The VLDB Journal
https://doi.org/10.1007/s00778-022-00770-0

REGULAR PAPER

A generic framework for efficient computation of top-k diverse results

Md Mouinul Islam[1] · Mahsa Asadi[1] · Sihem Amer-Yahia[2] · Senjuti Basu Roy[1]

Received: 18 January 2022 / Revised: 25 October 2022 / Accepted: 31 October 2022
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

Abstract
Result diversification is extensively studied in the context of search, recommendation, and data exploration. There are numerous algorithms that return top-k results that are both diverse and relevant. These algorithms typically have computational loops *that compare the pairwise diversity of records* to decide which ones to retain. We propose an access primitive **DivGetBatch()** that replaces repeated *pairwise comparisons of diversity scores of records by pairwise comparisons of "aggregate" diversity scores of a group of records*, thereby improving the running time of these algorithms while preserving the same results. We integrate the access primitive inside three representative diversity algorithms and prove that the augmented algorithms leveraging the access primitive preserve original results. We analyze the worst and expected case running times of these algorithms. We propose a computational framework to design this access primitive that has a pre-computed index structure **I-tree** that is agnostic to the specific details of diversity algorithms. We develop principled solutions to construct and maintain **I-tree**. Our experiments on multiple large real-world datasets corroborate our theoretical findings, while ensuring up to a 24× speedup.

Keywords Diversification · Top-k algorithms · Query processing

1 Introduction

Diversity has a wide variety of applications in search, recommendation [1,2,20,35,42,43] and data exploration. The goal of diversification algorithms is to return results that are relevant as well as cover user intent. In the data management community, returning top-k diverse results of a query has been extensively studied, and there exists many seminal works [14,23,49] that propose objective functions and efficient algorithms to achieve a trade-off between relevance and diversity.

The original implementation of many representative algorithms, such as, GMM [23], MMR [23], SWAP [49] that do not make any assumptions on the nature of the diversity functions, is iterative in nature and makes the decision of updating the top-k set by making a greedy choice based on the current top-k set and the remaining records that are not yet in top-k. These representative algorithms go through the cumbersome step of pairwise diversity computation of records between and across these two sets even to make a single update in the top-k set. Indeed, for a large database containing N records, this repetitive computation is expensive $\mathcal{O}(N)$, since typically $k << N$. We are also aware of a handful of existing works [21,32] that are specifically designed on geometric space and avoid this repetitive computation. However, to the best of our knowledge, most of the existing works assume this expensive computation to be necessary, when diversity is designed for arbitrary non-metric functions or even studied in general metric space. Contrarily, our effort here is to reduce that computation without making any explicit assumptions about the diversity function, that is, considering diversity functions to be fully arbitrary or even non-metric.

Our first contribution lies in identifying one major computational bottleneck in existing popular diversification algorithms and how to accelerate that process (Sect. 2.1). In Sect. 2.2, *we identify the basic ingredients of developing* **DivGetBatch()** *as an access primitive such that it remains*

✉ Senjuti Basu Roy
senjutib@njit.edu

Md Mouinul Islam
mi257@njit.edu

Mahsa Asadi
ma2266@njit.edu

Sihem Amer-Yahia
sihem.amer-yahia@univ-grenoble-alpes.fr

[1] New Jersey Institute of Technology, New Jersey, USA

[2] CNRS, Universite Grenoble Alpes, Grenoble, France

Published online: 28 November 2022

agnostic to any specific underlying diversity or distance computation function. This primitive is also guaranteed to produce identical top-k results as of the original diversity algorithms. The fundamental idea is to make the comparison go over a group of records, as opposed to record pairs, thereby accelerating the computation. In other words, the large number of N records are to be grouped in a small number of C nodes and some higher level diversity aggregates are to be maintained between the nodes. Toward that, we develop a generic computation framework that builds an index **I-tree** offline and maintains two other auxiliary data-structures (*MinsimMatrixNode* and *MaxsimMatrixNode*) that are highly generic in nature and suitable to handle updates. Indeed, the design of **I-tree** is rather simple and may appear to share resemblance with existing indexing techniques (Sect. 7 contains detailed discussion and empirical evaluation toward that). Our primary contribution lies in proposing a simple enough indexing technique that could be easily designed using off-the-shelf popular record partitioning algorithms, such as, K-Means [25], but study how to make it generic enough to work on a variety of diversification algorithms over arbitrary diversification functions. In fact, existing popular indexing techniques, such as *K-B-D-tree* [38], *kd-tree* [10], M-Tree [15], Ball-Tree [29], *R-tree* [24], assume that coordinate information of the records is available and used to create data structures to answer a large spectrum of distance queries, where distance may be based on Euclidean, cosine similarity, or general L_p norms. *However,* **I-tree** *assumes the records to be atomic and the diversity function to be arbitrary (refer to Sect. 8 for further comparison).*

Our second contribution is to develop query processing algorithms for *MMR*, *GMM*, and *SWAP* [14,23,49] using **DivGetBatch()** (Sects. 3, 4, 5). Fundamentally, we have rewired the original algorithms to run over pairs of groups of records as opposed to pairs of records to save up processing time. *We make nontrivial theoretical claims and proofs on the exactness and the running time of the augmented algorithms in expectation (assuming uniform data and query distributions) and in the worst case*. As an example, we prove that augmented $SWAP$ (*Aug-SWAP*) takes $\mathcal{O}(N/C * k * \log k + N)$ time in expectation compared to $\mathcal{O}(N * k * \log k)$ time of the original algorithm. It is easy to notice that augmented $SWAP$ is guaranteed to run faster than the original algorithm, as $Max(N/C * k * \log k, N)$ (C is the number of groups) is smaller than $N * k * \log k$. The summary of the complexity results is presented in Tables 1 and 2 .

Our third contribution is developing principled solutions for creating and maintaining **I-tree** (Sect. 6). **I-tree** is a complete m-ary tree [16] with height l. There exists many ways to build **I-tree** (e.g., hierarchical graph partitioning or clustering could be used). We identify that the main computational bottleneck of **I-tree** under batch updates lies in updating *MinsimMatrixNode* and *MaxsimMatrixNode*. Therefore, we formalize the index maintenance problem as an optimization problem, with the goal of minimizing the number of updates in these data structures. We present an integer programming-based exact solution **OPTMn** for that, and a greedy heuristic **GrMn** that is highly scalable in nature.

Our final contribution is experimental (Sect. 7). We use large real-world datasets, one large publicly available synthetic dataset to show that the augmented algorithms return results identical to their originals, while ensuring between a 3× to 24× speedup on large datasets. We study the effects of different parameters empirically and provide guidance for appropriate design choice. We empirically present exhaustive results to pre-process and maintain **I-tree**. Our empirical results corroborate our theoretical analyses.

Moreover, we compare the proposed index **I-tree** with a set of existing indexing structure, such as, M-Tree [15], KD-Tree [10], and Ball-Tree [29]. These latter trees are primarily designed for the Euclidean space. Our experimental results unanimously select **I-tree** as the winner. The augmented algorithms implemented using **I-tree** are at least 18× faster in query processing and as much as 170× faster for certain configuration. **I-tree** achieves more than 1.5× speedup during the index construction and at times it is more than 20× faster w.r.t. the baselines.

To summarize, we make the following contributions:

- We develop **DivGetBatch()**, an access primitive and show how to integrate it inside popular diversity algorithms to save up running time (Sects. 3, 4, 5). We present in depth theoretical analyses of the augmented algorithms.
- We propose a computational framework to support **DivGetBatch()**(Sect. 6). The framework consists of a pre-computed index **I-tree** and a query processing step. We also present non-trivial solutions to maintain **I-tree** under dynamic updates.
- We run an extensive experimentation that demonstrates the effectiveness of building and maintaining **I-tree** and **DivGetBatch()** and corroborates our theoretical claims (Sect. 7).

2 Background and approach

This section is organized in two parts. In Sect. 2.1, we present the background of the studied problem and define it. In Sect. 2.2, we present the fundamental ideas of our approach.

2.1 Motivating example and problem definition

The basic principle of existing diversification algorithms, such as *MMR*, *GMM*, and *SWAP*, is either to incrementally

Table 1 Technical results for running time analysis w.r.t. $|CandR|$

Algorithm	Variant	Expected time w.r.t $\|CandR\|$
MMR	Original	$\mathcal{O}(N * k^2)$
	Augmented	$\mathcal{O}(C * k^2 + N + \sum_{i=1}^{k} \|CandR_i\| * k$
GMM	Original	$\mathcal{O}(N * k)$
	Augmented	$\mathcal{O}(C * k + \sum_{i=1}^{k} \|CandR_i\|)$
$SWAP$	Original	$\mathcal{O}(N * k * \log k)$
	Augmented	$\mathcal{O}(N + \sum_{i=1}^{N} \frac{\|CandR_i\|}{N} * (C + k * \log k))$

Table 2 Technical results for running time analysis w.r.t. C, m, l

Algorithm	Variant	Expected time w.r.t C	Expected time w.r.t m and l
MMR	Original	$\mathcal{O}(N * k^2)$	$\mathcal{O}(N * k^2)$
	Augmented	$\mathcal{O}((N/C + C) * k^2 + N)$	$\mathcal{O}((N/m^l + m^l) * k^2 + N)$
GMM	Original	$\mathcal{O}(N * k)$	$\mathcal{O}(N * k)$
	Augmented	$\mathcal{O}(N/C + C) * k)$	$\mathcal{O}(N/m^l + m^l) * k)$
$SWAP$	Original	$\mathcal{O}(N * k * \log k)$	$\mathcal{O}(N * k * \log k)$
	Augmented	$\mathcal{O}(N/C * k * \log k + N)$	$\mathcal{O}(N/m^l * k * \log k + N)$

Index	Activity	Time	Space	Time	Space
I-tree	Construction	$\mathcal{O}(N * C^2 * t + N^2)$	$\mathcal{O}(C^2)$	$\mathcal{O}(N * m^{2l} * t + N^2)$	$\mathcal{O}(m^{2l})$
	Maintenance	$\mathcal{O}(N * \|Y\|)$	$\mathcal{O}(C^2)$	$\mathcal{O}(N * \|Y\|)$	$\mathcal{O}(m^{2l})$

build a top-k set of diverse results or to greedily replace records in a top-k list to find the most diverse ones. In both cases, the leading cost directly depends on the number of pairwise record comparisons. Imagine a toy database D containing $N = 10$ records. Since the records are considered atomic, Table 4 shows a record-record similarity matrix, $simMatrixRecord$, normalized between [0-1] for our example. Diversity between r_i, r_j is simply $1 - sim(r_i, r_j)$. Given a query Q, in order to produce $k = 2$ results, an algorithm such as MMR [14] first assigns all 10 records in D to a potential candidate set R. Then, it iterates over all 10 records once to find the best record in terms of MR score (based on diversity and relevance) and adds that to the result set S and discards that from R. It repeats the same process once more to produce the resulting set $S = \{r_{10}, r_8\}$. In particular, there is a repeated pairwise computation of the following kind:

```
While k ≤ 2:
      rec ← R[1]
       For   i = 2; i <= |R|; i++
          if   MR(Q, R[i], S) ≥ MR(Q, rec, S)
            rec ← R[i]
       EndFor
      S ← S⋃rec, R ← R − rec
      k ← k + 1
EndWhile
```

Problem Definition 1 Develop an access primitive **DivGetBatch()** and integrate it inside existing popular diversity algorithms. **DivGetBatch()** satisfies the following three criteria.

- It guarantees identical top-k results as that of the original algorithms.
- It is generic, i.e., it works for any diversity functions - diversity being metric or not.
- When integrated inside existing algorithms, it accelerates the computation and returns the results faster.

The proposed primitive simplifies the aforementioned implementation as follows - instead of iterating over the entire R set (which is $\mathcal{O}(N)$), it returns potentially a much smaller set of records $CandR$, from which the result set S would be updated.

```
CandR ← DivGetBatch(R, Q, S)
 While k ≤ 2:
    rec ← Max(MR(CandR, Q, S))
    S ← S⋃rec, CandR ← CandR − rec
    k ← k + 1
 EndWhile
```

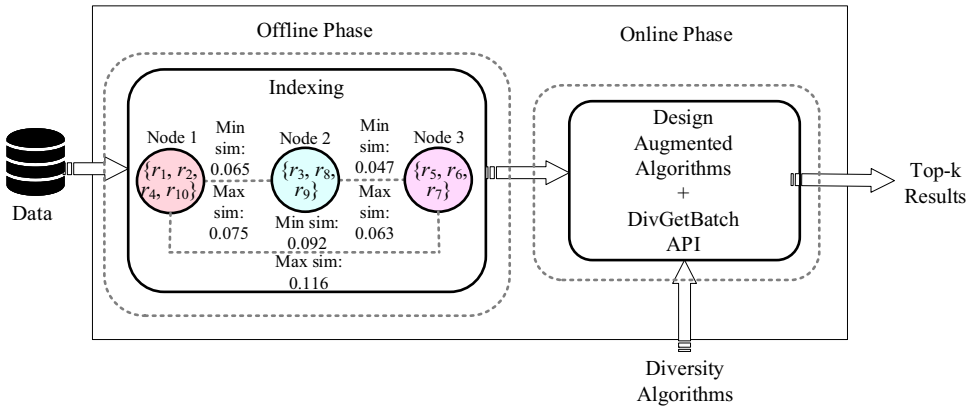


Fig. 1 Proposed computational framework

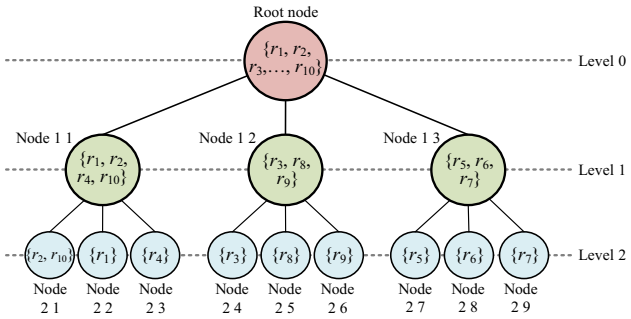


Fig. 2 **I-tree**

2.2 Approach

DivGetBatch() is designed by developing a computational framework, described in Fig. 1. The basic idea is to store "higher level aggregates"', such as *minimum and maximum diversity scores of a group of records instead of keeping individual pairwise diversity scores between the records*. We formally define the minimum and maximum diversity scores as *bounds* in later sections. As an example, if the same set of records are grouped in three nodes, as shown inside the indexing box of Fig. 1 and the maximum and minimum diversity scores are preserved between them, $node_2$ and $node_3$ can be discarded in the first iteration of processing of *MMR* pruning 6 out of the 10 records and returning only $\{r_1, r_2, r_4, r_{10}\}$ in R. This indeed leads to a significant speedup.

