MLR-Index: An Index Structure for Fast and Scalable Similarity Search in High Dimensions*

Rahul Malik, Sangkyum Kim, Xin Jin, Chandrasekar Ramachandran, Jiawei Han, Indranil Gupta, and Klara Nahrstedt

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
{rmalik4,kim71,xinjin3,cramach2,hanj,indy,klara}@illinois.edu

Abstract. High-dimensional indexing has been very popularly used for performing similarity search over various data types such as multimedia (audio/image/video) databases, document collections, time-series data, sensor data and scientific databases. Because of the *curse of dimensionality*, it is already known that well-known data structures like kd-tree, R-tree, and M-tree suffer in their performance over high-dimensional data space which is inferior to a brute-force approach *linear scan*. In this paper, we focus on an approximate nearest neighbor search for two different types of queries: *r-Range search* and *k-NN search*. Adapting a novel concept of a *ring* structure, we define a new index structure *MLR-Index* (**M**ulti-**L**ayer **R**ing-based **Index**) in a metric space and propose time and space efficient algorithms with high accuracy. Evaluations through comprehensive experiments comparing with the best-known high-dimensional indexing method *LSH* show that our approach is faster for a similar accuracy, and shows higher accuracy for a similar response time than *LSH*.

1 Introduction

A similarity search finds a small set of objects near a given query. Here, similarity between objects is often measured by their features, which are points in a high-dimensional vector space. Efficient similarity searches have become more and more important in various domains such as multimedia (audio/image/video), web documents, time-series, sensor and scientific areas because of the rapid mounting of these datasets and the increasing desires of modern users for fast and scalable systems.

Usually, feature vectors of these complex datasets lie in a high-dimensional space which are quite often bigger than several hundred dimensions. But for sufficiently high dimensions, existing index structures degrade to a linear scan approach which compares a query to each point of the database, that is not

* The work was supported in part by the U.S. NSF grants CNS-0520182, IIS-0513678, IIS-0842769, CAREER CNS-0448246, ITR CMS-0427089 and Air Force Office of Scientific Research MURI award FA9550-08-1-0265. Any opinions, findings, and conclusions expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

M. Winslett (Ed.): SSDBM 2009, LNCS 5566, pp. 167–184, 2009.

scalable [1]. Fortunately, in many cases, it is not necessary to insist on exact answers [2], and recent research has focused on this direction: searching approximate nearest neighbors (or *ANN*) [3,4,5,6]. But even for this relaxed problem, *ANN*, there exist fundamental challenges to achieve time and space efficiency with high accuracy performance.

An interesting approach *VQ-Index* [3] achieved an efficient compression by using a vector quantization technique which resulted in a smaller search space. Quantization was conducted by partitioning the space into regions and assigning a representative to each region so that data points were mapped to the representative of the region they fall into. One problem is that it is still had to perform a linear scan over the representatives repeatedly to retrieve similar representatives and then again over the candidates from the region of the resulted similar representatives, which made the searching procedure inefficient.

The best-known solution *LSH* [4,6] for *ANN* problem shows fast response time for a given query, but it needs to use multiple (sometimes over hundreds of) hash tables to achieve high accuracy, while a linear scan takes linear time but does not require any additional memory usage. Our object is to find a solution which shows a similar or better time efficiency to *LSH* using less memory.

In this paper, based on the vector quantization idea, but instead of conducting a linear scan over the representatives of the regions, we utilize a novel index structure *MLR-Index* (**M**ulti-**L**ayer **R**ing-based **Index**) which adapts a ring structure first introduced in a distributed network context [7]. Each representative (named *search node*) keeps track of $O(\log N)$ peers and organizes them into a set of concentric rings centered on itself with exponentially increasing radii. This multi-layer ring structure provides not only the immediate vicinity information but also outer pointers to remote regions. Instead of linear performance of simple vector quantization, it can find the nearest search node in $O(\log N)$ steps. Here, N is the number of search nodes not the number of all data objects. In our experiments, *MLR-Index* based similarity search algorithms showed several times faster response time than *LSH* for similar accuracy performance.

The followings are the contributions of this paper.

- We design a novel index structure *MLR-Index* that fits in a high-dimensional space with quantized vectors.
- We propose approximate algorithms which are time and space efficient with high accuracy for two different nearest neighbor search problems. We also propose even faster methods by using an additional data structure with similar accuracy.
- We develop scalable hierarchical search algorithms to maximize performances for dense regions which are clustered in multi-levels.
- We perform extensive experiments on real datasets comparing with the *state-of-the-art* high-dimensional indexing algorithm *LSH* [4,6] which show the superior performance of *MLR-Index*.

The remainder of the paper is structured as follows. Section 2 describes related works on high-dimensional similarity search. Section 3 defines terms and problems of this paper, and introduces concepts of the search node and its multi-layer

ring structure. Based on these concepts, an efficient method for finding nearest search node is described in Sect.4. Section 5 proposes the basic similarity search algorithms. Section 6 describes more efficient algorithms using m-NS structure. In Sect.7, a hierarchical search method for a dense region is proposed to utilize a multi-level clustered dataset. Experimental analyses are described in Sect.8. We conclude our work in Sect.9.

