Hike: A High Performance kNN Query Processing System for Multimedia Data

Hui Li
College of Computer Science and Technology
Guizhou University
Guiyang, China
cse.HuiLi@gzu.edu.cn

Ling Liu
College of Computing
Georgia Institute of Technology
Atlanta, USA
lingliu@cc.gatech.edu

Xiao Zhang, Shan Wang
School of Information
Renmin University of China
Beijing, China
{*swang, zhangxiao*}*@ruc.edu.cn*

***Abstract*—Internet continues to serve as the dominating platform for sharing and delivering multimedia contents. *kNN* queries are an enabling functionality for many multimedia applications. However, scaling *kNN* queries over large high-dimensional multimedia datasets remains challenging. In this paper we present Hike, a high performance multimedia *kNN* query processing system, it integrate the proposed novel Distance-Precomputation based R-tree (DPR-Tree) index structure, coupled with the fast distance based pruning methods. This unique coupling improves the computational performance of *kNN* search and consequently reducing I/O cost. Since Hike's index structure DPR-Tree by design generates two types of query independent precomputed distances during the index construction, and it can intelligently utilizes the precomputed distances to design a suite of computationally inexpensive pruning techniques, which make the Hike system can filter out irrelevant index nodes and data objects, and minimize the amount of duplicate computations among different *kNN* queries. We conduct an extensive experimental evaluation using real-life web multimedia datasets. Our results show that DPR-Tree indexing coupled with precomputed distance based query processing make the Hike system can significantly reduce the overall cost of *kNN* search, and is much faster than the existing representative methods.**

***Keywords-kNN* Search; High-dimensional Index; Multimedia Data;**

I. Introduction

We are entering the big data era and Internet continues to be the dominating playground for sharing and delivering big data of huge volume, fast velocity and high variety. Multimedia content as one of the most popular data types has enjoyed constant and rapid growth in both dataset size and resolution requirements (dimensionality of feature vectors). However, efficient search in large-scale multimedia systems, such as the similarity search techniques, especially the *kNN* search techniques, continue to be challenging in terms of high performance and scalability for many modern multimedia applications [1]. To accelerate the *kNN* queries over large high-dimensional multimedia datasets, the multi-dimensional indexes, which organize high-dimensional feature vectors of multimedia objects for fast disk I/Os, are often combined with index-specific optimization techniques, which can filter out irrelevant index nodes and data objects during the *kNN* query processing. Although a fair amount of research has been dedicated to both multi-dimensional indexing and kNN pruning techniques to facilitate multimedia *kNN* queries [2], [3], fast *kNN* retrieval over large high-dimensional multimedia datasets remains a demanding contest [1]. Concretely, as size and dimensionality increase, the multimedia *kNN* queries tend to become both I/O-intensive and computation-intensive [1], [4]. However, most of existing multi-dimensional indexing techniques focus primarily on reducing the amount of disk I/O access to improve the multimedia *kNN* query performance. We argue that it is equally important to develop computationally inexpensive optimization techniques that can filter out irrelevant index nodes and data objects at early stage of the *kNN* query processing.

In this paper, we present a high performance multimedia *kNN* query processing system named Hike, it integrate the proposed DPR-Tree (Distances Precomputed R-Tree) index structure, and a novel, computationally efficient *kNN* pruning approach *PDP* (Precomputed Distances based Pruning). Hike aims at speeding up multimedia *kNN* query processing by minimizing the overhead of distance computations. The novelty of Hike is two folds. First, we devise and integrate DPR-Tree, an efficient multimedia index structure by precomputing two types of query independent distances during index construction, which are utilized to speed up Hike's *kNN* query processing at runtime. Second, we develop a suite of novel *kNN* pruning techniques inside Hike, which use query dependent low-cost filters based on precomputed query independent distances stored in the index nodes.

Hike's *kNN* pruning design has two unique features: (i) It can utilize the precomputed distances to speed up the actual distance computations between query object and each of the candidate objects, and then filter out irrelevant data and index nodes as early as possible, which not only cuts down the unnecessary I/O cost, but also dramatically reduces the amount of computational overheads. (ii) It can identify and reuse the common computations incurred during the processing of different *kNN* queries to further reduce the computational cost of *kNN* search, which make Hike's DPR-Tree based *kNN* search become more scalable. We conduct extensive experiments on real-life multimedia datasets with different dataset sizes, up to 1 million images, and different

data dimensionalities, up to 256D, extracted from MSRA-MM 2.0 [5]. Our evaluation show that Hike is much faster than the existing representative approaches as it can significantly reduce I/O and computation cost for *kNN* search over large high dimensional multimedia datasets.

II. Related Work

A fair amount of research [2], [3] have devoted to high-dimensional indexing based multimedia *kNN* search in the last decade. We classify existing high-dimensional indexing approaches into three main categories based on the index structures: partitioning-based indexing, metric-based indexing and approximation-based indexing.

