Efficient RkNN Retrieval with Arbitrary Non-Metric Similarity Measures

Deepak P Prasad M Deshpande
IBM Research - India, Bangalore, INDIA
{deepak.s.p,prasdesh}@in.ibm.com

ABSTRACT

A RkNN query returns all objects whose nearest k neighbors contain the query object. In this paper, we consider RkNN query processing in the case where the distances between attribute values are not necessarily metric. Dissimilarities between objects could then be a monotonic aggregate of dissimilarities between their values, such aggregation functions being specified at query time. We outline real world cases that motivate RkNN processing in such scenarios. We consider the AL-Tree index and its applicability in RkNN query processing. We develop an approach that exploits the group level reasoning enabled by the AL-Tree in RkNN processing. We evaluate our approach against a Naive approach that performs sequential scans on contiguous data and an improved block-based approach that we provide. We use real-world datasets and synthetic data with varying characteristics for our experiments. This extensive empirical evaluation shows that our approach is better than existing methods in terms of computational and disk access costs, leading to significantly better response times.

1. INTRODUCTION

The Reverse Nearest Neighbor (RNN) search problem has received a lot of attention from the database community for its broad application range such as marketing, decision support and resource allocation [17]. Given a set of data objects and a query object, an RNN query retrieves all objects that have the query object as it's nearest neighbor. Similarities between objects are often computed as a monotonic aggregate of similarities in multiple attributes considered [14, 13]. Reverse k-NN (RkNN) generalizes the RNN query to find objects that have the query among its k nearest neighbors.

The goal of an RkNN query is to find the *influence* of a query object in the whole dataset. The classical motivating example [2] is the decision support task of identifying the optimal location for a new Pizza store. Given several location choices, the strategy is to pick the location that can attract the most number of customers. A RkNN query returns the customers who are likely to use the new store because of its geographical proximity as against the existing stores. The result set of RkNN could also precisely be that set of customers to whom the store owner may want to send promotional offer mailings since they are most likely to respond positively to it because of the same reason.

We arrived at the problem of having to identify the RkNN nearest objects in the scenario of business continuity planning for a service delivery organization that manages hundreds of thousands of servers. A core job role in such an organization is that of *system administrators (admins)* who manage servers, and solve problems that occur in the servers that they manage. Over time, they gain expertise in solving problems specific to certain software, operating systems and specific types of hardware. The expertise of a system administrator would then be represented as a vector of such acquired expertise (e.g., operating system, network type etc.); servers would also then map to such a space with appropriate values for the various categories. Matching admins to servers need not always be done on the basis of exact match; to choose the admin for a server running Linux, we may choose to prefer an AIX specialized admin over a Windows admin (if they are similar on other aspects) since AIX is known to be more similar to Linux than Windows. Similarity measures such as these (among OSes, Vendors of products etc) are often only available from domain knowledge, and hence usually do not adhere to metric properties. The suitability of an admin to a server would then be assessed using an aggregation function of the similarities between them based on the various attributes/categories considered. The *influence* of an admin is then assessed based on their RkNN set; the RkNN set gives the set of servers for whom the system admin in question is among the top-k best (highest scored) admins. Such scores are critical to the business since heavily skewed influence distribution among admins and attrition of highly influential admins are all causes of concern due to obvious reasons. A similar scenario arises in choosing a set of retail customers to send promotional mailings to, for a new offer on a particular product. Since the retail company would want to choose customers who are most likely to respond positively, the RkNN set is a good choice since that gives the set of users for whom the product would be in their top-k preferred products. The weights in such a case could be certain specific aspects of the product that the company may want to highlight. In cases where sets and categorical values are involved in the similarity search process, the similarity measures between attribute values may be non-metric; we discuss this further in Section 1.1.

RkNN query is more expensive than the k Nearest Neighbors query [19] due to the asymmetry, i.e., an object being the nearest neighbor of another doesn't imply that the latter is the nearest neighbor of the former, since it may have other objects closer to it. Thus, the naive solution for RkNN is significantly costlier (i.e., quadratic in the dataset size).

1.1 Non-metric Spaces

Many attributes in various domains are categorical and the similarities between the various values often come from domain knowledge. For example, operating system is a typical server attribute. Similarities between operating systems come from domain knowledge with the domain expert defining the similarities for each pair of operating systems. In such cases, the measures cannot be expected to comply with the metric requirement of triangle inequality (and hence, are not even *arbitrary metric* [2]). We will see in Section 2 that non-metric similarity measures are indispensible in many domains of interest. Attributes where the similarity between values is arbitrary do not have a total ordering among their values. For example, there is no global ordering of the operating systems available. However, upon presentation of a query, the values of the attribute may be ordered based on similarity to the query value for that attribute.

Multi-dimensional indexing structures such as R-tree are applicable only when there is an ordering of values for each attribute (Euclidean space). In the case of arbitrary similarity measures, an ordering of values for each attribute can be arrived at, when values are considered with respect to a chosen value for the attribute (typically, that of the object under consideration). However, since different objects may take different values for attributes, it is not possible to create a single index that can be used for different objects under consideration; this renders such an approach impractical. M-tree [11] avoids the need for a Euclidean distance, but it still requires the triangle inequality property. In the absence of triangle inequality property, the similarity between a pair of objects cannot be reasoned about by knowledge about their separate similarities to a third object. *This makes metric space indexing structures (e.g., R-Tree [16], kd-tree [6], M-Tree [11] etc.) for similarity search inapplicable.*

1.2 Our Contributions

In this paper, we address RkNN retrieval on databases of objects composed of multiple attributes. We focus on the setting where the dissimilarity between a pair of objects is measured as a monotonic aggregate of the dissimilarities between their attribute values [5, 14] where such dissimilarities could be non-metric. *We allow for the monotonic aggregation distance function to be specified at query time.*

Without the metric assumption, the naive method for RkNN requires scanning the database for each object to check if there are k objects closer to it than the query (see Appendix C). Objects are discarded as soon as k *closer* objects are found. This requires n partial/full scans and has a worst case complexity of $\mathcal{O}(n^2)$ where n is the number of objects. We have developed an IO friendly block based version of the naive, called *BR*, that has the same complexity, but requires fewer scans (Appendix C). Our main contribution is a new algorithm ALT-RkNN, based on the AL-Tree [13], a recently proposed non-metric space index structure. It works by pruning out sub-parts, leading to better average case complexity and requiring at most two scans of the database. We illustrate the efficiency of ALT-RkNN empirically through an extensive analysis.

Section 2 discusses related work. Section 3 defines the problem formally and Section 4 presents our AL-Tree based algorithm. Our experimental evaluation comprises Section 5 and Section 6 presents the conclusions.

2. RELATED WORK

The need for non-metric similarity functions has been argued in [15], that says that the triangle inequality property is too restrictive to model the (dis)similarities as perceived by humans. [21] opines that the conceptual notion of similarity is centered around various aspects, and that different aspects may be selected for comparing different pairs of objects. Such choices of aspects to compare pairs cannot be pre-determined; this makes metric assumptions inapplicable. Further, non-metric similarity measures have been found to be useful in similarity search in various kinds of data ranging from images to object trajectories [27, 7]. [24] points out specific cases in which each of the metric properties (viz., reflexivity, symmetry and triangle inequality) may not be intuitively satisfied. Two flavors of the RkNN problem have been studied. *Monochromatic* RkNN is the case where the query object comes from the search space (e.g., finding the influence of a user among a set of users) whereas *Bichromatic* is the case where the query object and the search space are composed of different (types of) sets of objects. The technique described in this paper can be applied to both *Monochromatic* and *Bichromatic* RkNN problems. Arbitrary metric spaces and even more constrained versions (e.g., Euclidean spaces) and their properties have been utilized to efficiently perform RkNN search. Such techniques work by exploiting metric space properties to affect pruning and hence are inapplicable for our setting; we briefly review approaches for metric space RkNN in Appendix A.

