A Performance-Guaranteed Approximate Range Query Algorithm for the ND-Tree

Gang Qian[1] Qiang Zhu[2] Sakti Pramanik[3]

[1]Department of Computer Science,
University of Central Oklahoma, Edmond, OK 73034, USA
gqian@uco.edu
[2]Department of Computer and Information Science,
The University of Michigan, Dearborn, MI 48128, USA
qzhu@umich.edu
[3]Department of Computer Science and Engineering,
Michigan State University, East Lansing, MI 48824, USA
pramanik@cse.msu.edu

Abstract

Range queries are a widely-used type of similarity queries that find all objects within a given distance from the query object. In this paper, we propose an approximate range query algorithm for the ND-tree, a multi-dimensional index for vectors with non-ordered discrete components. By sacrificing a little accuracy, approximate algorithms generally can greatly improve search performance. Our proposed approximate algorithm maintains a priority queue of tree nodes whose bounding rectangles (BR) intersect the query sphere. But it only accesses a user-specified portion of the queue. We propose a novel volume-based weighting scheme for the priority queue. The idea is that tree nodes whose BR has a larger intersection with the query sphere contain more result objects, thus should be accessed earlier. A closed-form formula is derived to calculate the volume of an intersection. Our experimental study using both synthetic and real data shows that the proposed algorithm can significantly improve query performance while maintaining high query accuracy.

1 Introduction

Similarity queries find objects in a database that are similar to a given query object. They have important applications in a variety of areas, including content-based image retrieval, data mining, and bioinformatics. Range queries are one type of similarity queries that find all objects within a given distance range from the query object.

The ND-tree [13, 14] is an R-tree [9] like index that supports range queries based on the Hamming distance in non-ordered discrete data spaces (NDDS). Unlike continuous data spaces (CDS) such as the Euclidean space, an NDDS contains vectors of discrete components from domains without an inherent order. The ND-tree was the first multidimensional indexing method specifically designed for large databases in application areas such as bioinformatics that have discrete and non-ordered data.

Although the ND-tree can efficiently support queries with small ranges, when the query range increases, query performance degrades, because much more disk accesses are needed to locate an exponentially increasing number of objects (vectors) in the query result. To efficiently support range queries with extended query ranges, we propose an approximate range query algorithm, called P-ARQ (Performance-guaranteed Approximate Range Query), for the ND-tree. The idea is to greatly improve query performance by sacrificing a little on accuracy of the query result. We define *query accuracy* of approximate range queries as recall, i.e., the percentage of objects returned among all the objects in the database that are within the query range.

In the proposed approximate algorithm, we do not guarantee that all result objects can be returned. Thus query accuracy is reduced. However, a user can specify the level of query performance desired (performance-guaranteed) and eventually retrieve all the missing result objects in an incremental manner if needed. Our experiments show that our algorithm significantly speed up range queries in the ND-tree for extended query ranges, while maintaining high query accuracy.

The rest of the paper is organized as follows. Section 2 covers related work. Essential NDDS concepts and the ND-tree structure are introduced in Section 3. The approximate algorithm P-ARQ is presented in Section 4. Section 5 discusses our experimental results. We conclude the paper in Section 6.

2 Related Work

Although we are the first to address approximate range queries in NDDSs, a number of approximate algorithms for nearest neighbor (NN) queries (another type of similarity queries) in CDSs were suggested in the literature.

Many approximate NN query algorithms are based on indices such as the R-tree [9] and the M-tree [2]. In [1], the approximate NN was obtained by first descending in the index tree to locate a leaf node whose bounding shape covered the query object q. Other leaf nodes were then accessed in the ascending order of their (minimum) distance from q. The algorithm ended when the distance between the next leaf and q was greater than a user-specified error bound ϵ. In [3], the above algorithm was further generalized into a probabilistic framework. The authors proposed an additional confidence value of the probability that the approximate NN could exceed the error bound ϵ. By allowing a small probability that the approximate NN be worse than ϵ, query performance was further improved over the original algorithm.