2 Related Works

Previously, two different types of data structures were proposed for efficiently indexing massive data objects in a vector space. Space partitioning idea was adapted for *grid file* [8], *K-D-B-tree* [9] and *quadtree* [10] that divided the data space along predetermined lines. Data-partitioning index trees such as *R-tree* [11], *R+-tree* [12], *R*-tree* [13], *X-tree* [14], *SR-tree* [15], *M-tree* [16], *TV-tree* [17] and *hB-tree* [18] divided the data space based on the distribution of data objects. These indexing methods worked well for a low-dimensional space, but most of them showed worse performance in higher-dimensional datasets [1,2,19].

VA-file [1] was the first indexing scheme that claimed to show better performance than brute-force linear scan in a high-dimensional similarity search. It tried to use compression techniques to improve the query performance. Since it quantized each dimension separately, the resulting regions became rectangular hyperboxes, which resulted in only a suboptimal partition of the data space. *VQ-Index* [3] tried to fix this problem by maintaining complex polytopes which represented data more accurately.

Recently, *LSH* [4,6] has become one of the best-known algorithms for a high-dimensional similarity search. It used a family of locality sensitive hash functions that hashed nearby objects into the same bucket with a high probability. For a given query, the bucket where it was hashed became the candidate set of similar objects. To achieve high search accuracy, *LSH* needed to use multiple (sometimes hundreds of) hash tables to generate a good candidate set.

3 Preliminary

In this section, we define two types of nearest neighbor search problems in a formal way, and introduce novel concepts of a search node and its *ring* structure that is used to find the nearest search node.

3.1 Problem Settings

We first define several terms that are commonly used in this paper. Euclidean distance is used as a metric in the dataset, but any other metric can be used for the following definitions.

Definition 1. *Define $\mathcal{V}$ to be a high-dimensional vector space $\mathbb{R}^d$ where $d \geq 20$ and $\mathcal{D}$ to be a finite set of data points where $\mathcal{D} \subset \mathcal{V}$. Define $dist(p, q)$ to be the Euclidean distance between two points p and q in $\mathcal{V}$. Define $B_p(r)$ to be a ball with radius r centered at p in $\mathcal{V}$.*

We assume the input data is already well-formatted as a high-dimensional vector space. (Developing better ways to extract features from a dataset is out of the scope of this paper.) By doing this, we are able to apply our algorithms to any kind of high-dimensional dataset such as scientific data and multimedia data which can be expressed as a high-dimensional vector space $\mathcal{D}$ defined in Def 1. Note that a query q is a point in $\mathcal{V}$, and a data point is a point in $\mathcal{D}$. That is, it is possible that a query q might be a point which is not a data point.

Now we state our two main problems in this paper. These two problems look similar, but we need to develop different solutions for each problem because of the efficiency issue caused by the large size of the data set. Traditionally, the solution of the r-range search problem has been used to solve k-nearest neighbor (or k-NN) problem, which used bigger r values repeatedly until at least k candidates are obtained and then sorted them to find the k nearest neighbors.

Problem 1. (***r*-Range Search**) Given a radius r and a query q, find all data points of $\mathcal{D}$ that reside in $B_q(r)$.

Problem 2. (***k*-NN Search**) Given a constant k and a query q, find k nearest data points of q in $\mathcal{D}$.

3.2 Search Node

In $\mathcal{V}$, we choose N points, called *search nodes*, that facilitates an efficient nearest neighbor search. Making all data points in $\mathcal{D}$ search nodes will cause a memory issue since each search node entails additional structures explained in Section 3.3. In this paper, we partition the data set $\mathcal{D}$ into N clusters and define *search nodes* as their centers. The radius of a cluster will be used to prune the search space of the nearest neighbors.

Definition 2. *Let C be a cluster of data points in $\mathcal{D}$ and p be its center. We say p is a* search node*. Denote $C(p)$ to be C and $\mathcal{S}$ to be a set of all search nodes. Define* radius(C(p)) *as the maximum distance between the center p of the cluster $C(p)$ and the data points in $C(p)$. That is, $radius(C(p)) = \max(\{dist(p,q)|q \in C(p)\})$. Define $size(C(p))$ as the number of data points in a cluster $C(p)$.*

There is a trade-off between the size of the cluster and the performance. Small sized clusters imply a small number of candidates that need to be checked or sorted which leads to faster response time. To make clusters small, we usually need a big number of clusters which requires more computations to conduct clustering algorithms. This burden appears only once as a preprocessing step, so we prefer small sized clusters for better performance. To achieve extreme performance, we apply multi-level clustering for big sized dense clusters since we might have dense clusters even for a large number of clusters. We utilize this hierarchical structure in Section 7 to achieve better performance.

To find the nearest neighbors of a query point, we conduct a search on $\mathcal{S}$ first, and then refine the search to the data points. A search node retains a ring structure for this procedure, and sometimes we use additional data structure, a list of the m nearest search nodes, for a faster search.