There has been previous work on RkNN under non-metric similarities; however, they cannot handle dynamically specified distance functions. With the dissimilarity function being *static* and fully specified beforehand, dissimilarities between pairs of objects could be pre-computed and stored. Thus, to decide on an object's membership in RkNN, its sufficient to compare its dissimilarity to the query against the object's k^{th} nearest neighbor dissimilarity in the database. This straightforward index has been found to be effective in RkNN processing [8]; more so when compressed [30]. Static functions disallow changing even the weights between attributes in a weighted sum distance function. Moreover, such approaches require large pre-computation time and storage, both being proportional to $D*K$ where D is the database size and K is the upper bound on the query time k.

Another related work is the distance based hashing [4] used for 1NN computation under non-metric similarity measures. However, the asymmetric relationship between kNN and RkNN makes it hard to adapt techniques for kNN to RkNN. A simple adaptation of a 1NN scheme described in [4] would necessitate more hash functions and require scanning each bucket to which at least one kNN candidate of each RkNN candidate maps to. This would still lead to an approximate answer,the kNN method being approximate.

3. PROBLEM DEFINITION

We now outline the RkNN problem and the setting more

formally. Let D be the set of objects in the database; each object in D has m attributes. The dissimilarity function d_i for attribute i is a function $d_i : A_i \times A_i \rightarrow \Re$ where A_i is the domain of the i^{th} attribute. The dissimilarity (distance) between two objects is defined as any monotone function of the distance between the corresponding attribute values:

$$d(Q, O) = f(d_1(v_1(Q), v_1(O)), \ldots, d_m(v_m(Q), v_m(O)))$$

where $v_i(O)$ denotes the value of the i^{th} attribute of O and $f(.)$ is any monotone combining function. Most commonly, such functions assume the form of a weighted sum:

$$d(Q, O, W) = \sum_i w_i * d_i(v_i(Q), v_i(O))$$

where $W = [w_1, \ldots, w_m], w_i > 0$ is the weight vector. We deal with this weighted sum form through this paper; however, the technique proposed is easily generalizable to any generic monotonic combining function (Appendix D.1). If each d_i is bounded in $[0, 1]$ and $\Sigma_i w_i = 1.0$, then the distance between any two objects is also bounded in $[0, 1]$.

Definition 1. **RkNN Query Problem:** Given a query object Q, k and a weight vector W, find all objects from D that have fewer than k objects closer (i.e., more similar) to them than Q. This corresponds to finding the set $S \subseteq D$:

$$S = \{s \in D : |\{u : u \neq s \wedge d(u, s, W) < d(Q, s, W)\}| < k\}$$

The inner set identifies the set of objects that are closer to s than the query object Q, when the distance is computed using the weight vector W. If the set has a size lesser than k, s is part of the RkNN set, S. The inner set need not be explicitly computed in an RkNN processing engine; it suffices to check whether there are at least k objects closer to s than the query to assess membership in the RkNN set.

We use $D[j]$ to refer to the j^{th} object in D whereas the value of its i^{th} attribute is denoted by $v_i(D[j])$. The domain of the i^{th} attribute, a_i, is denoted by A_i whereas the distance function for that attribute is denoted by $d_i(\ldots)$. The query object Q and the attribute weight vector $W = [w_1, w_2, \ldots, w_m]$ are the user inputs to the retrieval system.

4. RKNN RETRIEVAL USING AL-TREE

We now briefly describe the AL-Tree [13] and our approach that uses the AL-Tree for efficient RkNN retrieval.

4.1 The Attribute Level (AL) Tree

Consider the database D and a specific ordering of attributes. Each object can now be represented as a sequence of values, the i^{th} value corresponding to the i^{th} value in the chosen attribute ordering. The AL-Tree for D using the chosen ordering is then precisely the prefix tree[1] for the ordered database. In such a tree, all the leaf nodes are at the same level, i.e., level m, and each level in the tree corresponds to a specific attribute, according to the chosen ordering. The tree is compressed by collapsing each chain in the tree to the head of the chain; such compressed chains form leaf nodes at levels lesser than m. Each leaf in the tree maintains information about the objects that it stands for, and also any values for remaining attributes (in cases of leaf nodes representing collapsed chains). Any object in the database is uniquely associated with a leaf node, and all duplicate objects map to the same leaf node. For any node N, $Obj(N)$ is used to to denote the set of all its descendant objects.

[1]http://en.wikipedia.org/wiki/Trie

Id	OS Name	DB Name
1	MS Windows (MSW)	Informix
2	MS Windows (MSW)	Oracle
3	RedHat Linux (RHL)	Oracle
4	SuSE Linux (SL)	DB2
5	SuSE Linux (SL)	DB2

Table 1: Sample dataset

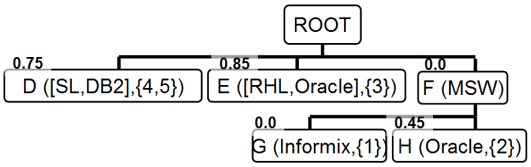

Figure 1: A Compressed AL Tree.

d_1	MSW	RHL	SL
MSW	0.0	0.8	1.0
RHL	0.8	0.0	0.1
SL	1.0	0.1	0.0

d_2	Informix	DB2	Oracle
Informix	0.0	0.5	0.9
DB2	0.5	0.0	0.5
Oracle	0.9	0.5	0.0

Figure 2: Distance Functions.

Example 1. Consider the dataset in Table 1 with the chosen attribute ordering. For OS='RHL', there is a single value of DB ('*Oracle*') in the database. The corresponding chain, hence, can be combined into a single node E in the corresponding AL-Tree represented by $([RHL, Oracle], \{3\})$. Similarly, the chain with OS Name of value 'SL' and DB of value '*DB2*' can be compressed into node D. The tree with both the chains compressed is shown in Figure 1. In this example, we represent the distance functions $d_1(\ldots)$ and $d_2(\ldots)$ in Figure 2. $d_1(\ldots)$ is non-metric; $d_1(MSW, SL) = 1.0$ is greater than the sum of the distances $d_1(MSQ, RHL) = 0.8$ and $d_1(RHL, SL) = 0.1$. Now, consider a query $Q = [MSW, Informix]$ and the weight vector $[0.5, 0.5]$. Figure 1 has the tree annotated with the *distance so far* for each node, from the query. The distance of the node H (i.e., $[MSW, Oracle]$), from the query is $0.5 * d_1(MSW, MSW) + 0.5 * d_2(Oracle, Informix)$ i.e., 0.45. The distance of F is $0.5 * d_1(MSW, MSW) = 0.0$. We will use such distances to effect pruning in our RkNN approach. Further, we use the Dataset in Figure 1 and the dissimilarities in Figure 2 with the Query $[MSW, Informix]$ and $k = 1$ as our running example; the result set for this is $\{[MSW, Informix]\}$.

4.2 RkNN Retrieval using AL-Tree

Now, we describe our approach for RkNN retrieval using the AL-Tree index. We start by describing conditions that enable pruning of AL-Tree nodes while searching for RkNN results; this serves to lead up to the proposed approach.

4.2.1 Pruning Conditions

Each node in the AL-Tree denotes a certain choice of values for a subset of attributes. For example, the node H in

Figure 1 denotes the choice $[OS = MSW, DB = Oracle]$. Let a candidate prefix be denoted by $C = [a_1 = c_1, a_2 = c_2, \ldots, a_j = c_j], j \leq m$. This node encloses all data objects (denoted by $Obj(C)$) that take the value c_i for each attribute i, $i \leq j$. Since each $d_i(.,.)$ would return a value in $[0, 1]$ (and assuming $\forall x, d_i(x, x) = 0.0$) the maximum distance among any pair of objects among $Obj(C)$ would be:

$$MaxD(C) = \Sigma_{i=j+1}^{m} w_i$$

The distance of the query to *each* object in $Obj(C)$ is at least that based on the attribute values fixed at C.

$$MinD(C, Q) = \Sigma_{i=1}^{j} w_i * d_i(v_i(Q), c_i)$$

We now specify a **Simple Pruning Condition** (SPC):

$$SPC = MinD(C, Q) > MaxD(C) \wedge |Obj(C)| > k$$

The first check verifies that every object c in $Obj(C)$ is closer to *all* other objects in $Obj(C)$ than the query. The second check ensures that $Obj(C)$ has at least $k + 1$ objects. *It is easy to see that if both the checks are satisfied, no object in $Obj(C)$ would be in RkNN since each of them has k objects (in $Obj(C)$) closer to it than the query.* In the process of exploring the AL-Tree for RkNN results, one can safely discard nodes that satisfy this simple pruning condition. We present the $MinD(.)$, $MaxD(.)$, and $|Obj(.)|$ values of each node in our running example (w.r.t $Q = [MSW, Informix], k = 1$) in Figure 3; it may be seen that SPC is able to prune D. SPC is a sufficient condition for pruning, but it is not necssary. Nodes not satisfying SPC are not guaranteed to be in the RkNN result. For example, SPC is unable to prune E & H regardless of them being not part of the result.

