Efficient Online Top-k Retrieval with Arbitrary Similarity Measures

Prasad M Deshpande
IBM India Research Lab
Bangalore, India
prasdesh@in.ibm.com

Deepak P
IBM India Research Lab
Bangalore, India
deepak.s.p@in.ibm.com

Krishna Kummamuru
IBM India Research Lab
Bangalore, India
kkummamu@in.ibm.com

ABSTRACT

The top-k retrieval problem requires finding k objects most similar to a given query object. Similarities between objects are most often computed as aggregated similarities of their attribute values. We consider the case where the similarities between attribute values are arbitrary (non-metric), due to which standard space partitioning indexes cannot be used. Among the most popular techniques that can handle arbitrary similarity measures is the family of threshold algorithms. These were designed as middleware algorithms that assume that similarity lists for each attribute are available and focus on efficiently merging these lists to arrive at the results. In this paper, we explore multi-dimensional indexing of non-metric spaces that can lead to efficient pruning of the search space utilizing inter-attribute relationships, during top-k computation. We propose an indexing structure, the AL-Tree and an algorithm to do top-k retrieval using it in an online fashion. The ALTree exploits the fact that many real world attributes come from a small value space. We show that our algorithm performs much better than the threshold based algorithms in terms of computational cost due to efficient pruning of the search space. Further, it outperforms them in terms of IOs by upto an order of magnitude in case of dense datasets.

Keywords

K-nearest neighbors, Index structures, similarity measures

1. INTRODUCTION

In this paper, we consider the problem of finding the top k objects that are similar to a given object from a database of objects. This is a fundamental problem that arises in many different situations such as information retrieval and any case of ranked retrieval of objects. Top-k algorithms evaluate similarities over multiples attributes of objects and combine the similarities using a monotone aggregation function. Top-k retrieval has been extensively studied in the past and many good algorithms and indexing structures have been developed [23]. The most popular simplifying assumption employed is that of metric spaces. The assumption that attribute similarities measures are metric, allows the creation of indexing structures that can exploit the triangle inequality property. Examples of metric-space based indexing structures for top-k retrieval include the widely used kd-tree [13] among others [20, 7, 21]. However, the triangle inequality property is too restrictive to model the (dis)similarities as perceived by humans (for example, [11]). This is particularly true in the case of similarities over categorical value spaces such as softwate products from multiple vendors, journals by different publishers etc. In this paper, we consider arbitrary similarity measures where metric space approches like kd-tree cannot be used. The most applicable method of handling the generalized problem is the family of threshold algorithms [8, 9, 10, 2]. These were designed as middleware algorithms that perform index scans over multiple pre-computed index lists, one for each attribute in the query, sorted in the descending order of similarity to the relevant query attribute. Different algorithms from this family vary in how they schedule between random and sequential accesses.

Many attributes of real-world objects come from extremely restricted spaces. As an example, consider the case where a top-k query is to be run over a database of servers within an organization where similarities are computed on server attributes such as hard disk capacity, memory capacity, operating system, network card details, speed of processors etc. A simplistic dataset (running example) of servers being represented by just two attributes, viz., OS and Memory Capacity as given in Table 1. Each of these attributes take values from a very restricted space; hard disks come in sizes which are multiples of gigabytes and there are not more than a score popular operating systems and versions. Due to the very small space of values, these datasets have a lot of objects which assume the same value for any attribute. Such datasets also appear in many other domains such as product databases. The attribute specific index lists which the family of threshold algorithms use, are not optimized to exploit the case where the number of objects per value of an attribute is large. Since many objects have the same value, a change in similarity score occurs infrequently leading to infrequent candidate pruning.

In this paper, we exploit the distribution of objects across various dimensions and similarity between various values of each attribute in building an indexing structure which we

EDBT'08, March 25–30, 2008, Nantes, France.

Id	OS Name	Memory
1	MS Windows (MSW)	512M
2	MS Windows (MSW)	2048M
3	RedHat Linux (RHL)	2048M
4	SuSE Linux (SL)	1024M
5	SuSE Linux (SL)	1024M

Table 1: Sample dataset

use as the backend for top-k query processing. We call this indexing structure as attribute-level tree (AL Tree). Each level in AL Tree represents an attribute, nodes in a level represent various values taken by the attribute and leaves represent the data with appropriate attribute values. The similarities between various values of each attribute are precomputed. At query time, we dynamically and virtually order the values of attributes in AL Tree, to efficiently search the space around the query object and output the nearest neighbors as they are encountered and inferred. The ALTree exploits the fact that many real world attributes come from a small value space, by searching over the value space rather than over objects. We show that this structure enables upto an order of magnitude improvement on response times on dense datasets over the conventional threshold based algorithms. It may be emphasized here that the performance of the proposed algorithm *improves dramatically with the density of the dataset* as opposed to conventional algorithms. Further, the proposed algorithm emits results one at a time in an online fashion. This enables the user to issue a query with high value of k and stop the computation as soon as the desired results are found.

The main contributions of our work are listed below:

- An attribute level indexing structure (AL Tree) that optimizes storage for dense datasets, which can be used as the backend for top-k query processing.
- An algorithm to return all objects within a specified threshold distance of the query object using the AL Tree backend.
- An on-line algorithm to return the top-k objects similar to the query object using the AL Tree backend.
- A detailed experimental evaluation of the algorithm showing its effectiveness over conventional techniques, especially on dense datasets.

The remainder of this paper is organized as follows. We begin with a discussion of related work in Section 2. In Section 3, we describe the problem formally. We describe the AL Tree indexing structure in Section 4 and algorithms for searching top-k in Section 5. We present our experimental results and comparison to CA and TA in Section 7 and finally conclude in Section 8.

2. RELATED WORK

Top-k query processing has been well studied on scenarios where the distance measure satisfies the metric property. A distance measure has to satisfy the traingle inequality property (among other properties such as symmetry and reflexivity) to qualify to be a metric distance measure. In such cases as metric distance measures, large areas of the space can be pruned out while searching for top-k results. Pivot points can be used to partition and thus restrict the search space [13, 20]. However, in practice, there are several dissimilarity measures that are non-metric [18]. These include the k-median distance that measures the k^{th} most similar portion of the compared objects and partial Hausdorff distance (pHD) for shape based image retrieval. Also, real world algorithms for measuring the distance may be complex based on data analysis, learning distances from the distribution, or domain knowledge [16, 14, 11]. AL Tree can be used for arbitrary non-metric similarity measures where the metric based indexing methods such as the kd-tree or R-Tree cannot be applied.

Another popular area where top-k queries are put to use is that of information retrieval, retrieval of multimedia content and web search engines. In case of text data, the number of attributes are as much as the size of the whole vocabulary and the values assumed by each attribute in a data document is the (most commonly) tf-idf value of the same. Multimedia and video retrieval also employ similar representations for data objects. Such a representation leads to a very sparse dataset and the distance measures used are metric. We address a different domain of denser datasets and non-metric similarities.