Three approximation techniques that control when an NN search algorithm should stop were proposed in [4] and [5], including allowed distance errors, percentages of child nodes to access, and limitations on query time. The three techniques were combined into a single hybrid approach in [5].

Several heuristics to prune tree nodes were presented in [10]. One such heuristic was to generate random points within the bounding shape of a node. If the percentage of random points that were closer than the current closest object to the query object was less than a threshold, the node was pruned. Another node-pruning approach was presented in [12]. It was based on indices that utilized bounding spheres, such as the SS-tree [17]. To find the approximate NN, the algorithm measured the hyper-angle between the centroids of the bounding spheres of child nodes and the query object with regard to the centroid of the bounding sphere of the parent node. It only accessed child nodes whose hyper-angles were less than a threshold.

There are approximate query algorithms that are not based on indices, such as the sequential access-based VA-File [16], locality-sensitive hashing [8], and clustering and dimensionality reduction [6]. A detailed survey and categorization of different types of approximate NN query algorithms can be found in [11].

In this paper, our focus is on developing an approximate range query algorithm based on the ND-tree. Although the algorithms proposed for CDSs are closely related to the problem that we want to solve, they are not directly applicable due to two reasons: 1) they are designed for NN queries, not for range queries; and 2) they are designed for CDSs, which have different properties from NDDSs.

3 Preliminaries

The concepts about NDDSs and the structure of the ND-tree were first introduced in [13, 14, 15]. To make this paper self-contained, we briefly describe them in this section.

3.1 Concepts and Notation for NDDS

A *d-dimensional NDDS* Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times ... \times A_d$, where $A_i (1 \leq i \leq d)$ is the *alphabet* of the i-th dimension of Ω_d. Each such alphabet consists of a finite set of letters, and there is no inherent ordering among the letters. For simplicity, we assume A_i's are the same in this paper. As shown in [14], the discussion can be extended to NDDSs with different alphabet sizes. A *vector* in Ω_d is defined as $\alpha = (a_1, a_2, ..., a_d)$ (or '$a_1a_2...a_d$'), where $a_i \in A_i$ $(1 \leq i \leq d)$. A *discrete rectangle* R in Ω_d is defined as $R = S_1 \times S_2 \times ... \times S_d$, where $S_i \subseteq A_i (1 \leq i \leq d)$ is called the *i-th component set* of R. The *area* or *volume of* R is defined as $|S_1| * |S_2| * ... * |S_d|$. The *overlap* or *intersection* of two discrete rectangles R and R' is $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times ... \times (S_d \cap S'_d)$. For a given set VS of vectors, the *discrete minimum bounding rectangle (DMBR)* of VS is defined as the discrete rectangle whose i-th component set $(1 \leq i \leq d)$ consists of all letters appearing on the i-th dimension of the given vectors in VS. The *DMBR* of a set of discrete rectangles can be defined similarly.

As discussed in [13, 15], the Hamming distance is a suitable distance measure for NDDSs. The Hamming distance between two vectors is the number of mismatching dimensions between them. Using the Hamming distance, the (*minimum*) *distance* between a vector $\alpha =$'$a_1a_2...a_d$' and a discrete rectangle $R = S_1 \times S_2 \times ... \times S_d$ is defined as:

$$dist(\alpha, R) = \sum_{i=1}^{d} f(a_i, S_i) \tag{1}$$

where $$f(a_i, S_i) = \begin{cases} 0 & if\ a_i \in S_i \\ 1 & \text{otherwise.} \end{cases}$$

Using the Hamming distance, a range query $range(\alpha_q, r_q)$ is defined as follows: given a query vector α_q and a query range of Hamming distance r_q, find all the vectors whose Hamming distance to α_q is less than or equal to r_q.

3.2 ND-tree

The ND-tree was introduced in [13, 15] to support efficient similarity queries in NDDSs. An overview of its structure is presented in this section.