	D	E	F	G	H
MaxD(.)	0.0	0.0	0.5	0.0	0.0
MinD(.,Q)	0.75	0.85	0.0	0.0	0.45
—Obj(.)—	2	1	2	1	1

Pruned Nodes
D

Figure 3: Illustration of SPC.

Now, let us consider how another prefix $C' = [a_1 = c'_1, \ldots, a_p = c'_p]$ could help in pruning C. We are interested in now calculating the maximum distance between pairs of points, of which one is from $Obj(C)$ and the other is from $Obj(C')$. Let this be denoted by $MaxD(C, C')$:

$$MaxD(C, C') = \Sigma_{i=1}^{min\{j,p\}} d_i(c_i, c'_i) + \Sigma_{i=min\{j,p\}+1}^{m} w_i$$

$MaxD(.,.)$ as defined above has two components; the actual distance between the values chosen in the candidates for those attributes for which both candidates have chosen a value (i.e., attributes for which the shallower candidate has chosen values), and the sum of the upper bounds of distances on the remaining attributes (which is simply the sum of the weights for those attributes). It is easy to see that $MaxD(C, C')$ is at least as much as $MaxD(C)$. Now, the condition where C' can be used to prune C is:

$$MinD(C, Q) > MaxD(C, C') \wedge |Obj(C) \cup Obj(C')| > k$$

The first part checks that each object in $Obj(C)$ has *all* other objects in $Obj(C) \cup Obj(C')$ closer to itself than the query and the second part checks whether there are enough objects in the considered set to ensure that $Obj(C)$ is not part of the RkNN result. When both conditions are satisifed, we say that C' *effects pruning of* C. When a node C' satisfies the first condition, i.e., $MinD(C, Q) > MaxD(C, C')$, we say that it *supports* C for pruning; this is because it contributes a set of objects that are closer to *each* object in $Obj(C)$ than the query. If such a supporter has enough descendants, it could satisfy the second condition, and thus effect pruning. In our example, E both supports and effects pruning of H since $MinD(H, Q) = 0.45$ is greater than $MaxD(H, E) = 0.4$ and $|Obj(E) \cup Obj(H)| = 2$ is greater than k. The fact that $MaxD(C, C')$ is always greater than $MaxD(C)$ implies that a node C can have other nodes effecting its pruning only if $MinD(C, Q) > MaxD(C)$.

Multiple supporters could together effect pruning of C. Consider the case where we want to check whether a supporter set $\{C_1, C_2, \ldots, C_f\}$ would enable pruning of C. Intuitively, each one of these C_is would then have to satisfy the condition $MinD(C, Q) > MaxD(C, C_i)$ and all of these should collectively build up a set of cardinality at least $k+1$.

$$MinD(C, Q) > max\{MaxD(C, C_i)\} \wedge |Obj(C) \cup \bigcup_i Obj(C_i)| > k$$

We could incrementally build up such sets of supporters until they have enough descendants across them to affect pruning of C. Let us consider the node E in our example and the set $\{D, H\}$; since $MaxD(E, D) = 0.3$ and $MaxD(E, H) = 0.40$, both D and H are supporters of E, $MinD(E, Q) = 0.85$ being greater. Also, the number of descendants across E, D & H turns out to be 4, which obviously is higher than k (i.e., 1). Thus, this set of supporters is able to prune E whereas the SPC was not able to. We are able to prune nodes E, F and H using this condition as illustrated in Figure 4. This condition obviously is more powerful than SPC. Our technique for RkNN retrieval using the AL-Tree works by exploring the AL-Tree starting from the root, maintaining support information of nodes and effecting pruning whenever possible. Pruning merely enables us to judge that none of the descendants of a candidate would be in the RkNN. Pruned candidates and their children may still be useful to prune other candidates.

C	Set (S)	MinD(C,Q)	max{MaxD(C,$s \in S$)}	$\lvert Obj(C) \cup \bigcup_s Obj(s) \rvert$
E	{D,H}	0.85	0.4	4
D	{E}	0.75	0.3	3
H	{E}	0.45	0.4	2

Figure 4: Nodes Pruned using Supporters.

A simple algorithm easily follows from the outlined pruning conditions; one that compares nodes pairwise and prunes out all prunable candidates (nodes). The rest form the result set. We illustrate such an algorithm in Appendix C. Such an approach, however, has a significant disadvantage; since RkNN result sets are typically small, most of such comparisons involve at least one node that has been pruned, and mostly are between nodes that would get pruned eventually. We present an improved approach that tries to remedy this and also strives to be more disk friendly, in the next section.

4.2.2 ALT-RkNN Algorithm

We now present the improved RkNN Algorithm (Algorithm 1) that exploits pruning to optimize on disk access and computational costs. The approach works by performing a depth-first traversal of the AL-Tree, updating the stack after each expansion to discard any candidates possible to

Alg. 1 ***ALT-RkNN***

```
1. C = [Root]
2. Root.spt = φ
3. while(¬(∀_{c∈C} Leaf(c)))
4.       c = pop(C)
5.       CC = children(c)
6.       ∀c' ∈ CC, c'.spt = c.spt
7.       ∀s ∈ {C ∪ CC}
8.            ∀c' ∈ CC, c' ≠ s
9.                 if(s ∈ C ∧ !supports(c, s) ∧ supports(c', s))
10.                     s.spt+ = |Obj(c')|
11.                if(s ∈ C ∧ !supports(s, c) ∧ supports(s, c'))
12.                     c'.spt+ = |Obj(s)|
13.                if(s ∈ CC ∧ supports(s, c'))
14.                     c'.spt+ = |Obj(s)|
15.      C = {c | c ∈ {C ∪ CC} ∧ (c.spt + |Obj(c)|) ≤ k}
16. C' = [Root]
17. ∀_{s∈C} s.spt = 0
18. while(C' ≠ φ ∧ C ≠ φ)
19.      c' = pop(C')
20.      if(!c'.isLeaf)
21.           C'.push(children(c')); continue;
22.      ∀s ∈ C
23.           if(MinD(s, Q) > MaxD(c', s))
24.                s.spt+ = |Obj(c')|
25.                if((|Obj(s)| + s.spt) > k)
26.                     C.remove(s)
27. return C
```

be discarded using the additional information from that expansion. We maintain, along with each candidate, a count of the number of descendants across candidates that support the present candidate in a variable *spt* (we do not need to know who the supporters are, since we are interested only in determining whether the node would get pruned or not); this excludes the descendants of itself. Pruned candidates are useful since they could effect pruning of other yet-to-be-seen candidates. Since we discard pruned candidates as and when they are found to be pruned, we may have a superset of the RkNN result set at the end of the depth-first traversal. We start with the only candidate in the set C being the *Root* node, which gets expanded to its children, CC, (D, E and F in our running example) in line 5. Children inherit their parents supporters and hence have their *spt*'s initialized thus in line 6. This property is due to the way in which $MaxD(.,.)$ is computed; $MaxD(.,.)$ between a pair of nodes is an upper bound of $MaxD(.,.)$ where a node is replaced by a child. *Thus, any node that supports another would support all of latter's children.* We use $supports(A, B)$ as a shorthand for the check whether A supports B. The expansion could cause two kinds of changes; the new children (i.e., CC) could find supporters among those nodes that did not support its parent (lines 11-12), whereas other candidates could find supporters among the new children (lines 9-10). Since we only maintain and inherit counts (and not sets of supporters), we need to be careful to exclude incrementing *spt* for supporters whose support is inherited through the parent. Also, nodes in CC may support one another; such adjustments are made in lines 13-14. In our example, D, E & F have their *spt*s initialized to 0 and eventually have it updated to 1, 2 and 0 respectively through lines 13-14. Line 15 discards pruned candidates from C (C is a stack, its modeled as a set for notational convenience); in our example, D and E would get excluded. The only candidate left, i.e., F would then be expanded. However, none of its children attain pruning conditions through lines 9-14 of the next iteration. This is because, E despite being able to effect pruning of H has been discarded earlier. We are left with a superset of RkNN results at the end of the traversal.