2.2.1 Offline vs. Online.

In this work, we assume that both data and query follow uniform distributions. A keen reader may notice that to accelerate diversity computation using **I-tree**, one has to "group" records and maintain some higher level aggregates between them. Grouping a large database of N records is time-consuming, as that would require partitioning them based on pairwise diversity. Indeed, this process of grouping must happen once and offline.

Precisely because of this, we resort to pre-process the records to group them and develop index **I-tree**, and use that later for processing diversity queries. This is the offline computation of the proposed framework.

Just like **DivGetBatch()**, **I-tree** is a general purpose complete tree like structure and could be designed in more than one way. It needs to satisfy three properties.

- **I-tree** has m arity and l height or levels (user inputs).
- Two highly important auxiliary data structures maintain similarity bounds between the nodes in **I-tree**: *MinsimMatrixNode* and *MaxsimMatrixNode* for maintaining minimum and maximum similarity bounds [1].
- For three nodes n, n', and n_j in **I-tree**, if n is a parent of n', and n_j is part of a different subtree and at the same level as n, the following relationship holds: Min $sim(n, n') \geq Minsim(n, n_j)$, and Max $sim(n, n') \geq Maxsim(n, n_j)$, (basically nodes that are part of the same subtree have higher min and max similarity bounds compared to the nodes that are not).

The indexing algorithm *BuildTree* (Algorithm 5) partitions (refer to the Subroutine *Partition*) the records. It also maintains additional data structures that contain similarity scores between nodes for efficient query processing. An example of a two-level index tree is shown in Fig. 2. At the first level, *BuildTree* creates a root node containing all N records and m children of the root node. From the point of abstraction, it is not important at this stage to describe how the data are partitioned. Basically, the goal is to keep similar records together while separating non-similar ones. There are multiple off-the-shelf techniques such as clustering and graph partitioning to carry out this task.

In our implementation, we use the popular k-means algorithm [25] for partitioning. The algorithm repeats the

[1] Diversity between a pair of records is simply $1 - $ *similarity* between them.

Table 3 Notations & interpretations

Notations	
D	Database containing N records
S	Result set
Z	Set of nodes that contain S
R	Remaining records in the dataset
Q	Query
k	Number of records in resulting set
m, l	Arity & Total number of levels in the **I-tree**
C	Number of nodes in the **I-tree**
$CandR$	Candidate record set returned by API
Y	A batch of new records to be updated in **I-tree**

partitioning procedure until it reaches l levels. We refer to Sect. 6 for further details (Table 3).

Next, we present the generic recipe of using **DivGetBatch()** online or during the query processing time.

2.2.2 Generic online algorithm using DivGetBatch()

The inputs to **DivGetBatch()** is **I-tree**, query Q, current candidate set of answers S, remaining records R, as well as the algorithm specific objective function f. The output is $CandR$, a set of candidate records that cannot be eliminated and require further processing by the original algorithm. **DivGetBatch()** explores **I-tree** level by level during query time and exploits two of its higher-level constructs: a. *Calculate-Bounds:* it computes similarity bounds [2] between Q and the nodes in **I-tree** based on a specific algorithm and objective function f. In particular, it computes an upper and a lower bound of diversity scores of the node. The goal is to decide if it is beneficial to go inside the node and explore the subtree under it. b. *Skip-Nodes:* based on the previous decision, the algorithm either skips the node and its entire subtree or explores the node.

Algorithm 1 shows the pseudo-code of the **DivGetBatch()** API.

3 *MMR* query processing with DivGetBatch()

The first algorithm we study is MMR [14] algorithm. We describe the original version of the algorithm and our augmented version and provide theoretical analysis on how our augmented version outperforms the original one.

[2] Please note diversity could be easily calculated from similarity bounds.

Algorithm 1 Generic **DivGetBatch()** API

```
Inputs: I-tree, S, R, Q, f
Outputs: CandR: remaining eligible set of records for next iteration
for y = 1 to l do
    for n in I-tree [y].nodes do
        uB, lB ← Calculate-Bounds(I-tree, n, y, f, S, Q, R)
        uBs ← ⋃ uB, lBs ← ⋃ lB
    end for
    M ← Skip-Nodes(I-tree, y, uBs, lBs)
    V ← { I-tree [y].nodes − M}
end for
CandR = {r | r ∈ n, n ∈ V}
return CandR
```

3.1 *MMR* algorithm

Maximal Marginal Relevance (MMR) algorithm is a seminal work on result diversification [14]. MMR is based on Marginal Relevance (MR) score (Eq. 1) that it maximizes in each iteration. Given a query, MR introduces a λ coefficient to strike a balance between the relevance score, computed between the records and the query, and the diversity score, computed between the records themselves.

MMR is greedy in nature that grows the size of the top-k set by adding records one by one in the top-k set by considering the relevance of the record and diversity with the previously selected records, using the formula below:

$$MMR(r) \leftarrow argmax_{r \in R \setminus S} MR(r),$$
$$MR(r) \leftarrow \lambda sim(r, Q) - (1 - \lambda) max_{r_j \in S} sim(r, r_j), \quad (1)$$

where Q is the query, S is the previously selected items, R is the remaining records in the dataset, r is a candidate record from R, and r_j is another record from S. λ is a tunable parameter. The time-consuming part of the algorithm lies in computing the MR score for each $r \in \{R \setminus S\}$ and returning the one with the highest MR score.

The MMR algorithm takes $\mathcal{O}(|R| \times |S|)$, when we add one new record to set S, demonstrating that it has an order of $N \times k$. The algorithm repeats k times and produces top-k results.

3.2 Aug-MMR algorithm

Aug-MMR algorithm is designed to circumvent this aforementioned time consuming computation by leveraging *DivGetBatch()*. The general idea is to return a small subset of records, as opposed to all $|R|$ records (which is $\mathcal{O}(N)$) in each iteration, thereby saving computation. The rest of the algorithm is identical to its original version and is presented in Algorithm 2.

We now describe subroutine 2, how **DivGetBatch()** exactly works in *Aug-MMR*. Inputs to **DivGetBatch()** are *I-tree*, S, R, Q, and f (i.e., objective function of MMR).

Table 4 Similarity matrix for records

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	Q
r_1	1.000	0.979	0.065	0.989	0.105	0.110	0.092	0.066	0.068	0.969	0.187
r_2	0.979	1.000	0.070	0.992	0.107	0.112	0.092	0.071	0.074	0.999	0.190
r_3	0.065	0.070	1.000	0.068	0.057	0.061	0.048	0.982	0.986	0.071	0.052
r_4	0.989	0.992	0.068	1.000	0.111	0.116	0.096	0.069	0.072	0.986	0.180
r_5	0.105	0.107	0.057	0.111	1.000	0.976	0.880	0.055	0.058	0.106	0.039
r_6	0.110	0.112	0.061	0.116	0.976	1.000	0.783	0.059	0.063	0.112	0.041
r_7	0.092	0.092	0.048	0.096	0.880	0.783	1.000	0.047	0.049	0.092	0.036
r_8	0.066	0.071	0.982	0.069	0.055	0.059	0.047	1.000	0.986	0.072	0.054
r_9	0.068	0.074	0.986	0.072	0.058	0.063	0.049	0.986	1.000	0.075	0.054
r_{10}	0.969	0.999	0.071	0.986	0.106	0.112	0.092	0.072	0.075	1.000	0.191

The output is $CandR$, the candidate set of records for which MR scores are to be computed to retain the best record. Based on Algorithm 1, we now describe the specifics of two higher-level constructs for *Aug-MMR*.

Calculate-Bounds This function leverages $MinsimMatrix\text{-}Node$ and $MaxsimMatrixNode$ to calculate lower ($lBMR$) and upper bounds ($uBMR$), respectively. The bounds essentially represent the score of a node based on f (Eq. 1) and mathematically can be expressed as follows:

$$lBMR_{node} \leftarrow \lambda Minsim(node, Q) - max_{node' \in Z}(1-\lambda)Maxsim(node, node'), \quad (2)$$

$$uBMR_{node} \leftarrow \lambda Maxsim(node, Q) - min_{node' \in Z}(1-\lambda)Minsim(node, node'), \quad (3)$$

where Z is the set of nodes that contain S,

$Minsim(node, Q)$ and $Maxsim(node, Q)$ are the minimum and the maximum similarity between any records in $node$ and Q, respectively, and $Minsim(node, node')$ and $Maxsim(node, node')$ are the minimum and the maximum similarity between any two records in $node$ and $node'$, respectively. Since $lBMR$ is the smallest score of $node$, it is calculated by taking the minimum of sim score in the first part of the equation and subtracting that from the maximum of sim score in the second part. Contrarily, $uBMR$ refers to the maximum MR score of $node$ (Eq. 3) and can be calculated by reversing the min and max of the (Eq. 2).

Algorithm 2 Aug-MMR

Inputs: I-tree, D, MMR, Q, k
Outputs: S: final top-k result set.
1: $R \leftarrow D, S = \phi$
2: **for** $t = 1$ **to** k **do**
3: $CandR \leftarrow$ **DivGetBatch(I-tree**, R, S, Q, MMR)
4: $S = \{S \bigcup MMR(r)_{r \in CandR}\}$
5: **end for**
6: **return** S

Skip-Nodes The argument of node skipping is simple—if the $uBMR$ score of a node is not larger than the $lBMR$ of another node, then the former node and its entire subtree could be pruned. The records from the remaining nodes form the $CandR$ set.

$$CandR \leftarrow \{N - \{r \in I - tree.n \mid uBMR_n < \max_{\forall n'}(lBMR_{n'})\}\} \quad (4)$$

This is done by finding the maximum value of $lBMR_{n'}$ of all nodes and then, discard ones with $uBMR$ less than it. *Running Example:* A step by step calculation of **DivGetBatch()** is shown in Table 5. The maximum and minimum similarity between $node_1$ and query Q is 0.180 and 0.191. In first iteration of Calculate-Bounds, lower bound of MR of $node_1$ which is $lBMR_{node_1} = 0.8*0.180-(1-0.8)*0 = 0.144$, and upper bound of MR of $node_1$, $uBMR_{node_1} = 0.8*0.191 - (1-0.8)*0 = 0.153$. Similarly, $lBMR_{node_2}$, $uBMR_{node_2}$, $lBMR_{node_3}$, and $uBMR_{node_3}$ are -0.047, 0.044, 0.029, and 0.033, respectively. In Skip-Nodes , the maximum of all $lBMR$s is found 0.144 which is $lBMR_{node_1}$.

$uBMR_{node_2}$ and $uBMR_{node_3}$ are smaller than $lBMR_{node_1}$. Therefore, $node_2$ and $node_3$ are discarded from further calculation in iteration 1. Records of $node_1$ $\{r_1, r_2, r_4, r_{10}\}$ are returned by **DivGetBatch()** to **Aug-MMR** algorithm. **Aug-MMR** performs calculation similar to original MMR on $\{r_1, r_2, r_4, r_{10}\}$ which results in $S = \{r_{10}\}$. Likewise, the maximum and minimum similarity between $node_1$ and $node_1$ are 0.969 and 1.0. In the second iteration of Calculate-Bounds, $lBMR_{node_1} = 0.8*0.180 - (1-0.8)*0.969 = -0.050$ and $uBMR_{node_1} = 0.8*0.191 - (1-0.8)*1.0 = -0.047$. Similarity, $lBMR_{node_2}$, $uBMR_{node_2}$, $lBMR_{node_3}$, and $uBMR_{node_3}$ are 0.028, 0.029, 0.010, and 0.009, respectively. In Skip-Nodes , the maximum of all $lBMR$s is $lBMR_{node_2} = 0.028$. $uBMR_{node_1}$ and $uBMR_{node_3}$ are smaller than $lBMR_{node_2}$. Thus, $node_1$ and $node_3$ are discarded from further calculation in iteration 2. Records of $node_2$ $\{r_3, r_8, r_9\}$ are returned by **DivGetBatch()** to **Aug-**

Table 5 First Two Iterations of **DivGetBatch()** in **Aug-MMR**

Functions	Nodes	Bounds	Iteration 1	Iteration 2
Calculate-Bounds	$node_1$	$lBMR$	$0.8 * 0.180 - (1 - 0.8) * 0 = 0.144$	-0.050
		$uBMR$	$0.8 * 0.191 - (1 - 0.8) * 0 = 0.153$	-0.047
	$node_2$	$lBMR$	$0.8 * 0.0191 - (1 - 0.8) * 0 = 0.0152$	0.028
		$uBMR$	$0.8 * 0.054 - (1 - 0.8) * 0 = 0.044$	0.029
	$node_3$	$lBMR$	$0.8 * 0.036 - (1 - 0.8) * 0 = 0.029$	0.010
		$uBMR$	$0.8 * 0.041 - (1 - 0.8) * 0 = 0.033$	0.009
Skip-Nodes			$lBMR$ array: 0.144, 0.041, 0.029	$lBMR$ array:
			$uBMR$ array: 0.153, 0.044, 0.033	-0.050, 0.028, 0.010
			$uBMR$ Array:	$node_2$, $node_3$ are skipped.
			$CandR = \{r_1, r_2, r_4, r_{10}\}$.	-0.047, 0.029 , 0.009
			$MMR(r_1, r_2, r_4, r_{10}) \leftarrow r_{10}$	$node_1$, $node_3$ are skipped.
			Number of records discarded is 6	$CandR = \{r_3, r_8, r_9\}$
				$MMR(r_3, r_8, r_9) \leftarrow r_8$
				top-2 set = $\{r_{10}, r_8\}$

MMR algorithm. **Aug-MMR** performs calculation similar to original MMR on $\{r_3, r_8, r_9\}$ which results in $S = \{r_{10}, r_8\}$

3.2.1 Aug-MMR algorithm proofs

Claim 1 **Aug-MMR** *returns identical top-k results as that of original* MMR.

