RC-Index: Diversifying Answers to Range Queries

Yue Wang Alexandra Meliou Gerome Miklau
College of Information and Computer Sciences
University of Massachusetts Amherst
{yuewang,ameli,miklau}@cs.umass.edu

ABSTRACT

Query result diversification is widely used in data exploration, Web search, and recommendation systems. The problem of returning diversified query results consists of finding a small subset of valid query answers that are representative and different from one another, usually quantified by a diversity score. Most existing techniques for query diversification first compute all valid query results and then find a diverse subset. These techniques are inefficient when the set of valid query results is large. Other work has proposed efficient solutions for restricted application settings, where results are shared across multiple queries. In this paper, our goal is to support result diversification for *general range queries over a single relation*. We propose the RC-Index, a novel index structure that achieves efficiency by reducing the number of items that must be retrieved by the database to form a diverse set of the desired size (about 1 second for a dataset of 1 million items). Further, we prove that an RC-Index offers strong approximation guarantees. To the best of our knowledge, this is the first index-based diversification method with a guaranteed approximation ratio for range queries.

1. INTRODUCTION

Query result diversification is an important aspect of user-facing applications, such as data exploration, Web search, and recommendation systems [42, 13, 11, 46]. The need for diversification arises when the system has to limit the number of query results: since human users can visually process limited information, interfaces typically need to limit the data they display on the screen to a few points on a map or a small number of items in a list. Diversification is one common way to present representative results to users, and it is employed by many real-world systems. For example, even though there are thousands of ATMs in Manhattan, a search in Google Maps typically reveals no more than 20 at any zoom level, and the chosen locations are typically dispersed in the viewing area. Product searches in online marketplaces, such as Amazon, also employ diversity: a search for laptops in a particular price range typically yields diverse brands and models on the first page of results, rather than similar laptops from different retailers.

Our goal is to provide an efficient and scalable solution to the problem of selecting a diverse subset of the result of *general* range queries[1] over a single relation. Systems that employ query result diversification typically try to optimize a specified diversity score function. The diversity score is defined over a bivariate distance function, which is domain- and application-specific, and measures the distance between any two items of a dataset. When a user issues a query, the system needs to retrieve a set S of k items such that: (1) every item in S is in the result of the query; (2) the diversity score of S is maximized. This is a challenging problem to solve efficiently and in a scalable way, as it is practically infeasible to compute the diversity score of every k-sized subset of the query result. In fact, this problem is known to be NP-hard [16, 40, 38]. One can only develop approximation algorithms to solve it.

Existing approaches fail to solve this problem effectively. Many existing diversification techniques follow a "process-first-diversify-next" approach [25]: first, they execute the query normally to retrieve all results, and they subsequently employ appropriate algorithms to identify a subset with high diversity score. The "process-first-diversify-next" techniques have a big problem: efficiency. In practical diversification scenarios, systems typically only need to retrieve a relatively small subset of items, thus, computing the entire query result is often computationally wasteful.

Some prior work on result diversification has achieved efficient solutions at the expense of generality. For example, some work focuses on extracting diverse items from data streams through continuous querying [33, 14, 15]. These techniques achieve efficiency by reusing prior diversification results as new items arrive. Similarly, in some data exploration scenarios, such as geolocation visualization, subsequent queries are correlated, which again allows reuse of overlapping items in query results [23, 24, 26]. These techniques do not perform well in more general cases where query results do not overlap significantly, even if the query predicates are over a fixed set of attributes. For example, if a user issues subsequent queries for laptops "under \$1000", "between \$1000 and \$1500", and "above \$1500", even though the queries are very similar, their results do not overlap, and thus these techniques do not apply.

In this paper, we propose a general, index-based algorithm for diversifying the results of multi-dimensional range queries over a single relation. At a high level, our algorithm transforms each range query into a set of subordinate searches, performs these searches using a novel index structure, the RC-Index, and finally extracts a diverse subset from the merged results of subordinate searches. RC-Indexes achieve efficiency by retrieving a very small set of candidate items compared to "process-first-diversify-next" techniques. Our algorithm extracts and merges candidates in a novel way that

[1] A range query specifies upper and/or lower boundaries on attribute values that all retrieved records should satisfy.

guarantees provably high diversification quality. We further show empirically that it finds better solutions than competing approaches, and we develop efficient algorithms to maintain RC-Indexes.

Our work addresses four important challenges.

Efficiency. Existing techniques for addressing diversification in ad-hoc queries require the retrieval of the entire query result before applying diversification algorithms to retrieve a diverse k-sized subset. In contrast, RC-Indexes retrieve a much smaller set of candidate items (up to 99.4% reduction compared to state of the art), which leads to order-of-magnitude faster response times.

Generality. One of our main goals in this paper is to support general range queries over a relation. Prior work achieved efficiency only by restricting generality, through the assumption that subsequent queries share a large portion of results. In contrast, RC-Indexes do not make such assumptions. The RC-Index is a new index-based access method to relations that allows for fast retrieval of diverse subsets of results for range queries.

Effectiveness. RC-Indexes provide theoretical guarantees with respect to diversification quality. The approximation ratio is based on tunable parameters, and at the limit approaches $\frac{1}{2}$, which is the optimal polynomial time ratio for our diversification problem [38].

Flexibility. RC-Indexes use existing indexing structures as submodules to support range search. Our implementation is based on range trees, but a system designer may opt for different index structures (e.g., k-d trees) if they are better-suited for a given application.

We organize our contributions as follows.

- Section 2 provides background and an overview of our solution.
- Section 3 introduces the core of our approach, a novel index structure called RC-Index, which combines range selection indexes (range trees) with diversity indexes (cover trees). We describe our system, which uses the RC-Index to answer range queries with diversification requirements.
- Section 4 presents our theoretical analysis, showing that RC-Indexes are effective and efficient at providing diverse results. Specifically, the diversity score of sets returned by our system approximates the optimal diversity score by a factor of $\frac{b-1-2b^{1-\delta}}{2(b-1)}$, where $b > 1$ and $\delta \geq 1$ are parameters that control the ratio and time/space complexity. When b or δ approaches infinity, the limit of the ratio is $\frac{1}{2}$. This is the optimal polynomial approximation for our diversification problem under pseudometric distance functions [38]. We also show that the RC-Index has low time and space complexity.
- Section 5 explains how to choose indexes to support a workload of queries with respect to RC-Indexes.
- Section 6 presents the evaluation of our prototype system over real-world and synthetic datasets, demonstrating that RC-Indexes are both efficient and effective at range result diversification. Specifically, we show that our approach achieves better diversity scores than the state of the art and is substantially faster as well: we can index 10^6 items in 330 seconds and answer a query in under a second. Our approach is also more general, as it subsumes prior work, handling both streaming and relational queries.
- Section 7 discusses related work.

2. OVERVIEW AND BACKGROUND

We begin this section with the definition of range query result diversification, an optimization problem over a diversity score function $f(S, dist)$. We then discuss two popular diversity score functions from prior work, and desirable properties for the domain-specific distance measures used by the diversity score functions. We continue to present an overview of our system solution in Section 2.3. The core of our approach is a special index structure, the RC-Index. The RC-Index is a novel combination of two types of index structures: cover trees and range trees. We describe these structures in Sections 2.4 and 2.5 before proceeding with the details of our framework in Section 3.

2.1 Result diversification

In this section, we define the problem of selecting a diverse subset of the results of a range query. We use X to denote a database with a single relation $R(\mathbb{A})$ over the set of attributes $\mathbb{A} = \{A_1, A_2, \ldots\}$. Let $n = |X|$ be the size of the relation. A range query q can apply interval filters to a subset of attributes $\mathbb{A}_q \subseteq \mathbb{A}$, where each attribute in $\mathbb{A}_q$ is assumed to have an ordered domain. A bivariate function $dist(x_i, x_j)$ measures the distance between items $x_i, x_j \in X$. The distance function is defined over a subset of attributes $\mathbb{A}_{dist} \subseteq \mathbb{A}$, which may or may not overlap with $\mathbb{A}_q$. Given a database X and a query q, we wish to find the subset of k items in the result $q(X)$ that maximizes a diversity score function defined over $dist$.

PROBLEM 1 (RANGE QUERY RESULT DIVERSIFICATION). *Given a set of items $X = \{x_1, x_2, \ldots\}$, a bivariate distance function $dist(\cdot,\cdot)$ on X, a range query q with result set $q(X)$, and a positive integer $k \leq |q(X)|$, range query result diversification selects $S \subseteq q(X)$ that maximizes a diversity score function f over dist:*

$$\begin{aligned} \max_{S} \quad & f(S, dist) \\ s.t. \quad & S \subseteq q(X) \\ & |S| = k \end{aligned}$$

The distance measure $dist$ is application-specific. The diversity score function $f(S, dist)$ has two popular forms [12, 42, 11]:

$$f_{min}(S, dist) = \min_{x_i, x_j \in S \,\wedge\, x_i \neq x_j} \{dist(x_i, x_j)\}$$

$$f_{sum}(S, dist) = \sum_{x_i, x_j \in S \,\wedge\, x_i \neq x_j} dist(x_i, x_j)$$

The first form, f_{min}, computes the minimum distance of the items in S; the corresponding diversification problem is called MAXMIN. The second form, f_{sum}, computes the sum of pairwise distances of items in S; the corresponding diversification problem is called MAXSUM. Even when the range query retrieves all items ($q(X) = X$), Problem 1 under MAXMIN or MAXSUM is NP-hard [16, 40, 11]. Therefore, Problem 1 for general $q(X)$ is also NP-hard.

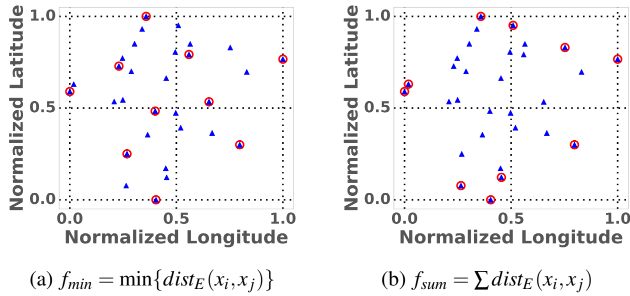
In practice, the diversification score is used to select appropriate top-k items. We demonstrate its application with an example:

EXAMPLE 1. *A user queries a dataset of ATM information to find those with a closing time after 8pm. The application displays 10 of the ATMs in the result with diverse locations based on their Euclidean distance $dist_E(\cdot,\cdot)$, using the following query q:*

```
SELECT *
FROM ATM_data
WHERE Close_time >= 20
LIMIT 10 DIVERSE(distE(Latitude, Longitude)) MAXMIN;
```

Query q applies a range filter to one attribute ($\mathbb{A}_q = \{$`Close_time`$\}$). The `LIMIT` *clause specifies that the query will return 10 items. Normally, the* `LIMIT` *clause would return any 10 items, but in this case, the clause is augmented with a diversification objective: We want the set of 10 results that maximizes the* MAXMIN *diversity score defined over the Euclidean distance $dist_E(\cdot,\cdot)$ on attributes* `Latitude` *and* `Longitude` *($\mathbb{A}_{dist} = \{$`Latitude`, `Longitude`$\}$).*

Figure 1a illustrates the result of the query of Example 1 over a small sample of the ATM dataset. All ATMs locations that satisfy



(a) $f_{min} = \min\{dist_E(x_i, x_j)\}$ (b) $f_{sum} = \sum dist_E(x_i, x_j)$

Figure 1: Diversification on a sample of ATMs in New York City under two diversity score functions. We select $k = 10$ ATMs (red circles), out of a total of 30 (blue triangles) in each figure.

the range predicate are denoted with blue triangles; the red circles indicate 10 diverse locations that may be selected. Figure 1b shows the result of the same query with the MAXSUM diversity score.

We note that f_{sum} selects points on the outskirts of the dataset, while f_{min} selects points closer to unselected points [13, 14]. Our proposed methodology supports both variants of the problem. For simplicity, we focus on f_{min}, i.e., MAXMIN, in the rest of this paper. We extend to MAXSUM in Appendix D.

2.2 Distance Function

The distance function is an important component of diversification, as the problem objective involves the maximization of pairwise distances of items in S. The distance function is domain- and application-specific, and can involve any of the attributes.

Intuitively, one can cluster X according to the distance metric $dist(\cdot,\cdot)$, and use this clustering to solve Problem 1. However, the challenge is that the attributes over which the distance is defined may or may not overlap with the attributes filtered by q: In Example 1, the distance is defined over the *latitude* and *longitude* attributes ($\mathbb{A}_{dist} = \{\texttt{Latitude}, \texttt{Longitude}\}$), while the range condition is over the `Close_time` attribute ($\mathbb{A}_q = \{\texttt{Close_time}\}$).