The set C is then refined to build the RkNN set by another pass over the AL-tree. The supporters counts of all nodes in C are reset (line 17). The second pass over the tree (starting at line 18) also proceeds in depth-first fashion and examines each leaf that is encountered against all candidates in C. If a leaf is capable of supporting a candidate in C, the *spt* of the latter is updated (line 24). Any possible pruning is performed then (lines 25-26). Since this pass traverses *all* leaves (it could stop earlier only if C becomes empty earlier), C is left with only RkNN results at the end. In our example, we start with G and H and are left with G at the end; H gets pruned when E is encountered in the traversal.

The strategy works like a *Filter-Refine* approach with the first phase filtering out pruned candidates; the second phase refines the remaining to weed out non-RkNN objects.

Memory Usage: Similar to algorithms for skyline and top-k query processing [13, 22], ALT-RkNN needs at least as much memory as the result set. During the first phase, the memory requirement is determined by the maximum size of the candidate set across iterations. The depth-first traversal avoids a lot of shallow internal nodes (which, being shallow, are less likely to get pruned easily) from being held in memory; however, ALT-RkNN may end up maintaining a lot of leaf nodes that could have been pruned using previously pruned candidates. Thus, the candidate set size is strongly dependent on the effectiveness of the pruning conditions. *The algorithm can be easily adapted to work with a constant upper bound on memory - we outline the approach here.* When the number of candidates exceeds the memory size, we spill some of the candidates to disk. Only leaf candidates are spilled to disk since internal node candidates need to be further expanded. Due to the depth first traversal, the number of internal node candidates is bounded by the depth of the tree, so they can be easily held in memory. The candidates written out to disk are potential RkNN candidates which need to be re-examined in the second phase, in addition to the non-spilled candidates. This is done in multiple passes, by loading as many candidates in each pass as can be held in memory. We show in Section 5 that memory requirements tend to be extremely small; the need to spill candidates to disk, hence, may not arise in practice.

Bichromatic Queries: Bichromatic RkNN queries involve data of two classes. The query object (e.g., a pizza store) comes from one class, whereas the objects to be retrieved are of a different class (e.g., customers). ALT-RkNN can be easily adapted to handle such cases; we give a rough sketch here. The AL-Tree is built on the common schema on the union of the two sets of objects, whereas each internal node maintains the list of descendants coming from the two classes separately. The support counts are incremented using just the number of descendants that are of the query type. Thus, the query type objects serve to prune out retrieved type objects and do not figure in the results themselves.

Extensions: In many scenarios, objects have numeric and categorical attributes. Further, the distance could well be a

generic monotonic aggregate as against a weighted sum. We discuss handling of such cases in Appendix D.

5. EXPERIMENTS

We now describe our experimental study where we compare our approach against the *Naive* and the block-wise *BR* algorithms (description in Appendix B).

5.1 Experimental Setup

Our experiments were run on an IBM X Series with Windows Server 2003 on a Pentium 3.4 GHz Processor with 2.0 GB RAM. We compare the algorithms based on the disk access (IO) costs, computational cost and response time.

IO costs are measured in terms of page IOs. Random page IO is costlier than sequential page IO; we set the cost ratio to 10 [22]. The aggregate IO cost is then the ratio-based weighted sum of the IO costs. The available memory can be used as a LRU cache to reduce IOs since repetetive requests for the same page could be served from the cache. We assume that the memory available is 5% of the dataset size for our experiments. An analysis of IO performance against varying ratios and varying memory in Appendix F.

The *computational time* is the sole indicator of the cost when the database can be held in memory. To isolate the computational costs from the IO costs, we use a scenario where all the objects and indexes are loaded in memory; all costs become purely computational (as IO is eliminated) then. For a disk based system, *response time* is the most significant measure, being the measure visible to the user. We simulate the disk based implementation assuming page access costs to be 1ms and 10ms for sequential and random access respectively for a page size of 32 KB. These estimates are based on reported figures on popular platforms [10, 1]. The *response time* is the sum of the computational time and the IO costs, measured in terms of time. All numbers reported are result of averaging over 100 random queries.

We use depth-first packing of the AL-Tree on disk with sibling ordering chosen randomly at index-creation. *The attributes in the AL-Tree are arranged in the increasing order of the number of distinct values. Top-k query processing benefits from such an ordering [13]; we observed the same for RkNN processing also. The intuition is that lesser number of distinct values leads to more objects in each sub-tree, enabling better group level reasoning that is used for pruning.*

5.1.1 Datasets

We use two real-world datasets, *ForestCover*[2] and *Census-Income* (CI)[3] having densities 0.0004 and 0.069 respectively, and a synthetic dataset generated from a normal distribution for our experiments. Details appear in Appendix E.

5.2 Performance on Real Data

In these experiments, we compare ALT-RkNN and BR algorithms against the *Naive* approach by varying the value of k from 1 to 20. We also analyze the memory requirement of the ALT-RkNN algorithm with varying k.

Computational Costs

The computational costs of the various algorithms over the CI and FC datasets are illustrated in Figure 5 and Figure 6 respectively; the Y-axis plots the time in ms against k on the X-axis. The *Naive* as well as the *BR* approach are seen to slow down linearly with increasing k (at different rates). Interestingly, the ALT-RkNN approach is seen to be relatively insensitive to varying k. This could partly be traced back to the skewed distribution of the real datasets. Objects are often clustered around certain attribute value combinations, and hence, internal nodes in those dense subtrees tend to have enough descendants to get pruned, or to be able to effect the pruning of other candidates. Such value-space effects are not visible to algorithms that deal with objects one at a time, and hence, *BR* and *Naive* are both unable to harness the skewed distribution of the datasets to their advantage. In summary, ALT-RkNN is seen to outperform both *BR* and *Naive* by large margins in computational costs.

IO Costs

Now, we study the IO costs of the various algorithms. The exhorbitant IO costs incurred by the *Naive* approach due to processing one object at a time is significantly brought down by employing a block based approach such as *BR*. ALT-RkNN, on the other hand, performs just two sequential scans over the AL-Tree index (*sequential scans since the tree is visited in the query-independent depth first packing order*); one for each pass. Figures 7 and 8 illustrate the effects with the IO costs plotted in logarithmic scale on the Y-axis against k on the X-axis. ALT-RkNN is seen to be an order of magnitude better than *BR* and upto three orders of magnitude better than *Naive*. Regardless of k, ALT-RkNN performs only two sequential scans on the index, and hence scales well with k in terms of IO costs. Some more analyses on IO performance appear in Appendix F.

Response Time

ALT-RkNN outperforms both the other approaches in the composite response time measure too. The charts for the CI and FC datasets are in Figure 9 and Figure 10 respectively (time in ms against k). For the CI dataset, the response time of *Naive* was around 30 times that of *BR* and ALT-RkNN; hence, it is not visible. ALT-RkNN performs increasingly well with denser datasets since higher density leads to more efficient pruning. It is hence, promising to note that ALT-RkNN outperforms both *BR* and *Naive* by healthy margins even on a very sparse dataset such as FC.

ALT-RkNN Memory Usage

Among the algorithms that we consider for our experiments, ALT-RkNN has a varying memory requirement, parameterized heavily by the effectiveness of the pruning conditions. We do have a way to limit the memory usage to a prespecified upper bound (Refer Section 4.2.2); here, we illustrate that the memory requirement is small enough that we practically do not have to resort to such techniques in real scenarios. We empirically analyze the memory requirement as a *fraction* of the dataset size. The behavior of this measure with varying k is plotted in Figure 11. The memory requirement for the ALT-RkNN is seen to be extremely small, almost 4 orders of magnitude smaller than the dataset size. ALT-RkNN, thus, requires an extremely small amount of memory to work with, in most practical scenarios.