Proof The proof is constructed using one helper lemma and one observation: Lemma 1 proves that **DivGetBatch()** never prunes a record that is part of the original top-k; Observation 1 shows that once the control comes back from **DivGetBatch()**, **Aug-MMR** works exactly as the original MMR in each iteration. Combining these lemma and observation, **Aug-MMR** returns identical top-k results as that of the original MMR. □

Lemma 1 **DivGetBatch()** *never prunes a record that is part of the original top-k.*

Proof As part of this proof, we first prove that Skip-Nodes *never discards* the record which has the highest MR score in that iteration.

Recall Property 1 and note that for every two nodes n and n' in the same subtree, if n is a parent of n', then n contains all records in n', thereby having larger $uBMR$ and $lBMR$ values. Therefore, if a node n is skipped, any child of n is also safe to be skipped.

We use helper Lemma 2 to prove that the actual MR score of any record in a node $node$ is bounded between $uBMR_{node}$ and $lBMR_{node}$. Let us assume, the next desired record $r_d \in node_d$ produces maximum MR value among all $R \setminus S$ records. MR_{r_d} is greater than $minMR_{node}$ for $\forall node$. Using Eq. 6:

$$\begin{aligned} MR_{r_d} &\geq max_{node \in \textbf{I-tree}[l].nodes} minMR_{node} \\ &\geq max_{node \in \textbf{I-tree}[l].nodes} (lBMR_{node}), \end{aligned}$$

Using Eq. 6, $MR_{r_d} = MaxMR_{node_d} \leq uBMR_{node_d}$. As a result,

$$\begin{aligned} uBMR_{node_d} &\geq MR_{r_d} \\ &\geq max_{node \in \textbf{I-tree}[l].nodes} (lBMR_{node}). \end{aligned} \tag{5}$$

According to Eqs. 5 and 4, $node_d$ will not be discarded, and all records inside $node_d$ including r_d will be returned by **DivGetBatch()** or send to the next level for further processing. This logic extends for all the iterations. Therefore, **DivGetBatch()** never prunes a record that is part of the original top-k. □

Lemma 2 *MR score of any record* $r \in node$ *(say* MR_r*) is bounded by upper and lower bound* $uBMR_{node}$ *and* $lBMR_{node}$*, respectively. That is,*

$$lBMR_{node} \leq MR_{r \in node} \leq uBMR_{node}. \tag{6}$$

Proof We will first prove that maximum relevance value (say MR_{rmax}) of any record (say $r_{max} \in node$) is less than equal to $uBMR_{node}$.

Where, MR_{rmax} can be expressed as:

$$\begin{aligned} MR_{rmax} = \lambda sim(r_{max}, Q) - (1 - \lambda) max_{r_j \in S} \\ \times sim(r_{max}, r_j)]. \end{aligned} \tag{7}$$

First part of Eq. 7 is always less than equal to first part of Eq. 3. That is:

$$\lambda sim(r_{max}, Q) \leq \lambda max_{r_i \in node} sim(r_i, Q) = \lambda Maxsim(node, Q), \quad (8)$$

Next, we show that second part of Eq. 7 is always greater than second part of Eq. 3.

Let us assume; $r_w \in S$ produces max value for the second part of Eq. 7. That second part can be rewritten as $(1-\lambda)sim(r_{max_{node}}, r_w)$. Let us assume, $r_w \in node_w$ where $node_w \in Z$. For any $node' \in Z$, we can write:

$$(1-\lambda)sim(r_{max}, r_w) \geq (1-\lambda)min_{r_i \in node, r_j \in node'} sim(r_i, r_j) \geq min_{node' \in Z}(1-\lambda)Minsim(node, node'), \quad (9)$$

From these two inequalities 8 and 9, we can conclude $MR_{rmax} \leq uBMR_{node}$ or, $MR_{r \in node} \leq uBMR_{node}$.

Similarly, the lower bound $lBMR_{node}$ can be shown as follows: $lBMR_{node} \leq minMR_{node}$.

Thus, any record in $node$ is certain to have MR value in between $uBMR_{node}$ and $lBMR_{node}$. □

Observation 1 *Once the control comes back from* **DivGetBatch()**, **Aug-MMR** *works exactly as the original MMR in each iteration.*

Aug-MMR has identical MR score calculation and MMR selection as that of the original MMR.

Claim 2 **Aug-MMR** *requires* $\mathcal{O}((N/C + C) * k^2 + N)$ *time in expectation.*

Proof In the original MMR algorithm, each iteration for finding one record takes $\mathcal{O}(N * k)$ times. For k iterations, the overall running time is therefore $\mathcal{O}(N * k^2)$. The running time of **Aug-MMR** does not need to go over all N records in each iteration. Instead, it relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree** and takes $\mathcal{O}(C * k)$ time. **DivGetBatch()** has to compute two subroutines:

$Calculate - Bound$ and $Skip - Nodes$. To compute these two functions, it takes $\mathcal{O}(N)$ time. Therefore, the overall running time is $\mathcal{O}(C*k^2+N)$, where C is the total number of nodes.

Part 2. Running time of the rest of computation: The rest of the computation depends on the size of $CandR$. Let us assume, **DivGetBatch()** returns $|CandR_i|$ records in the i-th iteration. Accordingly, we have:

$$T_{\textbf{Aug-MMR}} = \mathcal{O}\left(C * k^2 + N + \sum_{i=1}^{k} |CandR_i| * k\right).$$

The expected case analysis basically delves deeper into the analysis of *Part 2* and studies the expected running time considering different size of $CandR_i$ and its corresponding probability.

Let us assume, in iteration i, the $|CandR_i|$ records touch x number of nodes in **I-tree**. Indeed, x_i is the number of nodes with $|CandR_i|$ records in **I-tree**, that the augmented algorithms have to access during the query processing. Let us also assume node n_i contains v_i records. We start the proof assuming there is only one level in **I-tree** (i.e., $l = 1$) and then, generalize it later on. If $l = 1$, the expected running time of Part 2 calculation of **Aug-MMR** in the i-th iteration is:

$$E = \mathcal{O}\left(\sum_{i=1}^{C} prob(x_i) \times \text{computation cost}_{\textbf{Aug-MMR}}(x_i)\right).$$

Now, probability of returning x nodes = $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected.

We assume that both data and query follow uniform distributions; thereby, each node has an equal probability of getting selected or skipped. The probability of choosing a node is $1/C$. Therefore, the probability of not getting selected is $(1 - 1/C)$.

The size of the returned record set, i.e., $|CandR|$, if $x = i$ nodes are accessed:

$$\begin{aligned}|CandR|_i &= (1/C)^i * (1-1/C)^{C-i} * [(v_1 + v_2 + \ldots + v_i) \\ &\quad + (v_1 + v_3 + \ldots + v_{i+1}) + (v_2 + v_3 + \ldots + v_{i+1}) \\ &\quad + (v_3 + v_4 + \ldots + v_{i+2}) + \ldots] \\ &= (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} \\ &\quad * (v_1 + v_2 + \ldots + v_C) \\ &= (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} * N.\end{aligned}$$

Therefore, the overall expected cost of Part 2 is:

$$\begin{aligned}|CandR| &= N * \sum_{i=1}^{C} (1/C)^i * (1-1/C)^{C-i} * \binom{C-1}{i-1} \\ &= N * (1/C)/(1-1/C) * \sum_{i=1}^{C} (1/C)^{i-1} * \\ &\quad (1-1/C)^{C-(i-1)} * \binom{C-1}{i-1}. \\ &\quad Let\ j = i-1: \\ &= N * (1/C)/(1-1/C) * \sum_{j=0}^{C-1} (1/C)^j *\end{aligned}$$

$$(1-1/C)^{C-j} * \binom{C-1}{j}$$
$$= N * (1/C)/(1-1/C) * (1-1/C) *$$
$$\sum_{j=0}^{C-1} (1/C)^{j} * (1-1/C)^{(C-1)-j} * \binom{C-1}{j}$$
$$= N * (1/C)/(1-1/C) *$$
$$(1-1/C) * (1/C+1-1/C)^{C-1} = N/C.$$

Expected running time of **Aug-MMR** algorithm considering both Part 1 and Part 2 computation is:

$$E_{\textbf{Aug-MMR}} = \mathcal{O}((N/C+C) * k^2 + N).$$

Now, consider the case when $l > 1$. Probability of selecting a node in first level is $1/m$, given m is the arity of **I-tree**. Probability of selecting a node in second level = probability of selecting that node out of m node in that branch * probability of selecting it's parent = $1/m^2$. Similarly, probability of selecting a node at leaf node is $1/m^l = 1/C$. Thus, in the general case, when $l > 1$, expected running time of **Aug-MMR** is $\mathcal{O}((N/C+C) * k^2 + N)$, which is same as before. □

Worst-case **Aug-MMR** . In the worst-case, all N records are returned by **DivGetBatch()** in each iteration, which makes $\sum_{i=1}^{k} |CandR_i| = N * k$. Thus, the worst-case running time is $\mathcal{O}((N+C) * k^2)$.

4 *GMM* query processing with DivGetBatch()

The second algorithm we study is GMM algorithm. We describe the original version of the algorithm and our augmented version and similar to the previous section. We also provide proofs on how our augmented version outperforms the original one.

4.1 *GMM* algorithm

The next algorithm we study is GMM [23] that tries to find a subset of k most diverse records among N records by maximizing the minimum pairwise distance. GMM does not require any external query. Based on the original design, the first two records in the result set S are provided in constant time by an *oracle*. Then, the algorithm iteratively goes through all records in R and finds a record whose minimum *diversity* (maximum similarity) with the previously selected records is the largest (smallest). It continues until $|S|$=k. The objective function is:

$$GMM(r) \leftarrow argmax_{r \in R \setminus S} min_{r_j \in S} Div(r, r_j), \tag{10}$$

where $Div(r, r_j)$ is the diversity score between record r and r_j. A keen reader may notice that GMM uses diversity (Div) in the objective function, whereas, in our study, we store similarity between records. Unless specified otherwise, $Div = 1 - sim$. The two similarity matrices, one that captures the similarity between every pair of records, and the other that captures that of between nodes, could be used to calculate Div.

4.2 Aug-GMM algorithm

Aug-GMM leverages the **DivGetBatch()** API to reduce the number of records to iterate on. Algorithm 3 describes the pseudo-code, where the **DivGetBatch()** returns a small subset of records $CandR$ which later on is fed to the original GMM algorithm to get the $nextBest$ record.

Calculate-Bounds This function keeps track of the upper and lower bounds of scores between nodes ($uBGMM$ and $lBGMM$, respectively) using the same principles as that of the original GMM objective function (Eq. 10).

$$lBGMM_{node} \leftarrow min_{node' \in Z} \min Div(node, node'), \tag{11}$$

$$uBGMM_{node} \leftarrow min_{node' \in Z} \max Div(node, node'), \tag{12}$$

where Z is the set of nodes containing S, min $Div(node, node')$ and max $Div(node, node')$ are the minimum and the maximum diversity scores between any two records in $node$ and $node'$, respectively. In Eq. 11, minimum of the minimum diversity over all nodes in Z ensures the lower bound of GMM, such that all records in $node$ will have equal or greater value than $lBGMM_{node}$. Conversely, in Eq. 12, minimum of the maximum diversity over all nodes in Z ensures the upper bounds, such that all records in $node$ will have equal or lower GMM value than $uBGMM_{node}$.

Skip-Nodes This function is identical to Skip-Nodes of MMR in principle. The skip-rationale of **Aug-GMM** is:

$$CandR \leftarrow \Big\{ N - \Big\{ r \in I - tree.n \mid uBGMM_n < \max_{\forall n'} (lGMM_{n'}) \Big\} \Big\} \tag{13}$$

Running Example Let us assume $k = 3$ and the first two records of S are arbitrarily chosen as r_1 and r_3. Initially, $S = \{r_1, r_3\}$. From Fig. 1, r_1 and r_3 are inside $node_1$ and $node_2$, respectively. Hence, $Z = \{node_1, node_2\}$. Node-Node diversity $Div(node, node')$ can be calculated using Div = 1 - Sim. $Div(node_3, node_1)$ = (0.884, 0.908) and $Div(node_3, node_2)$ = (0.937, 0.9530). By using Eqs. (11) and (12), $lBGMM_{node_3}$ = 0.884 (as min of min div)

and $uBGMM_{node_3}$ = 0.908 (as min of max div). Similarly, $lBGMM_{node_1}$, $uBGMM_{node_1}$, $lBGMM_{node_2}$, and $uBGMM_{node_2}$ are 0, 0.031, 0, and 0.018. $lBGMM_{node_3}$ (0.884) is greater than $uBGMM_{node_1}$ (0.031) and $uBGMM_{node_2}$ (0.018). Using Equ. 13, $node_1$ and $node_2$ can be discarded. Obtaining records from $node_3$, $candR = \{r_5, r_6, r_7\}$ is returned from *DivGetBatch()*. Finally, $GMM(r_5, r_6, r_7) = r_5$ is called and the result set $S = \{r_1, r_3, r_5\}$ is achieved.

4.2.1 Aug-GMM algorithm proofs

Claim 3 **Aug-GMM** *returns identical top-k results as that of original GMM.*

Proof Akin to MMR proof, this proof is also constructed using one helper lemma and one observation: Lemma 3 proves that **DivGetBatch()** never prunes a record that is part of the original top-k; Observation 2 shows that in each iteration, once the control comes back from **DivGetBatch()**, **Aug-GMM** works exactly as the original GMM. Combining these lemma and observation, **Aug-GMM** returns identical top-k results as that of the original GMM. □

Lemma 3 **DivGetBatch()** *never prunes a record that is part of the original top-k.*

Proof As part of this proof, we first prove that Skip-Nodes *never discards* the record which has the highest GMM score in that iteration.