While there are no general restrictions on the distance metric with respect to the definition of Problem 1, some properties of the distance metric affect the problem complexity. Problem 1 under MAXMIN for general distance functions has no polynomial time algorithm that can provide a constant approximation ratio unless P=NP [38]. However, when the distance function is symmetric and satisfies the triangle inequality, a simple greedy heuristic provides a 1/2-approximation; no polynomial algorithm can provide an approximation ratio better than 1/2 unless P=NP [40, 38]. Following prior work in this area [40, 38, 14, 15], we also assume the distance function in this paper satisfies the three properties below:

DEFINITION 1 (DISTANCE FUNCTION). *The bivariate distance function $dist : X \times X \to \mathbb{R}_{\geq 0}$ in Problem 1 must be a pseudo-metric satisfying the following properties:*
$dist(x,x) = 0$
$dist(x,y) = dist(y,x)$ *(Symmetry)*
$dist(x,y) + dist(y,z) \geq dist(x,z)$ *(Triangle Inequality)*

Many common distance functions used in practice satisfy Definition 1. Here we list a few:
- The metrics induced by any L^p-norm with $p \geq 1$. These include Manhattan distance (L^1-norm) and Euclidean distance (L^2-norm).
- A graph-based metric in which items are vertices in a graph with non-negative edges and the distance between two vertices is the length of the shortest path connecting them.
- The "diversity ordering"-based distance [41]. This distance function defines a total ordering of the attributes such as *car make* $\prec$ *car model* $\prec$ *color* $\prec$ *year*, which means *car make* has higher priority than *car model* does and so on. The distance between two items is greater if the two items differ on a higher priority attribute.

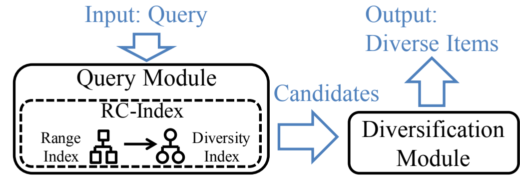


Figure 2: Solution overview. Our prototype system involves two modules: the Query Module and the Diversification Module.

Algorithm 1: Query Evaluation

Input: Range query q; Parameter k; RC-Index RC; Extra level δ.
Output: Set of diverse items S.
1 $T \Leftarrow$ `RangeQueryTreeExtraction`(q, RC) `// QM: Algo. 3`
2 $X^C \Leftarrow$ `CandidateExtraction`(k, T, δ) `// QM: Algo. 2`
3 $S \Leftarrow$ `GreedyDiversification`(X^C, k) `// DM: Algo. 4`

2.3 Solution and System Overview

The core of our approach is a novel index: the RC-Index.[2] Prior work [14] used indexes to solve the diversification problem. However, in that setting, one needs to first index the entire query result and then extract diverse items from the index. Rebuilding the index for each query result is too expensive. Instead, similar to a conventional B+ tree, we build an RC-Index on a dataset once but use it to answer various range queries. RC-Index combines two types of indexes: a Range Index and a Diversity Index. The Range Index is used to support range queries. It can be a B+ tree (for 1-dimensional queries), interval tree, R-tree, VA-file, k-d tree, range tree, etc. Each Diversity Index is a *cover tree* (described fully below) which is built on a subset of items. It organizes the items according to their pairwise distances. Items near the root are far from each other and items near the bottom are close to each other. It can help us limit the number of candidate items we extract, while ensuring the diversity of candidates.

We construct a framework on top of our core RC-Index to solve the diversification problem. Our framework has two modules as illustrated in Figure 2: the Query Module (QM) and the Diversification Module (DM). The QM builds an RC-Index offline to support range queries. Then, when a query arrives, the QM uses the index to extract a set of candidates and passes this set to the DM. Finally, the DM finds k diverse items to present to the user. Algorithm 1 shows the pipeline of query evaluation at a high level. Line 1 and Line 2 belong to QM. Line 3 is DM. We will elaborate in Section 3. The key to the success of this framework is that we limit the number of candidates extracted by the QM but also ensure their diversity. We will prove this desired property in Section 4.1.

Note that an RC-Index is built on a given set of attributes. This is a reasonable assumption because users' queries usually apply filters on a limited set of attributes in practice [1], and it is a standard assumption in conventional indexes like R-trees, k-d trees, and B+ trees. In Sections 3 and 4 we assume that the RC-Index is built on the query attributes $\mathbb{A}_q$; we relax this assumption in Section 5.

2.4 Cover Tree

[2] The RC-Index takes its name from the two data structures we use in our implementation: Range tree and Cover tree.

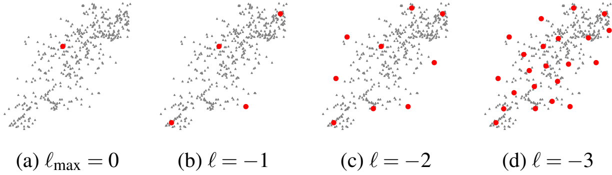


Figure 3: The 4 highest levels of a cover tree of ATMs in New York City with normalized coordinates. Items at each level ℓ are highlighted in red. The tree satisfies Nesting, Covering, and Separation.

A cover tree [5] is a data structure that naturally diversifies items. It is similar to navigating nets [27], which were originally designed for nearest neighbor search. Every level of a cover tree has a distance threshold. The distance between every pair of items at this level must be greater than this threshold. The threshold decreases from root to leaf. So, items at a higher level (i.e., closer to the root) are farther away from each other. Intuitively, one can use a cover tree over a dataset X to retrieve k diverse items from X by selecting any k items from the highest level that contains k or more items. Drosou and Pitoura [14] used cover trees for diversification. However, that work assumes queries over continuous data with sliding windows, so a single cover tree can answer consecutive queries that share results. In this paper, we are looking at a more general problem, as we wish to support range queries that may or may not share results. Our RC-Index uses cover trees internally, but it is a more complex structure, and our algorithms and approximation guarantees differ from the ones in the prior work.

A cover tree embeds items into a tree with multiple levels. Every level of the tree has an integer level number, ℓ. The root is at the highest level and we define this level as Level $\ell_{\max}$. Then each item may have other items as children. One special feature of cover tree is: as soon as an item x appears at Level ℓ_x as a child of another item, x will always have itself as a child at the next level $(\ell_x - 1)$. In addition, each level also has a distance threshold $\theta_\ell = b^\ell$, where $b > 1$ is a "base" distance parameter defined for each cover tree. Let C_ℓ be the set of items at Level ℓ. Formally, a cover tree must obey the following three invariants:

1. **Nesting**: $C_\ell \subseteq C_{\ell-1}$. If an item appears at level ℓ, it must appear at all levels below ℓ.
2. **Covering**: If $x_i \in C_\ell$ and x_j is its direct child, $dist(x_i, x_j) \leq \theta_\ell = b^\ell$. This implies that an item at Level ℓ covers all its direct children within a ball whose radius is θ_ℓ.
3. **Separation**: If $x_i, x_j \in C_\ell$ and $x_i \neq x_j$, $dist(x_i, x_j) > \theta_\ell = b^\ell$. This indicates that the pairwise distances between all distinct items at Level ℓ must be greater than θ_ℓ. In practice, items with $dist(x_i, x_j) = 0$ should be aggregated as a list of ids at all levels to satisfy the separation invariant.

The height of a cover tree depends on the data. The lowest level, $\ell_{\min}$, is determined by the closest pair of items due to the Separation invariant; formally, $\ell_{\min} = argmax_\ell \{dist(x_i, x_j) > b^\ell \mid x_i, x_j \in X\}$. The highest level, $\ell_{\max}$, depends on which item is chosen to be the root. It can be any integer in $(-\infty, +\infty)$ as long as the root covers all items. Although an item may appear in more than one level, we only need $O(n)$ space to store all items of a cover tree where n is the number of *distinct* items. We will defer the discussion of cover tree's time and space complexity to Section 4.

Figure 3 depicts the highest four levels of an example cover tree on a set of ATMs in New York City (south of Central Park). The ATMs' latitude and longitude are normalized to $[0, 1)$. We highlight items at each level as red points and display all other items as grey points. We set the base distance of this cover tree as $b = 2.0$. Recall $\ell_{\max}$ is a data-dependent value. The root of the cover tree in this example turns out to be at Level $\ell_{\max} = 0$.

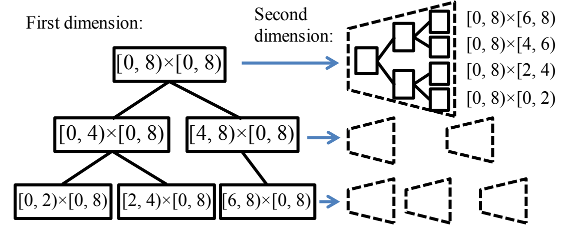


Figure 4: An example 2-dimensional range tree. Every node in the first dimension range tree points to another range tree in the second dimension.

2.5 Range Tree

A range tree [31, 3, 44, 29] is a nested tree structure. The left hand side of Figure 5 shows an example of a simple, 1-dimensional range tree. The root of the tree represents the entire range and every descendant non-leaf node corresponds to a subrange. The leaf nodes are the actual data items. Figure 4 illustrates a 2-dimensional range tree. In this particular example, we assume the data space is $[0, 8) \times [0, 8)$ and each inner range is evenly split into two subranges to simplify the presentation. In practice, the actual space and separator depends on the data distribution. We also omit leaf items for simplicity. The root of the whole range tree is $[0, 8) \times [0, 8)$ at the left side. It splits on the first dimension to get the two children $[0, 4) \times [0, 8)$ and $[4, 8) \times [0, 8)$. In the meantime, it points to another range tree at the right hand side. This range tree splits on the second dimension till the finest subrange like $[0, 8) \times [0, 2)$ and $[0, 8) \times [2, 4)$. Similarly, every inner node in the first dimension points to a range tree that splits on the second dimension. So when we implement a range tree, each node can have at most three children in two categories: $left$ and $right$ for the current dimension and *next* for exactly the next dimension. A node at the finest subrange does not have $left$ and $right$. Each node has at most one *next*: the nodes in the last dimension do not have *next*.

3. INDEX-BASED FRAMEWORK

In this section we focus on how the Query Module and the Diversification Module work. We prove their approximation ratio and complexity in Section 4.

3.1 Query Module: Sketch

We start by explaining how we design the Query Module. First of all, we need the Query Module's Diversity Index (DI) to extract diverse items from a large set of input items even before we consider any queries. A cover tree, which organizes items according to their distances, naturally satisfies our requirement. However, a DI with only one single cover tree is not enough to answer various user queries. For example, if we build only one cover tree $CT_{[0,8)}$ for items in range $[0, 8)$, how can we answer a query on $[2, 8)$? In the best case, if one subtree of $CT_{[0,8)}$ coincidentally contains all items in range $[2, 8)$ without any items outside $[2, 8)$, we can directly query this subtree to retrieve diverse items as we mentioned in Section 2.4. However, in general, we cannot rely on this case. As we point out in Section 2.2, the attributes used to compute distance (e.g. `Latitude` and `Longitude`) can be different from the attributes filtered by a range query (e.g. `Close_time`). So, a subtree of the cover tree $CT_{[0,8)}$ is *not* a cover tree of a subrange. In

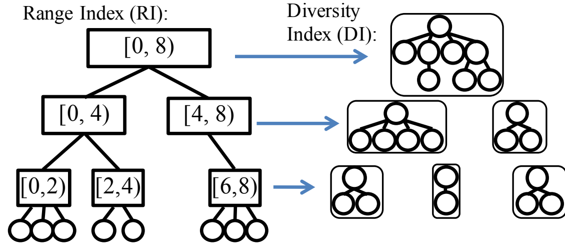


Figure 5: RC-Index: maps every Range Index (RI) node to a Diversity Index (DI) tree. It is the core of Query Module. This example Range Index supports 1-dimensional range queries.

other words, we cannot assume a cover tree's structure follows any *range* pattern.

So, we build the DI with multiple cover trees on different range partitions of X and extract diverse items from them *with an approximation ratio*, which is one of our main contributions. We base our approach on two important techniques. First, we show how we can carefully extract items from multiple non-overlapping cover trees to ensure the diversity score is no less than a factor times the optimal diversity score (Section 3.2). In other words, this approach is a constant-factor approximation algorithm. Second, we answer various queries while limiting the number of cover trees we build (Section 3.3). This is because we transform all range queries to a limited number of subordinate searches using the Range Index. For example, we may change a 1-dimensional range query on $[2,8)$ into two subordinate searches on $[2,4)$ and $[4,8)$. So, we only need to build cover trees for these subordinate searches. These two techniques, used together, form an index to sustain our Query Module, which we call an RC-Index.