5.3 Performance on Synthetic Normal Data

We study the performance of the various algorithms over

[2] http://kdd.ics.uci.edu/databases/covertype/covertype.data.html

[3] http://kdd.ics.uci.edu/databases/census-income/census-income.html

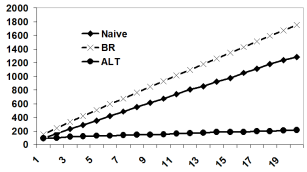

Figure 5: Computation (ms) vs. k (CI)

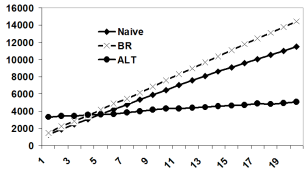

Figure 6: Computation (ms) vs. k (FC)

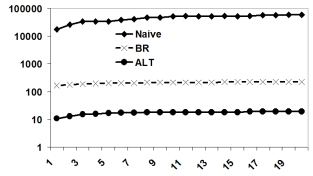

Figure 7: Disk I/O cost (ms) vs. k (CI)

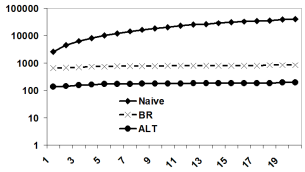

Figure 8: Disk I/O cost (ms) vs. k (FC)

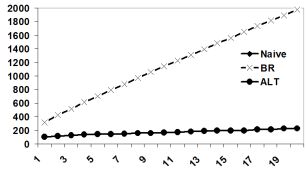

Figure 9: Response Time (ms) vs. k (CI)

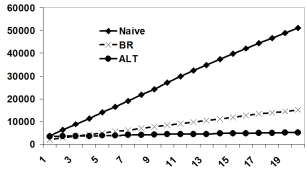

Figure 10: Response Time (ms) vs. k (FC)

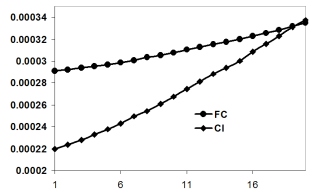

Figure 11: ALT-RkNN Memory Usage vs. k

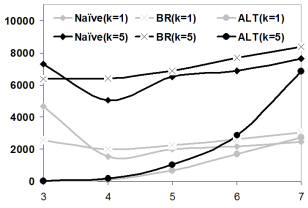

Figure 12: Response Time (ms) vs. #Attribs (Normal Data)

normally distributed (non-uniform) synthetic data. Synthetic data enables us to vary data size, number of values per attribute etc; Appendix E describes the dataset in detail.

Varying Dataset Sizes

For this experiment, we vary the dataset size from 0.1 to 1.2 million keeping the number of attributes and number of values per attribute constant at 5 and 50 respectively. This varies the data density from 0.0003 to 0.003. The response time plots for $k = 1$ and $k = 5$ are in Figures 13 and 14 respectively. The behavior of *Naive* is worth noting; at very low densities, the sparsity forces the *Naive* approach to go significantly down the list of data objects before objects close enough can be found. This operation is done for each object; causing huge IO costs leading to larger response times. ALT-RkNN is seen to consistently outperform both *Naive* and *BR* for both values of k. This suggests that ALT-RkNN is able to leverage the skew in the distribution to its advantage.

Varying Number of Values per Attribute

We now keep the dataset size and number of attributes constant at 1 million and 5 respectively, varying the number of values per attribute from 45 to 70 in steps of 5; this varies the density from 0.0005 to 0.005. The results for $k = 1$ (Refer Figure 15) show that ALT-RkNN significantly outperforms the others. At $k = 5$ (Refer Figure 16), the gap between ALT-RkNN and the other approaches widen further.

Varying Number of Attributes

AL-Tree achieves best performance when the number of attributes are lesser (the tree is then shallower) and the data is dense (that leads to increasing number of objects having the same value for an attribute). We evaluate the performance of the various algorithms with varying number of attributes. We keep the dataset size at 1 million and the number of values per attribute at 50, and vary the number of attributes from 3 to 7 (decreasing density and increasing tree depth in the process). The response time plots for $k = 1$ and $k = 5$ are presented in Figure 12. It is seen that ALT-RkNN outperforms both *Naive* and *BR* at lesser number of attributes. *Naive* and *BR*, being object based approaches, are relatively insensitive to the number of attributes since the dataset size remains constant. It is interesting that ALT-RkNN is still competetive to the object based approaches even with 7 attributes at a very low density of $1.28 * 10^{-6}$.

5.4 Discussion

From our experiments, we find that ALT-RkNN computational and IO costs are insensitive to k, however, the memory usage does go up by small amounts with increasing k. ALT-RkNN is orders of magnitude cheaper than other approaches in terms of IO costs, making it the very obvious choice for disk based implementations. The performance of ALT-RkNN is however seen to deteriorate gracefully with decreasing densities (as is the case with spatial indexes like R-Trees), but is still seen to be competetive to BR at even densities of the order of 10^{-6}. At very low densities, with very few attribute values repeating in the dataset, object based approaches become more appropriate.

6. CONCLUSIONS

We have proposed an approach, ALT-RkNN, for efficient RkNN query processing with arbitrary non-metric similarity measures. This approach exploits the various properties of the AL-Tree value space index to perform efficient RkNN search on the AL-Tree. We illustrate the effectiveness of our approach over the *Naive* approach and the *BR* approach that performs block-based accesses for RkNN processing through a series of experiments on real and synthetic datasets. ALT-RkNN is seen to outperform other approaches on real and normal data by large margins across varying IO cost ratios, memory sizes and densities. It is able to harness the skew in the data distribution to its advantage and is seen to perform better with increasing density and k. Thus, ALT-RkNN is seen to be the algorithm of choice for most real scenarios.

The AL-Tree has been found to be effective for top-k [13], skyline [22] and RkNN retrieval with arbitrary similarity measures. We are studying its applicability for queries such

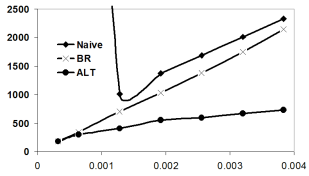

Figure 13: Response Time (ms) vs. Density (k=1) (Varying data size, Normal Data)

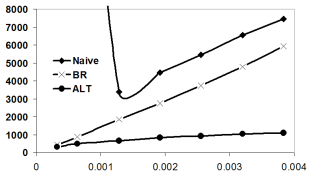

Figure 14: Response Time (ms) vs. Density (k=5) (Varying data size, Normal Data)

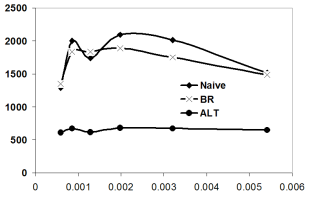

Figure 15: Response Time (ms) vs. Density (k=1) (Varying #Values, Normal Data)

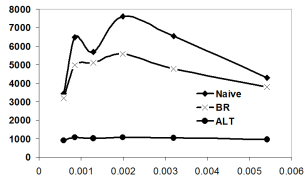

Figure 16: Response Time (ms) vs. Density (k=5) (Varying #Values, Normal Data)

as reverse skyline [12] and for finding a subset of interesting results for skyline & RkNN queries. AL-Tree is specifically designed with categorical attributes in mind; we are exploring how bucketing of contiguous values could help in dealing with numeric attributes, while retaining the bounds.

7. REFERENCES

[1] How fast is your disk? *http://www.linuxinsight.com/how_fast_is_your_disk.html*, January 2007.

[2] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD Conference*, pages 515–526, 2006.

[3] E. Achtert, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT*, pages 886–897, 2009.

[4] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *ICDE*, 2008.

[5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 1975.

[7] G.-H. Cha. Non-metric similarity ranking for image retrieval. In *DEXA*, pages 853–862, 2006.

[8] H. Chen, R. Shi, K. Furuse, and N. Ohbo. Finding rknn straightforwardly with large secondary storage. In *INGS*, 2008.