The typical partitioning based high-dimensional indexing methods include the variant of R-Tree indexes, which partition datasets either by MBR (Minimum Bounding Rectangle, *e.g.*, X-Tree [6]) or MBS (Minimum Bounding Sphere, *e.g.*, SS-Tree [7]), and divide space into smaller volume regions. In high-dimensional scenarios, the diagonal of MBR tends to be longer than the diameter of MBS, while the volume of MBS is often larger than the corresponding MBR. SR-Tree [8] combines the advantages of MBR and MBS. The bounding region in SR-Tree is specified by the intersection of a bounding sphere and a bounding rectangle, thus SR-Tree achieved better performance than those indexing approaches [9] that use either MBR or MBS at the cost of maintaining both MBR and MBS.

In metric based indexing methods, multimedia objects are mapped into a metric space. The basic principle of metric based indexing methods is to partition the dataset based on the distance between data point and some reference points. For a given query, the relevant data points can be retrieved based on both the distances between data points and reference points and the distances between reference points and query point. Example metric based indexing methods include M-Tree [10], iDistance [11] and several others [3]. iDistance, as an representative metric-based indexing method, employs the B+-Tree to index the single-dimensional distances between selected reference points and data objects. By reducing high dimensional distance computation to single dimension distance, iDistance is shown to perform efficiently for high-dimensional *kNN* search.

The third type of high-dimensional indexing approaches is based on approximation. They often have good performance at cost of accuracy loss to some extent. Approximation based indexing methods either use small, approximate representations to replace the original feature vectors, or employ various techniques to reduce the feature dimensionality in the indexing schema. Among them, DCR [12] and hashing based approaches (e.g., DSH[13]) rely on dimensionality reduction techniques, whereas P+-Tree [14] transform high-dimensional features into low-dimensional features (usually 1-dimensional features), and then execute search based on B+-Tree like techniques. A-Tree [15] is an approximation based high-dimensional index, motivated from VA-File [16], and SR-Tree. It uses relative approximation of MBR to represent data regions or data objects, and thus like VA-files, it can host larger fanout than other R-Tree variants, leading to more saving in I/O access and efficient computation of approximate *kNN* results.

Interestingly, most of high-dimensional indexing and search techniques for *kNN* queries have been focused on minimizing disk I/O access by developing new indexing structures to organize high dimensional multimedia datasets, combined with standard query dependent distance metrics, such as *MINDIST* and *MINMAXDIST*, for *kNN* query processing. One of the most widely used pruning techniques is the *INN* (Incremental Nearest Neighbor) search algorithm [17]. Unlike the *branch-and-bound* [18] approach who use both distance metrics *MINDIST* and *MINMAXDIST* for *kNN* pruning, *INN* algorithm only emploies *MINDIST* and priority queue during the *kNN* search, led to certain degree of reduction in the computational overheads. However, to the best of our knowledge, Hike's DPR-tree based approach is the first one that extends the popular R-tree index structure to embed precomputed query independent distances, and develops precomputed distance enabled optimization techniques to efficiently reduce the search space of *kNN* queries.

III. Design Overview of Hike System

A. Motivated Ideas

***kNN* query processing.** When a *kNN* query Q is issued, by conventional *kNN* search procedure, the query processing is performed in three steps: (i) First, we find the index node with the smallest MBR containing Q. We call the set of index nodes contained in this smallest MBR the candidate set of index nodes relevant to Q. (ii) Then the distance-based pruning, such as *branch-and-bound* [18], or *INN* [17] approaches will be employed to filter out those index nodes whose children nodes are closer to Q. (iii) Finally, the actual distance computations will be performed from Q to each of the remaining index nodes and each of the data objects contained in the candidate set relevant to Q to compute the *kNN* results. This distance based pruning and computation procedure applies to all queries.

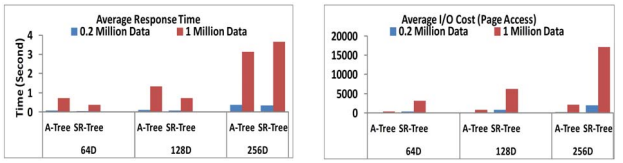
***kNN* cost distribution.** To better understand the cost distribution of high dimensional multimedia *kNN* search, we conducted a series of initial experiments on a large-scale real-life web multimedia dataset MSRA-MM 2.0 [5], consisting of 1 million images with dimensionality up to 256D. We randomly select 10,000 images as the query set. Details are described in Section VI.

We measured the ratio of computational overheads and I/O cost, and the cost distribution of all three distance computations during a multimedia *kNN* search based on the representative high dimensional index structures A-Tree and SR-Tree respectively. To understand how the dataset size and dimensionality impact the *kNN* query processing

performance, experiments are conducted by varying the dimensionality (ranging from 64 to 256) and the size of datasets (0.2 million v.s. 1 million images).