EXAMPLE 2 (CONTINUING EXAMPLE 1). *A user can create an RC-Index as follows to support the query q in Example 1:*

```
CREATE RC_Index ON ATM_data(Close_time)
DIVERSE(distE(Latitude, Longitude));
```

Our system can use this index to solve both MAXMIN *and* MAXSUM. *We focus on* MAXMIN *in this paper. Appendix D shows how to solve* MAXSUM.

Figure 5 depicts the high level idea of RC-Index of our Query Module. It consists of two indexes: Range Index (RI) and Diversity Index (DI). The RI is to support the range query. It can be a B+ tree, R-tree, k-d tree, range tree, and so on. On the left hand side in the figure, we show an example RI for 1-dimensional range queries. We display a range on each node to indicate its coverage. The root covers the range $[0,8)$. Then it splits the range into sub-ranges as children. The ranges correspond to interior nodes, while the leaves are actual items. Every RI node is mapped to a DI tree on the right hand side. For instance, the RI node $[0,4)$ is mapped to a DI tree that covers all five items within this range. At the bottom, every individual item is itself a DI tree, so we do not need to explicitly map leaves of the RI to a DI, which saves some space.

Our Query Module has the following features.

- We allow the attributes used for distance calculation $\mathbb{A}_{dist}$ to be different from the attributes filtered by the range query $\mathbb{A}_q$, which makes our approach widely applicable.
- An RC-Index supports not only any range query q on the query attributes $\mathbb{A}_q$ but also various k. For instance, if a system would like to return top 10 results to one user but top 20 results to another user, **one** index supports both queries. This k does not need to be hard-wired in the index. It only controls the number of candidates to extract from the index during query evaluation.
- An RC-Index supports a dynamic scenario where a system inserts and deletes items between queries. We explain index maintenance in Section 4.2.

Algorithm 2: Candidate Extraction

Input: Parameter k; Set of cover trees $T = \{CT_1, CT_2, \ldots\}$; Extra level δ.
Output: Set of candidate items X^C.

1 $X^C \Leftarrow \emptyset$
2 **foreach** $CT \in T$ **do**
3   $X^C \Leftarrow X^C \cup$ `ExtractTree`(CT, k, δ)
4 **return** X^C
5 **Function** `ExtractTree`(CT, k, δ)
6   **if** $|CT| \leq k$ **then**
7   **return** All items in CT
8   $\ell_k \Leftarrow \underset{\ell}{arg\,max} |C_\ell| \geq k$
9   $\ell \Leftarrow \max\{\ell_k - \delta, \ell_{\min}\}$
10   **return** C_ℓ

3.2 Query Module: Diversity Index

We introduce the Diversity Index first as it is the essential part of our framework. Recall that we use the cover tree [5] which is originally designed for nearest neighbor search. It organizes items according to distance, which allows us to extract diverse items from one or more cover trees with a performance guarantee. Note that we extract more than k candidate items from the Diversity Index in our Query Module. Later we choose exactly k diverse items from these candidates in our Diversification Module (Section 3.5).

Algorithm 2 explains how we extract more than k candidate items from multiple cover trees. We enumerate the cover trees and extract a level of items from each tree in `ExtractTree` (Line 5 to Line 10). This function finds the highest level with at least k items (Line 8) and goes δ levels down (Line 9) to return all items at that level.

EXAMPLE 3. *Consider applying Algorithm 2 to a single cover tree in Figure 3 with $k = 3$ and $\delta = 1$. Our function* `ExtractTree` *firstly makes sure this tree has more than 3 items. Then it finds the highest level with at least 3 items: Level $\ell_k = -1$. Finally it goes $\delta = 1$ level down to Level -2 and returns all its 9 items.*

The candidate items extracted by Algorithm 2 are diverse, which allows us to later select k items with high diversity score from these candidates. The δ parameter controls how many more items the system extracts. Greater δ leads to more candidate items, which means better approximation ratio but longer runtime. Formally, if the optimal diversity score on X is f^*, and the optimal diversity score on extracted X^C is f^C, we have $f^C \geq \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$. We prove this property and discuss the parameters in Section 4.1.

3.3 Query Module: Range Index

The previous Section 3.2 shows how to extract diverse candidates from a set of cover trees, while this section shows how to get this set of cover trees given a range query.

Given a range query, we transform it to several subordinate searches, each of which corresponds to a cover tree. Specifically, we use *Range Tree* [31, 3, 44, 29] to transform *any* range query to at most $\log^d n$ subordinate searches, where $n = |X|$ is the size of

Algorithm 3: Range Query Tree Extraction

Input: Range query q; RC-Index RC.
Output: Set of cover trees T.

```
1  T ⇐ QueryRangeTree(q, RC.root)
2  return T
3  Function QueryRangeTree(q, node)
4      if node.range ⊆ q.range then
5          return {node.CT}
6      if node.range ∩ q.range = ∅ then
7          return ∅
8      nowT ⇐ ∅
9      foreach child ∈ {node.left, node.right, node.next} do
10         nowT ⇐ nowT ∪ QueryRangeTree(q, child)
11     return nowT
```

Algorithm 4: Greedy Diversification

Input: Candidate set $X^C = \{x_1, x_2, ...\}$; Size of output k where $1 \le k \le |X^C|$.
Output: Set of diverse items S.

```
1  x_random ⇐ one random item in X^C
2  S ⇐ {x_random}
3  X^C ⇐ X^C − {x_random}
4  foreach x_i ∈ X^C do
5      dist[x_i] ⇐ dist(x_i, x_random)
6  while |S| < k do
7      x_farthest ⇐ argmax_{x_i ∈ X^C} dist[x_i]
8      S ⇐ S ∪ {x_farthest}
9      X^C ⇐ X^C − {x_farthest}
10     foreach x_i ∈ X^C do
11         dist[x_i] ⇐ min{dist[x_i], dist(x_i, x_farthest)}
12 return S
```

the data and $d = |\mathbb{A}_q|$ is the dimensionality. Range tree is one of the data structures that efficiently support range queries [4]. We use range tree as our Range Index for two reasons: (1) It can help us transform a range query to a reasonable number of subordinate searches. (2) It can answer a range query efficiently. In a conventional range query evaluation scenario, one can extract the entire $q(X)$ with time complexity $O(\log^d n + |q(X)|)$. So it is faster than k-d tree [2] or quad tree [17] whose worst case query complexity is $O(d \cdot n^{1-1/d} + |q(X)|)$ [28].

In the following example we transform a 2-dimensional query to four ssubordinate searches in a range tree:

EXAMPLE 4. *When receiving a 2-dimensional query* $[2,8) \times [2,6)$ *on a range tree in Figure 4, we split the first dimension and stop at* $[2,4) \times [0,8)$ *and* $[4,8) \times [0,8)$. *Then we split the second dimension to reach four nodes* $[2,4) \times [2,4)$, $[2,4) \times [4,6)$, $[4,8) \times [2,4)$, *and* $[4,8) \times [4,6)$.

We briefly present the time and space complexity of a conventional range tree. The query complexity is $O(\log^d n)$, because the range of each dimension is split to at most $\log n$ subranges. Its batch construction time is $O(n \log^d n)$. The amortized time complexity of insertion and deletion is $O(\log^d n)$ when we carefully maintain the balance of the tree [44, 29, 32]. Its space complexity is $O(n \log^d n)$.

3.4 Query Module: Candidate Extraction

In this section we put the Diversity Index and Range Index together to form a full RC-Index and show how our Query Module (QM) works. QM works in two steps as depicted in Line 1 and Line 2 of Algorithm 1. First, Algorithm 3 traverses the RC-Index to collect a set of cover trees for a range query. Second, we pass the cover trees to Algorithm 2 to get candidates. Here is an example:

EXAMPLE 5. *(Continuing Example 4) Given the range query* $[2,8) \times [2,6)$*, we traverse our index and stop at four nodes:* $[2,4) \times [2,4)$, $[2,4) \times [4,6)$, $[4,8) \times [2,4)$, *and* $[4,8) \times [4,6)$. *We collect their corresponding cover trees into a set* T. *Then we pass it to Algorithm 2 to get candidate items* X^C.

This approach successfully reduces the number of scanned items from $|q(X)|$, i.e. linear to n in the worst case, to polylogarithmic as we prove in Section 4.1.

3.5 Diversification Module

This module takes a set of candidate items $X^C = \{x_1, x_2, ...\}$ as input and outputs exactly k diverse items.

We deploy a simple $O(k \cdot |X^C|)$ greedy algorithm [40, 38] for the Diversification Module (DM). This greedy algorithm is expensive as it has to at least scan all input items, but it works well as our DM because the number of its input items is limited by the previous Query Module. Algorithm 4 introduces the detail of this greedy algorithm. Initially, it selects a random item from X^C to put into the output set S (Line 2). Then, it maintains an array to track the distances between selected and unselected items (Line 5). It iteratively chooses the item with the greatest distance from the selected items until it finds k items (Lines 6 to 11).

The complexity and approximation ratio of Algorithm 4 are as follows: The time complexity is $O(k \cdot |X^C|)$; The space complexity is $O(|X^C|)$; It is a 1/2-approximation algorithm [40, 38].

To solve the MAXSUM version of this problem, instead of MAXMIN, we modestly change Algorithm 4 as explained in Appendix D.

Query	Approximation Ratio:	$\frac{b-1-2b^{1-\delta}}{2(b-1)}$
	Space Complexity:	$O(k\gamma^{4(\delta+1)} \log^d n)$
	Time Complexity:	$O(k^2\gamma^{4(\delta+1)} \log^d n)$
Index	Batch Construction Time:	$O(\gamma^6 n \log^{d+1} n)$
	Amortized Insert/Delete:	$O(\gamma^6 d \log^{d+2} n)$
	Space Complexity:	$O(n \log^d n)$

Figure 6: The RC-Index is efficient at both querying and indexing.

4. PERFORMANCE ANALYSIS

We analyze the performance of query evaluation and index construction in this section. Figure 6 provides a summary. The discussion in this section assumes the RC-Index is built on the query attributes $\mathbb{A}_q$. We relax this assumption in Section 5.

4.1 Query Quality and Complexity

We prove the approximation ratio, time complexity, and space complexity for our overall query answering algorithm (Algorithm 1).

Approximation ratio

Given a set of cover trees that covers items in query result $q(X)$, we prove that Algorithm 2 extracts a high-quality set of candidate items X^C:

THEOREM 1. *Let the optimal diversity score on* $q(X)$ *be* f^* *and the optimal diversity score on extracted* X^C *be* f^C. *We have* $f^C \ge \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$, *where* b *is the base distance parameter of the cover trees we build and* δ *is the extra level parameter of Algorithm 2.*

In order to prove this theorem, we show there exists a subset $Y = \{y_1, ..., y_k\} \subseteq X^C$ whose diversity score f^Y is at least $\frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$. Specifically, let the optimal set be $S^* = \{x_1, ..., x_k\} \subseteq q(X)$ which leads to the optimal diversity score f^*. These $\{x_1, ..., x_k\}$ may come from one or more cover trees. Now we define $Y = \{y_1, ..., y_k\}$ based on S^*. Each y_i can be viewed as a "substitute" of x_i on x_i's cover tree CT:

• Case 1: If the tree CT has no more than k items, Algorithm 2 puts all items of this tree into X^C. So we define $y_i = x_i \in X^C$.

• Case 2: If the tree CT has more than k items, we define y_i as the ancestor of x_i at Level $(\ell_k - \delta)$[3], where ℓ_k is the highest level of tree CT with at least k items (Line 8). This y_i is also returned in X^C.

We would like to show x_i and y_i are close enough so that f^Y is not much worse than f^*. Formally, the distance between x_i and y_i satisfies the following lemma:

LEMMA 1. $dist(x_i, y_i) \leq \frac{b^{-\delta}}{1-b^{-1}} \cdot f^*$.

PROOF. Similarly, $dist(x_i, y_i)$ has two cases:

• Case 1: When the tree CT has no more than k items, since $y_i = x_i$, $dist(x_i, y_i) = 0 \leq \frac{b^{-\delta}}{1-b^{-1}} \cdot f^*$.

• Case 2: When the tree CT has more than k items, any k items at Level ℓ_k also exist at Level $(\ell_k - \delta)$ due to **Nesting** and should be extracted. They have already formed a solution with diversity score b^{ℓ_k} because of **Separation**. In addition, as f^* is the optimal diversity score of all items, $b^{\ell_k} \leq f^*$.