The other class of algorithms used in top-k query processing, which have been quite popular in recent years, is the class of threshold algorithms (TA) [10]. They maintain precomputed index lists on disk, one for each attribute value, sorted in descending order of attribute dissimilarity and assume that these lists can be accessed by random accesses or sequential accesses. Threshold Algorithms perform scans over these lists and aggregates similarity across various attributes on the fly, thus maintaining a lower bound for the rank-k result candidate and an upper bound for the scores of the rest of the candidates. They stop processing as and when they have prcessed enough to reach a condition that the former is greater than the latter, which ensures that the top-k candidates are indeed the top-k closest objects to the query. Different algorithms in the TA family differ in how they schedule the random accesses and sequential accesses over the index lists. We differ from threshold algorithms in two aspects: rather than assume a index list backend, we build a specialized index that can capture inter attribute dependencies and we return the top-k results in an online fashion. Early members of the TA family made extensive use of random access (RA) to index entries to resolve missing attribute-level dissimilarities of result candidates. But, for very large index lists with millions of entries that span multiple disk tracks, the resulting random access cost is 50 - 50,000 times higher than the cost of a sorted access (SA). To address this problem, it has been proposed not to use random access [10, 12]. This variant of TA is called NRA (No RA). However, this may make NRA to scan longer parts of index lists in order to discover incomplete information. Therefore, Fagin proposed a combined algorithm (CA) [9] that occasionally and carefully performs RAs for promising candidates which can contribute to major pruning of candidates. Scheduling of sequential and random accesses efficiently is critical for TA style algorithms and has been addressed by many. Strategies for scheduling RAs on "ex-

pensive predicates" are presented in [4, 17]. They considered restricted attribute sources, such as non-indexed attributes that do not support sorted access at all. The MPro method [4] computes a list of candidates for the indexed attributes and then schedules additional RAs on the non-indexed attributes. A integrated approach of knapsack related optimization technique for SA and a statistics based cost-model for RA scheduling is proposed in [2]. Many applications including web search need not have the truly best results and can do with approximate best results. It has been shown [19] that using probabilistic estimates of bounds in place of the overly conservative estimates of bounds can lead to performance improvements in TA style algorithms. Recently, there has been some work [15] at designing online versions of TA style algorithms without doing any random accesses. In [22], the authors relax the assumption that the aggregation functions are monotonic and propose an index-merge paradigm that performs progressive search over the space of joint states. In our work, we maintain the assumption that the similarity aggregation function is monotonic. For comparison, we choose CA and TA in our experiments since they are applicable to non-metric spaces and are the most widely studied algorithms.

3. PROBLEM DEFINITION

We will now define the problem formally. Let D be the set of objects in the database. Assume that each object in D have m attributes each. The distance 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 attribute i. The distance between two objects is defined as any monotone function of the distance between the corresponding attributes:

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

where $v_i(O)$ is the value of i^{th} attribute of object O and f is any monotone combining function. Most commonly, such combining 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 $W = [w_1, \ldots, w_m]$, $w_i \geq 0$. If each d_i is bounded in $[0, 1]$ and $\sum_i w_i = 1$, then d is also bounded between $[0, 1]$. Without loss of generality, we will use the weighted sum combining function in the rest of the paper (for simplicity). The notations used in this paper are listed in Table 2.

D	Database of objects
m	Number of attributes
A_i	Domain of attribute i
k	Number of objects to retrieve
O	Instance of an object
$v_i(O)$	i^{th} attribute value of O
Q	Query object
W $[w_1, w_2, \ldots w_m]$	Attribute weight vector
d_i	Distance function for attribute i

Table 2: Notations

In this setting, the top-k query problem is defined as follows:

DEFINITION 3.1. **Top-k Query Problem:** *Given a Q and W, find k objects from D that are nearest to the query object Q where the distance is computed using weights W. This corresponds to finding the set $R \subseteq D$, such that $|R| = k$, and*

$$O \in R \Leftrightarrow \forall O' \in D - R, d(Q, O, W) \leq d(Q, O', W)$$

One of the commonly considered variants of this problem, is that of computing a ranked set of results. Ranking the results based on their distance from the query object would enable the user to scan the set of results in that order. Another variant is that of computing and emitting the result set online, in the increasing order of distance from the query, which would enable the user to use the top results before the entire top-k results have been computed. Both these variants assume significance when k is large. We address the ranked online version of the top-k query problem in this paper.

4. THE ATTRIBUTE LEVEL TREE

In this section, we describe the Attribute Level Tree (AL Tree) data structure, the usage of indirection lists to achieve dynamic sibling ordering and a compaction technique for the AL Tree.

4.1 The AL Tree Data Structure

Consider a database of D objects having m attributes each. Each level in the AL Tree rooted at R is associated with a specific attribute in the dataset, and hence, each AL Tree assumes an ordering of attributes. Thus, each dataset has a unique AL Tree for a given ordering of attributes. Consider an ordering of attributes where the i^{th} attribute is denoted by a_i where $1 \leq i \leq m$. Each internal node of the AL Tree N is characterized by the level of the node $L(N)$, which denotes the number of edges from root to reach N, and a value $V(N)$ which would be one of the values from $A_{L(N)}$. We represent each such internal node by $(V(N))$. Each node N is associated with a set of data objects $S(N)$. Consider a node N, the route to whom from the root is R, N_1, N_2, Ě $N_{L(N)-1}$, $N_{L(N)}$ where $N_{L(N)} = N$. $S(N)$ would then comprise each object d which satisfies

$$\forall_{i=1}^{L(N)} v_i(d) = V(N_i)$$

The j^{th} step away from the root fixes the value of the attribute a_j, and thus the set of objects that a node N represents is the subset of the parent's objects restricted to those which assume the value $V(N)$ for the attribute $a_{L(N)}$. Thus, $S(N)$ would contain each object d from the set of objects in it's parent provided it satisfies the condition $v_{L(N)}(d) = V(N)$. It may be noted that internal nodes (apart from the root which represents the entire set of objects) represent sets of objects which has the same value for at least one attribute, whereas each leaf node represents sets of objects which have the same values for all the attributes, i.e., duplicates. Hence, every leaf node N, is characterized by the level $L(N)$, the value $V(N)$ and the object ids of the objects that the node represents, $S(N)$ and can be represented by $(V(N), S(N))$. Optionally, the leaf nodes may hold pointers to the location of the objects that it represents. The $S(N)$ attribute would always be a singleton list in a database which doesn't contain duplicates. It may be noted that the database can be fully re-created from the ALTree.

EXAMPLE 4.1. *Consider the dataset in Table 1. There are two attributes in the dataset, Operating System and Memory Capacity. Shown alongside the dataset is the AL Tree for the same assuming the ordering (OS, Memory) for attributes. The top-level nodes, A, B & C stand for the three values of the OS attribute, MSW, RHL and SL respectively. Among data objects that assume the value MSW for the OS attribute, there are two possible values, 512M and 2048M for the Memory attribute. Thus, the node A has two children, one for each of those values. The leaf node G stands for two objects which assume value 1024M for the Memory attribute (as that is the value associated with G) and the value SL for the OS attribute (as that is the value associated with G's parent, C).*

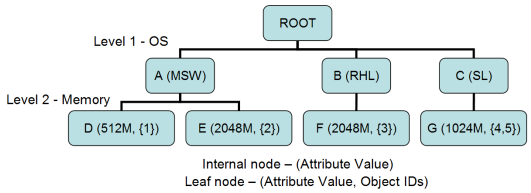

Figure 1: A sample AL Tree and dataset.