We use helper Lemma 4 to prove that the actual GMM score of any record in a node $node$ is bounded between $uBGMM_{node}$ and $lBGMM_{node}$. The rest of the proof is identical to Lemma 1 of *Aug-MMR*. □

Lemma 4 *GMM score of any record $r \in node$ (say GMM_r) is bounded by upper and lower bound $uBGMM_{node}$ and $lBGMM_{node}$, respectively. That is,*

$$lBGMM_{node} \leq GMM_{r \in node} \leq uBGMM_{node}.$$

Proof Let us first consider $uBGMM_{node}$, by assuming $F(node, r_j) = max_{r_i \in node} Div(r_i, r_j)$, it can be re-written as:

$$uBGMM_{node} \leftarrow min_{node' \in Z}[max_{r_j \in node'} F(node, r_j)], \tag{14}$$

Let us assume, maximum GMM value produced by any record in $node$ is $maxGMM_{node}$. According to Equ. 10, $maxGMM_{node}$ is expressed as follows:

$$\begin{aligned} maxGMM_{node} &= max_{r_i \in node}[min_{r_j \in S} Div(r_i, r_j)], \\ &= min_{r_j \in S}[max_{r_i \in node} Div(r_i, r_j)], \\ &= min_{r_j \in S} F(node, r_j), \\ &\leq min_{node' \in Z}[max_{r_j \in node'} F(node, r_j)], \\ &= uBGMM_{node}, \text{ [using Eq. 14]}. \end{aligned}$$

Similarly, it can be proved that $minGMM_{node} \geq lBGMM_{node}$. □

Observation 2 *Once the control comes back from* **DivGetBatch()**, **Aug-GMM** *works exactly as the original GMM in each iteration.*

Aug-GMM does exactly same calculation as the original GMM does on a set of records as a result it will produce the same record as GMM does in a single iteration.

Claim 4 **Aug-GMM** *requires $\mathcal{O}(N/C+C)*k)$ time in expectation.*

Proof In the GMM algorithm, each iteration for finding one record takes $\mathcal{O}(N)$ times. For k iteration, the overall running time is $\mathcal{O}(N*k)$. Similar to *Aug-MMR*, **Aug-GMM** does not need to go over all N records in each iteration, instead relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree** and takes $\mathcal{O}(C)$ time. **DivGetBatch()** has to compute two subroutines:

$Calculate - Bound$ and $Skip - Nodes$. To compute these two functions, it takes $\mathcal{O}(C)$ time. Therefore, the overall running time is $\mathcal{O}(C * k)$, where C is the total number of nodes.

Part 2. Running time of the rest of computation: Similar to *Aug-MMR*, the rest of the computation depends on the size of $CandR$. Let us assume, **DivGetBatch()** returns $|CandR_i|$ records in the i-th iteration. Hence, we have:

$$T_{\textbf{Aug-GMM}} = \mathcal{O}(C * k + \sum_{i=1}^{k} |CandR_i|).$$

The expected case analysis basically delves deeper into the analysis of *Part 2* and studies the expected running time considering different size of $CandR_i$ and its corresponding probability. By performing similar calculation as that of **Aug-MMR** as shown before, the expected cost of **Aug-GMM** is:

$$E_{\textbf{Aug-GMM}} = \mathcal{O}((N/C + C) * k).$$

□

Worst-case **Aug-GMM** In the worst-case, all N records are returned by **DivGetBatch()** in each iteration, which makes $\sum_{i=1}^{k} |CandR_i| = N * k$. Then, the worst-case running time is: $\mathcal{O}((N + C) * k)$.

Algorithm 3 Aug-GMM

Inputs: **I-tree**, D, GMM, k
Output: S: final top-k result set
1: $S \leftarrow$ two records selected by an oracle
2: $R \leftarrow \{D - S\}$
3: **for** $t = 1$ **to** $k - 2$ **do**
4: $CandR \leftarrow$ **DivGetBatch(I-tree**, R, S, GMM)
5: $S = \{S \bigcup GMM(r)_{r \in CandR}\}$
6: **end for**
7: **return** S

5 *SWAP* Query Processing with DivGetBatch()

The last algorithm we study is $SWAP$ [49]. We describe the original version and our proposed augmented version. Similar to the previous sections, we provide theoretical analysis.

5.1 *SWAP* algorithm

SWAP [49] is a greedy algorithm that produces top-k results based on a given query Q and a tunable parameter that controls how much relevance could at most drop between any two records in the top-k results. The algorithm starts by sorting the records w.r.t. relevance and initializing the top-k result set S with the k-records with the highest relevance score with Q. It finds a *candidate record* from the current top-k set that has the smallest diversity contribution based on Eq. 15. Indeed, in each iteration, it attempts to swap one record from R \ S with the candidate record. It starts scanning the remaining sorted relevance list from the top. In every iteration, it attempts to swap one record from the current top-k set with another from sorted R if the latter record has a higher contribution to diversity while ensuring the threshold of relevance drop. The algorithm terminates when the relevance drop is below the threshold, or R is fully scanned.

$$Divcont(r_i, S) = \sum_{r_j \in S} Div(r_i, r_j). \tag{15}$$

5.2 Aug-SWAP algorithm

Aug-SWAP is identical to the *SWAP*, i.e., it scans the sorted relevance list R, after initializing the top-k set S. It calls the **DivGetBatch()** API to retrieve a smaller set of candidate records $CandR$. These $CandR$ records are eligible to be considered during the next swap. If a record in R is not in $CandR$, then it is skipped. The rest of the process is identical to the original *SWAP* algorithm. Algorithm 4 contains the pseudo-code.

Calculate-Bounds Once the records are sorted w.r.t. relevance score, the diversity computation becomes query independent, and only between the records. This function calculates the upper and lower bounds of diversity contribution of nodes by leveraging $MinsimMatrixNode$ and $MaxsimMatrixNode$ considering the set of nodes Z that contains S, as below:

$$uBSWAP_{node} \leftarrow \sum_{node' \in Z} \max Div(node, node'), \tag{16}$$

$$lBSWAP_{node} \leftarrow \sum_{node' \in Z} \min Div(node, node'), \tag{17}$$

where $\max Div(node, node')$ and $\min Div(node, node')$ are the max and the min diversity between $node$ and $node'$. Naturally, the maximum (minimum) diversity is the maximum (minimum) of node diversities between $node$ and the nodes in Z.

Skip-Nodes This function will then check if $uBSWAP_{node}$ is less than the diversity contribution of the candidate record (18); If the condition is true, it will prune the node and the entire subtree under it. In such a case, none of the records inside this node are eligible for swap because they will not increase the overall diversity of S. **DivGetBatch()** returns the records for all non-pruned nodes:

$$CandR \leftarrow \Big\{ N - \Big\{ r \in I - tree.n \mid uBSWAP_n < min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j) \Big\} \Big\} \tag{18}$$

Running Example: Lets say, $k = 2$, $UB = 0.9$, sorted $R = \{r_8, r_7, r_2, r_1, r_4, r_9, r_3, r_6, r_{10}\}$, and initial top-2 records selected as S=$\{r_8, r_7\}$. Using Eq. 15, $Divcont(r_7, S) = 0.953$ and the $candidate$ is r_7. From Fig. 1, $Z = \{node_2, node_3\}$. Using Eqs. (16), (17), and Fig. 1, if Div = 1 - sim, we have:
$uBSWAP_{node_1} = max Div(node_1, node_2) = 0.935$,
$lBSWAP_{node_1} = min Div(node_1, node_2) = 0.925$.
Then, Eq. 18 is applied and $node_1$ is discarded, $node_2$, $node_3$ are returned by *DivGetBatch()*, and $CandR = \{r_3, r_9, r_5, r_6\}$. Next record in the sorted list is r_2, which is not in $CandR$. As a result, r_2 will be skipped.

5.2.1 Aug-SWAP algorithm proofs

Claim 5 **Aug-SWAP** *returns identical top-k results as that of original* SWAP.

Proof This proof is constructed using one helper lemma and one observation. Lemma 5 proves that **DivGetBatch()** does not skip a record that has a higher diversity contribution than that of the candidate record. Observation 3 shows that once all records returned in $CandR$, **Aug-SWAP** is identical to $SWAP$. Combining these lemma and observation, **Aug-SWAP** returns identical top-k results as that of the original $SWAP$. □

Algorithm 4 Aug-SWAP

Inputs: I-tree, D, UB, k, $SWAP$
Output: S: final top-k result set.

1: $R \leftarrow$ Sort D on score;
2: $S \leftarrow$TOPKITEMS(R, k)
3: $candidate \leftarrow argmin_{r_i \in S} Equation\ 15$
4: $CandR \leftarrow R$
5: $pos \leftarrow k+1$
6: **while** $candidate.score$ - $R[pos].score < UB$ **do**
7: **if** $R[pos]$ **in** $CandR$ **then**
8: **if** $Divcont(R[pos], S) > Divcont(candidate, S)$ **then**
9: $S \leftarrow \{S - candidate \bigcup R[pos]\}$
10: $CandR \leftarrow$ **DIVGETBATCH(I-tree**, R, S, Q, $SWAP$)
11: $candidate \leftarrow argmin_{r_i \in S} Equation\ 15$
12: **end if**
13: **end if**
14: pos++
15: **end while**
16: **return** S

Lemma 5 **DivGetBatch()** *never prunes a record that is part of the original top-k.*

Proof As part of this proof, we first prove that in each iteration Skip-Nodes *never discards* a record which has the higher diversity contribution than that of the candidate record.

Let us assume, $r_{cand} \in S$ has lowest diversity contribution in S.

$$Divcont(r_{cand}, S) = min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j)\}$$
$$= min_{r_i \in S} Divcont(r_i, S).$$

We use helper Lemma 6 to prove that the actual $DivCont$ score of any record in a node $node$ is bounded between $uBSWAP_{node}$ and $lBSWAP_{node}$. Let us assume, $r_d \in node_d$ is a record inside $node$, therefore,

$$uBSWAP_{node_d} \geq Divcont(r_d, S)$$
$$\geq Divcont(r_{cand}, S)$$
$$= min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j),$$

as a result,

$$uBSWAP_{node_d} \geq min_{r_i \in S} \sum_{r_j \in S} Div(r_i, r_j). \quad (19)$$

From Eqs. 18 and 19, it is evident that $node_d$ containing r_d will not be skipped by Skip-Nodes . This logic extends to all the iterations Skip-Nodes calls. Hence the proof. □

Lemma 6 *Divcont score of any record $r \in node$ is bounded by upper and lower bound $uBSWAP_{node}$ and $lBSWAP_{node}$, respectively. That is,*

$$lBSWAP_{node} \leq Divcont(r, S)_{r \in node} \leq uBSWAP_{node} \quad (20)$$

Proof By replacing the value of max $Div(node, node')$, the upper bound can be written as:

$$uBSWAP_{node} \leftarrow \sum_{node' \in Z} max_{r_i \in node, r_j \in node'} Div(r_i, r_j). \quad (21)$$

For any record $r \in node$ and $r_j \in S$, $r_j \in node_d$ and $node_j \in Z$,

$$Div(r, r_j) \leq max_{r_i \in node} Div(r_i, r_j),$$

Or,

$$\sum_{r_j \in S} Div(r, r_j) \leq \sum_{node' \in Z} max_{r_i \in node, r_j \in node'} Div(r_i, r_j),$$

As a result, $Divcont(r, S) \leq uBSWAP_{node}$. Similarly, we can prove: $Divcont(r, S) \geq lBSWAP_{node}$. □

Observation 3 *Once the control comes back from* **DivGetBatch()**, **Aug-SWAP** *works exactly as the original $SWAP$ does in each iteration.*

Aug-SWAP performs identical calculation of $SWAP$ on the records that are not pruned by *DivGetBatch()*.

Claim 6 **Aug-SWAP** *requires $\mathcal{O}(N/C * k * \log k + N)$ time in expectation.*

Proof In the original $SWAP$ algorithm, each iteration to select a new record to be swapped with the candidate record takes $\mathcal{O}(k * \log k)$ time. Therefore, for going over all records in R, it takes $\mathcal{O}(N * k * \log k)$. **Aug-SWAP** does not need to perform $\mathcal{O}(N * k * \log k)$, instead relies on **DivGetBatch()** to obtain a smaller set $CandR$ records.

Part 1. Running time of the API: A single iteration of **DivGetBatch()** needs to go over all the nodes in **I-tree**. **DivGetBatch()** has to compute two subroutines: $Calculate - Bound$ and $Skip - Nodes$. By updating only the most recent swapped records and using dynamic programming, the two subroutines' overall running time is $\mathcal{O}(C)$, where C is the total number of nodes. However, how many times the API gets called depends on the number of times the swap condition gets satisfied (recall lines 8-10 in **Aug-SWAP** algorithm).

Part 2. Running time of the rest of computation: The other major computation of this algorithm is the running time of a record be swapped, which is $\mathcal{O}(k * \log k)$ and $Divcont$ running time in the Algorithm 5 line 8, which is $\mathcal{O}(k)$. How many times $Divcont$ gets executed depends on Line 7 in the **Aug-SWAP** algorithm is satisfied. The number of times $SWAP$ gets executed depends on swap condition, which is Line 8 in the **Aug-SWAP** algorithm. Finally, the entire R

needs to be exhausted (as long as the bound drop threshold is satisfied), which takes $\mathcal{O}(N)$ time. As a result, we have:

$$\begin{aligned} T_{\textbf{Aug-SWAP}} = \mathcal{O}(&Number\ of\ times\ swap\ is\ satisfied \\ &* \textbf{DivGetBatch}()\ runtime + \\ &Number\ of\ times\ swap\ is \\ &satisfied * SWAP\ runtime + \\ &number\ of\ times\ line\ 7\ is\ satisfied * \\ &Divcont\ runtime + N). \end{aligned}$$

By considering running time of single $Divcont$, $SWAP$, and **DivGetBatch**() call, overall running time of **Aug-SWAP** becomes:

$$\begin{aligned} T_{\textbf{Aug-SWAP}} = \mathcal{O}(&Number\ of\ times\ swap\ is\ satisfied \\ &* C + Number\ of\ times\ swap\ is\ satisfied \\ &* k * \log k + number\ of\ times\ line\ 7 \\ &is\ satisfied * k + N). \\ = \mathcal{O}\Bigg(&\sum_{i=1}^{N} [probability\ of\ \ swap\ satisfied \\ &* C + probability\ of\ \ swap\ satisfied \\ &* k * \log k + probability\ of number\ of \\ &times\ line\ 7\ is\ satisfied * k] + N\Bigg) \end{aligned}$$

Expected size of $CandR$ is $\sum_{i=1}^{N} \frac{|CandR_i|}{N}$. Probability of line 7 satisfied = probability that $R[pos]$ is in $CandR$ = $\frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N}$. Without further information, the probability of a record getting swapped is 1/2 (same as not getting swapped). Probability of $SWAP$ = 1/2∗line 7 is satisfied = $1/2 * \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N}$. Expected running time (cost) of **Aug-SWAP** is:

$$\begin{aligned} E_{\textbf{Aug-SWAP}} &= \sum_{i=1}^{N} [1/2 * \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N} * (C + k * \log k) \\ &+ \frac{\sum_{i=1}^{N} \frac{|CandR_i|}{N}}{N} * k] + N \\ &= 1/2 * \sum_{i=1}^{N} \frac{|CandR_i|}{N} * (C + k * \log k) \\ &+ \sum_{i=1}^{N} \frac{|CandR_i|}{N} * k + N \\ &= \mathcal{O}(\sum_{i=1}^{N} \frac{|CandR_i|}{N} * (C + k * \log k) + N) \end{aligned}$$

First, we study the Part 2 computation having two costs associated with it, cost of $Divcont$ and cost that of $SWAP$. Based on Line 7 of Algorithm 5, if $CandR$ is large, it is likely to have $R[pos]$ inside it. In fact, if $CandR$ contains all R records, $R[pos]$ will always be there. For the purpose of illustration, let us assume, in the i-th iteration, $|CandR_i|$ records touch x number of nodes in **I-tree** and node n_i contains v_i records. Therefore, the probability that $R[pos]$ is in $CandR_i = \frac{\sum_{q=1}^{x} v_q}{N}$.