Since y_i is the ancestor of x_i and y_i is at Level $(\ell_k - \delta)$, according to **Covering**:

$$dist(x_i, y_i) < \sum_{\ell=-\infty}^{\ell_k - \delta} b^{\ell} = \frac{b^{\ell_k - \delta}}{1 - b^{-1}}$$

Thus: $dist(x_i, y_i) < \frac{b^{\ell_k - \delta}}{1 - b^{-1}} = \frac{b^{-\delta}}{1 - b^{-1}} \cdot b^{\ell_k} \leq \frac{b^{-\delta}}{1 - b^{-1}} \cdot f^*$ □

Now we can prove Theorem 1:

PROOF. According to triangle inequality and symmetry (Definition 1), for any x_i and the corresponding y_i:

$$dist(x_i, x_j) \leq dist(x_i, y_i) + dist(y_i, y_j) + dist(x_j, y_j)$$

Since f^* is the diversity score of $\{x_1, ..., x_k\}$, $dist(x_i, x_j) \geq f^*$. So for any distinct y_i and y_j:

$$\begin{aligned} dist(y_i, y_j) &\geq dist(x_i, x_j) - dist(x_i, y_i) - dist(x_j, y_j) \\ &\geq f^* - \frac{b^{-\delta}}{1 - b^{-1}} \cdot f^* - \frac{b^{-\delta}}{1 - b^{-1}} \cdot f^* \\ &= (1 - \frac{2b^{-\delta}}{1 - b^{-1}}) \cdot f^* = \frac{b - 1 - 2b^{1-\delta}}{b - 1} \cdot f^* \end{aligned}$$

Since $f^C \geq f^Y \geq dist(y_i, y_j)$, $f^C \geq \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$. □

Note that $\frac{b-1-2b^{1-\delta}}{b-1}$ must be greater than 0 to ensure a non-trivial bound. For instance, when $b = 2.0$ and $\delta = 3$, the bound is $1/2$. When $b = 3.0$ and $\delta = 2$, the bound is $2/3$.

We further show the bound in Theorem 1 is tight.

THEOREM 2. *Given any $\varepsilon > 0$, there exists a worst case making* $(\frac{b-1-2b^{1-\delta}}{b-1} + \varepsilon) \cdot f^* \geq f^C$.

Please see Appendix B for detailed proof.

Now we can derive the overall result quality of Algorithm 1:

[3] The items at Level $(\ell_k - \delta)$ are the same as those at Level $\ell_{\min}$ (i.e., $C_{\ell_k - \delta} = C_{\ell_{\min}}$) when $\ell_k - \delta < \ell_{\min}$.

THEOREM 3. *The approximation ratio of Algorithm 1 is* $\frac{b-1-2b^{1-\delta}}{2(b-1)}$.

PROOF. Algorithm 1's approximation ratio is the product of the two modules' approximation ratios. The Query Module's ratio is $\frac{b-1-2b^{1-\delta}}{b-1}$ as in Theorem 1. The Diversification Module's ratio is $\frac{1}{2}$. So, the ratio of Algorithm 1 is $\frac{b-1-2b^{1-\delta}}{b-1} \cdot \frac{1}{2} = \frac{b-1-2b^{1-\delta}}{2(b-1)}$. □

Note that the approximation ratio approaches $1/2$ as b increases or δ increases.

This bound is also tight because the approximation bounds of both modules are tight.

Query time complexity

A cover tree's complexity analysis requires a data-dependent *expansion constant*, γ. Given X, let the closed ball of radius r centered at x be $B(x, r) = \{x' \in S | dist(x, x') \leq r\}$. So the expansion constant is the smallest γ such that $|B(x, b \cdot r)| \leq \gamma |B(x, r)|$ for every $x \in X$ and $r > 0$ [5].

Each item can have at most γ^4 children [5]. Recall, the number of items at Level $(\ell_k + 1)$ is strictly less than k according to Algorithm 2. Thus, the number of items at Level $(\ell_k - \delta)$ is at most $k\gamma^{4(\delta+1)}$. Therefore we extract $O(k\gamma^{4(\delta+1)})$ items and the time complexity is also $O(k\gamma^{4(\delta+1)})$ on one single cover tree.

We have the following conclusion on multiple cover trees:

LEMMA 2. *The Query Module (Algorithms 2 and 3) returns a candidate item set X^C with $O(k\gamma^{4(\delta+1)} \log^d n)$ items.*

PROOF. A range query visits $O(\log^d n)$ nodes in the range tree. We map each node to a cover tree. For each cover tree, we extract $O(k\gamma^{4(\delta+1)})$ items. So, we extract $O(k\gamma^{4(\delta+1)} \log^d n)$ candidate items totally. □

COROLLARY 1 (OF LEMMA 2). *The worst-case time complexity of the Query Module (Algorithms 2 and 3) is $O(k\gamma^{4(\delta+1)} \log^d n)$.*

THEOREM 4. *The worst-case time complexity of Algorithm 1 is $O(k^2\gamma^{4(\delta+1)} \log^d n)$.*

This complexity is due to the complexity of the Diversification Module, $O(k \cdot |X^C|)$. Please see Appendix B for the proof.

While the time complexity depends on k quadratically, k is small in practice, in particular in the context of diversification. The exponential term $O(\gamma^{4(\delta+1)})$ looks large. But one should notice that the cover tree is designed for nearest neighbor search, and one search query takes $O(\gamma^{12} \log n)$ time [5], whose exponent of γ is also large. In practice, this exponential term turns out to be insignificant. We demonstrate this by showing the wall-clock runtime in Section 6.

Query space complexity

During query evaluation, we need to store the candidates we extract. So the space complexity is $O(k\gamma^{4(\delta+1)} \log^d n)$ as in Lemma 2.

Discussion of parameters b and δ

The parameters b and δ must satisfy several constraints: (1) the cover tree requires $b > 1$ [5]; (2) δ is a non-negative integer corresponding to extra levels in Algorithm 2; (3) the approximation ratio $\frac{b-1-2b^{1-\delta}}{2(b-1)}$ must be greater than 0.

Higher b means better (higher) approximation ratio but higher worst-case time complexity. Moreover, recall that the expansion constant is the smallest γ, such that $|B(x, b \cdot r)| \leq \gamma |B(x, r)|$; so, greater b brings greater γ and higher worst-case time complexity.

Algorithm 5: RC-Index Insertion

```
Input: New item x; RC-Index RC.
Output: Updated RC-Index RC.
1  Insert(x, RC.root)
2  return RC
3  Function Insert(x, node)
4      Insert x into the cover tree node.CT
5      Update node.im based on x
6      if node.im ∉ [β, 1−β] then
7          Batch construct the tree rooted at node with x included
8          return
9      foreach child ∈ {node.left, node.right, node.next} do
10         if x ∈ child.range then
11             Insert (x, child)
```

This property is intuitive because greater b makes the threshold $\theta_\ell = b^\ell$ of the cover tree change more quickly between levels, resulting in a shallower tree. In this case, Algorithm 2 extracts more candidates, which improves the approximation ratio but also increases runtime in the worst case. Similarly, greater δ improves the approximation ratio but increases worst-case time complexity. This is because greater δ makes Algorithm 2 extract more candidates from lower levels of the cover trees.

4.2 Index Complexity

We discuss the batch construction time complexity, insertion/deletion time complexity, and space complexity of RC-Index. We show that the RC-Index can be created and maintained efficiently, proving formal bounds on the cost of key operations.

Batch construction

We batch construct the RC-Index recursively. Given a node, we firstly build a cover tree for the node. Then we partition its range to two subranges and construct its *left* and *right* children in the same dimension. Finally we construct its *next* child in the next dimension. Its pseudo code is Algorithm 7 in Appendix.

THEOREM 5. *The batch construction of the RC-Index on n item takes* $O(\gamma^6 n \log^{d+1} n)$ *time in the worst case.*

Please see Appendix B for our proof through induction.

Insertion and deletion

Algorithm 5 lists the pseudocode for insertion. The deletion process is similar. Our idea is to apply a balance bound as in [35, 32]. Specifically, we define a rank of each node in our range tree as $rank(node) = (1 + \#\text{ nodes in subtree rooted at } node)$. A subtree here only means the subtree in the same dimension, excluding the nested subtrees in the next dimension. An empty tree has $rank = 1$ while a leaf node has $rank = 2$. Then we define the imbalance factor for each node as $im(node) = rank(node.left)/rank(node)$. Intuitively, an $im(node)$ closer to $1/2$ means a more balanced subtree rooted at $node$. A range tree after batch construction has $im \in [1/3, 2/3]$ for all nodes. In Algorithm 5, we insert a new item into the range tree and rebalance by batch construction of a subtree if its im falls out of $[\beta, 1-\beta]$, where $0 < \beta < 1/3$.

THEOREM 6. *The amortized time complexity of insertion or deletion of RC-Index is* $O(\gamma^6 d \log^{d+2} n)$.

Our proof is similar to the proof in [32]. We define an imbalance score of the tree. An insertion or deletion may increase the imbalance score while a rebuild always decreases the imbalance score. Please refer to Appendix B for the proof.

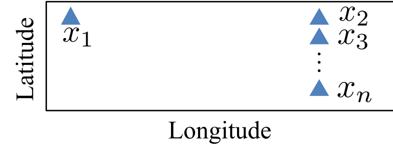


Figure 7: When $\mathbb{A}_q \supset \mathbb{A}_{RC}$, naïvely extending *RC* to answer q can result in arbitrarily bad diversity score.

Index space complexity

The actual space complexity of a cover tree on n items is $O(n)$, although some items may appear in multiple levels. We store only one record for each distinct item containing its id, highest level number, parent id, and children ids. So even though an item appears in many levels, we only store it once.

THEOREM 7. *The RC-Index on n items takes* $O(n \log^d n)$ *space.*

We prove the theorem by induction in Appendix B.

5. INDEX SELECTION

In this section, we discuss how to select indexes given a set of queries that filter on various sets of attributes. Before this section, we only consider building one specific RC-Index *RC* to answer a query q filtering on one set of attributes. Formally, suppose a range query q applies filters on an attribute set $\mathbb{A}_q$ and the Range Index of *RC* is built on an attribute set $\mathbb{A}_{RC}$. We have only considered the case where $\mathbb{A}_{RC} = \mathbb{A}_q$. We consider next some interesting properties of RC-Index when $\mathbb{A}_{RC}$ may not equal $\mathbb{A}_q$. These properties are useful when we discuss index selection.

THEOREM 8. *The database can use RC to evaluate q with approximation ratio* $\frac{b-1-2b^{1-\delta}}{2(b-1)}$ *if* $\mathbb{A}_q \subseteq \mathbb{A}_{RC}$.

This is simply due to the property of the range tree. The detailed proof is in Appendix C.

Then a natural question is: can we use *RC* when $\mathbb{A}_q \supset \mathbb{A}_{RC}$? Specifically, can we use *RC* to find a diverse set of items satisfying filters on $\mathbb{A}_{RC}$ and then apply filters on $(\mathbb{A}_q - \mathbb{A}_{RC})$? Unfortunately, the output diversity score of this approach can be arbitrarily bad. For example, consider a special case where $\mathbb{A}_q = \mathbb{A}_{dist} = \{\texttt{Latitude}, \texttt{Longitude}\}$. The distance function is still Euclidean. $\mathbb{A}_{RC} = \{\texttt{Latitude}\} \subset \mathbb{A}_q$. Assume $k = 2$. As Figure 7 illustrates, *RC* may mistakenly pick diverse items within $\{x_2, ..., x_n\}$ because their latitudes are different from each other. However, the best diverse set should be $\{x_1, x_n\}$. In this case, the diversity score is arbitrarily bad because $dist(x_2, x_n)/dist(x_1, x_n)$ can approach zero.

Following the proof of Theorem 8, we can easily derive the time complexity of evaluating q. Assume $d = |\mathbb{A}_{RC}|$ and $d_q = |\mathbb{A}_q|$. We have:

THEOREM 9. *The database can use RC to evaluate q with time complexity* $O(k^2 \gamma^{4(\delta+1)} \log^{d_q} n)$ *if* $\mathbb{A}_q \subseteq \mathbb{A}_{RC}$ *and* $d = O(k^2)$.

$d = O(k^2)$ means that the dimensionality of the RC-Index is not greater than order of k^2. This usually holds in practice. d is small because a user is unlikely to apply filters on more than 10 dimensions. But k could be easily greater than 5 (thus, $k^2 > 25$). $d = O(k^2)$ makes the complexity of extracting diverse items from cover trees dominate the whole complexity. Please see Appendix C for detailed proof.