4.2 Indirection Lists and Dynamic Sibling Ordering

The AL Tree assumes no ordering among sibling nodes. However, efficient top-k retrieval from the AL Tree requires a controlled and directed traversal of the tree according to the query. We accomplish query time sibling ordering in the AL Tree using pre-computed indirection lists, which we describe in this section. Let the values that attribute a_j can assume be denoted by be $A_j = \{A_{j_1}, A_{j_2}, \ldots A_{j_{c_j}}\}$. For every value v of attribute a_j, we maintain a list L, of values from A_j in the non-decreasing order of distance (non-increasing order of similarity) from v, i.e. the following holds,

$$d_j(L[p], v) \leq d_j(L[q], v), \forall p < q$$

For every attribute a_j, there would be c_j such lists (one list per value from A_j), each of length c_j. The total size of the collection of lists would hence be $\Sigma_j({c_j}^2)$ values. *However, as these lists are held on disk and each query needs to get only as many lists as the number of attributes (the ordering for children of sibling nodes would be the same for the same query as the similarity lists are per attribute-value entities), this approach is scalable.* These similarity lists specify the ordering for siblings of internal nodes.

Each internal node N, in the AL Tree would have a value based lookup function for the children nodes which can be defined as follows:

$$Child_N(v) = \begin{cases} C, \text{if } \exists C, \ni: (PARENT(C) = N) \\ \qquad\qquad \wedge (V(C) = v) \\ null, otherwise \end{cases}$$

Let the number of attributes be n and the query be $Q = (v_1, v_2, \ldots v_m)$ such that $v_j \in A_j$. Let the pre-computed list for v_j be denoted by L_j. We implement a lookup function for every internal node N, to lookup the i^{th} child of N (according to the order specified by the appropriate indirection list), denoted by $Child_N(i, v)$ as follows

1. $count = 0$
2. $for(j = 0; j < L.size(); j++)$
   (a) $if(Child_N(L[j])! = \phi) count++$
   (b) $if(count = j)$ return $Child_N(L[j])$
3. return ϕ

This returns the i^{th} child of N, when the children of N are arranged in the same order as they are in the list corresponding to the value specified in the ssecond parameter.

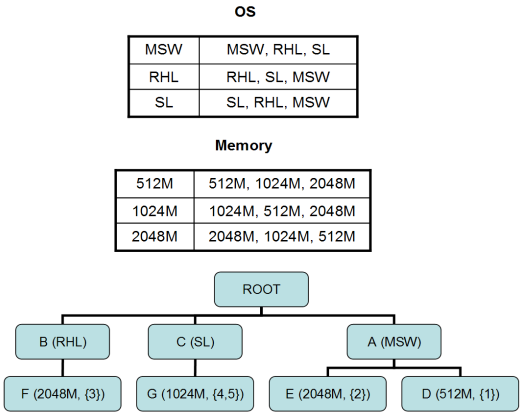

Figure 2: Similarity Lists & a Re-Ordered AL Tree.

EXAMPLE 4.2. *Consider the dataset in Table 1, and the query $(RHL, 2048M)$. The list for the value RHL (i.e., values of the same attribute in the decreasing order of similarity) would thus be $< RHL, SL, MSW >$ (the second row in the OS Similarity Matrix) and that for the value $2048M$ would be $< 2048M, 1024M, 512M >$. The re-ordered AL Tree for the query $(RHL, 2048M)$ is shown in Figure 1. Consider the scenario in the search where we have to find the 2^{nd} child of A i.e., the invocation of $Child_A(2, 2048M)$, the second parameter being the value of the second attribute from the query. The function progresses through the list $< 2048M, 1024M, 512M >$, firing value based lookup queries for each of the values in the list until it finds the 2^{nd} non-null child and returns it. The sequence of queries would be $Child_A(2048M)$, $Child_A(1024M)$ and $Child_A(512M)$. The query $Child_A(2, 2048M)$ stops traversing the list after making the call $Child_A(512M)$ because it finds the 2^{nd} non-null child then. Similarly, $Child_A(1, 2048M)$ stops after the call to $Child_A(2048M)$ whereas $Child_A(3, 2048M)$ returns null because it can't find a 3^{rd} child even after exhausting the list.*

The search algorithm that we present in a subsequent section invokes the function to get the i^{th} child only in sequence for a given query, i.e., the call to $Child_A(i, a)$ will not succeed the call to $Child_A(j, b)$ if $j > i$ whatever be the values of the A, a and b. This enables a straightforward optimization in the

$Child_N(\ldots,\ldots)$ function. This involves the function maintaining static counters to track the progress through each node (through its appropriate list), so that for a given invocation, the traversal through the list can start from where it stopped for the last invocation for the same node rather than from the beginning. *Note that the tree is not actually reordered per query. Rather the child nodes are accessed as needed in the query specific order.*

4.3 Compressed AL Tree

The AL Tree as described in Section 4.1 may contain internal nodes which have just one child and so do their descendants until the leaf node, i.e., certain internal nodes may be the head of a chain of nodes. Consider the attribute ordering $< a_1, a_2, \ldots a_m >$ for the AL Tree and a case where given the assignment of the sequence of values $< a_1 = k_1, a_2 = k_2, \ldots a_s = k_s >$ for the first s attributes, there can only be one possible assignment $< a_{s+1} = k_{s+1}, a_{s+2} = k_{s+2}, \ldots a_m = k_m >$ for the remaining $m - s$ attributes to reach an object in the dataset. Note that this is different from the functional dependency $a_1 a_2 \ldots a_s \rightarrow a_{s+1} \ldots a_m$ in that not every assignment of values for the first s attributes (but a specific assignment) determines the values of the remaining attributes. We denote such a dependency as $(a_1 a_2 \ldots a_s)_{(k_1 k_2 \ldots k_s)} \rightarrow a_{s+1} \ldots a_m$. Each such dependency for the least value of s leads to a chain of length $m-s$ in the AL Tree. Consider such a chain of nodes $N_1, N_2, \ldots N_{m-s}$. The search algorithm that we present in a later section works by traversing the internal nodes to reach leaves. Thus, although there is a single leaf node under N_1, visiting it would require $m - s$ steps (to visit each node in the chain) for the search algorithm.

A compressed AL Tree can be formed by collapsing each such chain into the head of the chain. With such a representation, the search procedure would be able to reach the leaf node in just one step as opposed to $(m - s - 1)$ steps in the earlier case. Such a representation would imply that each such node N_1 at the head of a chain of length $m - s$ would have to maintain additional information regarding the other attribute values of the remaining $m - s$ attributes as well as the object ids of the objects at the leaf of the chain. We represent each such compressed node N as $(L(N), < V(N), \ldots >, S(N))$ where $L(N)$ denotes the level of the node in the tree, $< V(N), \ldots >$ denotes the sequence of attribute values in the chain from head of the chain to leaf, and $S(N)$ denotes the set of object ids taken from the leaf node of the chain.