The expected running time of $SWAP$ in terms of C is: $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * probability of swap * cost of swap.

The probability of $x = i$ and $R[pos]$ is in $CandR_i$ is:

$$\begin{aligned} &= (1/C)^i * (1 - 1/C)^{C-i} * [(v_1/N + v_2/N + \cdots + v_i/N) \\ &+ (v_1/N + v_3/N + \cdots + v_i/N) + \ldots \\ &+ (v_{C-i}/N + \cdots + v_C/N)] \\ &= (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} * (\frac{v_1 + v_2 + \cdots + v_c}{N}). \\ &= (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1}. \end{aligned}$$

Therefore, the expected running time (cost) of $SWAP$ is,

$$\begin{aligned} E_{SWAP} = 1/2 * N * k * \log k * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \\ \binom{C-1}{i-1} = 1/2 * N/C * k * \log k. \end{aligned}$$

Expected running cost of $Divcont$ is $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * cost of $Divcont$. Therefore, the expected running time (cost) of $Divcont$ is:

$$\begin{aligned} E_{Divcont} &= N * k * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} \\ &= N/C * k. \end{aligned}$$

The expected cost of Part 2 becomes:

$$E_{Part_2} = 1/2 * N/C * k * \log k + N/C * k.$$

The expected running time (cost) of Part 1 is = $\binom{C}{x}$ * probability of x nodes getting selected * probability of $(C - x)$ nodes not getting selected * probability of $R[pos]$ is in $CandR_i$ * probability of swap * cost of **DivGetBatch**(). Using similar

calculation as above, expected cost of part 1 is:

$$E_{part_1} = 1/2 * N * \sum_{i=1}^{C} (1/C)^i * (1 - 1/C)^{C-i} * \binom{C-1}{i-1} * C = N/2.$$

Expected running time of **Aug-SWAP** algorithm considering both Part 1 and Part 2 computation is:

$$E_{\textbf{Aug-SWAP}} = 1/2 * N/C * k * \log k + N/C * k + N/2 + N = \mathcal{O}(N/C * k * \log k + N)$$

Now, consider the case when $l > 1$ for *Aug-SWAP*. Probability of selecting a node in first level is $1/m$, given m is the arity of **I-tree**. Probability of selecting a node in second level = probability of selecting that node out of m node in that branch * probability of selecting its parent = $1/m^2$. Similarly, probability of selecting a node at leaf node is $1/m^l = 1/C$. As the records are only returned from leaf nodes, the expected probability that $R[pos]$ is in $Cand R_i$ does not change for $l > 1$. The running time of **DivGetBatch()** $= \mathcal{O}(m^l) = \mathcal{O}(C)$ also stays same . The rest of the computation does not directly depend on l. As a result, expected running time of *Aug-SWAP* for $l > 1$ is same as before. □

Worst-case Aug-SWAP In the worst-case, none of the records are skipped, so the number of swap is $\mathcal{O}(N)$. Therefore, the worst-case running time is: $\mathcal{O}(N * C * k * \log k)$.

Our technical results are summarized in Tables 1 and 2 .

6 I-tree

The index is a hierarchical complete tree-like structure [28] that partitions D into multiple groups of records. Each node in **I-tree** consists of a group of similar records. The index structure maintains a higher level aggregate similarity between nodes [3]. **I-tree** is applicable not only to the studied three algorithms, but also to any content-based algorithm that is either based on replacing items in the top-k or building the top-k in an incremental fashion.

6.1 Index construction

The input to the indexing step is a $N \times N$ matrix, named *simMatrixRecord*. It represents the similarity scores between every pair of records, r_i and r_j, in the database and two additional parameters, l and m, which are the number of levels and arity of the tree, respectively. The output is a complete m-ary tree with l levels, referred to as *I-tree*.

Algorithm 5 Indexing Algorithm *BuildTree(node)*

Inputs: database D of N records, m: arity of the tree, l: number of levels,
Outputs: **I-tree**, *simMatrixNode*: node-node similarity matrix, *recordMap*: a mapping of all records and their belonging node id for each level.

1: $rootnode \leftarrow N$ records, $y = 0$
2: $nodelist[y] \leftarrow rootnode$
3: **while** $y \leq l$ **do**
4: **for** *node* **in** $nodelist[y]$ **do**
5: $cnodes \leftarrow$ *Partition(node, m)*
6: **I-tree** $[y][node]$.ADDCHILD(cnodes)
7: $w \leftarrow \bigcup cnodes$
8: *recordMap[y][r]* $\leftarrow$ *node id* containing record r in y
9: **end for**
10: *MinsimMatrixNode[y][i][j]* $\leftarrow$ Use Equation 23
11: *MaxsimMatrixNode[y][i][j]* $\leftarrow$ Use Equation 24
12: $nodelist[y] \leftarrow w$
13: $y \leftarrow y + 1$;
14: **end while**

The indexing algorithm *BuildTree* (Algorithm 5) partitions (refer to the Subroutine *Partition*) the records. It also maintains additional data structures that contain similarity scores between nodes for efficient query processing. An example of a two-level index tree is shown in Fig. 2. At the first level, *BuildTree* creates a root node containing all N records and m children of the root node. From the point of abstraction, it is not important at this stage to describe how the data are partitioned. Basically, the goal is to keep similar records together while separating non-similar ones. There are multiple off-the-shelf techniques such as clustering and graph partitioning to carry out this task.

In our implementation, we use the popular k-means algorithm [25] for partitioning. The algorithm repeats the partitioning procedure until it reaches l levels. Therefore, **I-tree** contains a total of C nodes such that:

$$C = \sum_{i=0}^{l} m^i = \frac{m^{l+1} - 1}{m - 1} = \mathcal{O}(m^l) \tag{22}$$

Inside *I-tree*, additional data structures are maintained:

a. A *recordMap* of size $N \times l$ that maps the id of a record with the id of its node in each level from $1 \ldots l$. b. *MinsimMatrixNode* and *MaxsimMatrixNode* that contain inter-node minimum and maximum similarities between any two nodes in the same level, respectively. Particularly, for two nodes n and n' in level y, *MinsimMatrixNode* and *MaxsimMatrixNode* contain:

$$MinsimMatrixNode[i, j] = Min_{r \in i, r' \in j} sim(r, r'), \tag{23}$$

[3] Diversity between a pair of records is simply $1 - similarity$ between them.

$$MaxsimMatrixNode[i, j] = Max_{r \in i, r' \in j} sim(r, r'), \tag{24}$$

where, $r \in n, r' \in n'$. Figure 1 contains these scores for 3 nodes of our running example.

6.2 Index maintenance

Even for a single insertion or deletion, **I-tree** requires the following two activities: a. insertion/deletion of that record from/into *I-tree*; b. updating *MinsimMatrixNode* and *MaxsimMatrixNode*, if these insertion/deletion require updating the minimum and maximum similarity scores between nodes. One can easily see that (a) could be achieved in a constant time when l =1 and $\mathcal{O}(l)$ when l greater than 1. However, a single insertion/deletion may require as many as $2 \times (C - 1)$ updates in these two matrices.

6.2.1 Batch update

We study how to maintain **I-tree** considering both insertions and deletions.

Batch Deletion Let us assume a batch of R records are to be deleted from *I-tree*. The process deletes these R records one by one and then checks how many entries in *MinsimMatrixNode* and *MaxsimMatrixNode* need update (if the deleted records contribute to these aggregate values, then that require updates in those two matrices, else not). The overall process takes $\mathcal{O}(|Y| \times C \times N)$ time.

Batch Insertion This problem is more complicated. If the records are inserted arbitrarily inside **I-tree**, then, each insertion may potentially cause a total of $2 \times (C-1)$ updates in the *MinsimMatrixNode* and *MaxsimMatrixNode* data structures. This is *the leading computational cost of batch insertion*. Moreover, when a batch of records are inserted, it is possible to have multiple records to get inserted inside the same node, and that should not be double-counted in the process. Finally, one needs to insert the records to those nodes, such that the aggregates stored in *MinsimMatrixNode* and *MaxsimMatrixNode* remain "tight" to enable effective pruning. These nuances are explored in formalizing the batch insertion problem.

Problem Definition 2 *(Batch Insert)* Let $MinsimMatrixNode[i, j]$ (similarly $MaxsimMatrixNode[i, j]$) denote the value after $|Y|$ insertions at the $[i, j]$-th entry at the *MinsimMatrixNode* (similarly $MaxsimMatrixNode$ matrix). Let $Minsim_{ij}$ and $Maxsim_{ij}$ be two binary variables, such that which $Minsim_{ij} = 1$ (similarly $Maxsim_{ij}$), if it requires an update after insertions, 0 otherwise. Our goal is to insert a batch of records Y such that, it minimizes $\sum_{i,j} Minsim_{ij} + \sum_{i,j} Maxsim_{ij}$, i.e., the total number of updates in these two matrices.

Algorithms We present an integer programming-based solution **OPTMn** for solving the batch insert problem. While **OPTMn** indeed produces the optimal solution, due to its exponential nature, it does not scale to a very large dataset considering a large number of insertions. As an alternative, we present **GrMn** a greedy heuristic algorithm which makes greedy choices and indirectly attempts to minimize the number of updates in *MinsimMatrixNode* and *MaxsimMatrixNode* matrices. The idea is to make a greedy decision by inserting each of the incoming records to that node which it is closest to (based on the underlying similarity measure) and then, check if that insertion requires any updates in *MinsimMatrixNode* and *MaxsimMatrixNode* matrices. The running time of this algorithm is $\mathcal{O}(|Y| \times N)$.

7 Experimental evaluation

Our experimental evaluations have three primary goals. First, we analyze if the augmented algorithms return identical results to their original counterparts using multiple large-scale datasets. Second, we examine the efficiency and scalability of the augmented algorithms and compare them with multiple baselines. Finally, we empirically study the cost of building and maintaining **I-tree**. For brevity, we present a subset of results that are representative.

Experimental setup All algorithms are implemented in Python 3.8. All experiments are conducted on a cluster server OSL machine with 32GB RAM memory, OS: Scientific Linux release 7.8 (Nitrogen), CPU: Intel(R) Xeon(R) CPU E3-1245 v6 @ 3.70GHz. Obtained results are the average of three separate runs. [4]

Diversity and similarity We use normalized Euclidean distance (dist) as diversity to validate our designed solutions in the geometric space, Cosine similarity [25] in general metric space. For non-metric distance, we use Movielens datasets and quantify the diversity between a pair of movies as the number of users who have rated either of these two movies but not both. We additionally use an arbitrary diversity function generated synthetically on Makeblobs dataset, such that it does not satisfy triangle inequality. Thus, diversity values are atomic for the last two cases and are not derived from the feature vectors. For all these cases, $sim = 1 - dist$.

Query selection In our experiments, queries are chosen randomly.

Performance measures We measure precision@k [25] for qualitative analysis. Efficiency of the proposed method is demonstrated with $|CandR|/N \times 100$, pruning = $1 - |CandR|/N \times 100$, as well as by presenting the running times of the algorithms in seconds and computing *speedup*

[4] The code and data could be found at https://anonymous.4open.science/r/divGetBatch-54BE/README.md

Table 6 Dataset statistics

Dataset	Size	#Total features	#Features used	Dataset type
Yelp	112,686	12	3	Real
MovieLens	1,000,209	3	2	Real
MovieLens non-metric	8,453	3	2	Real
UCI Gas dataset	13,911	128	128	Real
MakeBlobs	10,000,000	varied	20	Synthetic

Table 7 **Aug-MMR** vs MMR running time (s) on *MakeBlobs* with $l = 2$, $m = 6$

	Dataset Size			
Algorithm	5 k	10 k	50 k	10 k
Aug-MMR	4.33	8.69	43.57	306.11
MMR	19.77	40.16	197.28	1206.90

as follows:

$$speedup = \frac{T_{original-algorithm}}{T_{augmented-algorithm}} \tag{25}$$

where T denotes running time in seconds. Finally, we present time to build **I-tree** and the space required for that.

Datasets Experiments are conducted on five datasets, four real and one publicly available synthetic data. For real datasets, we use *Yelp*[5], *UCI Gas* dataset [6] that is high dimensional, *MovieLens* 1M records, and *MovieLens non-metric* dataset[7]. For synthetic data, we use *MakeBlobs* from the sklearn package.[8] An overview of the datasets is given in Table 6.

7.1 Baselines

In this section, we introduce diversity-based algorithms and index structure baselines that we compare to our proposed solutions.

7.1.1 Diversity baselines

For diversity-based methods, three representative algorithms are implemented.