Given the above two properties of RC-Index, now we discuss how to select indexes given a set of queries $Q = \{q_1, q_2, ...\}$.

Dataset	# Records (n)	Description
City	5,922	Greek cities
Bank	45,211	Marketing data of a Portuguese bank
Census	48,842	Census income of US
Forest	581,012	Forest cover type in Colorado, US
Gas	928,991	Gas sensors

Figure 8: We evaluate RC-Indexes on a variety of real-world datasets, ranging from 6,000 to about 1 million records.

Parameter	Values
# records, n	$\{10^3, 5\times10^3, 10^4, \mathbf{5\times10^4}, ..., 10^6\}$
# diverse items to return, k	$\{\mathbf{10}, 50, 100, 150, 200\}$
# query attributes	$\{\mathbf{1}, 2, 3, 4, 5, 6\}$
Base-distance of cover tree [5], b	$\{1.1, 1.5, \mathbf{2.0}, 3.0, 4.0\}$
Extra-level of Algorithm 2, δ	$\{0, 1, 2, \mathbf{3}, 4, 5\}$
Distance function	$\{L^1, \mathbf{L^2}\}$

Figure 9: We vary the parameters one at a time (with bold values indicating the default of the parameter when it is not varied).

Theorem 8 and 9 suggest that an RC-Index on $\mathbb{A}_{RC}$ can answer queries whose $\mathbb{A}_q \subseteq \mathbb{A}_{RC}$ with the same approximation ratio and time complexity as those when $\mathbb{A}_q = \mathbb{A}_{RC}$. So a naïve plan is to build a large index that covers all queryable attributes, i.e. $\mathbb{A}_{RC} = \cup_{q\in Q}\mathbb{A}_q$. The only problem of this plan is its large space complexity, $O(n\log^{|\mathbb{A}_{RC}|} n)$. We cannot build this large RC-Index when the database has a small space limit. Then the index selection becomes an optimization problem that optimizes the runtime and the approximation ratio of a workload while satisfying the space constraint. We defer the study of this problem in our future work.

6. EXPERIMENTAL EVALUATION

In this section, we present a thorough evaluation of RC-Indexes. Our results demonstrate that our approach can extract high quality diverse items efficiently, outperforming state-of-the-art techniques.

Data

We experiment with real-world and synthetic datasets. Figure 8 lists the real-world datasets. We use the City dataset containing 5,922 Greek cities and villages [13, 14]. Bank [34] and Census are two popular datasets with $\geq 10,000$ records from UCI Machine Learning Repository [30]. Forest [6] and Gas [20] are two larger datasets from the same repository. We remove categorical attributes because their small domains over-simplify the diversification problem. Then we randomly pick the remaining numerical attributes as $\mathbb{A}_q$ and $\mathbb{A}_{dist}$.

We also generate a synthetic dataset whose attributes follow a uniform distribution. In our synthetic data experiments, we vary the parameters one at a time as listed in Figure 9. The query q applies filters on a set of attributes. The distance function is on another two attributes. By default, our algorithm is $1/4$-approximate when $b = 2.0$ and $\delta = 3$. Some (b, δ) combinations make the ratio less than 0, but they work well in practice as we will show in Section 6.2.

Configuration

We use a MacBook with 2GHz dual-core Intel Core i7 and SSD. Our prototype system runs in PostgreSQL, utilizing 4GB memory.

We implement our prototype as user defined functions (UDFs) using the C language at server side. The user can specify the attributes that allow range queries and also a distance function on certain attributes. Then we create two auxiliary tables as our indexes: one for Range Index and the other for Diversity Index. The user can conduct batch construction, insert items, or query diverse items by invoking our UDFs.

Algorithm	Query Complexity	Approx. Ratio	Max Ratio
RC-Index	$O(k^2\gamma^{4(\delta+1)}\log^d n)$	$\frac{b-1-2b^{1-\delta}}{2(b-1)}$	$< 1/2$
Greedy	$O(kn)$	$1/2$	$1/2$
Tree	$O(n^2)$	$\frac{(b-1)}{2b^2}$	$\leq 1/8$
Tree++	$O(\gamma^6 n\log n)$	$\frac{(b-1)}{2b^2}$	$\leq 1/8$

Figure 10: Theoretical bounds of RC-Indexes and state-of-the-art baselines.

Such a UDF-based implementation is flexible because it does not force a user to modify the source code of PostgreSQL. A user may not have the privilege to install a customized PostgreSQL if s/he is not the admin. GiST [19] also has such flexibility but there is no simple way to manipulate the inner nodes of an index tree using GiST. We can adapt our UDFs to triggers in the future to improve user experience. The UDF implementation offers good performance as we will demonstrate below.

Comparison with baselines

We compare with three baseline approaches. They are the state of the art to the best of our knowledge.

• `Greedy`. The Greedy algorithm selects a random item to initialize the diverse result set S. Then for every unselected item x, it maintains an array of the minimum distance between x and any item in S. It iteratively adds the item with the maximum minimum distance into S and updates the distance array. Its pseudo code is the same as Algorithm 4 but it takes all items in $q(X)$ as input. So in the worst case, its query time is $O(kn)$. This is a $1/2$-approximation algorithm [40, 38].

• `Tree`. Drosou and Pitoura [14] have developed an approximation algorithm based on a single cover tree. They assume that items arrive continuously so they perform insertion and deletion while selecting diverse items from the cover tree. We adapt this approach to support arbitrary range queries by updating the cover tree between queries. Their streaming data scenario is different from ours so such an adaptation is slower than our approach. Its batch construction time for building a cover tree is $O(n^2)$. In addition, its maximum approximation upper bound is $1/8$ when $b = 2.0$. But our approach can achieve, for example, $1/4$ approximation ratio within very short query time.

• `Tree++`. This is the original cover tree algorithm [5], which can be viewed as a variant of the above approach. Instead of finding the farthest pair of items to construct a tree in `Tree`, we randomly pick an item to start the tree construction in `Tree++`, which reduces $O(n^2)$ complexity to $O(\gamma^6 n\log n)$.

Figure 10 shows the worst-case query complexity and approximation ratio of all algorithms we compare. At a glance, all three baselines' query complexity is $\Omega(n)$ because they have to scan the entire $q(X)$ when $|q(X)| = O(n)$ in the worst case. The complexity of our method has no $O(n)$ term with the help of indexes. Our approximation ratio $\alpha = (b-1-2b^{1-\delta})/2(b-1)$ is better than `Tree` and `Tree++`'s. Moreover, α's upper limit equals `Greedy`'s ratio $1/2$, which is the best possible approximation ratio of a polynomial algorithm unless P=NP [38].

Since `Greedy` has the best approximation ratio while finding the optimum solution takes exponential time, we compare the diversity scores against `Greedy`'s. We compute the *relative score* for each algorithm $Algo$ as $f_{Algo}/f_{\texttt{Greedy}}$ in the following experiments.

6.1 Quality and Scalability

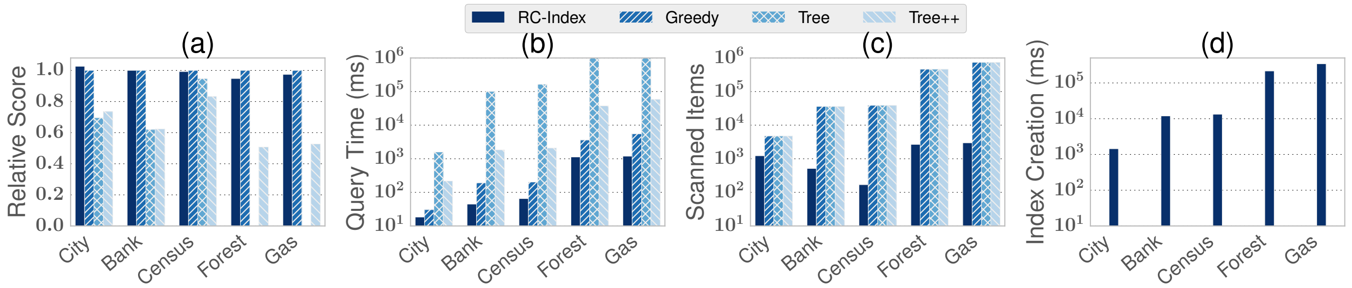


Figure 11: Real-world data: RC-Index is the fastest approach while ensuring a high quality result.

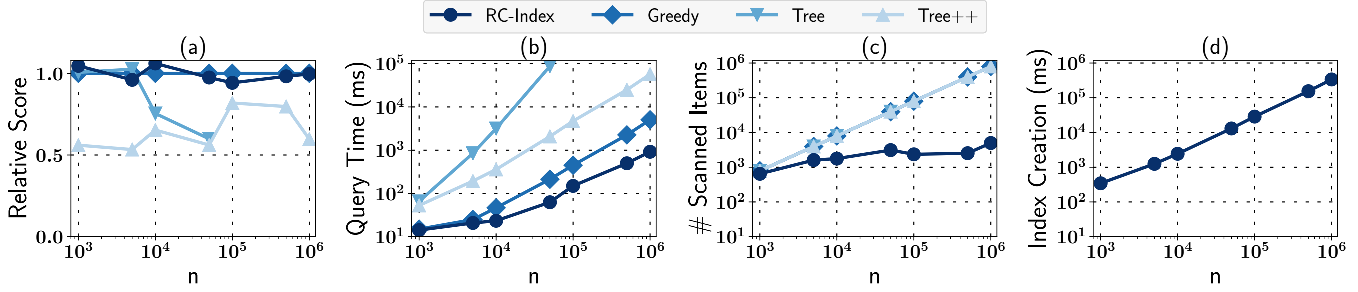


Figure 12: Synthetic uniformly-distributed data with varying size n: RC-Index is nearly an order of magnitude faster than the state of the art while ensuring a high quality result.

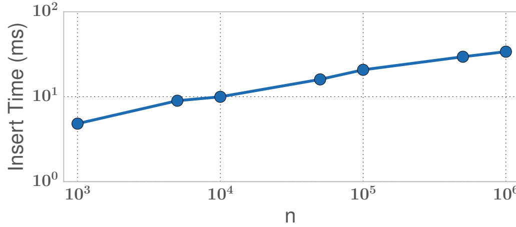


Figure 13: Per item insertion time for 100 items increases as n grows.

Our first experiment evaluates the performance of RC-Indexes on real-world data. Figure 11 shows that RC-Indexes outperform the baselines. Tree takes too long to execute on Forest and Gas, so its runtime is not depicted. (a) Figure 11a depicts the relative score compared to Greedy's. RC-Index's score is as good as Greedy's and is better than Tree and Tree++. (b) Figure 11b shows our query time is orders of magnitude less than the other three baselines. (c) Figure 11c tells us why RC-Index is efficient: it scans far fewer candidate items ($|X^C|$) than the other baselines do. (d) Figure 11d shows our indexing time is short: only 13.3 seconds for $48,842$ items or 343.8 seconds for $928,991$. It is a *one-time* cost after loading the data.

Our experiments on synthetic data show similar behavior. Figure 12 compares the performance on uniformly-distrbuted synthetic data where n varies within $\{10^3, 5\times10^3, 10^4, ..., 10^6\}$. We do not run Tree for $n \geq 10^5$ because its query time is too long. (a) Figure 12a shows that the quality of our result is as good as Greedy's, and they are both better than Tree and Tree++ when $n \geq 10^4$. In some cases, RC-Index outperforms Greedy even though its theoretical approximation ratio is worse than Greedy's. This is simply because the synthetic data is not the worst case data for RC-Index. (b) Figure 12b shows that RC-Index is an order of magnitude faster than the state of the art, Greedy and Tree++. Our query time is 0.9 second when there are 10^6 items. (c) Figure 12c shows that RC-Index scans far fewer candidate items than the other baselines do. (d) Finally, Figure 12d shows the batch index creation time of the synthetic data. RC-Index can index 10^5 items within only 28 seconds, or 10^6 items within 330 seconds. The indexing time increases almost linearly with regard to n (both in log scale in the figure), which is a desired property.

We repeat the same evaluation on synthetic data that follows a normal distribution $\mathcal{N}(0,1)$. We find little discernible impact and RC-Index still outperforms competitors. The result, which we include in Appendix F, is virtually identical to Figure 12.

We also plot the insertion time as n grows. For each n in $\{10^3, 5\times10^3, 10^4, ..., 10^6\}$, we insert 100 new items into RC-Index and compute the average insertion time. As Figure 13 illustrates, the insertion time grows slowly as n increases.