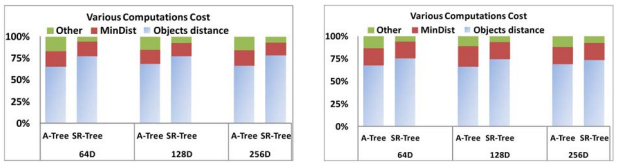
Figure 1 shows two interesting facts: (i) As dimensionality and data size increase, we see a sharp increase in both the response time and the I/O cost. (ii) When the dataset size is large and dimensionality is high, the computational cost may become a major factor in overall response time.

Figure 2 displays the computational overhead distribution for 0.2 million dataset and 1 million dataset separately. We observe that obtaining actual object distances to a query and *MINDIST*-based pruning consume a large portion of overall computational overhead for both A-tree and SR-tree. Furthermore, the actual distance computations between query object and data (or region) objects are most expensive. These observations motivate Hike's DPR-tree based approach. We conjecture that by using precomputed distances at runtime, we can significantly reduce all three types of distance computation overheads and effectively speed up the *kNN* search over large high dimensional multimedia datasets. Another observation obtained from this initial



(a) *Average Response Time* (b) *Average Page Access*

Figure 1. Average Response Time and Page Access with k=100, page size=64kb



(a) *0.2 million images, k=20, Page=128KB* (b) *1 million images, k=100, Page=64KB*

Figure 2. Analysis of Various CPU Cost

experimental study is that for different *kNN* queries over the same multimedia datasets, a fair amount of distance computations are repeated to some extent.

B. Overview of the Hike System

Base on the aforementioned analysis, we argue that one way to circumvent the expensive distance computations involved in high-dimensional multimedia *kNN* queries is to identify some distance metrics that are query independent and thus can be embedded into the index construction phase and maintained during index update. By paying one-time cost of pre-computation, we can minimize the amount of expensive distance computation at runtime and enjoy the performance speed-up of *kNN* query processing for many queries. This motivates us to design the Hike system, with a new index structure, DPR-Tree (Distance Precompution based R*-Tree), coupled with a Precomputed Distance based Pruning method (PDP). The architecture of Hike is illustrated as figure 3.

In Hike, we introduce two types of query independent distances that can be precomputed during the index construction and maintenance: (i) the distance between each child node and the center of their parent node, and (ii) the distance between the center of the MBR of each leaf index node and each of the data objects contained in this MBR. As to PDP, we also develop two efficient pruning techniques that utilize these precomputed distances to replace or minimize the frequent use of more costly distance pruning metrics, such as *MINDIST* or *MINMAXDIST*, and that by design remove or reduce the duplicated distance computations among different *kNN* queries. Inside Hike, there are two access approaches, sequential scan and DPR-Tree, can be utilized to fulfil the *kNN* search and obtain the results. Once got a *kNN* query, the cost estimator will employed to evaluate both I/O and computational overheads of this *kNN* search w.r.t different access methods. And the access method incur less overall overheads will be finally utilized to fulfil the *kNN* search task. The details of DPR-Tree and PDP will be presented in section IV and section V.

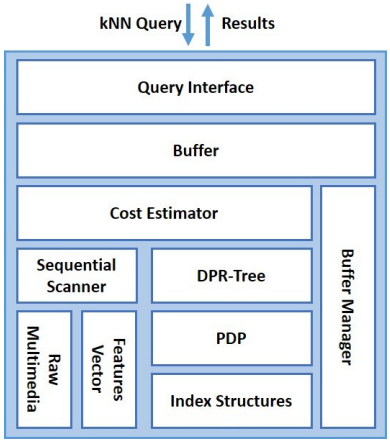
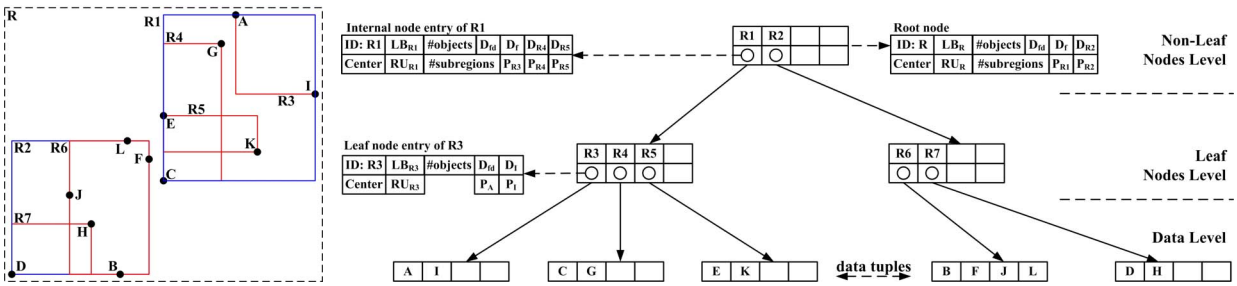