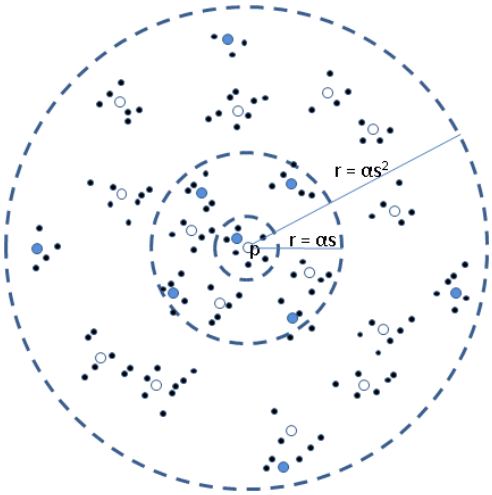


Fig. 1. Multi-layered ring structure with exponentially increasing radii

Definition 3. *For a search node p, we define* m-NS(p) *as a list of m nearest search nodes of p.*

3.3 MLR-Index

We adapt the concept of multi-layered ring structure from [7] to efficiently find the nearest search node (or its cluster) of a given query q. Each search node keeps track of a small, fixed number of other search nodes in $\mathcal{V}$ and organizes the list of peers into concentric, non-overlapping rings. This ring structure favors nearby neighbors by providing information on search nodes in the immediate vicinity. Moreover, its exponentially increasing ring radii enable a sufficient number of out-pointers to far away regions to facilitate rapid search.

Definition 4. *For a search node p, the i-th ring has inner radius* $r_i = \alpha s^{i-1}$ *and outer radius* $R_i = \alpha s^i$ *for* $0 < i < i^*$ *where* i^* *is a user defined parameter. For the innermost ring with* $i = 0$*, we define* $r_0 = 0$ *and* $R_0 = \alpha$*. All rings with* $i \geq i^*$ *are collapsed into a single outermost ring with* $r_{i^*} = \alpha s^{i^*}$ *and* $R_{i^*} = \infty$*.*

Each ring defined above keeps track of at most M search nodes in it. If there are more than M search nodes, then the subset of M search nodes that forms the polytope with the largest hypervolume are selected. Nodes that are geographically diverse instead of clustered together enable a node to forward a query to a larger region.

The number of search nodes per ring, M, represents the trade-off between performance and overhead. A large M increases the accuracy and the search speed by providing better query routing, but entails more memory at the same time. In [7], it is proved that by the use of a multi-layer ring with $M = O(log(N))$, we can find the nearest search node in $O(log(N))$ steps where N is the number of search nodes in $\mathcal{V}$. Note that N is much smaller than the total number of data points in $\mathcal{D}$.

Algorithm *FNSN*
input: A query point q
output: Nearest search node p of q
begin
1. randomly choose any search node p
2. $d \leftarrow dist(p, q)$, $\widetilde{d} \leftarrow \infty$
3. **while** $d < \widetilde{d}$
4. $\widetilde{d} \leftarrow d$
5. $d \leftarrow$ the minimum distance between the search nodes in $Ring(p)$ and q
6. **if** $d < \widetilde{d}$
7. $p \leftarrow$ the search node with the minimum distance d
8. output p

end

Fig. 2. Finding Nearest Search Node

Finally, we define *MLR-Index* which are composed of all structures mentioned above.

Definition 5. *We define* MLR-Index *to be a set of all search nodes together with their ring structures and radius values. For a search node p, we denote* MLR-Index(p) *to be p's ring structure together with* $radius(C(p))$. *Optionally, we add m-NS structure of each search node into* MLR-Index *to improve time efficiency.*

The total memory usage of *MLR-Index* is $N \times ((i^* + 1) \times M + m) = O(Nlog(N))$ for $M = O(log(N))$. Since $N \ll |\mathcal{D}|$, we can claim the space efficiency of *MLR-Index*.

4 Finding Nearest Search Node

To answer a range query or k-NN search, we begin with finding the nearest search node p of a query point q. The *FNSN* algorithm described in Fig.2 shows the procedure of finding the nearest search node. First, we randomly choose a search node p and measure the distance between p and q. Then, we compute the minimum distance between q and the search nodes in *MLR-Index* of p. If we find a closer search node, then find the minimum distance between its index structure and q. This procedure is repeated until we cannot find any closer search node.

We extend the *FNSN* algorithm with an additional input parameter A which retrieves the nearest search node except the nodes in A. In this way, we can find the second nearest search node $\widetilde{p}$ of q by applying *FNSN* algorithm twice: $p \leftarrow$ *FNSN*(q) and $\widetilde{p} \leftarrow$ *FNSN*$(q, \{p\})$. Similarly, repeating the procedure described above m times, we can find m-th nearest search node of q.

A theoretical bound that guarantees logarithmic performance of this procedure is shown in [7]. For N search nodes, the running time of *FNSN* becomes

$O(log(N))$ if each ring retains $M = O(log(N))$ search nodes, and sometimes can assure to find even the exact nearest search node if several conditions are satisfied. In the experiments, we show that a small M is sufficient for a fast response with high accuracy.

5 Basic Similarity Search

In this section, we propose two methods *BRS1* and *BKS1* for similarity search mainly using *MLR-Index* and the *FNSN* algorithm. The main idea is to execute *FNSN* on *MLR-Index* repeatedly until there exists no more cluster that intersects with the candidate solution set. Due to the logarithmic performance of *FNSN*, these basic algorithms work efficiently for a small number of iterations. Maximizing the candidate set (in a reasonable way for fast response time) ensures high accuracy performance, but later in Sect.6 we will discuss using an additional data structure that facilitates higher time efficiency by reducing the size of the candidate set. The details of *BRS1* and *BKS1* are shown in the following subsections.