Next we fix $n = 5\times10^4$ and vary k in $\{10, 50, 100, 150, 200\}$. As we can see in Figure 14a, the quality of the result is robust to changes in k. Figure 14b shows very little increase in query time as k increases.

Next, we vary d in $\{1,2,3,4,5,6\}$ while fixing $n = 5\times10^4$ and $k = 10$. Figure 15a demonstrates that the quality is stable. Figure 15b shows the query time increases because the query time complexity is proportional to $\log^d n$, but we are still faster than competitors.

Finally, we change the distance function from Euclidean to Manhattan distance and compare all algorithms as n grows. Figure 16 depicts the relative score and query time. Similar to the performance under Euclidean distance, RC-Index is nearly an order of magnitude faster than the other algorithms and provides high diversity score.

6.2 Parameter Selection

In this section, we test the sensitivity of our approach on synthetic data with 10^5. Two parameters, b and δ, impact our approximation ratio $\frac{b-1-2b^{1-\delta}}{2(b-1)}$ and query complexity $O(k^2\gamma^{4(\delta+1)}\log^d n)$

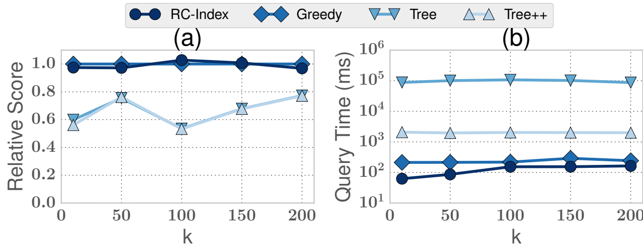


Figure 14: Varying k: `RC-Index`'s result quality stays the same and its query time increases slightly as k increases.

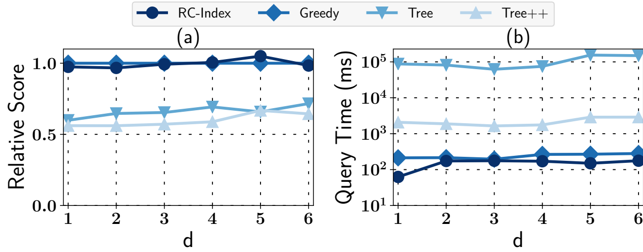


Figure 15: Varying d: `RC-Index`'s query time gets closer to Greedy's but the result quality is the same as d increases.

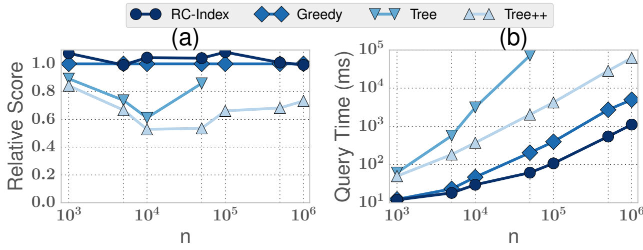


Figure 16: Comparison under Manhattan distance. The favorable performance of `RC-Index` is not sensitive to the distance function, as `RC-Index` outperforms all competitors.

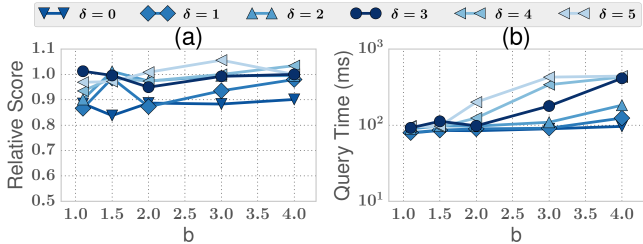


Figure 17: Varying b: greater b increases query time without significantly improving the result quality.

(the expansion constant γ is determined by b given the data). We vary them to see how performance changes. Note that some (b, δ) combinations result in $\frac{b-1-2b^{1-\delta}}{2(b-1)} \leq 0$, but they lead to good results in practice. We retain these values in the following figures.

Figure 17 depicts the performance using different b. We vary b in $\{1.1, 1.5, 2.0, 3.0, 4.0\}$ for six different δ between 0 and 5. On the one hand, greater b means greater approximation ratio in the worst case. But we do not observe such trend in Figure 17a, because the synthetic data following uniform distribution is not the worst-case data. It is unusual to run into worst-case scenario in practice. On the other hand, greater b means greater expansion constant γ. So the query time is longer as Figure 17b shows. When b and δ are large enough, the query time converges because the algorithm has extracted all items from $q(X)$.

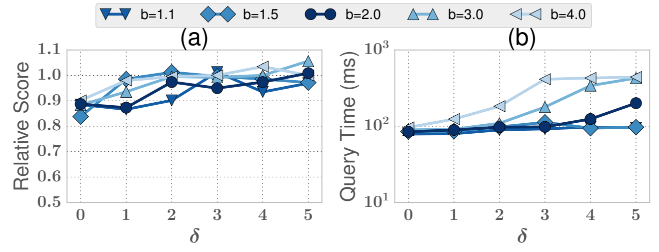


Figure 18: Varying δ: greater δ means better result but longer runtime.

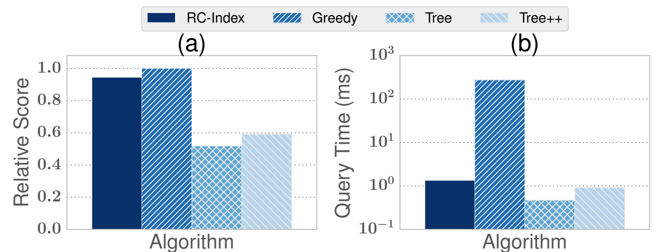


Figure 19: Our algorithm works well on streaming data.

Figure 18 shows how performance changes along with δ. We vary δ in $\{0, 1, 2, 3, 4, 5\}$ but fix b at different values this time. Intuitively, greater δ means our algorithm goes deeper in each cover tree to extract more candidate items. So the diversity score gets slightly better in general as Figure 18a depicts. The cost is that the query time gets longer as in Figure 18b. Again, the query time converges when b and δ are large, because the algorithm has extracted the entire $q(X)$.

6.3 Streaming Data

In this experiment, we examine the performance of `RC-Index` on streaming data which is the focus of prior work [14]. We redesign and reimplement the algorithms of Figure 10 and apply them on a data stream. We query this data stream through a sliding window. The window covers $50,000$ items. Every time we slide the window, we remove 10 old items, add 10 new items, and query the diverse items again.

Figure 19 shows the average relative score and query time of the four algorithms on 10 queries. `Tree` and `Tree++` have an advantage in this experiment as they were designed for streaming data. Nevertheless, the diversity score of RC-Index is close to the best score and better than both `Tree` and `Tree++`. In terms of query time, `Tree` [14] is the fastest, but RC-Index is only slightly slower.

6.4 Index Selection

In this experiment we demonstrate Theorem 8 and 9. Given a range query q that applies filters on a set of attributes $\mathbb{A}_q$, one can build an RC-Index whose Range Index is on an attribute set $\mathbb{A}_{RC}$ to answer q as long as $\mathbb{A}_q \subseteq \mathbb{A}_{RC}$. Theorem 8 and 9 show that the diversity score and query time are irrelevant to the size of $\mathbb{A}_{RC}$. Here we fix $|\mathbb{A}_q| = 1$ and vary $d = |\mathbb{A}_{RC}|$ in $\{1, 2, 3, 4, 5, 6\}$. As Figure 20 illustrates, the diversity score and query time are unaffected as d grows.

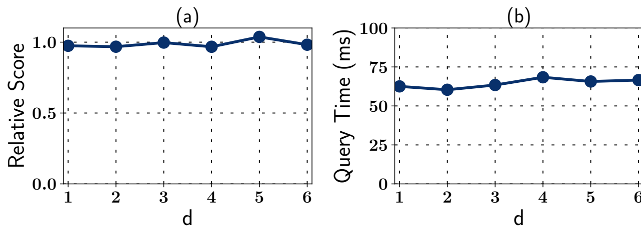


Figure 20: RC-Index supports partial range query well.

7. RELATED WORK

Search result diversification is about selecting a small subset of diverse items to present to the user when a large set of items satisfy the user query. It finds its application in many scenarios such as data exploration, Web search, and recommendation systems. According to the surveys [12, 39, 46], diversification problems can be classified into three categories: (1) content-based (or similarity-based) diversification finds items that are dissimilar to each other; (2) intent-based (or semantic coverage-based) diversification finds items relevant to various topics to help user further disambiguate the query; (3) novelty-based diversification finds items that are different from the previously retrieved ones for the user. Our work focuses on content-based diversification, which mainly maximizes the distance between selected items and the relevance of items to the query [18, 10, 11]. In our setting, we assume the query filtering conditions encode relevance to the user.

There are two most common objective functions in content-based diversification problems: MAXMIN and MAXSUM. MAXMIN maximizes the minimum pairwise distances between selected items while MAXSUM maximizes the sum of pairwise distances. Both problems have been studied earlier in operational research as dispersion problems. The original motivation is to locate undesired facilities like nuclear reactors among the given nodes in a network. These two problems are proven to be NP-hard in the discrete and continuous cases [16, 40, 38]. We focus on MAXMIN as its results are more representative in many applications like geolocation based diversifications. Ravi et al. [38] prove that MAXMIN is NP-hard and has no polynomial-time relative approximation algorithm for general distance functions. But when the distance function obeys the triangle inequality, a greedy heuristic, or GMM, results in a $1/2$-approximation algorithm, and no polynomial algorithm can achieve a better performance guarantee unless P=NP.

However, researchers cannot directly migrate this greedy $1/2$-approximation algorithm to a database because it is too expensive. The time complexity of this algorithm is $O(kn)$ where n is the number of items in a query result. A database can have a million items satisfying a query in a data exploration or product search scenario. So $O(kn)$ can be very large even when k is as small as ten. Its bottleneck is the scan of $O(n)$ items. Yu et al. [45] have developed a similar approach which starts with k items and swaps better items with them greedily. Carbonell and Goldstein [7] iteratively select items with maximal marginal relevance. Vieira et al. [42] merge more scores and apply randomization in the greedy algorithm. Khan et al. [25] classify the above techniques as "process-first-diversify-next", which are expensive because they may scan or sort $O(n)$ items in the worst case. Our index-based approach avoids this issue to offer greater efficiency.

Many researchers use indexes or buffers to improve the efficiency for diversification. Vee et al. [41] work on categorical distance and use an index to probe diverse items. Qin et al. [37] consider binary distance. So they reduce the problem to weighted independent set and solve it with the help of the graph structure. In contrast, we are dealing with more general distance functions. Some researchers reuse selected items to further shorten query evaluation time. Several papers [33, 14, 15] work on extracting diverse items from a continuous data stream. While new items arrive, the user continuously queries the stream including both new and old items, so they can mix processed old items with new items as a result. Some other papers [23, 24, 26] assume correlation between consecutive queries during data exploration. So they can also reuse previously returned items. Our approach does not make such assumptions. We support any range queries efficiently with a quality guarantee, making our approach applicable to more general scenarios. If range queries do overlap, we can adapt our query evaluation algorithm by maintaining an intermediate cover tree as a buffer of candidates to guarantee approximation ratio and efficiency. Several papers [21, 8] are based on the idea of "core-sets", where they partition the data, compute a small diverse subset for each partition, and combine the subsets to answer a query. They have two drawbacks in answering a range query in our problem setting. First, their $1/3$ approximation ratio is worse than ours. Second, their query time complexity and insertion/deletion complexity are worse than ours. Having more partitions leads to greater query complexity, but having fewer partitions leads to greater insertion/deletion complexity. These two complexities cannot be both better than ours due to the linear scan embedded in the algorithm. Drosou and Pitoura [13] also use an index, but they solve a variant of the diversification problem where the distance between selected items only needs to be greater than a given threshold. Khan and Sharaf [25] prune items according to the sorted partial distance. But they focus on MAXSUM and the performance guarantee is unclear. We focus on MAXMIN and our algorithm has a formal approximation bound.

Our approach uses cover trees [5], which were originally designed for nearest neighbor search. Some other early data structures like ball tree [36], metric skip list [22], and navigating net [27] also have similar features. We use a cover tree as our Diversity Index because of its simplicity and good runtime performance in practice, but our approach is not necessarily limited to this data structure.

The study of range query dates back to 1970s. Researchers propose k-d tree [2], quad tree [17], B+ tree [9], VA-file [43], range tree [31, 3, 44, 29], and so on. We use range tree because of its good time complexity [44, 29, 32]. The range tree in our approach can be replaced by any other range query index if necessary.