Figure 3. Overview of Hike System

IV. DPR-Tree Index Structure

A. Data Structure

DPR-tree is the built-in index structure of Hike system, it extends an R-tree index at both internal nodes and leaf nodes. Two types of distances are precomputed at the index construction phase and modified upon insertion, deletion and update. The first type is the distance between each child node and the center of the MBR of its parent index

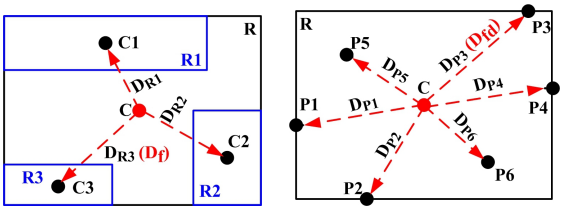


(a) *Example dataset* (b) *Structure of DPR-Tree*

Figure 4. Example dataset and the corresponding DPR-Tree structure

node. This precomputed distance is stored at each internal index node. The second type is the distance between the center of the MBR of its leaf node and each of the data objects contained in this MBR. This precomputed distance is store at the corresponding leaf node. Based on the second type of distance, we can compute the distance between the center of the MBR of an index node and its farthest data object, denoted as D_{fd}. In addition, the following distance information is also precomputed and maintained in the DPR-Tree: the *center* of a MBR (the method of obtain it is similar as in SR-Tree), the distances between the center of an internal node's MBR and the center of each of its children nodes, the distances between the center of a leaf node and each data object included in the MBR of this leaf node. Finally, for each index node R_i, the distance between the center of the MBR of R_i and its farthest data object D_{fd} is also computed and stored at node R_i. We can consider D_{fd} as the radius of the bounding sphere of R_i.

Figure 4(a) gives an example dataset and Figure 4(b) shows its corresponding DPR-Tree structure. In addition to maintain information as it in R*-tree, node R_1 also stores its center and the precomputed distance from its center to each center of its three children nodes R_3, R_4 and R_5. Similarly, leaf node R_3 stores its center, the precomputed distance from its center to each of its data objects (A and I) and the distance from its center to the farthest data object, *i.e.*, A in this example.



(a) *Internal Nodes Level* (b) *Leaf Nodes Level*

Figure 5. Precomputations in DPR-Tree

Figure 5 illustrates the computation of the two types of precomputed distances. In Figure 5(a), *R* is an internal node and includes 3 subregions: *R1*, *R2*, *R3*. *C1*, *C2* ,*C3* are the centers of their MBRs respectively, and D_{R1}, D_{R2}, D_{R3} are the precomputed distances between the *center* of R and each *sub-region center* of R: $D_{Ri} = dist(C, C_i)$ $(1 \leq i \leq 3)$. Because D_{R3} has the longest distance to the center *C* of R, compared to other subregions of R, we mark it as D_f. The precomputed distances in leaf nodes level are illustrated in Figure 5(b). This leaf node include six data objects, *P1*, ..., *P6*, their distances to the *center C* of R, $D_{P1}, \ldots, D_{P6}$, are precomputed and stored in the leaf node. D_{P3} is the data object in the MBR of R, which has the farthest distance to the center of R. At both leaf and internal node level, precomputed distances are stored in a sorted array following the descending order of the precomputed distance to the center of R, called *dists*. We use $dists_i$ to refer to the distance from R's center C to the *i*-th longest data object (if R is a leaf node) or index node object. D_{fd} or D_f is the first element in the corresponding *dists*.

In summary, an internal node *R* in a DPR-tree with degree M consists of the precomputed distances (*dists*) to its children nodes, the distance from its center to the farthest data object D_{fd} located in the MBR of R, in addition to the standard R-tree data structure, such as *ID*, MBR information (left bottom *LU* and right upper *RU*), number of included data objects ($\#objects$), the subregions ($\#suregions$) and the entries E_1, ... , E_m $(M_{min} \leq m \leq M)$, where $2 \leq M_{min} \leq \lceil M/2 \rceil$. Each entry E_i correspond to a child node of R with 4 components: range bound information LB_i and RU_i, the number of included data objects $\#object_i$ and a pointer $Pointer_i$ to the child node. Similarly, a leaf node L also consists of center *C* and the precomputed distances ($dists$) from the center of L to every data object, in addition to the standard R-tree structure.

B. Index Construction and Maintenance

In the Hike system, the index construction algorithm of a DPR-Tree is similar to that of an R-Tree with two additional operations for each index node: one for determining the position of the node center and the other for distance precomputation. The update operation in DPR-Tree is implemented by removing the old entry and then inserting the updated new entry to the DPR-Tree. The deletion algorithm

of DPR-Tree is similar to R-Tree family, and it triggers the recomputation of the precomputed distances at those index nodes which are being merged due to the deletion of a given index node.