The ND-tree has a disk-based balanced structure, which has some similarities to that of the R-tree [9] in CDSs. Let M and m ($2 \leq m \leq \lceil M/2 \rceil$) be the maximum and the minimum numbers of entries allowed in each node of an ND-tree, respectively. An ND-tree satisfies the following two requirements: (1) every non-leaf node has between m and M children unless it is the root, which may have a minimum of two children; (2) every leaf node contains between m and M entries unless it is the root, which may have a minimum of one entry/vector.

A leaf node in an ND-tree contains an array of entries of the form (*op*, *key*), where *key* is a vector in an NDDS Ω_d and *op* is a pointer to the object represented by *key* in the database. A non-leaf node N in an ND-tree contains an array of entries of the form (*cp*, *DMBR*), where *cp* is a pointer to a child node N' of N in the tree and *DMBR* is the discrete minimum bounding rectangle of N'. Since each leaf or non-leaf node is saved in one disk block, while their entry sizes are different, M and m for a leaf node are usually different from those for a non-leaf node. Figure 1 shows an example of the ND-tree that indexing genomic sequences with alphabet $\{a, g, t, c\}$ [13].

After an ND-tree is constructed for a data set in an NDDS, an exact range query $range(\alpha_q, r_q)$ can be performed using the tree. The main idea is to start from the root and prune those nodes whose DMBRs are out of the query range until the leaf nodes containing the desired vectors are found. The exact range query algorithm can be found in [15].

4 Performance-Guaranteed Approximate Range Query Algorithm

In this section, we introduce P-ARQ, the performance-guaranteed approximate range query algorithm for the ND-tree.

4.1 Main Ideas

P-ARQ allows a user to specify an input parameter p, which is the percentage of the leaf nodes accessed by P-ARQ. For example, assume that an exact range query will access 100 leaf nodes in an ND-tree. If the user specifies $p = 50\%$, then P-ARQ will only access 50 leaf nodes. This effectively reduces the number of disk accesses and improves query performance.

Accessing only a portion of the tree nodes is actually not a new idea. For example, as discussed in Section 2, one technique proposed in [4, 5] was to access only a specified portion of children in a non-leaf node. P-ARQ differs in that the user-specified percentage p is based on the number of leaf nodes that would be accessed by an exact query. This approach gives a user a clear idea of how fast the approximate query will be comparing to the exact query. Thus, query performance of P-ARQ is guaranteed for the user. Through experiments, we also observed that using p has the advantage of getting consistent query accuracy, that is, the percentages of result objects returned are similar for the same p value across different query ranges and data set sizes (see Section 5).

P-ARQ accesses every non-leaf node that is within the query range during the process of a range query. This allows better query accuracy, since every non-leaf node that is within the query range may have a number of leaf nodes in its subtree that contain vectors within the query range. Pruning such a non-leaf node may reduce query accuracy of P-ARQ. On the other hand, query performance is not really affected by examining all these non-leaf nodes, as the number of leaf nodes are much greater than the number of non-leaf nodes in an ND-tree. P-ARQ is presented as follows:

ALGORITHM 4.1.1 : **P-ARQ**
Input: (1) range query: $range(\alpha_q, r_q)$; (2) user-specified percentage: p; (3) ND-tree with root node *RN* for the underlying database.
Output: set *VS* of vectors within the query range.
Method:

```
1. let VS = ∅;
2. push RN into an empty priority queue NQueue of tree nodes;
3. let leafCnt = 0; /* Number of leaves processed */
4. while NQueue ≠ ∅ do
5.   let CN = dequeue(NQueue);
6.   if CN is a non-leaf node then
7.     for each entry E in CN do
8.       if dist(α_q, E.DMBR) ≤ r_q then
9.         if E.cp points to a non-leaf node then
10.          enqueue E.cp with weight ∞;
11.        else /* E.cp points to a leaf */
12.          enqueue E.cp with a certain weight w;
13.        end if;
14.      end if;
15.    end for;
16.  else /* CN is a leaf */
```