5.1 *r*-Range Search

The *BRS1* algorithm described in Fig.3 is the basic algorithm for the r-range search problem that uses *MLR-Index*. It sequentially finds next nearest search nodes until there is no more search node whose cluster intersects with $B_q(r)$ for a given query q and a query range r. The union of the clusters which intersect with r-range search area $B_q(r)$ becomes a set $\widetilde{C}$ whose data points form a candidate set of the range search query. Since each cluster has a small number of data points as mentioned in Sect.3.2, the candidate set is also in a small size which enables efficient computation.

Algorithm *BRS1*
input: A query point q and a range parameter r
output: Data points within $B_q(r)$
begin
1. $p \leftarrow FNSN(q)$
2. **while** $dist(p, q) - radius(C(p)) \leq r$
3. $\quad \widetilde{C} \leftarrow \widetilde{C} \cup C(p)$
4. $\quad S \leftarrow S \cup \{p\}$
5. $\quad p \leftarrow FNSN(q, S)$
6. output points in $\widetilde{C}$ which lies within $B_q(r)$

end

Fig. 3. Basic r-Range Search Using Next Nearest Search Node

The following theorem shows the theoretical legitimacy of this approach.

Theorem 1. *Let q be a query and p_i be the i-th nearest search node of q. Let R be the maximum radius of all clusters. Suppose j is the smallest number such that $dist(q, p_j)$ becomes bigger than $R + r$. Then, there exists no i that is bigger than j such that any data point in $C(p_i)$ lies within $B_q(r)$.*

Proof. Proof by contradiction. Suppose that $i > j$ and there exists a data point a in $C(p_i) \cap B_q(r)$. Then, $r \geq dist(q, a) \geq dist(q, p_i) - R \geq dist(q, p_j) - R > r$. This results in a contradiction. Hence, proved.

In the experiments, we found that it was enough to use $radius(C(p))$ instead of the maximum radius R of all clusters. Since usually a small number of search nodes near the query point is sufficient for high accuracy, we might conduct *FNSN* for a fixed number of iterations for a reasonable accuracy with faster response time. Note that, we iteratively find the next nearest search nodes, not retaining them into the memory in advance.

5.2 *k*-NN Search

The *BKS1* algorithm described in Fig.4 is the basic algorithm for the k-NN search problem that uses *MLR-Index* structure. As we did for range search, this method

---

Algorithm *BKS1*
input: A query point q and k
output: k-NN data points of q
begin
1. $p \leftarrow FNSN(q)$
2. **if** $|C(p)| \geq k$
3. $d \leftarrow$ dist(q,k^{th} nearest point of q in C(p))
4. $\widetilde{C} \leftarrow$ apply *BRS1* with $r = d$
5. **else**
6. $A \leftarrow C(p)$
7. $S \leftarrow S \cup \{p\}$
8. **while** $|A| < k$
9. $\widetilde{p} \leftarrow FNSN(q, S)$
10. $S \leftarrow S \cup \{\widetilde{p}\}$
11. $A \leftarrow C(\widetilde{p})$
12. $d \leftarrow$ dist(q,k^{th} nearest point of q in A)
13. $\widetilde{C} \leftarrow$ apply *BRS1* with $r = d$
14. output k-NN points of q within $\widetilde{C}$
end

Fig. 4. Basic k-NN Search Using Next Nearest Search Node

also executes the *FNSN* algorithm repeatedly to find nearest search nodes of a query q. First, we find the nearest search node p by applying *FNSN*. If $C(p)$ contains at least k points, then find the k^{th} nearest point of q within $C(p)$ and draw a ball $B_q(r)$ considering its distance as a radius r. Then apply the *BRS1* method with the resulted radius r and find the k nearest data points among them.

If $C(p)$ contains less than k points, then execute the *FNSN* algorithm until the union A of the clusters of the next nearest search nodes of q contains at least k points. Then find the k^{th} nearest point of q within A and draw a ball $B_q(r)$ considering its distance as a radius r. Then apply the *BRS1* method with the resulted radius r and find the k nearest data points among them.

The following theorem shows the theoretical legitimacy of this approach.

Theorem 2. *Let q be a query and p_i be the i-th nearest search node of q. Let R be the maximum radius of all clusters. Let j be the smallest number such that $|\bigcup_{i \le j} C(p_i)|$ contains at least k data points. Let o be the k^{th} nearest data point of q in $|\bigcup_{i \le j} C(p_i)|$ and d be the distance between o and q. Suppose that m is the smallest number such that $dist(q, p_m)$ becomes bigger than $R + d$. Then, there exists no i which is bigger than m such that $C(p_i)$ contains any of the k nearest data points of q.*

Proof. Proof by contradiction. Suppose $i > m$ and that there exists a data point $\hat{o}$ in $C(p_i)$ which is one of k-NN of q. Then, $dist(q, \hat{o}) \geq dist(q, p_m) - R > d$. But $B_q(d)$ already contains at least k data points. This results in a contradiction. Hence, proved.

In *BKS1*, we do not have to retain additional structures like m-NS that require more memory usage. The running time of *FNSN* is $O(log(N))$ and it becomes more efficient to find nearest search nodes in a series, since they are not far away from each other. Usually, we need to find only a few next nearest search nodes to achieve high accuracy. Note that, similar to *BRS1*, we found that $radius(C(p))$ was enough to use instead of the maximum radius R of all clusters in the experiments.