[9] O. Cheong, A. Vigneron, and J. Yon. Reverse nearest neighbor queries in fixed dimension. *CoRR*, abs/0905.4441, 2009.

[10] W. Chung, Gray and Horst. Windows 2000 disk io performance. *Microsoft Research TR*, June 2000.

[11] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[12] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.

[13] P. M. Deshpande, D. P, and K. Kummamuru. Efficient online top-k retrieval with arbitrary similarity measures. In *EDBT*, pages 356–367, 2008.

[14] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[15] K. Goh, B. Li, and E. Chang. Dyndex: A dynamic and nonmetric space indexer. In *ACM Intl. Conference on Multimedia*, 2002.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[17] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD Conference*, pages 201–212, 2000.

[18] H.-P. Kriegel, P. Kröger, M. Renz, A. Züfle, and A. Katzdobler. Reverse k-nearest neighbor search based on aggregate point access methods. In *SSDBM*, pages 444–460, 2009.

[19] K. C. K. Lee, B. Zheng, and W.-C. Lee. Ranked reverse nearest neighbor search. *IEEE TKDE*, 20(7):894–910, 2008.

[20] J. Lin, D. Etter, and D. DeBarr. Exact and approximate reverse nearest neighbor search for multimedia data. In *SDM*, pages 656–667, 2008.

[21] G. Murphy and D. Medin. The role of theories in conceptual coherence. In *Psychological Review*, 1985.

[22] D. P, P. M. Deshpande, D. Majumdar, and R. Krishnapuram. Efficient skyline retrieval with arbitrary similarity measures. In *EDBT*, 2009.

[23] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, pages 91–98, 2003.

[24] T. Skopal and J. Lokoc. Nm-tree: Flexible approximate similarity search in metric and non-metric spaces. In *DEXA*, pages 312–325, 2008.

[25] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *In SIGMOD Workshop on DMKD*, pages 44–53, 2000.

[26] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.

[27] M. Vlachos, D. Gunopulos, and G. Kollios. Robust similarity measures for mobile object trajectories. In *DEXA 2002*, 2002.

[28] C. Xia, W. Hsu, and M.-L. Lee. Erknn: efficient reverse k-nearest neighbors retrieval with local knn-distance estimation. In *CIKM*, 2005.

[29] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.

[30] J. L. Yanmin Luo, Canhong Lian and H. Chen. Finding rknn by compressed straightforward index. In *ISKE*, 2008.

[31] M. L. Yiu and N. Mamoulis. Reverse nearest neighbors search in ad hoc subspaces. *IEEE TKDE*, 19(3):412–426, 2007.

APPENDIX

A. RKNN SEARCH IN METRIC SPACES

The Reverse Nearest Neighbor query (RkNN query with k=1) and its variants were introduced in [17]; this also proposes the RNN-Tree that facilitates the processing of such queries. The RNN-Tree, a variant of the R-Tree [16] makes use of a static distance function to pre-compute the distance to the nearest neighbor from each object in the database. Vicinity circles of this radius are built around each data point; for a query, the RkNN result is the set of all points within whose vicinity circles the query point falls. The RdNN [29] tree supports both NN and RkNN queries for various values of k using a single index when k is specified at index creation time. Such vicinity circles cannot be precomputed with dynamic distance functions; the setting that we explore in this paper. Estimates of kNN distances [28] have been shown to help querying with various values of k in RkNN. The MRkNN-CoP-Tree[2] is the first approach for RkNN search over general metric spaces where the value of k is specified at query time (as opposed to index creation time, as for the earlier approaches) uses estimates of kNN distances at run-time to guide the search process. Metric spaces enable reasoning about the distance between a pair of objects using their distances to a third object using the triangle inequality. Such reasoning can be done at an aggregate level to prune multiple objects in a single operation [3]. Euclidean spaces, being a specific type of metric spaces, have been exploited to affect other types of optimizations in RkNN search [9, 18, 20]. *Filter-Refine* approaches comprise two phases where the filter phase is used to select a subset of the objects from the database that are necessarily a superset of the RkNN results (but may include false hits); the refine phase identifies such false hits and removes them. It has been shown that fast boolean range queries in metric-space indexes [23] and certain properties of well-studied R-Tree indexes [25] are good tools in the filter phase. [26] employs a best-first search on the R-Tree index in the filter phase. [31] presents a unifying filter refine framework for RkNN search incorporating various filtering and verification approaches for metric spaces. None of these techniques can be applied in the case of non-metric spaces. Approaches for approximate RkNN try to compromise on the accuracy of the results, and may incur some false misses in a bid to speed up the processing. An approach [23] restricts the search space to the k' nearest neighbors of the query, where larger values of k' reduce the chances of false misses.

B. SEQUENTIAL ALGORITHMS FOR RKNN

Sequential algorithms do not build any indexes and work by scanning the database; hence, they are applicable for non-metric dissimilarity functions. We first consider a naive algorithm and then an improved version of it.

B.1 Naive Algorithm

A naive approach for RkNN retrieval would check every object for neighbors that are closer to it than the query; if there are fewer than k such neighbors, the object in question would be part of the RkNN result set. As illustrated in Algorithm 2, for each object $D[i]$ under consideration, the count of objects nearer than the query object and a flag are initialized to 0 and *true* respectively in line 3. The count is incremented as and when more objects are found to be closer

Alg. 2 ***Naive***

```
1. R = φ
2. ∀_i, 0 ≤ i < |D|
3.       count = 0, flag = true
4.       ∀_j, j ≠ i ∧ 0 ≤ j < |D|
5.             if (d(D[i], D[j], W) < d(D[i], Q, W))
6.                   if (count++ = k){flag = false; break;}
7.       if(flag) R = R ∪ {D[i]}
8. return R
```

to $D[i]$ than the query object; this continues till the count reaches k, at which point it can obviously be confirmed to be not part of RkNN set (by resetting the flag in line 6). All those objects that are found to have fewer than k neighbors nearer to them than the query, are added to the result set in line 7. This completes the process. The naive approach may do as many complete scans of the database as there are objects, and thus could incur high IO cost.

B.2 Blockwise with Refilling (BR)

The Naive algorithm is similar to doing a self join of the database D with itself. The join condition between two objects O_1 and O_2 is that O_2 is closer to O_1 than the query object Q. For each object O, the number of objects it joins with is counted and if this number is less than k, O is part of the result set. Similar to block based join algorithms, we can have a block based version of the Naive algorithm as shown in Algorithm 3. Rather than scanning the database once for each object as in the Naive, it makes use of the available memory to hold several candidates and processes them in the same scan of the database. Assuming that there is sufficient memory to hold t pages with r objects each, it starts by scanning $t-1$ pages (leaving one page free for doing the sequential page-by-page scan of the database) and creating a candidate set C in line 1. Then it scans the database and compares the objects with the candidates in memory in lines 7 to 10. If a candidate has more than k objects closer to it than the query, it is removed from the candidate set. The refilling step does further optimization by reading in the next batch of r objects (i.e. the next page) whenever r objects have been eliminated from the candidate set. That way more objects are processed in the same scan of the database. However, this implies that for each candidate, we have started comparing that from different points in the database, depending on when that candidate was loaded. To handle this, we keep track of the number of objects each candidate is compared against in line 5. If a candidate has been compared with all objects in the database ($|D|$ in number), it can be removed from the candidate set and added to the result if its *count* is less than k.