MMR [14]: computes an objective score based on two parameters: relevance to the query and diversity with other records. As shown in Eq. 1, they are combined in a linear expression with a λ coefficient. The algorithm repeats this computation k times to produce top-k.

GMM [23]: finds the k most diverse records by selecting the maximum of minimum distances between undiscovered records and previously selected ones at each iteration (Eq. 10). Like MMR, it also iteratively builds the top-k set.

SWAP [49]: This greedy algorithm first finds the initial top-k records, then greedily interchanges records that are part of the current top-k with the ones that are remaining, if the *swap* improves diversity contribution (Eq. 15).

SPP [21]: Space Partitioning and Probing (SPP in short) is an algorithm that minimizes the number of accessed objects while finding exactly the same result as MMR. SPP belongs to a family of algorithms that rely only on score-based and distance-based access methods and does not require retrieving all the relevant objects. SPP is designed only for the geometric space.

7.1.2 Index structure baselines

We implement three additional baselines to compare against **I-tree**. These indexing techniques are limited to metric space and cannot be applied on arbitrary diversity function not satisfying triangular inequality.

KD-tree [10]:$KD\text{-}tree$ is a multidimensional Binary Search Tree. The tree is created by bisecting each dimension and finding the median. $KD\text{-}tree$ can perform searches in multidimensional space for efficient nearest neighbor search.

Ball-tree [29]: $Ball\text{-}tree$ is a binary tree in which every node defines a D-dimensional hypersphere or ball, containing a subset of the points to be searched. Each node in the tree defines the smallest ball that contains all data points in its subtree. This gives rise to the useful property that for a given test point t outside the ball, the distance to any point in a ball B in the tree is greater than or equal to the distance from t to the surface of the ball. $Ball\text{-}tree$ only supports binary splits.

The arity of the tree in both $KD\text{-}tree$ and $Ball\text{-}tree$ is fixed to 2.

M-Tree [15]: $M\text{-}tree$ is similar to $Ball\text{-}tree$, but supports multiple splits. Every node n and leaf lf residing in a particular node N is at most distance r from N, and every node n and leaf lf with node parent N keeps the distance from it. It also has the similar property of $Ball\text{-}tree$, which is for a

[5] https://www.yelp.com/dataset/documentation/main

[6] https://archive.ics.uci.edu/ml/datasets/gas+sensor+array+drift+dataset

[7] https://grouplens.org/datasets/movielens/

[8] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

given test point t outside the node, the distance to any point in a node in the tree is greater than or equal to the distance from t to the surface of the node.

We are incorporating Node-Node distance matrix to these baseline tree index structures so that they can be used for **I-tree** API.

Cover-Tree [12]: Another popular indexing structure is cover tree which is used to enable efficient nearest neighbor search in metric space. To be able to work with **DivGetBatch()**, the indexing technique must work in a fashion that the parent nodes of the index structure (in this case a tree) cover the records that are present in their sub-tree. This allows us to effectively maintain the inter-diversity bounds across the nodes and when a node gets pruned, all its children also does. Contrarily, in a cover tree, only the leaf nodes together contain and cover all the records and no other intermediate/higher level nodes does. Therefore, it is not obvious how to adapt this indexing technique and integrate it inside our proposed access primitive.

7.1.3 Index maintenance baselines

OPTMn and **GrMn** are compared with two baselines.

NonIncrMn *Algorithm:* In *NonIncrMn*, **I-tree** is built from scratch after every $|Y|$ insertions. **NonOlMn** *Algorithm:* This algorithm makes a local decision to insert each record based on Problem 2, without accounting for overlapping updates inside the same node in **I-tree**.

7.2 Summary of results

Our first set of experiments (Sect. 7.3) verify that our results from all three augmented algorithms are *identical* to their original counterparts. We measure precision@k [25] for different k, and our empirical results obtain 100% precision score.

Our next set of experimental results demonstrate (Sect. 7.4) that the running time of the augmented algorithms are consistent with our theoretical analyses. We achieve a 19× and 24× speedup for **Aug-MMR** and *Aug-GMM*, on *Makeblobs* 10M and *MovieLens* 1M data, respectively. We achieve a 3× speedup for **Aug-SWAP** on *MakeBlobs* 1M dataset. These results corroborate that our proposed framework is suitable to scale on large datasets. We also show that **I-tree** works on any arbitrary distance functions while other baselines are designed for only metric distance functions. We have conducted experimental analysis on two different non-metric distance functions (one obtained from the real data), these experimental results demonstrate that **Aug-MMR** attains 82% pruning compared to the baseline solutions, resulting in about 2.7 times speed up on an average. On the other hand, the results obtained from high dimensional UCI Gas dataset demonstrate that the proposed framework is still effective even in higher dimension, as **Aug-MMR** attains about 1.7 speed up on an average.

Figure 11 demonstrates the index construction and the query processing time trade-off of **I-tree** , and we compare that with our implemented baseline indexes, KD-tree, Ball-Tree, M-Tree. These results convincingly demonstrate that **I-tree** enables the fastest query processing time, while requiring comparable index construction time. The results demonstrate that **I-tree** is always more than 18× faster in query processing and as much as 170× faster for certain configurations. For preprocessing, it is always more than 1.5× faster and at times it is more than 20× faster. We also present $|CandR|$ percentage and pruning percentage of **I-tree** compared to other index baselines in Tables 9 and 10 which shows that **I-tree** outperforms all baselines with having 90% pruning.

The results in (Sect. 7.4.3) convincingly demonstrate that **I-tree** is lightweight to compute and space efficient (for the largest dataset, it takes 109 minutes to build the index, which is acceptable because it is done offline and only once). Finally, in Sect. 7.4.3, we demonstrate that our proposed solution **OPTMn** is an ideal choice for incremental index maintenance, while the greedy heuristic **GrMn** is highly scalable while being not too inferior from the optimal solution **OPTMn** qualitatively. **GrMn** takes 22 minutes to insert 100 k data into 1M dataset, while building **I-tree** from scratch is unrealistic as **NonIncrMn** takes 2 hours.

7.3 Quality analysis

The goal of these experiments is to empirically validate if the augmented algorithms produce the same results as their original counterparts. Additionally, we present how effective **DivGetBatch()** is in pruning records by presenting the size of $CandR$.

We have calculated precision@k while varying k from 10 to 50, considering the original and augmented algorithms. We obtain the precision@k equal to 100% always.

7.4 Scalability analysis

We run two types of scalability experiments. (i) demonstrate the efficacy of the augmented diversification algorithms and compare them appropriately with the baselines; (ii) demonstrate the efficacy of the indexing technique—present index construction and maintenance time, and compare them appropriately with the baselines. Additionally, we also present the memory requirements of **I-tree**. We analyze these effects by increasing dataset size and other pertinent parameters.

Table 8 $|CandR|$ percentage returned by **DivGetBatch()** on *MovieLens*

Algorithm	Dataset size 5 k (%)	10 k	50 k (%)	100 k (%)	500 k (%)	1 M (%)
Aug-MMR	13	5.21	0.56	0.09	0.08	0.08
Aug-GMM	59.96	15.48	4.16	2.67	0.31	0.4
Aug-SWAP	14.96	28.11	10.07	48.74	9.27	0.66

Table 9 $|CandR|$ percentage returned by **DivGetBatch()** using different index structures for **Aug-MMR** on *MakeBlobs*

Algorithm	Dataset size 5 k (%)	10 k (%)	50 k (%)	100 k (%)
I-tree	10	10	10	10
KD-tree	96.72	96.72	96.87	97.34
Ball-tree	96.7	95.62	96.56	96.56
M-tree	97.92	97.19	98.32	98.07

Table 10 Pruning percentage by **DivGetBatch()** using different index structures for **Aug-MMR** on *MakeBlobs*

Algorithm	Dataset size 5 k (%)	10 k (%)	50 k (%)	100 k (%)
I-tree	90	90	90	90
KD-tree	3.3	3.3	3.1	2.6
Ball-tree	3.3	4.3	3.4	3.4
M-tree	2	2.8	1.6	1.9

Table 11 Number of access percentage for **Aug-MMR** and SPP on *MakeBlobs*

Algorithm	Dataset size 5 k (%)	10 k (%)	50 k (%)	100 k (%)
I-tree	10	10	5.2	2.79
SPP	20.44	9.57	27.31	26.52

7.4.1 Augmented diversification algorithms

We first vary dataset size, then additional parameters that impact the query processing time. To demonstrate efficacy, we present two things. (1) The percentage of remaining records returned by *DivGetBatch()*, which is which is $|CandR|/N \times 100$ and pruning $(1-|CandR|/N \times 100$. (II) Query processing time in seconds.

Effectiveness in Pruning In Table 8, we present the number of remaining records returned by *DivGetBatch()*, which is $|CandR|$ using *MovieLens* dataset. We can observe that there is a remarkable reduction compared to the original dataset. For example, **Aug-MMR** returns only 814 records. The biggest number is for **Aug-SWAP** with 66513 records, but still returning only 6% of the records.

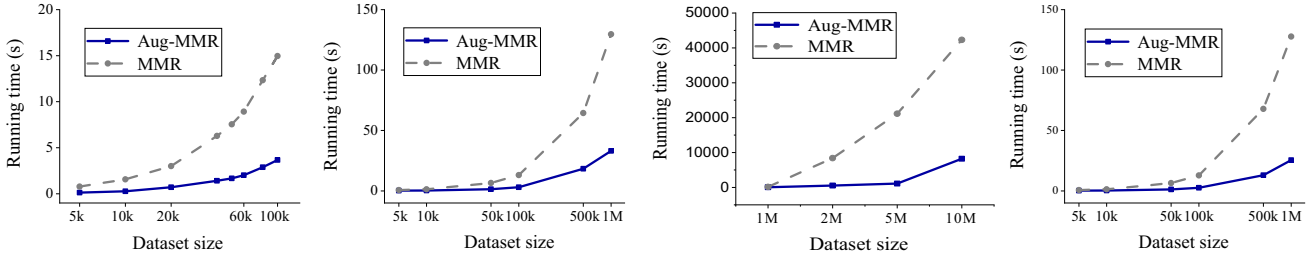
Tables 9 and 10 show $|CandR|$ and pruning percentage returned by *DivGetBatch()* for **Aug-MMR** algorithm using different index structures and MakeBlobs dataset. We can see that by fixing $C = 32$, $KD\text{-}tree$, $Ball\text{-}tree$, and $M\text{-}tree$ pruning are below 5%, while **I-tree** pruning considerably outperforms all baseline which is 90%.

Effectiveness in Number of Accesses In order to perform a fair comparison between our augmented algorithms and SPP, we compare the number of I/O accesses SPP does and present that number for **Aug-MMR** (SPP is designed to optimize that access). We calculate the number of accesses in *DivGetBatch()* by counting the distinct records present in $CandR$ in k rounds. The results are presented in Table 11. We can see that **Aug-MMR** has less number of access. For example on 100 k data, **I-tree** has 2799 number of access while SPP has 26521 number of access.

Varying Dataset Figures 3, 4, 5, 6 and 7 compare the running times of our three augmented algorithms and their baselines using our three datasets. As N increases, the running times of each algorithm and its baseline increase, but we observe that our algorithms are consistently faster and they scale significantly better. Figure 3 shows Aug-MMR's scalability on all three datasets. We fix m to 1000, $k = 20$ and $l = 1$ for all dataset sizes while N is increased from 5000 up to 1M. We can see that on *MovieLens*, varying N from 5000 to 1M, **Aug-MMR** is 5× faster than *MMR*. Figure 5 shows **Aug-GMM** 's scalability. On *MovieLens*, varying N from 5000 to 10M, **Aug-GMM** is 24× faster than *GMM*. Consistent with the theoretical analysis, **Aug-GMM** is faster than **Aug-MMR** for the same settings because **Aug-MMR** has an additional k term in the expected cost equation. Figure 7 shows **Aug-SWAP** 's scalability on all three datasets. For the 1M data of *MakeBlobs*, we obtain a 3× speedup over *SWAP*. We obtain a 1.33× speedup for *Movielens* because the total number of swaps in *MovieLens* are higher.

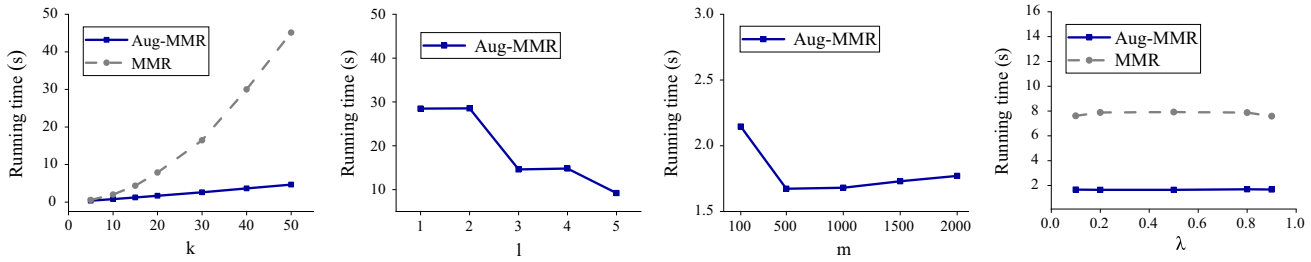
We also measure the scalability of **Aug-MMR** compared to MMR using large scale data sizes of 2 M, 5 M, and 10 M using makeBlobs dataset. The results are shown in Fig. 3c in which with $m = 1000$ and $l = 1$, we have up to 19× speedup.