6 *m*-NS Based Similarity Search

In this section, we propose two methods *BRS2* and *BKS2* for similarity search which uses additional structure m-NS (***m*-N**earest **S**earch Nodes) for *MLR-Index*. The main idea is to expand the search space by utilizing m-NS instead of repeatedly executing *FNSN* on *MLR-Index* to achieve faster response time. Even though *FNSN* shows logarithmic performance, a large number of *FNSN* iterations must be a bottleneck of the performance of *BRS1* and *BKS1*. The details of *BRS2* and *BKS2* are shown in the following subsections.

6.1 *r*-Range Search

Based on the heuristic that nearest search nodes of a query point q and those of q's nearest search node p are quite common, we add m-NS list of each search

Algorithm *BRS2*
input: A query point q and a range parameter r
output: Data points within $B_q(r)$
begin
1. $p \leftarrow FNSN(q)$
2. **for** $\widetilde{p} \in m\text{-}NS$
3. **if** $dist(\widetilde{p}, q) - radius(C(\widetilde{p})) < r$
4. $\widetilde{C} \leftarrow \widetilde{C} \cup C(\widetilde{p})$
5. output points in $\widetilde{C}$ which lies within $B_q(r)$

end

Fig. 5. Basic r-Range Search Using m-NS

node together into *MLR-Index* for a faster search with similar quality of results. The details of this method is described in Fig 5. We first use *FNSN* algorithm to find the nearest search node p of a given query q. Then, we find the search nodes among m-NS whose clusters intersect with $B_q(r)$. By doing so, we get a set $\widetilde{C}$ of union of those clusters, whose data points form a candidate set of the range query. Finally, the desired points are found within $\widetilde{C}$. In this way, we can reduce the search space from the whole dataset to a small number of clusters. This algorithm *BRS2* uses an additional precomputed data structure m-NS for each node to get a faster response with similar accuracy performance to *BRS1*.

6.2 *k*-NN Search

As we did for a range search problem, we can also achieve faster response time than *BKS1* by use of an additional data structure m-NS of each search node. The main procedures of this algorithm *BKS2* are described in Fig 6. At first, we execute the *FNSN* algorithm to find the nearest search node p from a query point q. If $C(p)$ contains at least k points, then find the k^{th} nearest point of q within $C(p)$ and draw a ball $B_q(r)$ considering its distance as a radius r. Include the clusters of m-NS(p) search nodes which intersect with $B_q(r)$ into the candidate set $\widetilde{C}$.

If $C(p)$ contains less than k points but the union of clusters A of $\widetilde{m}$-NS(p) contains at least k points ($\widetilde{m} \leq m$), then find the k^{th} nearest point of q within A and draw a ball $B_q(r)$ considering its distance as a radius r. Include the clusters of m-NS(p) search nodes which intersect with $B_q(r)$ into the candidate set $\widetilde{C}$.

Sometimes, we might have a case where even the union of the clusters of m-NS(p) contains less than k points. In this case, since we cannot find k-NN using the clusters of m-NS, we use the *FNSN* algorithm to find more clusters near the query point q. Only in this case, we cannot utilize a precomputed m-NS list of nearest search nodes. But since this case rarely happens, we still can use this algorithm for the purpose of higher efficiency.

Algorithm *BKS2*
input: A query point q and k
output: k-NN data points of q
begin
1. $p \leftarrow FNSN(q)$
2. **if** $|C(p)| \geq k$
3. $\widetilde{C} \leftarrow k$-NN of q within C(p)
4. $d \leftarrow$ dist(q,k^{th} nearest point of q in C(p))
5. **for** $\widetilde{p} \in m\text{-}NS(p)$
6. **if** $dist(\widetilde{p}, q) - radius(C(\widetilde{p})) < d$
7. $\widetilde{C} \leftarrow \widetilde{C} \cup C(\widetilde{p})$
8. **else if** |clusters of $m\text{-}NS(p)| < k$
9. apply *FNSN* repeatedly until we get $\widetilde{m}$ such that |clusters of $\widetilde{m}\text{-}NS(p)| \geq k$
10. $\widetilde{C} \leftarrow$ union of clusters of $\widetilde{m}\text{-}NS(p)$
11. **else**
12. find smallest $\widetilde{m}$ such that |clusters of $\widetilde{m}\text{-}NS(p)| \geq k$
13. $A \leftarrow$ union of clusters of $\widetilde{m}\text{-}NS(p)$
14. $\widetilde{C} \leftarrow k$-NN of q within A
15. $d \leftarrow$ dist(q,k^{th} nearest point of q in A)
16. **for** $\widetilde{p} \in m\text{-}NS(p) - \widetilde{m}\text{-}NS(p)$
17. **if** $dist(\widetilde{p}, q) - radius(C(\widetilde{p})) < d$
18. $\widetilde{C} \leftarrow \widetilde{C} \cup C(\widetilde{p})$
19. output k-NN points of q within $\widetilde{C}$
end

Fig. 6. Basic k-NN Search Using m-NS

7 Hierarchical Similarity Search

As previously mentioned, we do the partitioning on the dataset as the vector quantization for efficient data compression. If a cluster is dense, then we may partition it again into smaller clusters to enable more efficient search. In this section, we provide algorithms that utilize this kind of hierarchical property. Note that we extend the *FNSN* algorithm to work in a restricted domain to enable searching the nearest node within a partition.