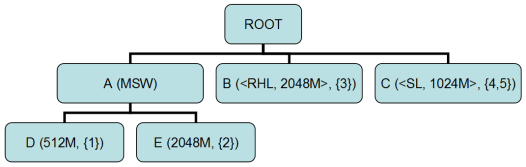

Figure 3: A Compressed AL Tree.

EXAMPLE 4.3. *Consider the dataset in Table 1, and the corresponding AL Tree. The dependencies* $a_{1_{RHL}} \rightarrow a_2$ *and* $a_{1_{SL}} \rightarrow a_2$ *exist in this dataset, leading to the chains* $< B, F >$ *and* $< C, G >$ *respectively. We collapse the chain* $< B, F >$ *to the single node at* B *represented by* $(< RHL, 2048M >, \{3\})$ *because* B *represents the collapsed chain containing itself representing value* RHL *and the leaf node* F *which stands for the value* $2048M$ *and stands for the single object* 4. *The compressed tree with both the chains compressed would be as in Figure 3.*

5. SEARCHING THE AL TREE FOR TOP-K NEAREST NEIGHBORS

The search through the AL Tree employs controlled expansion of the tree using the information from the ordering of the siblings in the tree. Consider a query $Q = (v_1, v_2, \ldots v_m)$ such that $v_j \in A_j$ associated with weights $W = (w_1, w_2, \ldots w_m)$, wherein the distance between Q and an object $O = \{o_1, o_2, \ldots o_m\}$ would be computed as $d(Q, O, W)$. We track the progress of the expansion (search) procedure by maintaining information in the form of 4-tuples, which are referred to hereafter as *candidates*. A candidate points to a node in the AL Tree, and maintains information about which child of the node it would lead the search to, and distance information for the current node. We define a candidate $C = < N, M, d, d_{min} >$ to point to a node N, M being the leftmost child of N yet to be seen in the search process. d refers to the distance of the Query object to the candidate C based on the attributes seen so far (at each level from the root to N) whereas d_{min} is the lower bound on the distance of any child of C from the query object, based on the AL Tree nodes seen so far. We defer further explanation of the functionality of d_{min} to a subsequent paragraph. Let $N_0 = R$, N_1, N_2, ..., $N_{L(N)} = N$ be the route to N from the root R. The distance based on the attributes seen so far is calculated as the aggregated distances of the value at each node in the route to the appropriate value of the query object, i.e.,

$$d = \sum_{i=1}^{L(N)} (d_i(V(N_i), v_i) * w_i)$$

The search procedure works by expanding candidates. The expansion of a candidate leads to visiting a new node, not seen so far, or to the emission of some objects to the result stream. The expansion of C would lead to emission of the $|S(N)|$ objects into the top-k output if N is a leaf node. The expansion of C would yield the following candidates if N is an internal node

1. $< M, P, e, e >$ where $e = d + d_{L(M)}(V(M), v_{L(M)}) * w_{L(M)}$ and P is the leftmost child of M AND
2. Either of
   (a) $< N, M', d, d + d_{L(M)}(V(M), v_{L(M)}) * w_{L(M)} >$ where M' is the right adjacent sibling of N OR
   (b) *null*, if M is the rightmost child of N

We refer to the first candidate of the expansion as the child of C, and the second candidate (if non-null) as the variant of C. The child is the expansion of C to M, the child of N as directed by C. The variant is a modification of C and points to N itself, but differs from C in that it directs the expansion of itself to the next child of N. There is yet another difference between C and the variant, and that is

1. $S = \{< Root, Child_{Root}(v_1, 1), 0.0, 0.0 >\}$
2. while(S != ϕ AND |results emitted| $< k$)
   (a) Pick P, the candidate in S which has the least value for the d_{min} and expand P
   (b) If P is a leaf node, emit the P_C objects that it stands for
   (c) Else $S = (S$ - $\{P\}) \cup \{$non-null expansions of $P\}$

Figure 4: The Search Algorithm.

in the d_{min}. The d_{min} of the variant, as can be observed, assumes the same value as the d of the child. This is because; any child resulting out of the expansion of the variant of C cannot be at a distance closer than the actual distance of the child of C generated out of the same expansion. Thus, *the actual distance of the child generated out of the expansion of any node is a lower bound for the distance of any child generated out of the variant from the same expansion.* This unique property exists because children are arranged in the non-decreasing order of distances from the query. Further, the child of C has its d and d_{min} assuming identical values because no expansion of M has taken place yet, and thus, the lower bound for the distance of any child of the generated child is the same as its actual distance so far. Thus, the d_{min} of any candidate holds the lower bound of distance of any child of the candidate as estimated by the nodes of the tree seen so far. This lower bound if used by the search procedure outlined in a subsequent paragraph to pick the next candidate to be expanded.

EXAMPLE 5.1. *Consider the dataset in Table 1, and the query* $(MSW, 1024M)$ *with the weights* $(1.0, 1.0)$. *The ordered tree corresponding to this query would be as in Figure 1. Assume that each unit of difference causes a distance of one unit. Let* $< A, D, 0.0, 0.0 >$ *be a candidate which is picked for expansion in the search process. Both* d *and* d_{min} *are* 0.0 *because there the actual distance so far is* 0.0 *(as the query as well as the node A, assumes the value of MSW for the first parameter and because no children of A have been expanded so far). The expansion of* $< A, D, 0.0, 0.0 >$ *leads to the child* $< D, \varphi, 1.0, 1.0 >$ *and the variant* $< A, E, 0.0, 1.0 >$. *The child has* d *set to* 1.0, *because the* 2^{nd} *attribute's value in the query i.e.,* $1024M$, *has a distance of* 1.0 *with the attribute value for C, i.e.,* $512M$. *The fact that E occurs to the right of D implicitly means that the distance between the attribute value for E and the query's second attribute is at least as much as* 1.0. *Hence, the variant has a value of* 1.0 *for* d_{min}, *the lower bound of distance for any child of the variant.*

The search procedure maintains a list of candidates, and works by expanding the one in the list with least value for the d_{min}, until at least k nearest neighbors are emitted. It may be noted that it guarantees that the nearest neighbors are emitted in the non-decreasing order of distance from the query. The procedure is outlined in Figure 4 for the Query $Q = (v_1, v_2, \ldots v_m)$.

Upon completion of the algorithm, either k candidates would have been emitted or S would have been exhausted. The latter condition occurs if the database itself contains less than k objects.

EXAMPLE 5.2. *Consider the dataset in Table 1, and the query (MSW, 1024M) with the weights (1.0, 1.0). We illustrate how S changes in the course of the search for k=1. S starts as* $\{ < Root, A, 0.0, 0.0 > \}$. *Upon expansion of the sole element, it becomes* $\{ <A, D, 0.0, 0.0 >, < Root, B, 0.0, 0.0 > \}$. *As both the candidates have the same value for* d_{min}, *we choose the element pointing to A to expand, arbitrarily breaking the tie. As S changes to* $\{ < D, \varphi, 1.0, 1.0 >, < A, E, 0.0, 1.0 >, < Root, B, 0.0, 0.0 > \}$, *we pick the candidate pointing to Root to expand, leading to the new set* $\{ < D, \varphi, 1.0, 1.0 >, < A, E, 0.0, 1.0 >, < B, F, 1.0, 1.0 >, < Root, C, 0.0, 1.0 > \}$. *Among the candidates having 1.0 as the* d_{min}, *we choose the one pointing to D for expansion, which results in the emission of the element with object id 1, thus completing the search process.*

THEOREM 5.1. *If* $d(O_1, Q, W) < d(O_2, Q, W)$ *then* O_1 *is emitted before* O_2.