V. DPR-Tree based kNN Search

In DPR-Tree based *kNN* search, we proposed a pruning technique that utilize the two types of built-in precomputed distances to speed up query processing. It filtering out irrelevant index nodes and data objects as early as possible while minimizing the amount of expensive distance computation and the duplicated distance computation. Concretely, the DPR-tree based *kNN* search consists of two steps:

Step 1: Find a *near-optimal* index lookup sphere S_{opt} for a given *kNN* query, then prune out those index nodes whose MBRs do not overlap with the chosen lookup sphere. By *near-optimal* we mean that the sphere is centered at Q and contains at least the top k nearest data objects. A careful selection of such index lookup sphere S_{opt} can reduce I/O access to those index nodes that are irrelevant to Q and minimize the unnecessary computations corresponding to those irrelevant index nodes and data objects.

Step 2: For the index nodes that are overlapped with the optimal sphere S_{opt}, we propose the *Precomputed Distances based Pruning (*PDP*)* method to prune irrelevant next level index nodes and irrelevant data objects to obtain the final *kNN* results of Q. The *PDP* method utilizes precomputed distances to filter out irrelevant data objects and index nodes and minimizes the demand for expensive distance computations. Meanwhile, the unnecessarily repeated computations among different *kNN* queries is also significantly reduced.

A. Finding Near-Optimal Search Region

Given a DPR-Tree index T with root node T_{root}, let H denote the height of index and assume the top L levels of index residing in memory ($1 \leq L \leq H$). Let S represent a query search sphere centered at Q, $RC(S)$ denote the set of candidate index nodes in the L-th level of index, which overlap with S and $OC(S)$ denote the number of data objects covered by $RC(S)$. Given a *kNN* query Q, we first need to use the DPR-Tree to find a *Near-Optimal* sphere S_{opt} with respect to (w.r.t.) Q from the given L levels of the index. We call S_{opt} a near-optimal index lookup range if it is centered at Q with radius r_{opt} and contains at least k data objects, namely $k \leq OC(S_{opt}) \leq \omega \cdot k$, where ω ($\omega > 1$) is a system defined parameter to guarantee that $RC(S_{opt})$ always contains enough data objects.

Concretely, given a kNN query Q, we first find the index node with smallest MBR containing Q. Then we iteratively browse the list of subregions of this index node until the list is exhausted. At each iteration, we choose one subregion and using the distance between Q and the center of this subregion as the lookup radius of the candidate sphere S and compute $RC(S)$. Then we test if $k \leq OC(S) \leq \omega \cdot k$. If yes, S is the near-optimal search sphere S_{opt} we look for and the algorithm terminates. Otherwise, we have either $OC(S) \geq \omega \cdot k$ or $OC(S) \leq k$. In the former case, S is too large w.r.t. the near-optimal search sphere, and in the later case, S is too small w.r.t. the near-optimal search sphere. Thus we return to the next round of iteration.

It should be noted that we only perform above iteration over the top L level of index structure that resident in main memory. Thus, no I/O overheads are incurred during the process of computing the *Near-Optimal* search sphere S_{opt} and $RC(S_{opt})$.

B. Pruning with Precomputed Distance

After finding the optimal search region S_{opt} with lookup radius r_{opt}, and the set of candidate $RC(S_{opt})$ for a given *kNN* query Q, we examine all the candidates in $RC(S_{opt})$ to prune out (i) those index nodes that do not contribute to the results of Q, and (ii) those data objects that are not the closest k objects with respect to the query Q. We name this DPR-tree based pruning algorithm as *PDP* (Precomputed Distances based Pruning), its sketch is show as algorithm 1, it consisting of two algorithmic components: *PDP_I* (*PDP* in Internal Nodes Level, algorithm 2) and *PDP_L* (*PDP* in Leaf Nodes Level, algorithm 3), they are devised for precomputed distanced based pruning at both internal and leaf nodes level independently.

Algorithm 1: PDP($RC(S_{opt}), Q, r_{opt}$)

```
1 kNN ← ∅;
2 foreach R ∈ RC(S_opt) do
3     if IsLeaf(R) then
4         PDP_L(RC(S_opt), R, Q, r_opt, C_R);
5     else
6         PDP_I(kNN, R, Q, r_opt, C_R);
7 Return kNN;
```

1) Internal Nodes Level Pruning:

Given a query Q and $RC(S_{opt}) = \{R_i | Overlap(S_{opt}, R_i) = true\}$, where R_i touched the optimal search region S_{opt}, the task of internal nodes level pruning is to remove those index nodes in $RC(S_{opt})$, which are irrelevant to the results of Q. Let C_R denotes the center of node R. The internal nodes level pruning algorithm *PDP_I* is sketched in Algorithm 2. For each node $R \in RC(S_{opt})$, the distance from Q to the center of R, denoted by d_{QC_R}, will be computed (line 1). Based on d_{QC_R} and the precomputed distances stored at the node R, we can construct two low cost pruning metrics: the lower and upper bound distances of $dist(Q, R_i)$, *i.e.*, $d_{low}(Q, R_i)$ and $d_{up}(Q, R_i)$, which are employed to prune the children nodes of R, *i.e.*, R_i. According to the triangle inequality theorem, we have $d_{low} = d_{QC_R} - dist(C_R, C_{R_i})$ and $d_{up} =$

d_{QC_R} + $dist(C_R, C_{R_i})$ (lines 3-4). In ***PDP_I***, $R_i.D_{fd}$ is the distance from R_i's center C_{R_i} to the farthest data object R_i, $dist(C_R, C_{R_i})$ is the distance between center of *R* and center of R_i.