7.1 r-Range Search

To answer a hierarchical range query, we apply the basic range query algorithms levelwise. We first find high level clusters that intersect with a query range $B_q(r)$, and for the resulted clusters we perform basic range query algorithms again to retrieve data points within radius r of a query point q. In this way, we can perform the search efficiently even though the first level of clusters contain many data points. That is, instead of performing a brute-force linear scan on a

Algorithm *HRS*
input: A query point q and a range parameter r
output: Data points within $B_q(r)$
begin
1. find high level clusters that intersects with $B_q(r)$ using BRS
2. **for** each high level cluster C found above
3. apply BRS on C
4. output data points within $B_q(r)$
end

Fig. 7. Hierarchical r-Range Search

Algorithm *HKS*
input: A query point q and k
output: k-NN data points of q
begin
1. $p \leftarrow FNSN(q)$
2. apply *BKS* within C(p)
3. $d \leftarrow$ dist(q,k^{th} nearest point of q in C(p))
4. apply *HRS* with parameter d
5. output k-NN points of q
end

Fig. 8. Hierarchical k-NN Search

large number of data points of a first level cluster, we do a logarithmic search on its smaller partitions and perform a linear scan on a reduced set of candidates which minimizes the computational cost.

7.2 *k*-NN Search

To search the k nearest neighbors of a given query q in a hierarchical structure, we make use of all algorithms we developed so far. We first find the nearest search node p at the first level. Then we apply *BKS* within the first-level cluster $C(p)$ and get the k nearest points of q from it. We calculate the k-th minimum distance d from q to points in $C(p)$, and based on d we apply our hierarchical range search algorithm *HRS* to the entire space except $C(p)$. Finally, we combine the results and output the k nearest data points of q by sorting them in ascending order based on the distance between q and them.

Note that we can apply any version of the *BRS* algorithms in the hierarchical search algorithms *HRS* and *HKS*.

8 Experiments

This section presents extensive experimental evaluations on our proposed algorithms. Experiments were conducted on a computer with the following configuration: CPU Intel Pentium 4 2.26GHz, Memory 2GB and OS Redhat Linux 2.6.18 Kernel. Two standard high-dimensional datasets, *Corel* [20] and *CovType* [21], were used for the experiments from *UCI* repository of machine learning databases. The *Corel* dataset contains image features extracted from a Corel image collection. The four sets of features are based on the color histogram, color histogram layout, color moments, and co-occurence. *CovType* was originally proposed to predict forest cover type from cartographic variables which were derived from US Geological Survey (USGS) and US Forest Service (USFS) data. Table 1 summarizes the features of two datasets.

Table 1. Statistics for datasets used in experiments

Dataset	Size	Dimensions	Application
Corel	66,616	89	Content based image retrieval
ConType	581,012	54	Finding similar forest cover type

To evaluate time performance, we measured the average execution time per query. To evaluate retrieval quality, we used the recall measure. Linear scan was used for the exact solutions of recall measure. We did not show precision and error rate results in this paper because they showed similar tendencies with recall. We used linear scan and the most recent version of *LSH* [6] for the comparisons with our proposed algorithms. *LSH* has a parameter to control the speed and the quality, and we set it as 0.5 in our experiments as higher query speed results in lower retrieval quality.

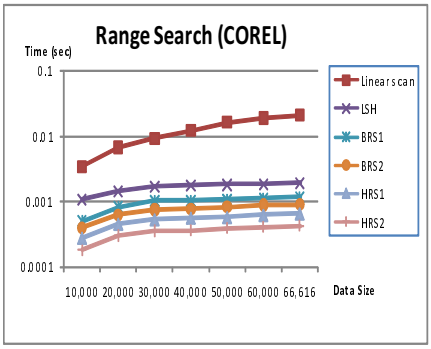
8.1 r-Range Search

Fig.9 shows the average query time of r-range search on two different datasets *Corel* and *CovType*. Our algorithms show discriminant performance over linear scan and *LSH*. Note that the vertical axes in Fig.9 are in logarithmic scales.

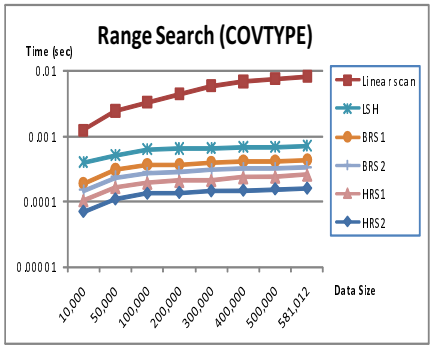
The following experiments used *Corel* dataset with $size = 66616$, $r = 3$, $m = 3$, $i^* = 5$, $M = 20$, and $N = 2000$ by default.

Fig.10 shows the average query times and the corresponding recall values for various r values. For a small query range r, most of the algorithms show similar performances, but once r becomes bigger, our proposed algorithms show better performances than linear scan and *LSH*. It is noticeable that *LSH* showed slower query time and lower recall than our algorithms for each r value.

Fig.11(a) and Fig.11(b) show the average query times and the corresponding recall values for various M values. Here, M is the number of search nodes per ring. If the ring retains more search nodes, it must show faster and more accurate results, which turned out to be true.