Moreover, we run **Aug-MMR** on high-dimensional euclidean distance considering more number of features using 1M and 2M makeBlobs dataset. for 1M data, 1M and 20 features, MMR takes 12492.64 (s), and **Aug-MMR** takes 2817.14 (s). For 2M data and 20 features, MMR takes 25812.43 9



(a) Yelp **(b) MakeBlobs** **(c) MakeBlobs, large scale** **(d) MovieLens**

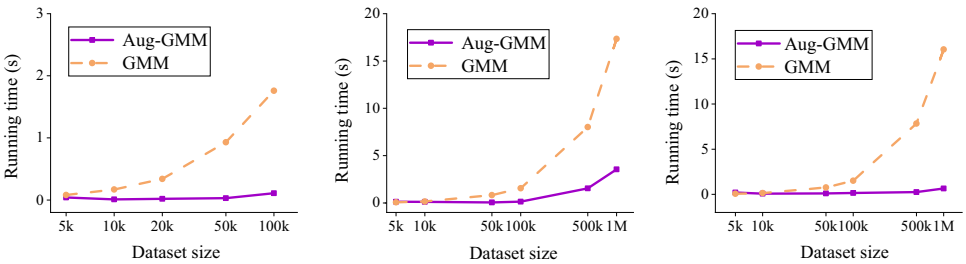
Fig. 3 **a** Yelp **b** MakeBlobs **c** MakeBlobs, large scale **d** MovieLens. **Aug-MMR** vs *MMR* scalability



(a) Varying k **(b)** Varying l **(c)** Varying m **(d)** Varying λ

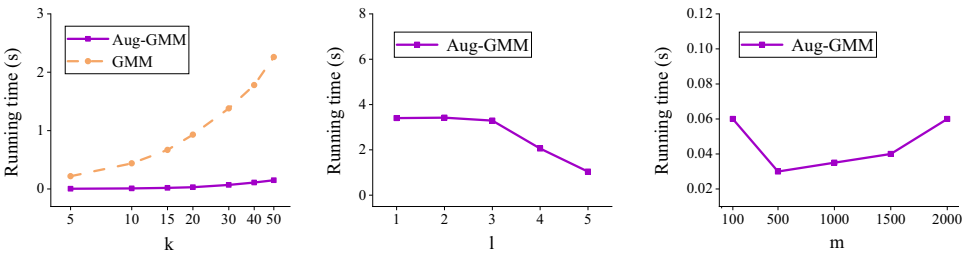
Fig. 4 **a** Varying k **b** Varying l **c** Varying m **d** Varying λ. **Aug-MMR** vs *MMR* varying parameters

Fig. 5 **a** Yelp **b** MakeBlobs **c** MovieLens. **Aug-GMM** vs *GMM* scalability



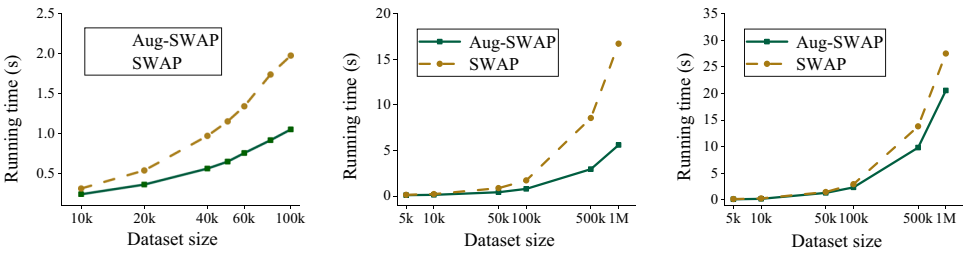
(a) Yelp **(b) MakeBlobs** **(c) MovieLens**

Fig. 6 **a** Varying k **b** Varying l **c** Varying m **Aug-GMM** vs *GMM* performance varying parameters

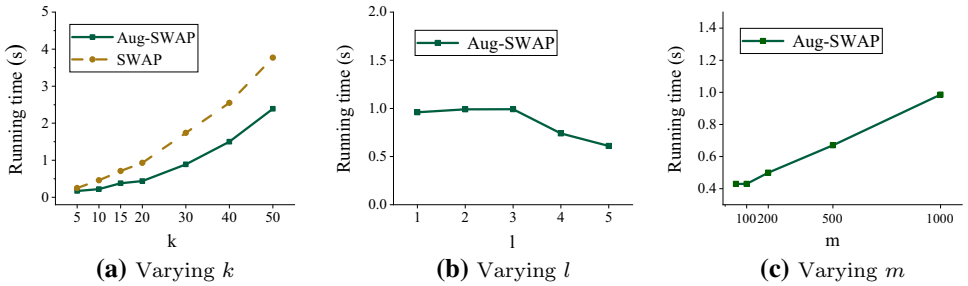


(a) Varying k **(b)** Varying l **(c)** Varying m

Fig. 7 **a** Yelp **b** MakeBlobs **c** MovieLens. **Aug-SWAP** vs *SWAP* scalability



(a) Yelp **(b) MakeBlobs** **(c) MovieLens**



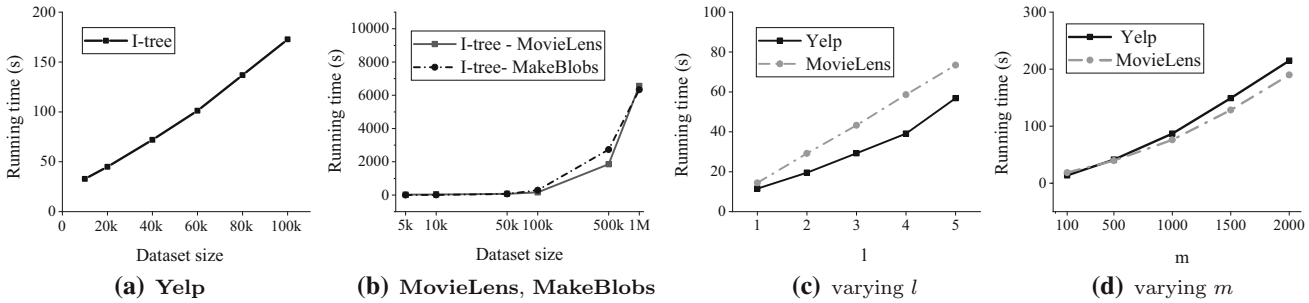


Fig. 8 **a** Varying k **b** Varying l **c** Varying m **Aug-SWAP** vs *SWAP* varying parameters

Fig. 9 **a** Yelp **b** MovieLens, MakeBlobs **c** varying l **d** varying m. **I-tree** construction time

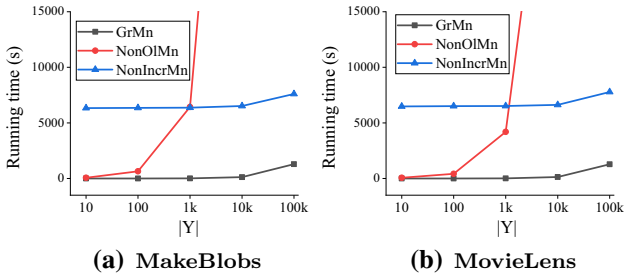


Fig. 10 **a** MakeBlobs **b** MovieLens. **I-tree** maintenance time varying $|Y|$

(s), **Aug-MMR** takes 6317.20 (s) which in both case show 4× speedup.

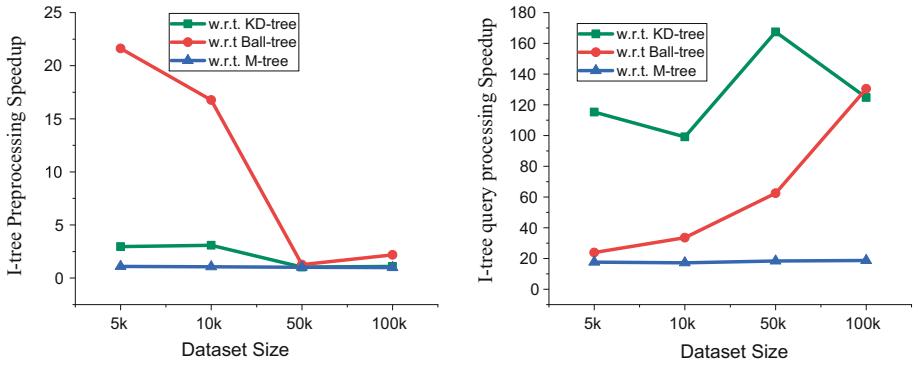
Additionally, Fig. 12 presents the scalability of the proposed **Aug-MMR** algorithm compared to MMR using UCI Gas dataset with 10 k records and 128 features. We set $\lambda = 0.8$ and vary k from 10 to 25. By increasing k, **Aug-MMR** shows more scalability than MMR. **Aug-MMR** is about 1.7 times faster than the baseline implementation.

Finally, we run **Aug-MMR** on l more than 1 to show the efficiency of our proposed algorithm using multi-level **I-tree**. Table 7 shows that for l=2, **Aug-MMR** speedup is almost 4× for all dataset sizes.

Varying Parameters We study the effect of different parameters on running time. Some parameters belong to the offline indexing algorithm, such as the number of levels (l) and arity of **I-tree** (m) and the total number of nodes (C). Other parameters are part of the online augmented algorithms. For example, k for the number of returned records and λ coefficient for **Aug-MMR** . In Figs. 4, 6, 8, we vary parameters using *Yelp* dataset with a fixed size of 50000 records. In our experiment, optimum parameter settings for offline indexing are obtained by performing multiple runs and selecting the best. The index created using those parameter settings can be used in multiple runs of the online phase.

Varying k. Figures 4a, 6a, and 8a present how running time changes as we vary k from 5 to 50 for different baselines while fixing l, m, and λ to 1, 500, and 0.8, respectively. The running time increases quadratically for MMR and *Aug-MMR*, linearly for GMM and *Aug-GMM*, and in $\mathcal{O}(k * \log k)$ fashion for $SWAP$ and *Aug-SWAP*. These results are as consistent with our theoretical analysis, because of the presence of k^2 term in the MMR and *Aug-MMR*'s expected cost, k in GMM and *Aug-GMM*'s expected cost, and $k * \log k$ of that of $SWAP$ and *Aug-SWAP*. *Varying m*. Figures 4c, 6c, and 8c show the impact of varying m on the running time of the three algorithms. While varying m, we fix other parameters: $k = 20$, $l = 1$. The choice of m depends on the distribution of the dataset. As we increase m, the bounds for augmented algorithms become tighter while time for **DivGetBatch()** increases. We can see that there is a drop in running time and which indicates the optimum value for m for these three algorithms. For example, in **Aug-MMR** and *Aug-GMM*, the ideal value is $m = 500$ and for *Aug-SWAP*, it is $m = 100$.

Varying l. Figures 4b, 6b, and 8b show the impact of varying l on the running time of the three algorithms. We fix other parameters: $k = 20$, and setting m to 2. C, the total number of nodes in **I-tree** becomes 2, 7, 15, 31, 63, respectively, for $l = 1, 2, 3, 4, 5$. In general, by fixing m and increasing l, C increases, and overall running time decreases. This is con-



(a) **I-tree** Index Preprocessing speedup w.r.t baselines

(b) **I-tree** Query Processing speedup w.r.t baselines

Fig. 11 **a I-tree** Index Preprocessing speedup w.r.t baselines **b I-tree** Query Processing speedup w.r.t baselines. Index Construction and Query Processing time for tree baselines and **I-tree**

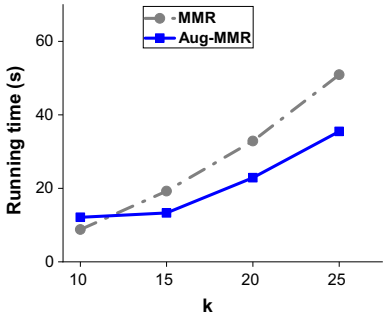


Fig. 12 **Aug-MMR** vs MMR running time on UCI Gas data

sistent with our theoretical analysis, as the expected running time contains a $1/C$ term.

Varying λ. Figures 4d, 6d, and 8d show that varying λ in *MMR* and **Aug-MMR** does not significantly change the running time. We have fixed $k = 20$, $l = 1$, and $m = 500$. The result is evident by observing the expected cost equations of *MMR* and **Aug-MMR** algorithms which do not contain a λ term. Though MR scores changes with λ, it has very little effect on the overall running time of *MMR* and **Aug-MMR** algorithms.

7.4.2 Varying diversity functions

Table 13 shows the results for **Aug-MMR** compared to MMR using different distance measures: euclidean distance measure, cosine similarity as general metric, and a non-metric distance function. Using 100 k data from MakeBlobs dataset and m= 1000, l = 1 and number of features = 2, we can see that **Aug-MMR** performs 4× better than MMR using both euclidean and cosine similarity metrics. For non-metric arbitrary distance function, the distance between records does not satisfy triangular inequality. Using this method, we see 15% improvement, since the relevance and diversity scores are created arbitrarily and the result depends on the data distribution.

Table 12 shows overall comparison for **I-tree** and other baselines. SPP uses $KD\text{-}tree$ as its index so we did not add it to the table. We can see that, unlike other baselines, **I-tree** can be used in non-metric functions and outperforms with 90% pruning of the original dataset.

Table 14 shows the results for **Aug-MMR** compared to MMR using non-metric distance function computed from MovieLens non-metric dataset. The total number of movies is 8,453, $\lambda = 0.8$, and $k = 20$. The diversity between a pair of items (movies) is calculated as the number of users that have rated either of those movies, but not both. Table 14 demonstrates that **Aug-MMR** outperforms *MMR* with 82.66% pruning of the original dataset, resulting in about 2.7 times speed up on an average.

7.4.3 Index construction and maintenance

Comparison with Baselines-Index Construction vs. Query Processing In these set of experiments, we compare the index construction and query processing time trade-off of **I-tree** and compare that with of $KD\text{-}tree$, $Ball\text{-}tree$, and $M\text{-}tree$ considering *Aug-MMR*. We adapt k-means and k-medoids [25] for building **I-tree** with number of iterations set to 300. The dataset that is used in this experiments is MakeBlobs. Figure 11 presents the **I-tree** speedup compared to other baselines for index preprocessing and query processing time. The results demonstrate that **I-tree** is always more than 18× faster in query processing and as much as 170× faster for certain configurations. For preprocessing, it is always more than 1.5× faster and at times it is more than 20× faster.

Index Construction Now that it is obvious that **I-tree** outperforms the other indexing baselines, we further profile its efficacy.