Algorithm 2: PDP_I($RC(S_{opt}), R, Q, r_{opt}, C_R$)

```
1  d_QC_R ← dist(Q, C_R);
2  foreach R_i ∈ R do
3      d_low(Q, R_i) ← (d_QC_R − dist(C_R, C_R_i));
4      d_up(Q, R_i) ← (d_QC_R + dist(C_R, C_R_i));
5      if d_low(Q, R_i) > r_opt + R_i.D_fd then continue;
6      else if d_up(Q, R_i) ≤ r_opt + R_i.D_fd then
7          PutInto(R_i, RC(S_opt));
8      else
9          if MINDIST(Q, R_i) ≤ r_opt then
10             PutInto(R_i, RC(S_opt));
```

Given a child node R_i, the internal node level pruning is performed in three steps: (1) If the corresponding lower bound distance $d_{low}(Q, R_i)$ is greater than $r_{opt} + R_i.D_{fd}$, then R_i is bound to be outside of search region S_{opt} and should be filtered out (lines 5); (2) Otherwise, if the upper bound distance $d_{up}(Q, R_i)$ is less or equal to $r_{opt} + R_i.D_{fd}$, it means that R_i either is contained by or overlapped with S_{opt}. Thus R_i is kept in the result candidate set *RC* for further analysis (lines 6-7); (3) If neither of the two low-cost pruning based on precomputed distances can determine whether to keep or filter out R_i, we resort to the *MINDIST* metric to make the final decision (lines 8-10). If $MINDIST(Q, R_i)$ is less or equal to r_{opt}, then R_i overlaps with S_{opt}, thus R_i is kept in $RC(S_{opt})$ for further analysis.

Algorithm 3: PDP_L(kNN, R, Q, r_{opt}, C_R)

```
1  d_QC_R ← dist(Q, C_R);
2  foreach e ∈ R do
3      d_low(Q, e) ← (d_QC_R − dist(C_R, e));
4      if d_low(Q, e) > r_opt then continue;
5      else
6          d ← dist(Q, e);
7          if IsFull(kNN) and (d < kNN_k) then
8              ReplaceInto(e, kNN);   Sort(kNN);
9          else if IsNotFull(kNN) then
10             PutInto(e, kNN);   Sort(kNN);
```

2) Leaf Nodes Level Pruning:

In our DPR-Tree based *kNN* search, each survived leaf node R from the internal node level pruning will be examined for leaf node level pruning by algorithm ***PDP_L***. It identifies the data objects contained in R, which are either outside the search sphere S_{opt} or do not belong to the top k closest data objects w.r.t the query point *Q*.

In algorithm ***PDP_L***, first, we computer the distance between Q and the center of the node *R*, d_{QC_R}, then we examine each data object *e* contained in leaf node *R* in an iterative process and each iteration consists of 3 steps:

(1) Instead of computing *dist(Q, e)* in $O(d)$ time (d is the data dimension), we compute $d_{low}(Q, e)$, the lower bound distance between e and Q in O(1), by utilizing two distances we already have at hand and the triangle inequality theorem: $d_{low}(Q, e) = d_{QC_R} - dist(C_R, e)$. Note that $dist(C_R, e)$ is the precomputed distance stored in the leaf node *R*.

(2) We use the lower bound based pruning to filter out those data objects that are outside of S_{opt} of the query Q. If $d_{low}(Q, e)$ is greater than r_{opt}, then the data object *e* is outside of S_{opt} and cannot belong to the top k closest objects, thus it should be removed without further distance computation (line 4). Note that the saving from not conducting actual (Euclidean) distance computation can be significant when the dimensionality d is not small (O(d) v.s. O(1) in terms of time complexity).

(3) Otherwise, e is inside S_{opt}. We need to compute the actual distance between query *Q* and data object *e* to determine if e belongs to the top k closest objects to Q (lines 6-11). A sorted *kNN* candidate object queue, called *kNN* priority queue, is maintained for each query Q. Two cases are considered: (i) When the queue is full, we test if *dist(Q, e)* is less than the distance between Q and the *k*-th object in the priority queue of Q. If not, e is disregarded; and if yes, we invoke the *ReplaceInto* function to replace the k-th object with *e* (lines 7-8) and then re-sorting the objects in the priority queue of Q using the *Sort* routine. (ii) When the priority queue is not full, we simply add *e* into the queue and sort the data objects in the queue again (line 10).