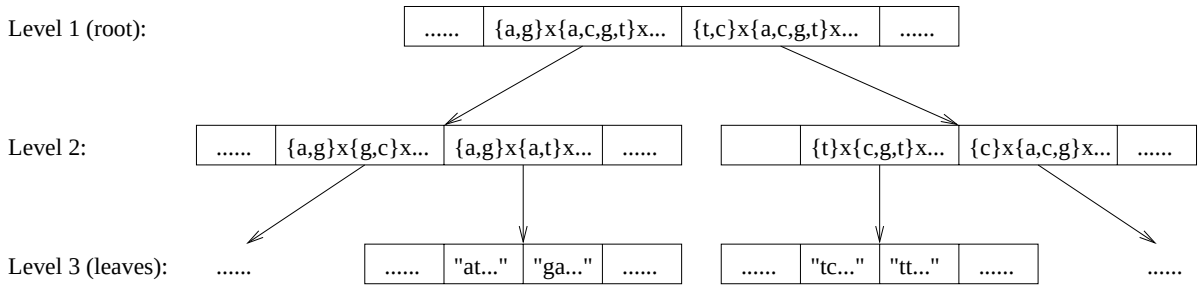

Figure 1: An example of the ND-tree

17. **for** each vector v in CN **do**
18. **if** $dist(\alpha_q, v) \leq r_q$ **then**
19. let $VS = VS \cup \{v\}$;
20. **end if**;
21. **end for**;
22. $leafCnt = leafCnt + 1$;
23. $percent = leafCnt/(leafCnt + size(NQueue))$;
24. **if** $percent \geq p$ **then**
25. break;
26. **end if**;
27. **end if**;
28. **end while**;
29. **return** VS.

P-ARQ uses a priority queue $NQueue$ to keep track of the tree nodes to be processed. In step 10, a weight of infinity (∞) is associated with every non-leaf node within the query range. This guarantees that all non-leaf nodes are processed first. In step 22, $leafCnt$ counts the number of leaf nodes accessed so far. If the percentage of leaf nodes accessed exceeds p, the while-loop stops immediately (steps 23 - 25). Note that, since all the non-leaf nodes are processed first, when the algorithm starts to process leaf nodes, all the leaf nodes that are within the query range are already enqueued. Therefore, $leafCnt + size(NQueue)$ in step 23 gives the total number of leaf nodes that are within the query range and all of them would be accessed if an exact query were performed.

In step 12, a certain weight w is associated with each leaf node in the priority queue. It is an important value that determines which leaf nodes are accessed and which are not, since P-ARQ only accesses a portion of the priority queue. Obviously, query accuracy is not optimized if we use the natural order (that is, assign the same weight value to every leaf node). One possible approach is to employ the NN query heuristic, which uses the (minimum) distance between the query and the bounding rectangle of the leaf node as the weight. The idea is that if a leaf node is closer to the query, then it is likely to contain more vectors within the query range. Our experiment shows that this heuristic leads to good query accuracy. On the other hand, the minimum distance still does not directly reflect the number of result objects in a leaf node. To further improve query accuracy, we propose a new volume-based weighting scheme for P-ARQ, which is presented in the next section.

4.2 Volume-based Weighting Scheme

Instead of using the natural order or the minimum distance as the weight for leaf nodes in P-ARQ, we propose to use the volume of the intersection between the DMBR of the leaf node and the query sphere, which is a hyper-sphere with the query object as its centroid and the query range as its radius. The size (volume) of the intersection directly reflects the number of objects (vectors) that are within the query range in the leaf node. Therefore, it is a better heuristic than the minimum distance to arrange the order of the leaf nodes in the priority queue ($NQueue$).

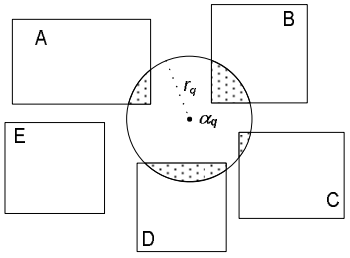

Figure 2: Illustration of volume-based weighting

The volume-based weighting scheme is illustrated in Figure 2. In the two-dimensional data space, the query sphere intersects bounding rectangles A, B, C and D. Based on the size of the intersections, the order of the leaf nodes in the priority queue represented by the bounding rectangles will be D, B, A and C.