PROOF. Each candidate $C =< N, M, d, d_{min} >$ represents a subset of the leaves of the tree. To be precise, it represents all leaves that are reachable from N by passing through the child M or through other children to the right of M. Let $L(C)$ represnt the set of leaves represented by C. We have the following observations based on the way the candidates are generated.

1. The subsets represented by each candidate are disjoint
2. Each leaf is covered by at least one candidate
3. $d_{min} \leq d(l, Q, W)$ for all $l \in L(C)$

The first two obeservations hold since we start with the candidate $< R, Child_R(v_1, 1), 0, 0 >$ which represents all the leaves of the tree. Also, at each step, the candidate $C =< N, M, d, d_{min} >$ chosen is replaced by two candidates that partition $L(C)$ – one subset consists of leaves reachable through M and the other consists of leaves reachable through children to the right of M. The third observation follows from the way d_{min} is calculated and the fact that children of each node are explored in the increasing order of distance from the query.

The proof is by contradiction. Let O_2 be emitted before O_1. From the previous observations, at the time O_2 is emitted, there is some candidate, say $C_1 =< N, M, d, d_{min} >$ that covers the leaf O_1. The candidate corresponding to O_2 is $C_2 =< O_2, null, d(O_2, Q, W), d(O_2, Q, W) >$. Since C_2 was chosen over C_1 as a result, it implies that $d(O_2, Q, W) \leq d_{min}$. Since $d_{min} \leq d(O_1, Q, W)$, we get $d(O_2, Q, W) \leq d(O_1, Q, W)$ which is a contradiction. □

Theorem 5.1 implies that results are emitted in the increasing order of distance, thus making this an online algorithm.

Also, the first k results represent the top-k similar objects. Note that the top-k result set may not be unique since there can be multiple results with identical similarity scores. This algorithm is guaranteed to return k objects such that there is no other object closer to the query than the result objects.

Choosing the next candidate to expand: Step 2(a) in the algorithm presented in Figure 4 may be complicated in cases where there are multiple candidates with the minimal value for the d_{min}. *Although the result would be identical regardless of the choice*, certain heuristics would help in arriving at the results by visiting fewer nodes and within fewer iterations (on the average). We list a couple of such heuristics with justifications as below:

1. Among two candidates which point to nodes in different levels in the tree, choose the candidate pointing to the deeper node. This is because the candidate at the deeper node has more chances of containing objects closer to the query as it's distance estimate has already factored in more attributes.
2. Among those candidates that point to nodes in the same level, choose the candidate that has the second element pointing to the leftmost child of the first element in preference to others. This is because d_{min} assumes the value of the real distance only for those candidates that satisfy the above condition. For all other candidates, d_{min} represents a lower bound of the distance.

5.1 Using AL Tree for Range Queries

In sparse datasets, the top-k nearest neighbors may be too far off from the query point to make any sense. In such cases, as well as in a variety of other cases, range queries which expect all the data objects within a specific distance r from the query point are useful. Such queries can be handled by a simple adaptation of the algorithm presented in Figure 4 where we keep emitting nearest neighbors until they are within r distance of the query point, at the same time maintaining only such candidates in the candidate set which have their d_{min} lesser than r. In this section, we present an algorithm which uses the AL Tree to answer range queries. A recursive function $R(N, Q, r)$ which returns all objects within r distance of the query object Q using a weight vector W, considering only attributes below a specific node N in the tree can be defined as below.

1. if $r < 0.0$ return $\emptyset$
2. if N is a leaf, return $S(N)$
   (a) $RS = \emptyset$
   (b) for each $M \in C(N)$
   - $r' = w_{L(M)} * d(v_i(Q), V(M))$
   - $RS = RS \cup R(M, Q, r - r')$
   (c) return RS

The function works by recursively calling the function on its child nodes with smaller values for r computed by factoring in the distance increment (r') incurred by moving to the child. The range query to find objects within a distance of r from Q can then be computed as $R(Root, Q, W, r)$. It may be noted that this algorithm is significantly IO-friendly as compared to the top-k search algorithm as it does not revisit any node in the AL Tree, but does not emit the results online.

6. DISK-BASED IMPLEMENTATION

A disk based implementation of an AL Tree is necessary for large datasets. In this section, we describe our approach of packing the AL Tree on to disk and address the issue of updating such a disk based implementation when the dataset changes over time.

6.1 Subtree Packing

The search through the AL Tree proceeds by picking candidates and expanding them, which involves accessing the child nodes of the node (pointed to, by the candidate). To optimize disk accesses in such a setting, we use a variant of breadth first packing. We start with the root and do breadth first packing until the page is full. The frontier nodes (nodes which have been packed in the current page, and have at least one of their children left to be packed) are collected, and the breadth first packing is done on sub-trees rooted at those nodes and the resulting frontier nodes are collected. This process is continued until all the nodes in the tree are packed. This packing is optimized for top-k search on the AL Tree as it tries to pack entire subtrees in a page, this clustering nodes that are processed together in a page.

6.2 Updating the AL Tree on Disk

Addition of a new tuple to the AL Tree at most affects or creates one node in the tree due to the compressed nature of the AL Tree (Section 4.3). If a new node needs to be created, it can be created on a new page on the disk without affecting the correctness of the algorithm. If the tuple contains a new attribute value (one which is not already present in the similarity lists), the similarity lists for that attribute needs to be re-computed. If a lot of new nodes get added and existing nodes get deleted, the tree becomes very dispersed on the disk and the disk IO performance of the algorithm deteriorates (although the correctness is not affected). In such cases, a periodic re-run of the Subtree packing procedure can be done to restore the disk performance optimized packing structure for the tree. AL Tree was designed for an application where updates are infrequent, where periodically repacking the tree is not an issue. Deletion of a node is a straightforward operation which involves deletion of the link to that node, freeing up the space occupied by that node in the page.

7. EXPERIMENTS

In this section we describe our experimental results. We performed a detailed study of the AL Tree based algorithm and compared it to the Combined Algorithm (CA) and Threshold Algorithm (TA). We chose CA and TA for comparison since they are applicable to non-metric spaces.

7.1 CA and TA Adaptation

We make two modifications to the threshold style algorithms that we compare against on the lines of the block based optimizations used in [2].

1. In the TA and CA algorithms, we access lists in blocks. This has implications on the access cost computations which we explain in Section 7.2.

2. We update the best scores for candidates in CA once per block of list items. The book-keeping in CA is prohibitively expensive since it needs to update the best possible score for each candidate in each iteration. Updating once per block of list items leads to huge savings in computational costs.