VI. Experimental Evaluation

In this section, we perform two sets of experimental evaluations using the MSRA-MM 2.0 dataset of over 1 million images [5] w.r.t. compare the performance of Hike (DPR-Tree) with three popular and representative high-dimensional index algorithms, SR-Tree, A-Tree and iDistance.

Experimental Setup. MSRA-MM 2.0 dataset is a web-scale data set extracted from Microsoft's Bing image search engine, its 128D wavelet texture features and 256D RGB color histogram features are used and 1000 images are randomly selected as the query set. During the evaluation, we carefully set the variable length code of A-Tree as 6, and the number of reference points for iDistance is set to 100 to achieve good performance. We set the default page size for each index method to 32KB for 128D datasets and 64KB for 256D datasets. All experiments are conducted on a Redhat Enterprise Linux 5 server with Interl Xeon 2.4 GHz E5645

CPU and 8GB RAM. We measure the *average response time* and the overhead of *page access* for all four indices.

A. Effect of Data Size

Figure 6 shows the effect of data size on the performance of all four indexing approaches with *k* set to 100. Figures 6(b) and 6(d) measures the page access of the four indices. DPR-tree has smaller page access compared to SR-tree and iDistance, though A-Tree has significantly smaller page access in comparison and iDistance is the worst with highest page access. Figures 6(a) and 6(c) show that DPR-Tree achieves best performance in terms of response time nearly in all experiments. The only exception is for the 256D datasets with data size less than 400K. The response time of DPR-Tree in this setting is 100%-170% times faster than SR-Tree and A-Tree but slightly worse than iDistance. This is because Hike's DPR-Tree often has much larger number of reference points than iDistance.

We also observe that although the fanout of A-Tree and SR-Tree may be slightly greater than DPR-Tree, they are still slower than DPR-Tree *w.r.t* response time. There are a number of reasons:

(1) SR-Tree and A-Tree employ expensive *MINDIST* based metrics for *kNN* pruning, which has been proved to involve heavy computations overheads (recall Section III);

(2) In A-Tree based *kNN* pruning, computations of *MINDIST* require VBR (Virtual Bounding Rectangle) information of the index nodes. However, since the bounding regionsin A-Tree are approximated by their relative positions with respect to their parent's bounding region, and if a node being examined for pruning is an index node, the positions of VBRs are calculated by the costly *decode* function, which are based on the MBR of the node and the subspace codes of all children entries. Thus the *kNN* pruning in A-Tree is often computation-intensive and costly, which consequently degrades the performance of *kNN* search and making A-tree having the worst performance in response time compared to all other three approaches;

(3) The proposed DPR-Tree based *kNN* search utilized two types of precomputed distances stored in each internal and leaf nodes, which make the distance based pruning more effective, consequently the page access of DPR-Tree is optimized and less than SR-Tree.

As to iDistance, although it is more efficient in *kNN* search compared to SR-Tree and A-Tree, it has the largest page access among all methods in all experiments. This is because the iDistance based *kNN* search "sphere" is mapped to a single dimensional range in the underlying B+-tree, which expands gradually until reaching the final *kNN* result. It is known that in high-dimensional sapce, a small search sphere and sphere extension often involves many overlapped regions and objects, which require to be further disk access and pruning [3]. Therefore, iDistance based *kNN* search tends to consume large number of page accesses, especially when data size is big and data dimensionality is high.

In contrast to idistance, A-Tree incurs smallest page access because it uses relatively approximate representation of MBRs, which save lots of space in index nodes, thus A-Tree has a larger fanout and the page access is significantly smaller. Unfortunately, this disk I/O advantage is offset by the high in-memory distance computation cost (include the overheads of *decode*) of A-tree based pruning method, making A-tree having the worst performance in response time compared to other three approaches.

Even though DPR-tree has a slightly smaller fanout compared to R-tree due to the space used for storing precomputed distances at each index node, DPR-tree has showed better page accesses than iDistance and SR-tree, thanks to its precomputed distance based pruning, which is both fast and also more effective in pruning out irrelevant index nodes and data objects at early stage of the query processing.

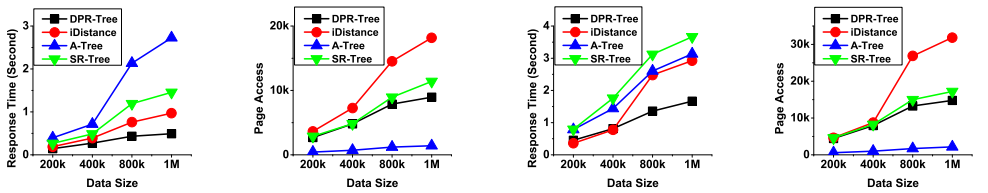
B. Effect of k

We conduct four sets of experiments to measure the effect of different *k* values on the performance of all four indexing approaches using two datasets, the 128D and 256D MSRA-MM 2.0 datasets of 1 million images.