The use of the Monte Carlo method was proposed in [10] to estimate the number of objects (vectors) in a node that are closer to the query object than the current approximate NN found for NN queries in

a CDS. Our method of using the exact intersection volume is obviously more accurate than the Monte Carlo method. The calculation of the intersection volume between a hyper-sphere and a hyper-rectangle is complex in a high-dimensional CDS. However, due to the nature of an NDDS, we can derive a closed-form formula to calculate the intersection volume between a query sphere and a DMBR in an NDDS.

We first give a clear definition of the volume of the intersection between a query sphere and a DMBR. Following the definition of the area/volume of a discrete rectangle in Section 3, we define the *volume of the intersection* between a query sphere and a DMBR as the total possible number of vectors that are within the query range in the DMBR. Formally, given a range query $range(\alpha_q, r_q)$, where α_q = '$a_1 a_2 ... a_d$', and a discrete rectangle $R = S_1 \times S_2 \times ... \times S_d$, the volume of the intersection between $range(\alpha_q, r_q)$ and R is:

$$volume = \sum_{i=0}^{r_q} H(\alpha_q, R, i) \quad (2)$$

where function $H(\alpha_q, R, i)$ returns the total possible number of vectors that are at distance i from α_q in R.

Based on the definition of the Hamming distance, a vector at distance i from the query vector α_q has i *mismatching dimensions* and $d-i$ *matching dimensions* with α_q. Accordingly, function H can be calculated using the following equation:

$$H(\alpha_q, R, i) = \sum_{\forall DS = ds(i)} \left(b(DS_d - DS) * \prod_{\forall j \in DS} c(j) \right) \quad (3)$$

where

$ds(x)$ returns a set of x dimensions,

DS_d is the set of all d dimensions,

$$b(set_{dim}) = \begin{cases} 0 & \text{if } \exists\ x \in set_{dim},\ a_x \notin S_x \\ 1 & \text{otherwise} \end{cases}, \text{ and}$$

$$c(x) = \begin{cases} |S_x| - 1 & \text{if } a_x \in S_x \\ |S_x| & \text{otherwise} \end{cases}.$$

In Equation 3, function $ds(x)$ returns an arbitrary set (DS) of x dimensions from a total of d dimensions in the NDDS. DS is considered as the set of i mismatching dimensions. The sizes of the component sets on these dimensions of the discrete rectangle R are used to calculate the possible number of vectors in the intersection in the product part ($\prod$) of the equation. As shown in function $c(x)$, if the component set of R on a certain dimension x ($\in DS$) contains a_x, then the mismatching letter count for dimension x of R is $|S_x| - 1$. Note that it is possible that $S_x = \{a_x\}$, in which case $c(x) = 0$. Function $b(set_{dim})$ takes a set of dimensions as its input parameter. The $DS_d - DS$ part of the equation is a set minus/difference operation. DS_d represents the set of all d dimensions in the NDDS. The set difference operation produces the set of the rest of the $d - i$ dimensions, other than those in DS. These dimensions are considered as matching dimensions. Therefore, the corresponding component sets in R on these matching dimensions must have the corresponding letters in α_q. Otherwise, the dimension that does not have the α_q letter cannot be a matching dimension and, accordingly, function $b(set_{dim})$ returns a zero for this particular combination of i mismatching and $d - i$ matching dimensions.

The calculation of $H(\alpha_q, R, i)$ can be simplified. We first identify *mismatching-only dimensions*, which are dimensions whose corresponding component set in R does not contain the corresponding letter in α_q. Such dimensions can never be a matching dimension. For example, suppose that the component set of R on dimension 5 is $S_5 = \{b, d, e\}$. If a_5 = 'c' in the query α_q, then dimension 5 is a mismatching-only dimension.

Similarly, we also identify *matching-only dimensions*, which are dimensions whose corresponding component set in R contains only the corresponding letter in α_q. Such dimensions can never be a mismatching dimension. For example, suppose that the component set of R on dimension 6 is $S_6 = \{c\}$, while a_6 = 'c' in α_q. Then dimension 6 is a matching-only dimension.