7.2 Experimental Setup

We compare our AL Tree based algorithm against TA, CA and the classical full merge algorithm (followed by partial sort). Our comparisons are based on both computational costs and IO costs. IO costs are measured in terms of either sequential or random page IOs, of which the latter is significantly costiler than the former. Studies on CA have assumed the ratio of cost of random item access, c_r to the cost of sequential item access, c_s to be between 1000 and 100000. IO accesses are typically done at the page level; if a page can hold t list items, the ratio of costs between random page accesses, p_r and sequential page accesses, p_s would be $(c_r/c_s)/t$. Our implementation uses a t of 1000 and we set the cost ratio (p_r/p_s) to 10 for our experiments, i.e., $(c_r/c_s) = 10000$. Algorithms that access data sequentially, such as CA and full merge, need just one page per attribute for scanning the lists. On the other hand, AL Tree can make use of extra memory to cache pages that are read once as nodes (and hence pages) may be revisited. Unless otherwise mentioned, we use a LRU cache of the size of 7.5% of the dataset for our experiments. A sequential item access for TA and CA implies stepping through one item on each of the lists. Thus, it is a particularly advantageous scenario for TA and CA if there are as many disks (or diskheads) as there are attributes, so that m pages (one per attribute) are accessible by sequential IO at any given configuration. On the other hand, AL Tree is a single tree structured entitity and does not have any straightforward partitioning to leverage the presence of multiple disks. *For our experiments, we use multiple disks (one per attribute) for the CA and TA implementation whereas we use a single disk for storing the AL Tree. It may be noted that this setting is very favorable to CA and TA as the next page for any list is accessible by sequential IO. However, it is somewhat unrealistic since this setting requires a number of disks equal to the number of attributes, which makes it dataset dependent.* The random access for CA and TA for a particular object involves filling in the values for the unseen attributes for that object. This would typically need as many random accesses as there are unseen attributes. [2] describe an approach where enough information can be stored in memory to get all the unseen attributes using one random access. We use such an implementation for CA for our experiments. For the disk based implementation of the AL Tree based algorithm, we do subtree packing(Section 6.1) of the AL Tree. We performed our experiments on an IBM X Series running Windows Server 2003 on an Intel Pentium Four 3.4 Ghz Processor with 2.0 GB of RAM.

As we are interested in analysing the AL Tree based algorithm in a very general setting, we use synthetic datasets upfront to illustrate the behavior by varying parameters such as k and *data density*. Data density is computed as the ratio of the number of data objects to the total possible number of distinct tuples in the space. We generate synthetic datasets with uniform random distribution and random distances so that each value of each attribute has a uniform representation in the dataset. Further, we use Gaussian distributions with random similarity measures to gain insights as to how to order the attributes during the creation of the AL Tree. Lastly, we analyze our algorithms against the classical threshold algorithms on real datasets from the UCI Machine Learning Repository [6] and validate the observations from the experiments on synthetic datasets.

7.3 Computational Costs

To isolate the computational costs from IO costs, we use a setting where all the objects and indexes are loaded in memory. So there is no IO cost involved and the cost is purely computational. We measured the execution times, the number of iterations and the maximum candidate set size. For CA and TA, the number of iterations is a measure of the number of items processed from each list. For AL Tree it refers to the number of executions of the loop (picking a candidate and expanding it). For all algorithms, the maximum candidate set size refers to the maximum number of candidates held in memory during the run. It is a measure of the memory overhead of these algorithms and also indicative of the execution time in the case of CA and AL Tree. For all experiments, the number of attributes is 5 with 25 distinct values per attribute unless otherwise mentioned. For CA, the c_r/c_s was 10000 which correponds to p_r/p_s of 10. The numbers reported are an average over 100 random queries.

7.3.1 Varying density

In this subsection, we analyse the execution times for AL Tree, CA and TA for varying densities while keeping the dataset size constant. Density is varied by varying the size of the value space (i.e., the number of attributes and the number of values per attribute). We vary the number of values per attribute from 62 to 18 fixing k and the dataset size at 5 and 1 million respectively. Figure 5 plots the execution time against the varying density. Note that AL Tree and TA are plotted on the first Y axis, while CA is plotted on the second Y axis since there is a order of magnitude difference in the times. TA and CA remain more or less stable because they do object based indexing and the number of objects is held constant throughout. *Both CA and TA are not able to exploit the increased density as they do not do value space indexing, whereas AL Tree registers a sharp improvement in performance as the density increases*, especially during the initial period when the density increases from very low values to a substantial figure. To analyze the execution times, we plot the number of iterations in Figure 6. This graph follows the same trend as the execution times. CA and TA do many more iterations since they have to process a significant percentage of the lists. AL Tree uses the index to explore the neighbourhood of the query, so can find the results in lesser number of iterations. The number

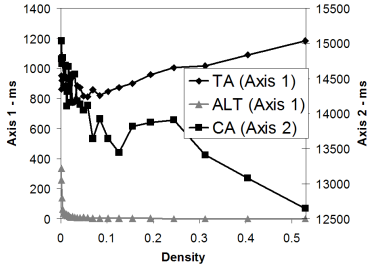

Figure 5: Execution Time vs Density (Section 7.3.1)

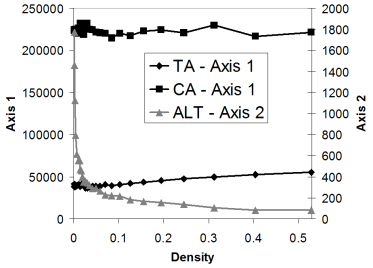

Figure 6: Number of Iterations vs Density

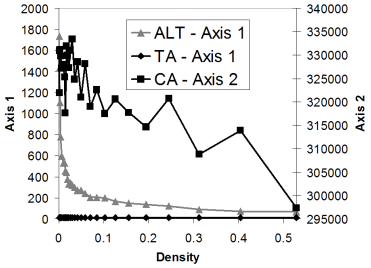

Figure 7: Maximum candidate set size vs Density

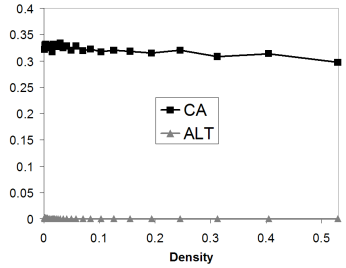

Figure 8: Candidate set size ratio vs Density

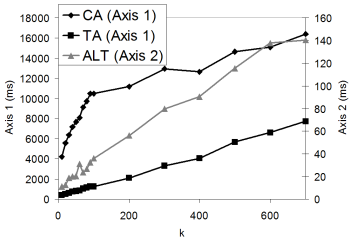

Figure 9: Execution Time for varying k

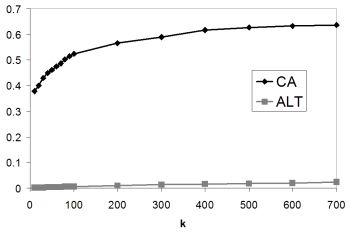

Figure 10: Candidate set size ratio vs k

of iterations for AL Tree reduces with density since the top-k are in a smaller neighbourhood.