Figures 7(b) and 7(d) measure the page access by varying k. We observe that A-Tree has consistently small I/O overheads for both 128D and 256D datasets and is less sensitive to the increase of k. DPR-tree, SR-tree and iDistance have increased I/O access cost as k increases and DPR-Tree incurs consistently less page access cost than SR-Tree and iDistance, whereas iDistance consistently shows the highest number of page access with varying k. Figure 7(a) and Figure 7(c) show the average response time by varying k from 20, 50 to 100 for 128D and 256D dataset respectively. As *k* increases, the average response time of all four indexing approaches also increases. The DPR-Tree based approach to *kNN* search consistently outperforms the other three methods for both 128D and 256D datasets with varying k values, thanks to its efficient precomputed distance based pruning techniques.

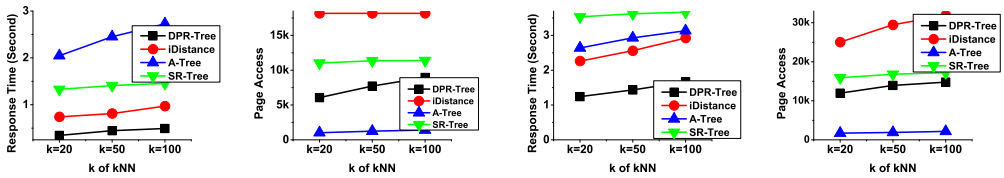
VII. Conclusions

Efficient search of multimedia content is one of the main Internet computing capabilities for many big data applications and systems. In this paper we have presented Hike, a high performance multimedia *kNN* search system, it integrated the novel precomputed distances based index structure DPR-Tree, coupled with a fast precomputed distances based kNN search method *PDP* to speed up the high-dimensional *kNN* search over large multimedia datasets. Our experimental evaluation have shown that Hike based *kNN* search offers substantial performance enhancement and significantly outperforms the existing state-of-the-art high-dimensional indexing approaches.



(a) *128D, Response time* (b) *128D, Page access* (c) *256D, Response time* (d) *256D, Page access*

Figure 6. kNN search comparison w.r.t data size on 128D and 256D datasets (*k=100*)



(a) *128D, Response time* (b) *128D, Page access* (c) *256D, Response time* (d) *256D, Page access*

Figure 7. kNN search comparison w.r.t k on 1 million 128D and 256D datasets

ACKNOWLEDGMENT

This work was partially supported by Science and Technology Fund of Guizhou Province (No. J [2013]2099), Fund of National Natural Science Founda-tion of China (No. 61462012), and The Major Applied Basic Research Program of Guizhou Province (NO.JZ20142001-05).

REFERENCES

[1] M. S. Lew, N. Sebe, C. Djeraba, and R. Jain, "Content-based multimedia information retrieval: State of the art and challenges," *TOMCCAP*, vol. 2, no. 1, pp. 1–19, 2006.

[2] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Comput. Surv.*, vol. 33, no. 3, 2001.

[3] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[4] T. Skopal, J. Lokoc, and B. Bustos, "D-Cache: Universal distance cache for metric access methods," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 5, pp. 868–881, 2012.

[5] H. Li, M. Wang, and X.-S. Hua, "MSRA-MM 2.0: A large-scale web multimedia dataset," in *ICDM Workshops*, 2009, pp. 164–169.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree : An index structure for high-dimensional data," in *VLDB*, 1996, pp. 28–39.

[7] D. A. White and R. Jain, "Similarity indexing with the SS-tree," in *ICDE*, 1996, pp. 516–523.

[8] N. Katayama and S. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," in *SIGMOD*, 1997, pp. 369–380.

[9] N. G. Colossi and M. A. Nascimento, "Benchmarking access structures for the similarity retrieval of high-dimensional multimedia data," in *IEEE ICME (II)*, 2000, pp. 1215–1218.

[10] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 426–435.

[11] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive b^+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.

[12] Z. Huang, H. Shen, J. Liu, and X. Zhou, "Effective data co-reduction for multimedia similarity search," in *SIGMOD*, 2011.

[13] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, "DSH: data sensitive hashing for high-dimensional k-nnsearch," in *SIGMOD*, 2014, pp. 1127–1138.

[14] R. Zhang, B. C. Ooi, and K.-L. Tan, "Making the pyramid technique robust to query types and workloads," in *ICDE*, 2004, pp. 313–324.

[15] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, "The A-tree: An index structure for high-dimensional spaces using relative approximation," in *VLDB*, 2000, pp. 516–526.

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

[17] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, 1999.

[18] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, 1995, pp. 71–79.