We call those dimensions that can be either mismatching or matching as *dual dimensions*. Let d_{mis}, d_m and d' represent the numbers of mismatching-only, matching-only and dual dimensions respectively. We have $d_{mis} + d_m + d' = d$. Obviously, given $H(\alpha_q, R, i)$, if either $d_{mis} > i$ or $d_m > d-i$, we have $H(\alpha_q, R, i) = 0$, because it is impossible that any vector in R can be at exactly distance i from α_q.

Once the mismatching-only and matching-only dimensions are identified for a given query α_q, the calculation of function $H(\alpha_q, R, i)$ can be done based on the following equation:

$$H(\alpha_q, R, i) = \begin{cases} 0 & \text{if } d_{mis} > i \\ 0 & \text{if } d_m > d - i \\ prod_{mis} \times H'(R, i) & \text{otherwise} \end{cases} \quad (4)$$

where

$$prod_{mis} = \prod_{\forall j \in DS_{mis}} |S_j|$$

in which DS_{mis} is the set of mismatching-only dimensions, and

$$H'(R, i) = \sum_{\substack{\forall DS' = \\ ds'(i - d_{mis})}} \prod_{\forall j \in DS'} (|S_j| - 1)$$

in which function $ds'(x)$ returns an arbitrary set of x dimensions from the d' dual dimensions.

In Equation 4, $prod_{mis}$ represents the partial product of the possible number of objects (vectors) from all the d_{mis} mismatching-only dimensions. The $ds'(i - d_{mis})$ part of function $H'(R, i)$ picks from the d' dual dimensions the rest of the dimensions to make up a total of i mismatching dimensions. When a dual dimension j is picked as a mismatching dimension, it contributes $|S_j| - 1$ to the calculation, as shown in the product part ($\prod$) of function $H'(R, i)$. Note that the d_m matching-only dimensions do not explicitly appear in the equation because they contribute a partial product of 1 to the total possible number of vectors in the intersection.

The simplified approach works because every dimension of the DMBRs at the leaf level in a large ND-tree are usually split once or more, creating a number of mismatching-only and matching-only dimensions. The calculation of function H can be further simplified using a lookup table of pre-computed powers of S_j and $S_j - 1 (1 \leq j \leq d)$, if the alphabet sizes of all dimensions are the same, as assumed in Section 3.

5 Experimental Results

Our experiments were conducted on a personal computer with Intel Core2 Duo 2.00GHz CPU, 2GB memory and 400GB hard disk. Query performance was measured based on the number of disk accesses, and query accuracy was evaluated based on recall, i.e., the percentage of the exact query result objects returned by the approximate algorithm. The data sets used in the experiments included both synthetic and genomic data. Synthetic data sets were randomly generated with 10 dimensions and an alphabet size of 10 on all dimensions. Genomic data sets were extracted from bacteria genomic sequences from the GenBank [7], which were broken into subsequences/vectors of 18 characters long (18 dimensions). All programs were implemented in C++. The minimum space utilization percentage for a disk block was set at 30% for the ND-tree and the disk block size was set at 4KB.

5.1 Heuristics Evaluation

In this section, we present experimental results that were used to evaluate the effectiveness of the heuristics employed in P-ARQ. Synthetic data was used in these experiments. Each result was the average from executing 100 random test queries. We compared three different weighting schemes for the priority queue employed in P-ARQ. They were 1) the volume of the intersection between the query sphere and the DMBR of a leaf node; 2) the minimum distance between the query and the DMBR of a leaf node; and 3) a constant weight (natural order). The results are presented in Figures 3, where the data set used had two million vectors. Note that we only need to compare query accuracy of the three schemes, because they have exactly the same query performance given the same p value.