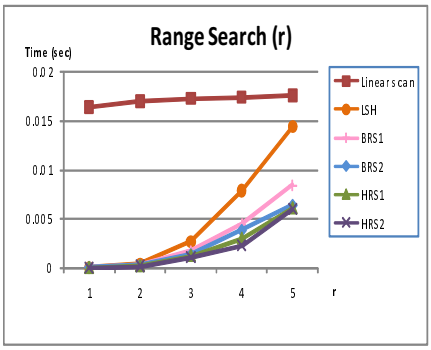


(a) Range search running time for COREL Dataset

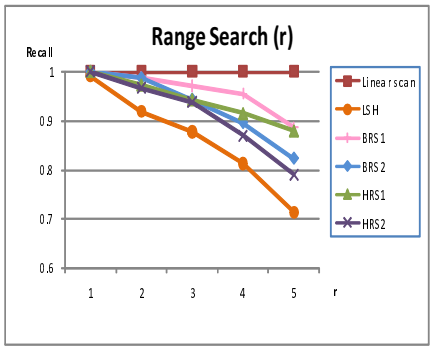


(b) Range search running time for COVTYPE Dataset

Fig. 9. Range search running time for various datasets



(a) Range search running time for various r values



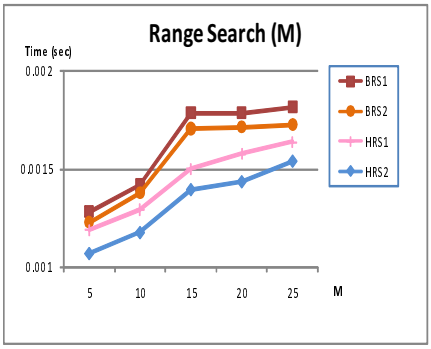
(b) Range search recall for various r values

Fig. 10. Range search for various r values

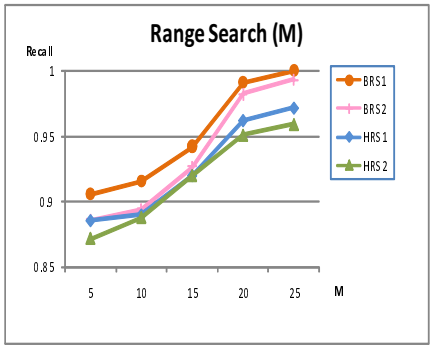
Fig.11(c) and Fig.11(d) show the average query times and the corresponding recall values for various N values. Here, N is the number of clusters, or total number of search nodes in a dataset. As the number of clusters becomes bigger, the size of each cluster becomes smaller which leads to higher accuracy but a slower response time because it needs to search more clusters.

For faster response time, we developed m-NS structure which is used in *BRS2* and *HRS2*. Fig.11(e) and Fig.11(f) show that m-NS structure improves not only the response time but also the accuracy of the algorithms. Retaining 2-NS showed higher than 96% of accuracy for both methods.

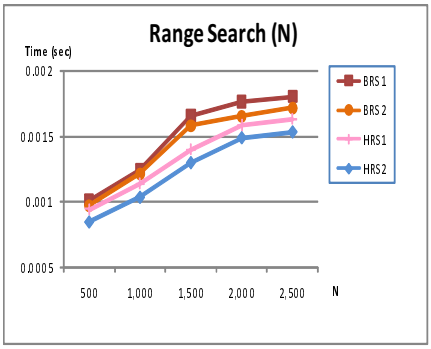
To achieve higher performance, we can increase our two parameters M and m, but it also causes a bigger amount of memory usage.



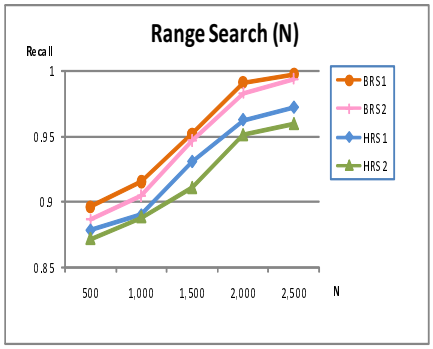
(a) Range search running time for various M values



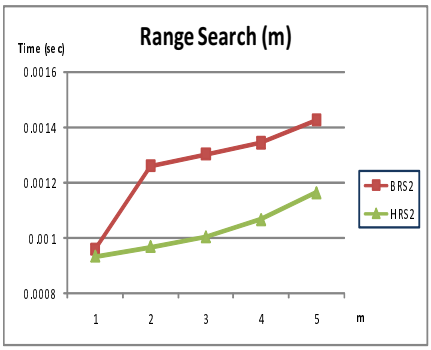
(b) Range search recall for various M values



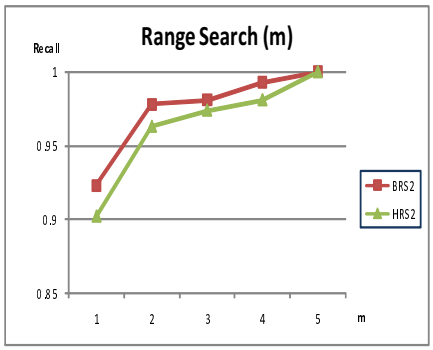
(c) Range search running time for various N values