C. SIMPLE RKNN ALGORITHM

In this section, we present a simple RkNN Algorithm (Algorithm 4) to illustrate how pruning conditions outlined in Section 4.2.1 could be used to search the AL-Tree for RkNN. The algorithm starts off with just the root node in the candidate set and progresses by repeatedly picking a node from the candidate set and expanding it to it's children - which are then added to the candidate set in lieu of the picked node (line 4). This simple algorithm keeps track of what

Alg. 3 ***Blockwise with Refilling (BR)***

Config: t **pages, each capable of holding** r **objects**

```
1. Scan t pages and create candidate set C
2. While C is not empty
       /* Scan the database */
3.     For each object O in the DB
4.         For each candidate c in C
5.             c.compared++
6.             if(c = O)continue
7.             if O is closer to c than Q
8.                 c.count++
9.                 if (c.count = k)
10.                    remove c from C
11.            if (c.compared = |D|)
12.                remove c from C
13.                if (c.count < k) add c to R
14.            if r candidates have been removed
15.                scan next page and create new
                   candidates in C
16. return R
```

Alg. 4 ***Simple RkNN***

```
1. CS = {Root}
2. while(¬(∀c∈CS Leaf(c)))
3.     C = getNext(CS)
4.     CS = CS ∪ children(C)
5.     for all (c ∈ CS, c' ∈ children(C))
6.         compute MaxD(c, c')
7.     Identify pruned nodes and mark them in CS
8. Return all objects from non-pruned nodes in CS
```

nodes are pruned when each expansion is performed (lines 5-7). This process goes on until the candidate set contains only leaf nodes; at this point, all nodes that have not yet been pruned comprise the RkNN result set.

Line 5 of the algorithm is a very expensive process since it involves computation of the $MaxD(.,.)$ for every candidate in CS (CS also includes pruned candidates). This is counter-intuitive to our original goal of using the pruning conditions to prune out candidates and using such pruning to optimize on computational costs. Our ALT-RkNN approach (Section 4.2.2) makes use of a candidate maintenance strategy optimizing various such costs.

D. EXTENSIONS TO ALT-R$_K$NN.

In this section, we consider handling of generic monotonic aggregate dissimilarity functions and objects where certain attributes are numeric.

D.1 Generic Monotonic Aggregate Dissimilarity Functions

The ALT-RkNN algorithm works by checking between $MaxD(.,.)$ and $MinD(.,.)$ and affecting pruning whenever possible. We will now show that such functions are well-defined in the case of any monotonic aggregate function. Consider two objects where their dissimilarity on the i^{th} attribute is denoted by d_i, $0.0 \leq d_i \leq 1.0$. Let the dissimilarity function be denoted by $f(\ldots)$. The following inequality then holds, $f(\ldots)$ being a monotonic aggregate:

$$f(d_1, \ldots, d_k, 0.0, 0.0, \ldots, 0.0) \leq f(d_1, d_2, \ldots, d_m)$$

$$\leq f(d_1, \ldots, d_k, 1.0, 1.0, \ldots, 1.0)$$

Thus, any of the d_is being replaced by 0.0 would lead to a lower bound of $f(\ldots)$ whereas any of them being replaced by 1.0 would give an upper bound of the same (the condition above is a special case where only some trailing d_is are changed). Now, consider two candidate prefixes, $X = [a_1 = x_1, \ldots, a_j = x_j], j \leq m$ and $Y = [a_1 = y_1, \ldots, a_k = y_k]$ with their dissimilarity on the i^{th} attribute being denoted by d_i. $MaxD(.,.)$ and $MinD(.,.)$, being upper and lower bounds of pair-wise dissimilarities between objects (one from $Obj(X)$ and another from $Obj(Y)$), can then be defined as follows:

$$MaxD(X, Y) = f(d_1, d_2, \ldots, d_{min\{j,k\}}, 1.0, \ldots, 1.0)$$

$$MinD(X, Y) = f(d_1, d_2, \ldots, d_{min\{j,k\}}, 0.0, \ldots, 0.0)$$

$MaxD(X)$, the upper bound of dissimilarity between objects in $Obj(X)$ would then be intuitively the same as $MaxD(X, X)$. The ALT-RkNN algorithm can trivially be adapted to such monotonic aggregate dissimilarity functions by using the $MinD(.,.)$ and $MaxD(.,.)$ bounds defined above instead of their counterparts in Section 4, the rest remaining the same.

D.2 Handling Numeric Attributes

Spatial indexes such as R-trees [16] are found to be very effective for similarity queries on numeric attributes. In most cases, objects tend to have some numeric attributes in addition to categorical ones. The dissimilarities between two objects would then typically be the sum of their dissimilarities on the numeric attributes (as computed using some metric dissimilarity measures such as $L1$ or $L2$ norm etc.) and that on categorical attributes as computed using the appropriate monotonic aggregate dissimilarity function. We outline a simple approach to utilize R-Tree indexes within the ALT-RkNN framework. This involves building an R-tree on numeric attributes at each leaf node of the AL-Tree (built on categorical attributes), each such tree indexing just the objects mapping to its respective leaf node in the AL-Tree. The ALT-RkNN algorithm could then progressively deepen the search beyond the AL-Tree leaf node by utilizing the R-Tree hierarchy; we now illustrate how the $MinD(.,.)$s and $MaxD(.,.)$s could be computed for such a traversal. The bounds between pairs of AL-Tree nodes are trivial; that between an AL-Tree node and an R-Tree node would then be the bounds between the former and the AL-Tree leaf node that the latter is associated with. Among R-Tree nodes, the actual distances between their corresponding AL-Tree leaves are known; the upper bound is then formed by adding the maximum distance between any pair of corners of the corresponding bounding rectangles (each R-Tree node has a corresponding bounding rectangle) whereas the lower bound is formed by adding the minimum distance between any two virtual points, one from each bounding rectangle. The ALT-RkNN adaptation is then obvious, given the upper and lower bounds. An alternative approach would be to invert the ordering and build an AL-Tree at the leaf node of every R-Tree node; bounds can then be similarly defined.

E. DATASETS USED

This section details the various datasets used in empirical evaluation, and the rationale behind the choice. The *Forest-Cover*[4] dataset (FC) contains data of the Forest Cover type for 581012 cells, each of size 30X30 meters over regions in the United States. The attributes chosen from the dataset had 67, 551, 2, 700, 2, 7 and 2 distinct values leading to a low data density of 0.04%[5]. The *Census-Income* dataset (CI)[6] contains census data for 199523 people for 1970, 1980 and 1990 from the Los Angeles area. We choose a subset of attributes, namely *Age, Education, Number of Minor Family Members, Number of Weeks Worked and Number of Employees*, from the dataset, based on their utility in measuring similarities between people. The attributes chosen have 91, 17, 5,53 and 7 distinct values respectively leading to a high density of 6.9%. Datasets of widely varying densities were chosen since that would help to generalize the empirical observations better. The similarities between different values of attributes are chosen randomly from the interval [0-1]. The ALT-RkNN algorithm makes use of group level reasoning that is enabled by the tree structure of the AL-Tree data structure. Skewed distribution, as is often the case with real-world datasets, poses a favorable case for the ALT-RkNN algorithm since the value space compression is effectively used to enable effective group-level reasoning and pruning. To be more comprehensive, we used synthetic datasets as well. Usage of synthetic data enables us to test for varying densities as well as varying data sizes. We generated synthetic data with uniform random and normal distributions. Uniform random nature of data distributes data evenly across the various values of attributes, and is hence, an unfavorable case for the ALT based approach. Our experiments illustrate that the ALT-RkNN approach is still competetive even in those adverse scenarios (Refer Appendix G). Normal distribution[7] is often considered to exhibit many characteristics that are often associated with real data. Hence, we choose to study the performance of the approaches when applied to synthetic data that come from a normal distribution as well. For all the synthetic datasets, we generate the similarities between attribute values using a random number generator.

Synthetic Normal Data

Normal data is characterised by a probability distribution concentrated around the *middle* values; this makes it tricky for generating non-metric space data since there is no global ordering of values in such spaces. However, to build a normal distribution, we assume an ordering of values for each attribute, and generate data to ensure that the distribution is normal and hence is heavily concentrated around the middle values in the chosen ordering. It may be noted that we still generate similarities between different values randomly; hence values around the middle of the chosen ordering are not designed to have higher similarities to each other. We use a uniform random number generator and rejection sampling[8] to generate normal data. We choose the variance to be 3, and the mean to be the index of the middle attribute in the ordering chosen for data generation. The difference in distribution of values in the normal data (as against uniform random data) is expected to significantly influence the performance of various approaches.