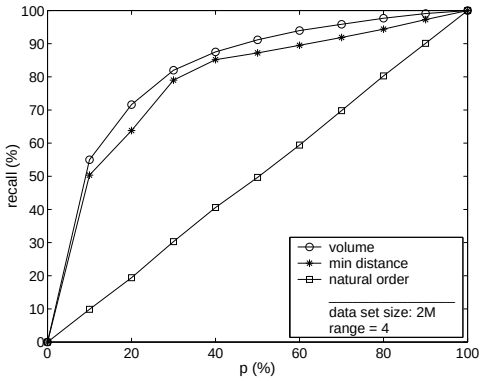

Figure 3: Weighting schemes at different p values

Figure 3 show query accuracy at different p values with a query range of 4. From the figure, we can see that the natural order scheme does not work well because the percentage of result objects that it returns is just proportional to p, the percentage of the leaf nodes accessed. The volume-based weighting scheme has higher query accuracy than the minimum distance-based one. Its query accuracy is on average 5% better than that of the minimum distance-based scheme across different p values. Since the volume-based scheme uses more information available in an ND-tree, it generally leads to better query accuracy. The figure also show that P-ARQ achieves a high recall rate of at least 90% at a p value of 50%.

5.2 Performance and Accuracy

In this section, we present experimental results that evaluate query performance and accuracy of P-ARQ. Both synthetic and genomic sequence data sets were used in the experiments. Due to space limitations, we only present results using genomic data. As before, results were the average measure for executing 100 random test queries for each case.

Figures 4 and 5 show query performance and accuracy of P-ARQ respectively at different query ranges using genomic data. Note that query performance is represented as a percentage of disk accesses of the corresponding exact query.

As designed, query performance of P-ARQ is directly tied to the user specified parameter p, which is the

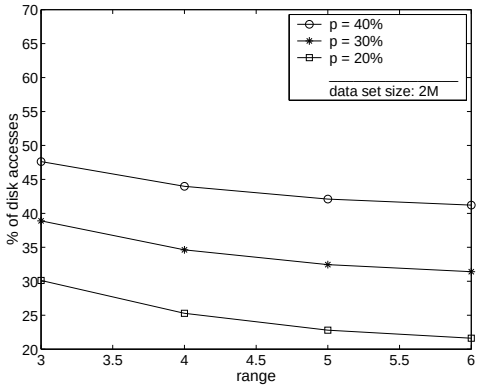

Figure 4: Query performance at different query ranges

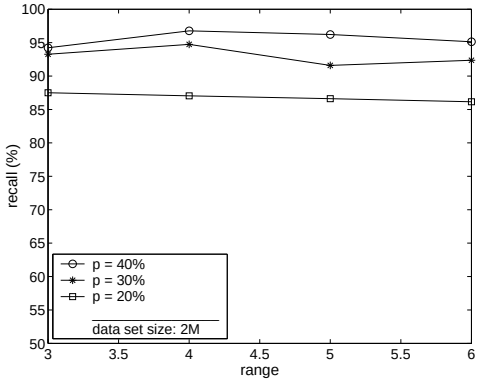

Figure 5: Query accuracy at different query ranges

percentage of leaf nodes that are accessed by the exact query. Since P-ARQ accesses all non-leaf nodes that are within the query range, as shown in Figure 4, the actual percentage of disk accesses is slightly higher than the p value. It gets closer to p as the query range increases, because the portion of non-leaf nodes becomes smaller compared to that of leaf nodes at larger query ranges.

In Figure 5, we can see that P-ARQ reaches a recall rate of more than 90% at $p = 30\%$ using the genomic data set. As mentioned in Section 4, one advantage of using the percentage of leaf nodes as the user-specified parameter p is that query accuracy of approximate queries is consistent across different query ranges. This is demonstrated in Figure 5 as the recall rates of the same p values do not change much at different query ranges. This property of P-ARQ makes it easy for a user to choose one p value, use it for different query ranges, and still expect to get similar query accuracy.