(d) Range search recall for various N values



(e) Range search running time for various m values

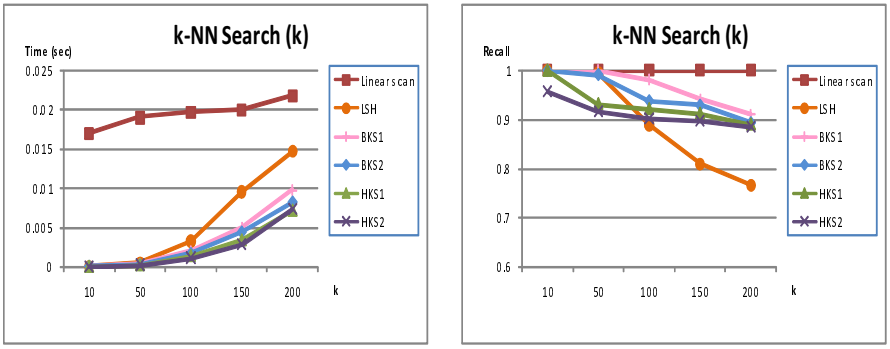


(f) Range search recall for various m values

Fig. 11. Range search for various system variables

8.2 k-NN Search

The k-NN search experiments were done by randomly choosing 100 samples from each dataset. Because of the page limit, we only analyze several results over different k values in this section, since k-NN search showed similar tendencies with r-range search for various settings of our system parameters.



(a) k-NN search running time for various k values

(b) k-NN search recall for various k values

Fig. 12. k-NN search for various k values

Fig.12 shows the average query times and the corresponding recall values for various k values. For a small number of $k \leq 50$, our proposed algorithms show comparable performances to *LSH*, but once k becomes bigger, our algorithms maintain high accuracy in a stable way while *LSH* dramatically drops down its accuracy.

Overall, we observe the following results for both r-range search and k-NN search algorithms:

- Using the additional m-NS structure is faster than only using the *ring* structure as *MLR-Index* to achieve similar retrieval quality, because directly using m-NS avoids additional time cost of performing *FNSN* searches repeatedly.
- Hierarchical search algorithms are faster than non-hierarchical algorithms to achieve similar retrieval quality. The reason is that the hierarchical search algorithms enable faster search on dense clusters.
- Our proposed algorithms are significantly faster than linear scan, and also much faster than best-known method *LSH*, even achieving higher accuracy at the same time. For *LSH*, setting different parameter values might make them faster, but it will cause lower accuracy. Since our algorithms achieve better performance both on speed and quality compared with the current setting of parameters, the experimental results are sufficient to prove the advantage of our algorithms.

9 Conclusions

In this paper, we proposed a novel index structure *MLR-Index* (**M**ulti-**L**ayer **R**ing-based **Index**) for high-dimensional indexing problems. Since the vector quantization technique and the ring structure of *MLR-Index* made the search space smaller, we developed an algorithm *FNSN* to find the nearest search node efficiently. By use of *MLR-Index* and *FNSN*, we designed several high-performance solutions for the approximate nearest neighbor search problems including the r-range search problem and the k-NN search problem. We could design even faster search algorithms with similar high accuracy by adding an additional structure m-NS (***m*-N**earest **S**earch Nodes) into *MLR-Index* and utilizing levelwise clusters for dense partitions. Extensive experimental results comparing with *linear scan* and the best-known method *LSH* showed that our approach was effective and efficient with higher accuracy and faster response time.

References

1. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB, pp. 194–205 (1998)
2. Beyer, K.S., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 217–235. Springer, Heidelberg (1999)
3. Tuncel, E., Ferhatosmanoglu, H., Rose, K.: Vq-index: an index structure for similarity searching in multimedia databases. In: MULTIMEDIA, pp. 543–552 (2002)
4. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB, pp. 518–529 (1999)
5. Jeong, J., Nang, J.: An efficient bitmap indexing method for similarity search in high dimensional multimedia databases 2, 815–818 (2004)
6. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: VLDB, pp. 950–961 (2007)
7. Wong, B., Slivkins, A., Sirer, E.G.: Meridian: a lightweight network location service without virtual coordinates. SIGCOMM Comput. Commun. Rev. 35(4), 85–96 (2005)
8. Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst. 9(1), 38–71 (1984)
9. Robinson, J.T.: The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In: SIGMOD, pp. 10–18 (1981)
10. Finkel, R.A., Bentley, J.L.: Quad trees: A data structure for retrieval on composite keys. Acta Inf. 4, 1–9 (1974)
11. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
12. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The r+-tree: A dynamic index for multi-dimensional objects. In: VLDB, pp. 507–518 (1987)
13. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The r*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec. 19(2), 322–331 (1990)

14. Berchtold, S., Keim, D.A., Kriegel, H.P.: The x-tree: An index structure for high-dimensional data. In: VLDB, pp. 28–39 (1996)
15. Katayama, N., Satoh, S.: The sr-tree: an index structure for high-dimensional nearest neighbor queries. SIGMOD Rec. 26(2), 369–380 (1997)
16. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)
17. Lin, K.I., Jagadish, H.V., Faloutsos, C.: The tv-tree: an index structure for high-dimensional data. The VLDB Journal 3(4), 517–542 (1994)
18. Lomet, D.B., Salzberg, B.: The hb-tree: a multiattribute indexing method with good guaranteed performance. ACM Trans. Database Syst. 15(4), 625–658 (1990)
19. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33(3), 322–373 (2001)
20. Ortega-Binderberger, M., Porkaew, K., Mehrotra, S.: Corel Image Features Data Set (1999), `http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features`
21. Blackard, J.A., Dean, D.J., Anderson, C.W.: Covertype Data Set (1998), `http://archive.ics.uci.edu/ml/datasets/Covertype`