8. CONCLUSIONS

In this paper, we study the problem of query result diversification. We propose a novel index structure, the RC-Index, and algorithms to significantly reduce the number of items we extract to answer a range diversification query. Compared to state-of-the-art algorithms which are linear or quadratic, our query time is sub-linear with respect to the number of items that satisfy the query. Moreover, our framework guarantees an approximation ratio for the diversity score, tunable through two parameters. When the parameters are large, the approximation ratio approaches $1/2$, which is the best possible ratio of a polynomial algorithm unless P=NP. Our experiments, using both synthetic data and real data, show that the quality of our result is close to (or even better than) the `Greedy` algorithm whose approximation ratio is $1/2$. In future work, we plan further study on generalized algorithms for continuous relevance scores, index selection, and distributed RC-Indexes.

9. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[3] J. L. Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5):244–251, 1979.

[4] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[5] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[6] J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131 – 151, 1999.

[7] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, pages 335–336, 1998.

[8] M. Ceccarello, A. Pietracaprina, G. Pucci, and E. Upfal. Mapreduce and streaming algorithms for diversity maximization in metric spaces of bounded doubling dimension. *Proc. VLDB Endow.*, 10(5):469–480, 2017.

[9] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[10] T. Deng and W. Fan. On the complexity of query result diversification. *Proc. VLDB Endow.*, 6(8):577–588, 2013.

[11] T. Deng and W. Fan. On the complexity of query result diversification. *ACM Trans. Database Syst.*, 39(2):15:1–15:46, 2014.

[12] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Rec.*, 39(1):41–47, 2010.

[13] M. Drosou and E. Pitoura. Disc diversity: Result diversification based on dissimilarity and coverage. *Proc. VLDB Endow.*, 6(1):13–24, 2012.

[14] M. Drosou and E. Pitoura. Dynamic diversification of continuous data. In *EDBT*, pages 216–227, 2012.

[15] M. Drosou and E. Pitoura. Diverse set selection over dynamic data. *IEEE Transactions on Knowledge and Data Engineering*, 26(5):1102–1116, 2014.

[16] E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48 – 60, 1990.

[17] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, Mar. 1974.

[18] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[19] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[20] R. Huerta, T. Mosqueiro, J. Fonollosa, N. F. Rulkov, and I. Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157:169 – 176, 2016.

[21] P. Indyk, S. Mahabadi, M. Mahdian, and V. S. Mirrokni. Composable core-sets for diversity and coverage maximization. In *PODS*, pages 100–108, 2014.

[22] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC*, pages 741–750, 2002.

[23] H. A. Khan, M. Drosou, and M. A. Sharaf. Dos: An efficient scheme for the diversification of multiple search results. In *SSDBM*, pages 40:1–40:4, 2013.

[24] H. A. Khan, M. Drosou, and M. A. Sharaf. Scalable diversification of multiple search results. In *CIKM*, pages 775–780, 2013.

[25] H. A. Khan and M. A. Sharaf. Progressive diversification for column-based data exploration platforms. In *ICDE*, pages 327–338, 2015.

[26] H. A. Khan, M. A. Sharaf, and A. Albarrak. Divide: Efficient diversification for interactive data exploration. In *SSDBM*, pages 15:1–15:12, 2014.

[27] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *SODA*, pages 798–807, 2004.

[28] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inf.*, 9(1):23–29, Mar. 1977.

[29] D. T. Lee and C. K. Wong. Quintary trees: A file structure for multidimensional datbase sytems. *ACM Trans. Database Syst.*, 5(3):339–353, 1980.

[30] M. Lichman. UCI machine learning repository, 2013.

[31] G. S. Lueker. A data structure for orthogonal range queries. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 28–34, 1978.

[32] G. S. Lueker and D. E. Willard. A data structure for dynamic range queries. *Information Processing Letters*, 15(5):209 – 213, 1982.

[33] E. Minack, W. Siberski, and W. Nejdl. Incremental diversification for very large sets: A streaming-based approach. In *SIGIR*, pages 585–594, 2011.

[34] S. Moro, P. Cortez, and P. Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22 – 31, 2014.

[35] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[36] S. M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[37] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *Proc. VLDB Endow.*, 5(11):1124–1135, July 2012.

[38] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

[39] R. L. T. Santos, C. Macdonald, and I. Ounis. Search result diversification. *Found. Trends Inf. Retr.*, 9(1):1–90, Mar. 2015.

[40] A. Tamir. Obnoxious facility location on graphs. *SIAM J. Discret. Math.*, 4(4):550–567, Sept. 1991.

[41] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. A. Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[42] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[43] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

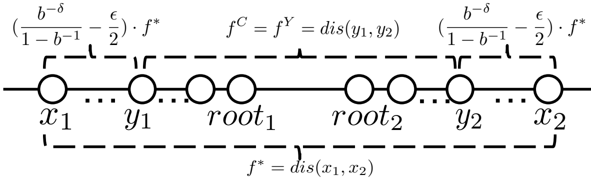


Figure 21: The bound in Theorem 1 is tight in the worst case.

Algorithm 7: RC-Index Batch Construction

Input: Set of items X; Dimensionality d.
Output: RC-Index RC.

1 $RC.root \Leftarrow$ `Construct`$(1, X.sortByDimension(1))$
2 **return** RC
3 **Function** `Construct`$(nowD, sortedX)$
4   **if** $|sortedX| = 0$ **then**
5   **return** NULL
6   $node.left \Leftarrow$ `Construct`$(nowD, firstHalf(sortedX))$
7   $node.right \Leftarrow$ `Construct`$(nowD, secondHalf(sortedX))$
8   **if** $nowD < d$ **then**
9     $nextX \Leftarrow sortedX.sortByDimension(nowD+1)$
10     $node.next \Leftarrow$ `Construct`$(nowD+1, nextX)$
11   $node.CT \Leftarrow$ construct cover tree on $sortedX$
12   **return** $node$

Algorithm 6: Cover Tree Insertion

Input: Cover Tree CT; New item x
Output: Cover Tree CT with x inserted

1 Raise the level of $CT.root$ until it covers x
2 $x.parent \Leftarrow CT.root$
3 $x.level \Leftarrow CT.root.level - 1$
4 $factor \Leftarrow CT.b/(CT.b-1)$
5 $conflict \Leftarrow \{CT.root\}$
6 $level \Leftarrow CT.root.level$
7 $levelD \Leftarrow (CT.b)^{level}$
8 **while** $conflict \neq \emptyset$ **do**
9   **foreach** $v \in conflict$ **do**
10   **if** $dist(x,v) < factor \cdot levelD$ **then**
11   **if** $dist(x,v) \leq levelD$ **then**
12         $x.parent \Leftarrow v$
13         $x.level \Leftarrow level - 1$
14   **else**
15       $conflict \Leftarrow conflict - \{v\}$
16   $level \Leftarrow level - 1$
17   $levelD \Leftarrow levelD/CT.b$
18   $conflict \Leftarrow \{v | v.parent \in conflict \wedge v.level \geq level\}$

[44] D. E. Willard. *Predicate-Oriented Database Search Algorithms*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1978.

[45] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: Diversification in recommender systems. In *EDBT*, pages 368–378, 2009.

[46] K. Zheng, H. Wang, Z. Qi, J. Li, and H. Gao. A survey of query result diversification. *Knowl. Inf. Syst.*, 51(1):1–36, 2017.

APPENDIX

A. COVER TREE OPERATIONS

Our cover tree insertion, deletion, and querying algorithms are slightly different from the original one in the cover tree paper [5]. We must support general base distance $b > 1$ for our approach while the original algorithms only works when $b = 2$. The major difference is that we need to compute an upper bound of the radius of a node. Suppose an item v is at level ℓ_v. Its direct children is within a ball of radius b^{ℓ_v}. Its "grandchildren" is within a ball of radius $b^{\ell_v} + b^{\ell_v - 1}$ and so on. The limit of the radius is $\lim_{\ell\to\infty} b^{\ell_v}(1-b^{-\ell})/(1-b^{-1}) = b^{\ell_v} \cdot b/(b-1)$. So we precompute $factor = b/(b-1)$ as the upper bound factor for all items at different levels. An item x may conflict with an descendant of item v if $dist(x,v) < factor \cdot b^{\ell_v}$. Algorithm 6 shows the insertion procedure. Deletion and querying algorithms are very similar.

B. PROOFS OF RC-Index PERFORMANCE

Approximation ratio

We prove Theorem 2:

PROOF. We prove it by providing a worst case example (Figure 21). In this case, $k = 2$ and the distance function is Euclidean. All items are on a straight line. Items from x_1 to $root_1$ belong to one cover tree rooted at $root_1$. Items from $root_2$ to x_2 belong to another cover tree rooted at $root_2$.

As illustrated, there are finite number of items between x_1 and y_1 leading to $dist(x_1, y_1) = (\frac{b^{-\delta}}{1-b^{-1}} - \frac{\varepsilon}{2}) \cdot f^*$ obeying Lemma 1. There are also δ items between y_1 and $root_1$, each of which is at a different level. The other cover tree on the right is symmetric.

So the extracted X^C will be all items between y_1 and y_2. The optimal diversity score of X^C is therefore $f^C = dist(y_1, y_2) = (\frac{b-1-2b^{1-\delta}}{b-1} + \varepsilon) \cdot f^*$. □

Query time complexity

The proof of Theorem 4 is as follows:

PROOF. The time complexity is the sum of the complexities of the Query Module (Algorithm 3 and 2) and the Diversification Module (Algorithm 4). The complexity of the Query Module is in Corollary 1. The complexity of the Diversification Module is $O(k \cdot |X^C|)$, which equals $O(k^2 \gamma^{4(\delta+1)} \log^d n)$ because $|X^C| = O(k\gamma^{4(\delta+1)} \log^d n)$ based on Lemma 2.

So the worst-case time complexity is dominated by Algorithm 4, $O(k^2 \gamma^{4(\delta+1)} \log^d n)$. □

Batch construction

Proof of Theorem 5:

PROOF. We prove it through induction.

- When $d = 1$, we build a 1-dimensional range tree, each node of which is a cover tree. We first sort all items into an array in $O(n \log n)$ time so that we can identify the items of each inner node in linear time [4]. Then we use divide and conquer algorithm to build the RC-Index. At each node, we find the median using the sorted array, divide the items into two children, build RC-Index for each child, and finally construct a cover tree for the current node. We know that building a cover tree on n items takes $O(\gamma^6 n \log n)$ time. So we have the following recurrence relation:

$$t(n) = 2 \cdot t(n/2) + O(\gamma^6 n \log n)$$

According to the master theorem, $t(n) = O(\gamma^6 n \log^2 n)$. The overall time complexity $T_1(n)$ including sorting is still $T_1(n) = O(n \log n) + t(n) = O(\gamma^6 n \log^2 n)$.

[4] This is faster than recursively applying the $O(n)$ selection algorithm in practice.

• Suppose Theorem 5 holds when $d = m$, so $T_m(n) = O(\gamma^6 n \log^{m+1} n)$. Now we prove it also holds when $d = m+1$.

We construct a $(m+1)$-dimensional RC-Index by recursively splitting the first dimension and building an m-dimensional RC-Index for each node. Similarly, we also sort the items by the first dimension with $O(n\log n)$ time. Then we divide and conquer to build each child. Finally, we construct a cover tree for the current node. So we have the recurrence relation:

$$\begin{aligned} t(n) &= 2\cdot t(n/2) + T_m(n) + O(\gamma^6 n \log n) \\ &= 2\cdot t(n/2) + O(\gamma^6 n \log^{m+1} n) + O(\gamma^6 n \log n) \\ &= 2\cdot t(n/2) + O(\gamma^6 n \log^{m+1} n) \end{aligned}$$

According to the master theorem, $t(n) = O(\gamma^6 n \log^{m+2} n)$. So the overall complexity is $T_{m+1}(n) = O(n\log n) + t(n) = O(\gamma^6 n \log^{m+2} n)$. □

Insertion and deletion

Given the imbalance score

$$IM(tree) = \sum_{v_i \in tree} im(v_i) rank(n_i) \log^{d_i+1} rank(v_i)$$

we have:

LEMMA 3. *One insertion/deletion increases $IM(tree)$ by $O(d\cdot\log^{d+2} n)$.*

PROOF. For each node n_i, an insertion/deletion can increase the imbalance factor $im(v_i)$ by at most $O(1/rank(v_i))$. So $im(v_i)rank(v_i)$ always increase by $O(1)$.