[4] http://kdd.ics.uci.edu/databases/covertype/covertype.data.html

[5] Data density is computed as the ratio of the number of data objects to the maximum number of distinct tuples in the space

[6] http://kdd.ics.uci.edu/databases/census-income/census-income.html

[7] http://en.wikipedia.org/wiki/Normal_distribution

[8] http://en.wikipedia.org/wiki/Rejection_sampling

F. IO PERFORMANCE ANALYSIS

In this section, we analyze the IO performance of the *Naive*, *BR* and ALT-RkNN algorithms against varying cost ratios and varying available memory sizes on the *Forest-Cover* and *Census-Income* datasets (Refer Appendix E).

IO Performance with Varying Ratios

We now analyze the IO costs of the various techniques against varying ratios between random and sequential access costs. The ratio depends on the characterstics of the storage system and vary a lot [22]; this makes an analysis of IO costs with varying ratios interesting. We plot the IO costs against varying ratios in Figure 19. This confirms that the margins in terms of IO costs among the various approaches (as observed earlier in Figure 7) hold up for a variety of ratios. A similar behavior was observed for the FC dataset too; we omit the chart due to space constraints.

IO Performance with Varying Memory Sizes

Cache sizes are critical to disk intensive algorithms that tend to revisit disk blocks. The *Naive* approach may visit a page upto $|D|$ times, once for each object in the database. The ALT-RkNN approach works by making upto two full passes over the AL-Tree index; the first pass calculates the shortlist of candidates from which false positives are eliminated in the second pass. Both these passes are at most as costly as simple sequential scans over the index since the candidates are processed in the query-agnostic depth first manner that they are stored. However, this means that the ALT-RkNN could visit each page upto twice. With more available cache, the *BR* approach could hold more objects' state information together, thus reducing the number of scans of the database. Thus, all the approaches that we evaluate in this paper are influenced by cache sizes, by small or large amounts. As described in Section 5.1, the available memory is used as a LRU cache for the *Naive* and ALT-RkNN algorithms and for holding the candidates in the BR algorithm. Now, we analyze the performance of these approaches over varying memory sizes. We vary the memory size from 3% of the dataset size to 15%. The variation in IO costs for the CI dataset is shown in Figure 17. The *Naive* approach shows a steep fall in IO costs with increasing cache sizes since larger number of revisits are served by the cache. The *BR* and ALT-RkNN approaches have significantly small IO costs even in very low cache percentages, and are hence not much affected with variations in cache sizes. The chart for the larger and sparser FC dataset presented in Figure 18 shows a slightly different behavior in that the savings achieved by *Naive* with increasing cache sizes is not as conspicuous as for the CI dataset. This is because the FC dataset, being sparser, forces the *Naive* to go significantly deep down (thus scanning a larger number of objects) for every object before it can be eliminated. Thus, when it gets to the next object under consideration, the first page would have already been eliminated from the cache. This shows that the *Naive* is benefitted by increasing cache sizes on dense datasets whereas the *BR* and ALT-RkNN are relatively insensitive to it.

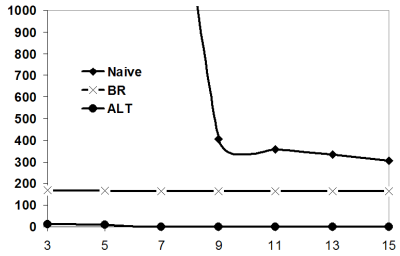

Figure 17: IO cost (ms) vs. Memory % (CI)

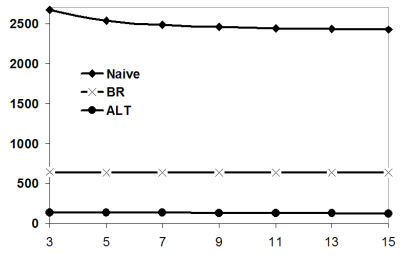

Figure 18: IO cost (ms) vs. Memory % (FC)

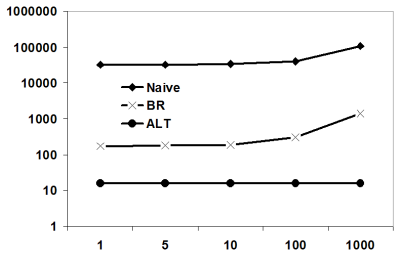

Figure 19: Disk IO Cost vs. Ratio (CI)

G. EXPERIMENTS ON UNIFORM RANDOM DATA

As stated in Appendix E, uniform random datasets pose an adverse scenario for the ALT-RkNN approach since even distribution of data among values causes least value space compression. Further, uniform random data is precisely the setting that causes subtrees to have maximally diverse descendants (with higher average distance among them); thus reducing the chances of early attaininment of the pruning conditions in Section 4.2.1. This is also an unfavorable case for *BR* and *Naive* approaches since they have reduced probabilities of finding very close neighbors (by virtue of the maximally diverse dataset induced by the uniform random distribution). This hypothetical setting of uniform random data provides us a means to analyze how ALT-RkNN (and *BR*) compare against the *Naive* approach on adverse scenarios. We study the behavior with varying data densities, where density is first varied by varying data size, and then by varying the number of values per attribute.

Varying Dataset Sizes

For this experiment, we vary the dataset size from 100000 to 1.2 million keeping the number of attributes and number of values per attribute constant at 5 and 50 respectively. This varies the data density from 0.0003 to 0.003. It is expected that increasing dataset sizes would lead to higher response times. The response time graphs for experiments with $k = 1$ and with $k = 5$ are represented in Figures 22 and 23 respectively. The behavior of *Naive* is similar to that in synthetic normal data; incurring very high IO costs at low densities. The overall trend of increasing response times with increasing data sizes is reflected in both the charts. Such effects are applicable to the *BR* approach too; however, it is less pronounced since *BR* decides on a large number of objects during each such pass in contrast to *Naive* which progresses by making decisions per object. At not-so-low densities, the ALT-RkNN is very competetive to *Naive* (Ref. Figure 22) whereas *BR* is seen to give better response times. For $k = 5$, both *ALT-RkNN* and *BR* are quite comparable to each other on response times; they outperform the *Naive* approach approximately by a factor of 3.

Varying Number of Values Per Attribute

We now study the effects of density (at constant dataset size) on the various algorithms. We keep the dataset size and number of attributes constant at 1 million and 5 respectively and vary the number of values per attribute from 45 to 70 in steps of 5; this varies the data density from 0.0005 to 0.005.

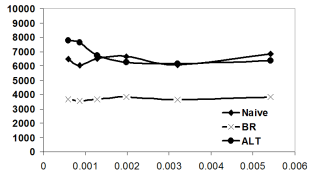

Figure 20: Response Time (ms) vs. Density (k=1) (Varying #Values, Uniform Random Data)

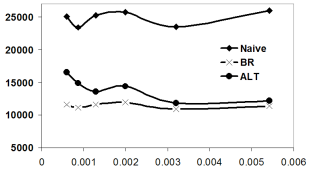

Figure 21: Response Time (ms) vs. Density (k=5) (Varying #Values, Uniform Random Data)

The better value space compression achieved by the AL-Tree when the number of values per attribute is low does not help much since the query also comes from the same space (this makes pruning less effective); this causes the query to have an increasing number of objects that take same values on various attributes. Approaches like *BR* and *Naive* that work by comparing objects are not benefitted by reduced number of values per attribute since they work by comparing objects. The results for $k = 1$ and $k = 5$ have been plotted in Figures 20 and 21 respectively. The response times do not exhibit significant variation (much on expected lines) with density since the dataset size is held constant at 1 million. Similar to the results for varying dataset sizes, the *Naive* and ALT-RkNN approaches are comparable in response times at $k = 1$ whereas *BR* and ALT-RkNN outperform *Naive* by close to a factor of 2 at $k = 5$.

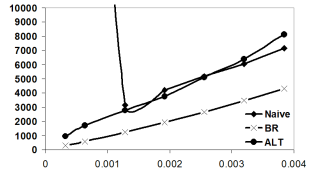

Figure 22: Response Time (ms) vs. Density (k=1) (Varying data size, Uniform Random Data)

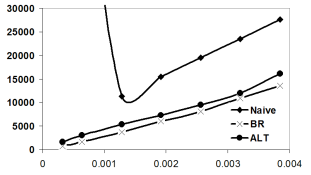

Figure 23: Response Time (ms) vs. Density (k=5) (Varying data size, Uniform Random Data)