In Fig. 9a and b, we vary dataset size and fix other parameters, $m = 1000$, $l = 1$. As we can observe in Fig. 9a, on the 100 K *Yelp* dataset, indexing time is 172.69 seconds. In Fig. 9b, indexing time is 105 minutes on the 1M *MakeBlobs* dataset, and 109 minutes on the 1M *MovieLens*. Figure 9c

Table 12 Index comparisons

Index	Metric functions	Non metric Functions	90% Pruning
I-tree	✓	✓	✓
KD-tree [10]	✓	×	×
Ball-tree [29]	✓	×	×
M-tree [15]	✓	×	×

Table 13 **Aug-MMR** vs *MMR* running time on *MakeBlobs* 100 k records

Algorithm	Distance function Euclidean	Cosine	Non-metric
Aug-MMR	3.08	4.64	13.06
MMR	13.12	15.36	15.27

Table 14 **Aug-MMR** vs *MMR* on *Movielens non-metric* data

Algorithm	Running time (s)	Average pruning
Aug-MMR	0.19	82.66%
MMR	0.52	0

and d shows that the running time increases linearly when parameters m and l are systematically increased. In Fig. 9c, by varying l, we fix dataset size to 50000, and m to 2 (since $C = m^l$, by increasing l, the total number of nodes will increase). Finally, in Fig. 9d, we vary m, while fixing dataset size to 50000 and $l = 1$. These figures demonstrate that the preprocessing time increases linearly with varying parameters. **I-tree** takes 253 MB of space for 1M data with $m = 1000$ and $l = 1$.

Index Maintenance For analyzing the index maintenance, we use two datasets, *MakeBlobs* and *MovieLens*. We compare **OPTMn** and its efficient counterpart **GrMn** with the baselines *NonOlMn*, and *NonIncrMn*. As expected, **OPTMn** has the least number of updates, but due to its inherent exponential nature, it does not scale beyond 10 k dataset size with more than $|Y| = 1000$ records. Table 15 presents these results. We also see *GrMn*, even though not the optimal one, but stays consistently close to *OPTMn*. This table also shows that **GrMn** is better than the baselines in both running time and number of updates. Figure 10a and b presents running time comparisons on very large datasets. **GrMn** is highly scalable, and the other two baselines take more time than *GrMn*. These results corroborate that **GrMn** is a suitable alternative to solve the index maintenance problem (Figs. 11, 12).

Incremental Index Maintenance vs Maintenance from Scratch Table 16 shows comparison between **GrMn** and **NonIncrMn** index update algorithms. We present index preprocessing time in the offline phase, and query processing time in the online phase for the **Aug-MMR** algorithm.

Table 15 **I-tree** maintenance on *MakeBlobs* 10 k records

$\|Y\|$	Algorithm	# updates	Running time (s)
10	**OPTMn**	14	3.59
	GrMn	76	0.007
	NonOlMn	14	0.29
	NonIncrMn	2446	1.30
100	**OPTMn**	59	512.42
	GrMn	76	0.05
	NonOlMn	142	2.97
	NonIncrMn	2447	1.44
1000	**OPTMn**	59	18768.68
	GrMn	76	0.43
	NonOlMn	1068	34.58
	NonIncrMn	2449	1.45

Clearly, **GrMn** requires smaller preprocessing time and higher query processing time compared to *NonIncrMn*. As it could be seen from Table 16, with 10,000 updates, the query processing time of **GrMn** becomes almost 5× slower than that of *NonIncrMn*. Contrarily, the preprocessing time of **GrMn** is about 4.5× faster than that of **NonIncrMn** at that setting. Since query processing time is more important and must be optimized, it seems, for 10,000 updates, it is better to build the index from scratch instead of maintaining it incrementally.

8 Related work

8.1 Results diversification

Result diversification remains to be an active research topic with extensive applications in recommendation and search [1,2,4,13,20,30,33,35,36,40–44], including very recent works that study diversity for fairness and popularity [31,39,51].

8.2 Content-based algorithms

Content-based algorithms, which are our primary focus here, are of two kinds: *Interchange algorithms* first select k relevant records and then, exchange selected records with remaining records to increase the overall diversity (*SWAP* [49] is an example). *Incremental greedy algorithms* iteratively build the

Table 16 **I-tree** maintenance **GrMn** vs construction from scratch **NonIncrMn** running time on *MakeBlobs* 10 k records

$\|Y\|$	Insertion Algorithm	Preprocessing time-offline (s)	Query processing time-online (s)
10	GrMn	0.007	1.25
	NonIncrMn	1.30	0.55
100	GrMn	0.05	1.33
	NonIncrMn	1.44	0.60
1000	GrMn	0.43	1.96
	NonIncrMn	1.45	0.80
10000	GrMn	1.02	8.18
	NonIncrMn	4.65	1.61

top-k set by selecting the best record at each round. Notable examples of this latter kind are Maximal Marginal Relevance (*MMR*) method [14], Greedy Max-Min (*GMM*) [23], Max-Sum [22], IA-SELECT [5], and dLSH [2].

SPP [21] is a bounded diversification algorithm that produces same result as MMR while minimizing the number of accessed records. In [17], Drosou et al. introduce both greedy and interchange algorithms for the diversity over continuous data. In [19], the authors propose greedy algorithms for considering diversity over dynamic data by presenting $Insert$ and $Delete$ operations over the cover tree indexing structure. They also exploit the *GMM* algorithm for returning diversified top-k results. In [18], the authors propose greedy algorithms for diversity over a representative subset of objects, $DisC$, which is a mapping of the original data. They also present a degree of diversification, radius r, instead of k size results. Their proposed algorithms exploit the M-$tree$ [15] indexing structure.

From a different perspective, one can categorize diversification algorithms into three groups: record-level, feature-level, and category-level. In record-level algorithms (*MMR*, *GMM*, and *SWAP*), the input is the distance value between records regardless of which record feature is more important. The score value is calculated based on an objective function that calculates distances/diversity. The inputs of feature-level algorithms are record features. Examples are DivGen and GenFilt [6]. The feature with the highest score is obtained from all records based on a ranking, and the goal is to skip some features and prune records having low scoring features. In the category-level algorithms, records are grouped into multiple categories. Such algorithms apply some constraints that will return no more than one or two records from the same category [3,50].

8.3 Comparison with existing indexes

Compared to our proposed **I-tree**, existing indexing techniques are vector space-based methods where coordinate information of the records is used to create data structures to answer a large spectrum of distance queries, where distance may be based on Euclidean, cosine similarity, general L_p norms, and so on. Popular solutions in low to moderate dimensional space include *K-B-D-tree* [38], *kd-tree* [10], *R-tree* [24], *R*-tree* [9], *SS-tree* [45] or more recent *X-tree* [11], *UB-tree* [8], *SR-tree* [27]. All these methods use the domain object feature vectors to measure the distance between objects and create a similarity index. As opposed to that, we consider the records to be atomic (and not a collection of vectors), and the diversity function could be metric or not. Therefore, these methods do not extend to solve our problem.

There exists other popular tree data structures like $Cover$-$tree$ [12], $Ball$-$tree$ [29] and M-$tree$ [15] that are used for nearest neighbor search. Unlike our **I-tree**, these trees can only be used for metric distance functions.

In summary, we present an access primitive **DivGetBatch()** *that leverages a precomputed data structure* **I-tree** *to integrate MMR, GMM, and SWAP to expedite their processing time. The design of our primitive is independent of features and categories and is applicable with any distance measure, making it generic and useful. We study MMR, GMM, and SWAP, since we believe these are notable choices in the existing diversity literature space, and many more recent works adapt these algorithms* [2,7,17–19,26,34,37,46–48].

9 Conclusion

We propose an access primitive **DivGetBatch()** to expedite diversification algorithms while returning their exact top-k results. We present a computational framework to develop **DivGetBatch()** that contains a pre-computed index structure **I-tree** and describe how to rewire popular diversification algorithms using **DivGetBatch()**. Unlike existing indexes that primarily work on vector spaces (assuming the records have co-ordinates), we consider the records to be atomic as opposed to a collection of vectors. We make rigorous the-

oretical analysis of the exactness and running times of the augmented algorithms. We present principled solutions to maintain **I-tree** under batch updates. Our experiments on large real-world datasets corroborate our theoretical analysis and show that our solution yields a 24× speedup on large datasets.

In the future, we are interested to study how to enable approximate top-k result diversification with guarantees leading to even faster running times. We also intend to explore how to adapt our proposed framework if diversity is assumed to satisfy metric property, in particular, the triangle inequality.

References

1. Abbar, S., et al.: Diverse near neighbor problem. In: SoCG pp. 207–214 (2013)
2. Abbar, S., et al.: Real-time recommendation of diverse related articles. WWW pp. 1–12 (2013)
3. Abbassi, Z., et al.: Diversity maximization under matroid constraints. In: SIGKDD (2013)
4. Agarwal, P.K., et al.: Efficient indexes for diverse top-k range queries. PODS pp. 213–227 (2020)
5. Agrawal, R., et al.: Diversifying search results. WSDM 5–14 (2009)
6. Angel, A., Koudas, N.: Efficient diversity-aware search. SIGMOD pp. 781–792 (2011)
7. Balog, K., et al.: Transparent, scrutable and explainable user models for personalized recommendation. SIGIR (2019)
8. Bayer, R.: The universal b-tree for multidimensional indexing: General concepts. In: ICWCA, Springer, pp 198–209 (1997)
9. Beckmann, N., et al.: The r*-tree: An efficient and robust access method for points and rectangles. SIGMOD, pp 322–331 (1990)
10. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
11. Berchtold, S., et al.: The x-tree: An efficient and robust access method for points and rectangles. VLDB, pp 28–39 (1996)
12. Beygelzimer. A., et al.: Cover trees for nearest neighbor. ICML (2006)
13. Cai, Z., et al.: Diversified spatial keyword search on rdf data. The VLDB Journal pp. 1–19 (2020)
14. Carbonell, J., et al.: The use of mmr, diversity-based reranking for reordering documents and producing summaries. SIGIR pp. 335–336 (1998)
15. Ciaccia, P., et al.: M-tree: An efficient access method for similarity search in metric spaces. Vldb **97**, 426–435 (1997)
16. Cormen, T.H., et al.: Introduction to algorithms. MIT press (2009)
17. Drosou, M., et al.: Diversity over continuous data. IEEE Data Eng Bull **32**(4), 49–56 (2009)
18. Drosou, M., et al.: Disc diversity: result diversification based on dissimilarity and coverage (2012). arXiv preprint arXiv:1208.3533
19. Drosou, M., et al.: Diverse set selection over dynamic data. TKDE **26** (2013)
20. Esfandiari, M., et al.: Multi-session diversity to improve user satisfaction in web applications. TWC, pp 1928–1936 (2021)
21. Fraternali, P., et al.: Top-k bounded diversification. SIGMOD, pp 421–432 (2012)
22. Gollapudi, S., et al.: An axiomatic approach for result diversification. WWW pp. 381–390 (2009)
23. Gonzalez, T.F.: Clustering to minimize the maximum intercluster distance. TCS **38**, 293–306 (1985)
24. Guttman, A.: R-trees: A dynamic index structure for spatial searching, ACM **14**(2), (1984)
25. Han, J., et al.: Data mining concepts and techniques third edition. Morgan Kaufmann Series **5**(4), 83–124 (2011)
26. Hope, T., et al.: Accelerating innovation through analogy mining. SIGKDD (2017)
27. Katayama, N., et al.: The sr-tree: An index structure for high-dimensional nearest neighbor queries. Sigmod Record **26**(2), 369–380 (1997)
28. Knuth, D.E.: The Art of Computer Programming, Fundamental Algorithms, vol 1, 3rd edn. Addison Wesley Longman Publishing Co., Inc., (book) (1998)
29. Kumar, N., et al.: What is a good nearest neighbors algorithm for finding similar patches in images? In: European conference on computer vision, Springer, pp 364–378 (2008)
30. Mafrur, R., et al.: Dive: diversifying view recommendation for visual data exploration. CIKM pp. 1123–1132 (2018)
31. Maropaki, S., et al.: Diversifying top-k point-of-interest queries via collective social reach. In: CIKM pp. 2149–2152 (2020)
32. Mouratidis, K.: Geometric aspects and auxiliary features to top-k processing. In: MDM (2016)
33. Nikookar, S., et al.: Diversifying recommendations on sequences of sets. VLDB Journal (2022)
34. Parreño, F., et al.: Measuring diversity. a review and an empirical analysis. EJOR **289**(2), 515–532 (2021)
35. Puthiya Parambath, S.A., et al.: A coverage-based approach to recommendation diversity on similarity graph. In: RecSys pp. 15–22 (2016)
36. Qin, L., et al.: Diversifying top-k results (2012). arXiv preprint arXiv:1208.0076
37. Ren, P., et al.: Leveraging contextual sentence relations for extractive summarization using a neural attention model. SIGIR, pp 95–104 (2017)
38. Robinson, J.T.: The kdb-tree: a search structure for large multidimensional dynamic indexes. SIGMOD, pp 10–18 (1981)
39. Singh, A., et al.: Fairness of exposure in rankings. In: SIGKDD pp. 2219–2228 (2018)
40. Tsai, C.H., et al.: Beyond the ranked list: User-driven exploration and diversification of social recommendation. In: 23rd ICIUI pp 239–250 (2018)
41. Vargas, S., et al.: Rank and relevance in novelty and diversity metrics for recommender systems. RecSys (2011)
42. Vargas, S., et al.: Coverage, redundancy and size-awareness in genre diversity for recommender systems. RecSys pp. 209–216 (2014)
43. Wang, D., et al.: Sequence-based context-aware music recommendation. Information Retrieval Journal pp. 230–252 (2018)
44. Wang, L., et al.: Diversified and scalable service recommendation with accuracy guarantee. IEEE TCSS (2020)
45. White, D.A., et al.: Similarity indexing with the ss-tree. In: ICDE pp. 516–523 (1996)
46. Wu, W., et al.: Personalizing recommendation diversity based on user personality. UMUAI **28**(3), 237–276 (2018)
47. Wu, Y., et al.: Beyond greedy search: pruned exhaustive search for diversified result ranking. SIGIR, pp 99–106 (2018b)
48. Jg, Y., et al.: Recent advances in document summarization. KIS **53**(2), 297–336 (2017)
49. Yu, C., et al.: It takes variety to make a world: diversification in recommender systems. EDBT pp. 368–378 (2009)

50. Zanitti, M., et al.: A user-centric diversity by design recommender system for the movie application domain. In: Companion Proceedings of WWW, pp 1381–1389 (2018)
51. Zehlike, M., et al.: Fa* ir: A fair top-k ranking algorithm. In: CIKM pp. 1569–1578 (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.