In the first dimension where $d_i = d$, we visit a path of $O(\log n)$ nodes. The sum of imbalance factor increments is thus $\sum_{\text{path}} O(1)\cdot \log^{d_i+1} rank(v_i) = \sum_{\text{path}} O(1) \log^{d+1} rank(v_i) \le \sum_{\text{path}} O(1) \log^{d+1} n = O(\log^{d+2} n)$. In the second dimension where $d_i = d-1$, we visit $O(\log^2 n)$ nodes for all subtrees. The sum of imbalance factor increments is also $O(\log^{d+2} n)$. We have similar results for all dimensions.

So one insertion/deletion increases $IM(tree)$ by $O(d\cdot\log^{d+2} n)$ for all dimensions. □

Here is the proof of Theorem 6:

PROOF. A pure insertion/deletion without batch construction visits $O(\log^d n)$ nodes and perform insertion/deletion on the corresponding cover trees. So the complexity is $O(\log^d n)\cdot O(\gamma^6 \log n) = O(\gamma^6 \log^{d+1} n)$.

Now we analyze the cost of batch construction. Let the imbalance score of a tree be

$$IM(tree) = \sum_{v_i \in tree} im(v_i) rank(v_i) \log^{d_i+1} rank(v_i)$$

where v_i are the nodes and d_i is the node's dimension. A node in the first dimension has $d_i = d$ and a node in the last dimension has $d_i = 1$.

Suppose we perform n operations each of which can be an insertion or deletion. So the tree has at most n items. Through an easy induction, we can prove that one insertion/deletion can increase $IM(tree)$ by $O(d\cdot\log^{d+2} n)$ (Lemma 3). So after n operations, $IM(tree)$ is at most $IM^+ = O(n\cdot d\cdot \log^{d+2} n)$.

In addition, each insertion/deletion can result in several batch constructions of subtrees. Assume u constructions occur during the process. Each construction reduces the imbalance factor im of a node from $(0,\beta)\cup(1-\beta,1)$ to $[1/3,2/3]$. In other words, it decreases im by at least $(1/3-\beta) = O(1)$. Formally, suppose every construction occurs at v_j in dimension d_j, it decreases im by at least $O(1)\cdot rank(v_j)\cdot \log^{d_j+1} rank(v_j)$. So all u constructions together decreases $IM(tree)$ by $IM^- = \sum_{j=1}^{u} O(1)\cdot rank(v_j)\cdot \log^{d_j+1} rank(v_j)$.

Initially, IM_0 is zero. After n operations, $IM_n = IM^+ - IM^-$ is positive. So $IM^- < IM^+$.

Based on Theorem 5, the cost of all constructions is $cost = \sum_{j=1}^{u} O(\gamma^6 rank(v_j)\cdot \log^{d_j+1} rank(v_j))$. So $cost = O(\gamma^6\cdot IM^-) < O(\gamma^6\cdot IM^+) = O(\gamma^6\cdot n\cdot d\cdot \log^{d+2} n)$. So the amortized construction complexity of each operation is $O(\gamma^6 d \log^{d+2} n)$, dominating the pure insertion/deletion cost $O(\gamma^6 \log^{d+1} n)$. Therefore the amortized complexity of each operation is $O(\gamma^6 d \log^{d+2} n)$. □

Index space complexity

We prove Theorem 7 through induction:

PROOF. • When $d = 1$, each node of the range tree is a cover tree. The root has n items, each of its two children has $n/2$ items, and so on. The space complexity of a cover tree is $O(n)$, so the 1-dimensional RC-Index takes $P_1(n) = O(n\log n)$ space.

• Suppose the space complexity is $O(n\log^m n)$ when $d = m$, we now prove the space is $O(n\log^{m+1} n)$ when $d = m+1$.

Remember we build $(m+1)$-dimensional RC-Index by splitting the first dimension and building an m-dimensional RC-Index for each node. So we have the following recurrence relation:

$$\begin{aligned} P_{m+1}(n) &= 2\cdot P_{m+1}(n/2) + P_m(n) + O(n) \\ &= 2\cdot P_{m+1}(n/2) + O(n\log^m n) + O(n) \\ &= 2\cdot P_{m+1}(n/2) + O(n\log^m n) \end{aligned}$$

So $P_{m+1}(n) = O(n\log^{m+1} n)$ based on the master theorem. □

C. INDEX SELECTION

Now we prove Theorem 8:

PROOF. Recall how we traverse a range tree to answer a query q. Initially, q corresponds to a single search at the root of the range tree. If q applies a filter on the first dimension, we transform the root to $O(\log n)$ subordinate searches. However, if q applies no filter on the first dimension, we simply goes from the root to its child in the next dimension without branching new searches. Similar procedure happens in all dimensions. We still end up with multiple subordinate searches and extract items from the corresponding cover trees. Obviously, the approximate ratio in Theorem 3 still holds. So we can use RC to answer query q as long as $\mathbb{A}_q \subseteq \mathbb{A}_{RC}$. □

Here is the proof of Theorem 9.

PROOF. As we explain in the proof of Theorem 8, every time we apply a filter on a dimension, we branch $O(\log n)$ subordinate searches. So our algorithm ends up with $(\log^{d_q} n)$ subordinate searches in the range tree in $O((d-d_q)\log^{d_q} n)$ time where $(d-d_q)$ implies we go directly from one node to its child *next* in the next dimension without branching new searches. Finally our algorithm extracts items from $O(\log^{d_q} n)$ cover trees to diversify them in $O(k^2\gamma^{4(\delta+1)}\log^{d_q} n)$ time. So the time complexity is $O\big((d-d_q)\log^{d_q} n + k^2\gamma^{4(\delta+1)}\log^{d_q} n\big)$. Since $(d-d_q) \le d = O(k^2)$, the overall query time is dominated by $O(k^2\gamma^{4(\delta+1)}\log^{d_q} n)$. □

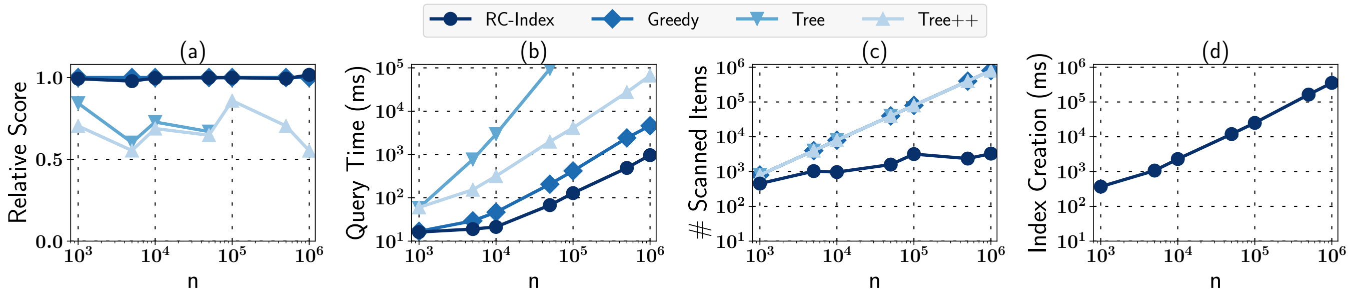


Figure 22: Synthetic normally-distributed data with varying size n: RC-Index is nearly an order of magnitude faster than the state of the art while ensuring a high quality result.

D. MAXSUM

We show how to extend our framework to support MAXSUM in this section.

We build the same RC-Index on the items. Then we apply the same Algorithm 3 and 2 to extract candidates. Finally, we can change the greedy Algorithm 4 for our diversification module to a very similar greedy algorithm in [38] with $1/4$ approximation ratio to support MAXSUM. We prove the approximation ratio of this pipeline is $\frac{b-1-2b^{1-\delta}}{4(b-1)}$.

Assume the optimal solution is $S^* = \{x_1, x_2, ..., x_k\}$ with diversity score f^*. The candidate items we extract is X^C. Suppose we construct the same $Y = \{y_1, y_2, ..., y_k\}$ on X^C with diversity score f^Y as we do for Theorem 1. Let $K = \frac{k(k-1)}{2}$ where $k \geq 2$ for non-trivial cases. We have the following lemma and theorem.

LEMMA 4. $dist(x_i, y_i) \leq \frac{b^{-\delta}}{1-b^{-1}} \frac{1}{K} \cdot f^*$.

PROOF. Similarly, $dist(x_i, y_i)$ has two cases:

• Case 1: When the tree CT has no more than k items, since $y_i = x_i$, $dist(x_i, y_i) = 0 \leq \frac{b^{-\delta}}{1-b^{-1}} \frac{1}{K} \cdot f^*$.

• Case 2: When the tree CT has more than k items, any k items at Level ℓ_k also exist at Level $(\ell_k - \delta)$ due to **Nesting** and should be extracted. They have already formed a solution with diversity score b^{ℓ_k} because of **Separation**. In addition, as f^* is the optimal diversity score of all items, $b^{\ell_k} K \leq f^*$.

Since y_i is the ancestor of x_i and y_i is at Level $(\ell_k - \delta)$, according to **Covering**,

$$dist(x_i, y_i) < \sum_{\ell=-\infty}^{\ell_k-\delta} b^{\ell} = \frac{b^{\ell_k-\delta}}{1-b^{-1}}$$

So $dist(x_i, y_i) < \frac{b^{\ell_k-\delta}}{1-b^{-1}} = \frac{b^{-\delta}}{1-b^{-1}} \cdot b^{\ell_k} \leq \frac{b^{-\delta}}{1-b^{-1}} \cdot \frac{1}{K} f^*$ □

THEOREM 10. *Let the optimal diversity score on X be f^*, and the optimal diversity score on extracted X^C be f^C, $f^C \geq \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$.*

PROOF. According to triangle inequality and symmetry (Definition 1), for any x_i and the corresponding y_i:

$$dist(x_i, x_j) \leq dist(x_i, y_i) + dist(y_i, y_j) + dist(x_j, y_j)$$

equivalent to

$$dist(y_i, y_j) \geq dist(x_i, x_j) - dist(x_i, y_i) - dist(x_j, y_j)$$

Since f^* is the diversity score of $\{x_1, ..., x_k\}$, $\sum_{1 \leq i < j \leq k} dist(x_i, x_j) = f^*$. So for any distinct y_i and y_j:

$$\begin{aligned}
\sum_{1 \leq i < j \leq k} dist(y_i, y_j) \geq & \sum_{1 \leq i < j \leq k} \big(dist(x_i, x_j) - dist(x_i, y_i) - dist(x_j, y_j)\big) \\
= & \sum_{1 \leq i < j \leq k} dist(x_i, x_j) \\
& - \sum_{1 \leq i < j \leq k} \big(dist(x_i, y_i) + dist(x_j, y_j)\big) \\
\geq & f^* - \sum_{1 \leq i < j \leq k} \left(\frac{b^{-\delta}}{1-b^{-1}} \frac{1}{K} \cdot f^* + \frac{b^{-\delta}}{1-b^{-1}} \frac{1}{K} \cdot f^* \right) \\
= & f^* - K \left(2 \cdot \frac{b^{-\delta}}{1-b^{-1}} \frac{1}{K} \cdot f^* \right) \\
= & (1 - \frac{2b^{-\delta}}{1-b^{-1}}) \cdot f^* \\
= & \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*
\end{aligned}$$

Since $f^C \geq f^Y \geq dist(y_i, y_j)$, $f^C \geq \frac{b-1-2b^{1-\delta}}{b-1} \cdot f^*$. □

After extracting the candidates, we apply a $1/4$-approximation algorithm for MAXSUM in [38]. So the overall approximation ratio is $\frac{b-1-2b^{1-\delta}}{4(b-1)}$.

E. DISCUSSION

One natural generalization of the problem is when range queries are replaced by continuous relevance scores. Then the problem becomes a multi-objective optimization problem whose object is to simultaneously maximize relevance and diversity. Specifically, assume the relevance score $rel(x, q)$ is part of the λ-weighted objective function [10]:

$$\begin{aligned}
f_{\min}(S, dist, rel) = & \lambda \cdot \min_{x_i, x_j \in S \wedge x_i \neq x_j} \{dist(x_i, x_j)\} \\
& + (1-\lambda) \cdot \min_{x \in S} \{rel(x, q)\}
\end{aligned}$$

where $S \subseteq q(X)$ and $|S| = k$.

The intuitive way to handle relevance score $rel(x, q)$ is to prioritize items with higher $rel(x, q)$ when we build an RC-Index: make high-score items be the parent to low-score items whenever possible. However, the approximation ratio needs to be carefully considered in future work.

F. MORE EXPERIMENTAL RESULTS

We vary n and experiment with data whose attributes follow normal distributions $\mathcal{N}(0, 1)$. As Figure 22 illustrates, RC-Index still outperforms the competitors.