Table 1 show query performance and accuracy of P-ARQ at different data set sizes using genomic data. The p values are set at 30%. From the table, we can see the same trend of query performance as shown in Figures 4, that is, the percentage of disk accesses is directly related to the p value and is slightly higher than p due to the accesses of non-leaf nodes by P-ARQ. On the other hand, query accuracy of P-ARQ is almost always at a recall rate of above 90% from $500,000$ to $3,000,000$ vectors. This shows that P-ARQ scales well with increasing data set sizes. This also demonstrates the consistency of query accuracy of P-ARQ for a given p value at different data set sizes.

6 Conclusions

In this paper, we propose an approximate range query algorithm, P-ARQ, for the ND-tree in NDDS. P-ARQ allows a user to specify a parameter p, the percentage of leaf nodes accessed by the exact query algorithm. Since leaf nodes represent the majority of the disk accesses for processing a range query, the value of p effectively determines when P-ARQ stops. Thus, it gives a user a guarantee on query performance.

We propose a new volume-based weighting scheme and a closed-form formula for volume calculation to arrange the order in which tree nodes are accessed by P-ARQ. This allows P-ARQ to first process those leaves that potentially have more result objects, resulting in better query accuracy. We have also seen that query performance and accuracy of P-ARQ are quite consistent across different query ranges and data set sizes. Note that it is easy to further manipulate the priority queue in P-ARQ by specifying which portion of the queue is to be accessed. This will allow a user to incrementally access all the query result ojbects when time permits.

In future work, we plan to conduct a theoretical analysis of query performance and accuracy of P-ARQ.

7 Acknowledgments

Research supported by National Center for Research Resources, a component of the US National Institutes of Health (under grant # P2PRR016478), the US National Science Foundation (under grants # IIS-1319909 and # IIS-1320078), University of Central Oklahoma, Michigan State University, and the University of Michigan.

References

[1] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45(6):891–923, 1998.

Table 1: Query performance and accuracy at different data set sizes ($p = 30\%$)

dataset	$r_q = 3$		$r_q = 4$		$r_q = 5$		$r_q = 6$	
size	*io%*	*recall%*	*io%*	*recall%*	*io%*	*recall%*	*io%*	*recall%*
500000	36	95	33	95	32	93	31	94
1000000	38	95	34	95	32	92	31	94
2000000	39	93	35	95	32	92	31	92
3000000	39	93	35	94	33	89	32	91

[2] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on VLDB*, pages 426–435, 1997.

[3] Paolo Ciaccia and Marco Patella. PAC nearest neighbor queries: approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th International Conference on Data Engineering*, pages 244–255, 2000.

[4] Antonio Corral, Joaquín Cañadas, and Michael Vassilakopoulos. Approximate algorithms for distance-based queries in high-dimensional data spaces using R-trees. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, pages 163–176, 2002.

[5] Antonio Corral and Michael Vassilakopoulos. On approximate algorithms for distance-based queries using R-trees. *Computer Journal*, 48(2):220–238, 2005.

[6] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 503–511, 2001.

[7] GenBank. http://www.ncbi.nlm.nih.gov/Genbank, June 2014.

[8] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity searching in high dimensions via hashing. In *Proceedings of the 25th International Conference on VLDB*, pages 518-529, 1999.

[9] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–57, 1984.

[10] King-Ip Lin, Mike Nolen, and Koteswara Kommeneni. Utilizing indexes for approximate and on-line nearest neighbor queries. In *Proceedings of the 9th International Symposium on Database Engineering and Application*, pages 83–88, 2005.

[11] Marco Patella and Paolo Ciaccia. Approximate similarity search: a multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36–48, 2009.

[12] Sakti Pramanik, Scott Alexander, and Jinhua Li. An efficient searching algorithm for approximate nearest neighbor queries in high dimensions. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 865-869, 1999.

[13] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. The ND-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proceedings of the 29th International Conference on VLDB*, pages 620–631, 2003.

[14] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *ACM Transactions on Database Systems*, 31(2):439–484, 2006.

[15] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Transactions on Information Systems*, 23(1):79–110, 2006.

[16] Roger Weber and Klemens Böhm. Trading quality for time with nearest neighbor search. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 21–35, 2000.

[17] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, 1996.