Figure 7 shows the maximum candidate set sizes. TA uses a fixed candidate set size of the current top k objects at any time. For CA, the maximum candidate set size keeps decreasing with density because it needs to scan lesser down the lists to reach the results. The maximum candidate set size of AL Tree reduces sharply with density in the earlier stages where there is upto an ten-fold decrease. Overall, CA has a order of magnitude higher memory overhead than AL Tree, whereas TA has the least memory overhead. Figure 8 plots the ratio of the maximum candidate set size to the number of objects for CA and AL Tree. The interesting thing to note is that as data density decreases, the ratios for both increase. As the data density approaches 0, the ratios for both will be 1. This indicates that at very low densities, the nearest neighbour could be spread out over the entire space, so indexing doesn't really help. In fact, previous studies [3] have questioned the meaningfulness of the nearest neighbour query in high dimensional spaces that are sparse, since the distance to the nearest neighbour approaches the distance of the farthest neighbour for these datasets. Also, a simple linear scan is often better for nearest neighbour search in very sparse high dimensional spaces.

7.3.2 *Varying k*

In these experiments, the data set size was fixed at 300000 with a density of 0.03 and k was varied from 5 to 500. Figure 9 plots the execution times as k is increased. As expected, the execution times of all the algorithms increases with k. As before, AL Tree performs much better than CA and TA. The number of iterations follows a similar trend and is not shown here. Figure 10 plots the ratio of the maximum candidate set size to the number of objects for CA and AL Tree. It can be seen as k increases, the ratio increases, indicating that at higher values of k, we explore a larger percentage of the entire space leading to larger execution times. With an online algorithm, such as the AL Tree based one, the user need not be concerned with using a large value of k since they can stop the query as soon as they get satisfactory results.

7.4 IO Costs

While dealing with huge datasets, objects and indexes have to be stored on disk, thus making the IO cost an important quantity to measure and optimize. In this subsection, we describe a set of experiments to compare the IO cost of the AL Tree based algorithm with CA and TA, and with the classical full merge algorithm. We report IO accesses in terms of page IO accesses, assuming, unless otherwise mentioned, a (p_r/p_s) ratio of 10 (which corresponds to a c_r/c_s ratio of 10000). Although we consider a ratio of 10 consistently, we compared the behavior of AL Tree and CA on varying ratios wherein we found that although smaller ratios are advantageous for AL Tree (as it does more random accesses), AL Tree outperforms CA on IO costs upto a page access ratio of around 250 (i.e., for c_r/c_s upto 250000). Thus, the weighted page IO cost would be computed as $s + (p_r/p_s) * r$ where s and r stand for the number of sequential and random page accesses respectively. The numbers reported are an average over 100 random queries.

7.4.1 *Varying density*

In this subsection, we analyse the IO costs for AL Tree, CA and TA for varying densities while keeping the dataset size constant. Density is varied by varying the size of the

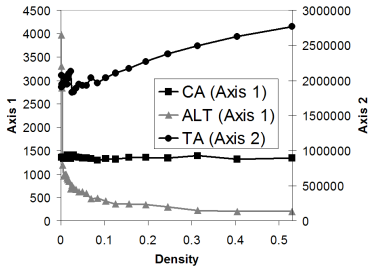

Figure 11: Page IO Cost vs Density (Section 7.4.1)

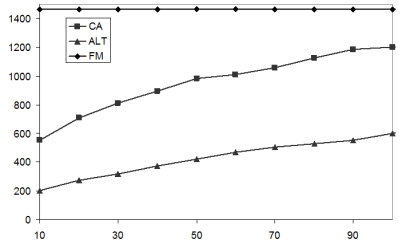

Figure 12: Page IO Access Cost vs k

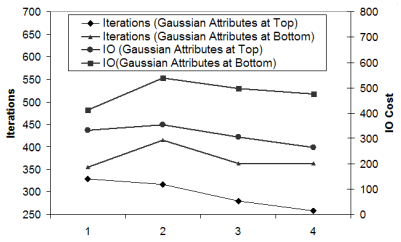

Figure 13: IO Cost/Iterations vs No of Skewed Attributes Used

value space (i.e., the number of attributes and the number of values per attribute). We vary the number of values per attribute from 62 to 18 fixing k and the dataset size at 5 and 1 million respectively. As can be seen from Figure 11, the performance of CA remains quite stable (as it uses object-based indexes) whereas that of AL Tree improves with density (as it used value-based index). AL Tree outperforms CA by a good margin and it outperforms TA (IO Cost plotted on a different axis) quite significantly. As seen earlier, for low density datasets, the number of candidates explored increases sharply which in turn leads to traversing more nodes in the AL Tree resulting in more random IOs. Also, since the search algorithm may visit a node multiple times employing a best first strategy, the working set size tends to increase with the number of nodes visited. This leads to trashing increasing the IO cost significantly since most of these IOs are random IOs.

7.4.2 Varying k

In these experiments, the dataset size was fixed at 300000 with a density of 0.55 and k was varied from 5 to 100. Figure 12 plots the weighted IO cost as k is increased for the CA, AL Tree and Full Merge algorithms. The Full Merge IO cost remains constant as it has to access all the lists (by sequential access). The IO costs for CA increase much faster than that of the AL Tree and the former is at least twice as much as the latter for any value of k that we experimented with.

7.5 Disk Based Implementation Performance

The performance of a disk based implementation of any top-k query mechanism depends on both the computational expense and the IO access cost. Thus, the metric that is considered of high significance is the total *response time* for a disk based implementation. Having analysed the computational costs and IO costs separately in the last sections, we analyse the response time of AL Tree against the classical threshold algorithms in this section. We simulate the disk-based implementation where we assume page access costs to be 1ms and 10ms for sequential and random access respectively using a disk page size of 8192 bytes. It may be noted that these estimates are in tune with disk performance figures reported in literature [5] [1].

We do response time analysis for varying density using the same configuration as in Sections 7.4.1 and 7.3.1 and by varying k using the configuration in Section 7.4.2 and 7.3.2. As can be seen in Figure 15 and Figure 16 respectively, AL Tree performs significantly better as compared to the classical algorithms in both the cases.

7.6 AL Tree Specific Experiments

In this section, we describe a set of experiments to analyse the AL Tree based algorithm with respect to ordering of attributes and caching.

As the order of attributes is fixed at AL Tree creation time, it should be good enough to cover a wide variety of cases. So far, we have been dealing with uniform distributions over different values of each attribute. But, real datasets are often skewed, and attributes differ in the amount of skew. We analyzed the AL Tree by using Gaussian (skewed) distributions for a subset of attributes. We analyze the trend by varying the number of Gaussian attributes from 1 to 4 in a dataset of 7 attributes, and considered the case where they are placed together at the top of the tree and the case where they are placed together at the bottom of the tree. As can be seen in Figure 13, placing the skewed attributes at the top of the tree is a favorable case for the AL Tree based search procedure. As Gaussian distributions are very skewed, the number of objects that assume different values per attribute vary widely. These results may be generalized to conjecture that lesser the number of values per a given attribute (being an extreme case of attribute skew), higher up should it be in the AL Tree as attributes with very few values are an extreme case of attributes with highly skewed distribution. Since the tree is processed from top to bottom, the number of iterations is sensitive to the number of nodes at the higher levels in the tree. By having attributes with lesser number of values at the top, the number of nodes at higher levels is reduced.

