、効率的に処理できるような手法を考える必要がある。これらの課題を解決し、ARkR をさらに効率化することを目指してゆく。

## 文　　献


[1] Vlachou, A., Doulkeridis, C., Kotidis, Y., et al.: Reverse top-k queries. In: ICDE, pp. 365-376 (2010)

[2] Zhang, Z., Jin, C., Kang, Q.: Reverse k-ranks query. PVLDB 7(10), 785-796 (2014)

[3] Y. Dong, H. Chen, K. Furuse, and H. Kitagawa. Aggregate Reverse Rank Queries LNCS, 9828:87-101 (DEXA 2016)

[4] Vlachou, A., Doulkeridis, C. , Kotidis, Y., et al.: Monochromatic and bichromatic reverse top-k queries, pp. 1215-1229 (2011)

[5] Vlachou, A., Doulkeridis, C., Kotidis, Y., et al.: Branch-and-bound algorithm for reverse top-k queries. In: SIGMOD Conference, pp. 481-492 (2013)

[6] Vlachou, A., Doulkeridis, C., Kotidis, Y.: Identifying the most influential data objects with reverse top-k queries, pp. 364-372 (2010)

[7] Vlachou, A., Doulkeridis, C., et al.: Monitoring reverse top-k queries over mobile devices. In: MobiDE, pp. 17-24 (2011)

[8] Antonin Guttman, R-trees: a dynamic index structure for spatial searching, Proceedings of the 1984 ACM SIGMOD international conference on Management of data, June 18-21, 1984, Boston, Massachusetts