AL Tree is a tree based data structure which revisits nodes (and hence, pages) due to the best first search strategy employed. This makes it quite sensitive to cache sizes. We studied the behavior of the AL Tree based algorithm for varying cache sizes. Figure 14 plots the weighted IO cost against the cache size represented as a fraction of the total dataset size. This experiment was performed on a 3% dense dataset having 5 attributes. The IO performance improves drastically for small increments in cache sizes when the cache is small. This is expected because of thrashing when working with small caches because the working set is not in the cache. As more of the working set is in the cache, the performance improvements decrease for the same cache size increments. For this specific dataset, the working set

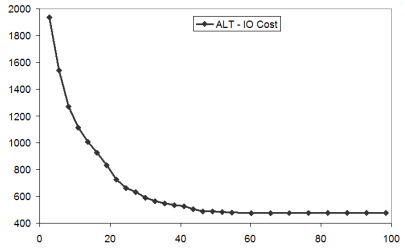

Figure 14: IO Cost vs Cache-Dataset Ratio

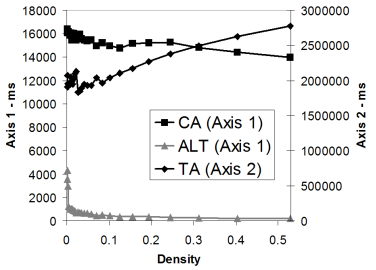

Figure 15: Response Time vs Density

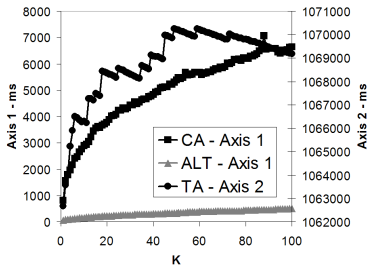

Figure 16: Response Time vs k

can be inferred to be close to 25% of the dataset size.

7.7 Performance Analysis on Real Datasets

We analyzed the performance of the AL Tree against CA on real world datasets. Real-world datasets are usually very skewed (non-random) and thus may be significantly different from the synthetic random datasets, on which we reported results in the previous sections. We use two real-world datasets from the UCI Machine Learning Repository[6] for our experiments in this section.

7.7.1 Census Income Dataset

The Census Income Dataset contains details census data about 199523 people for 1970, 1980 and 1990 from the Los Angeles area [1]. We chose a subset of attributes from the dataset, based on their utility in measuring similaritites between objects. The attributes chosen for the Census dataset were *Age, Education, Number of Minor Family Members, Number of Weeks Worked and Number of Employees* which assume 91, 17, 5, 53, and 7 distinct values respectively in the dataset. The density of this dataset on the selected attributes is 0.07. Taking cue from the observations from Section 7.6, we ordered the attributes based on the increasing number of distinct values per attribute in the AL Tree from top to bottom. As can be seen from Figure 17 (which has the Y axis plotted on log-scale), the response time for AL Tree is much better than that of CA for the Census dataset over varying values of k by orders of magnitude. Further, the graph also shows the IO-cost constituent (in terms of time) for both the methods. It may be noted that AL Tree spends a considerable amount of its time in performing IO, mainly because of node revisits.

7.7.2 ForestCover Dataset

The ForestCover dataset contains data on the Forest Cover type for 581012 30 X 30 meter cells over regions in the US [2]. The attributes chosen from the ForestCover dataset had 67, 551, 2, 700, 2, 7 and 2 distinct values (The ForestCover dataset has 44 binary attributes among the 55 attributes present). This dataset is markedly different from the Census Income Dataset in that the data is very sparse with a density of 0.0004. We compare the performance of the AL Tree based algorithm and CA for varying values of Cache Sizes. We performed two sets of experiments by varying k, one with the cache size set to 7.5% of the dataset size, whereas

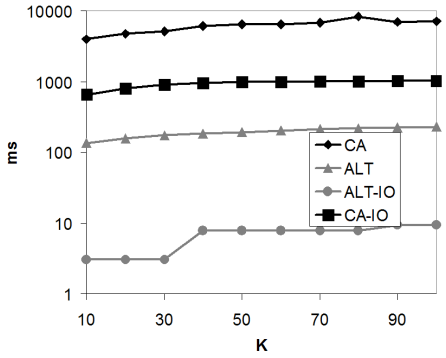

Figure 17: Response Time vs k (Census Data)

the other set was run setting cache size to 0.05% of the dataset size. As CA does not revisit entries in the disk, it is unable to take advantage of the increased cache size. On the other hand, our algorithm is able to effectively utilize the cache as it revisits parts of the tree very often. As can be seen in Figure 18, the performance of AL Tree deteriorates quite drastically when the cache size is reduced to a very small cache fraction of the dataset (0.05% in this case). Yet, it may be noted that even with a very small cache size, AL Tree outperforms CA on very sparse datasets such as ForestCover.

8. CONCLUSIONS AND FUTURE WORK

The top-k problem is an important problem in many domains. Non-metric distance functions are essential to effectively capture the notion of similarity in many cases. Most of the existing multi-dimensional indexing methods are applicable only to metric spaces. To this end, we have developed the AL Tree structure that takes advantage of there being few distinct values per attribute. The top-k algorithm based on the AL Tree efficiently explores the value space to find the top results in an online fashion. We did a detailed comparison of AL Tree with TA and CA. Threshold algorithms store per attribute similarity lists whereas AL Tree captures the inter attribute dependencies as well, leading to a better pruning of the search space. Our results show that in terms of computation costs, AL Tree outperforms TA and CA by orders of magnitude. Exploring the space efficiently saves a lot of cost over merging large lists. In terms of the IO cost,

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

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

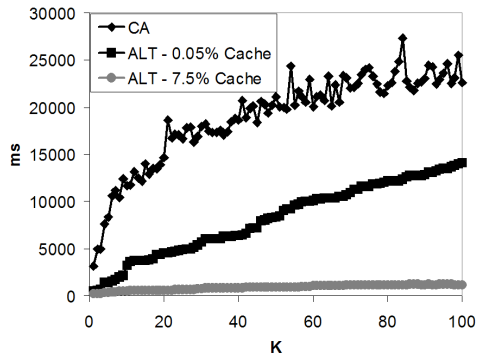

Figure 18: Response Time vs k (Forest Cover)

AL Tree performs better for dense datasets whereas CA performs better when the dataset is very sparse. As such AL Tree is the recommended structure for dense datasets and when memory available is large so that computational costs become dominant.

For future work, the IO performance of AL Tree warrants further study. In this paper, we have used an algorithm that tries to pack subtrees onto a disk page. There could be better ways of mapping the tree to disk that need to be explored. The access pattern on the AL Tree is such that the same node and page may be visited multiple times during a search. While this is not an issue for in memory computation, it leads to many more IOs for disk based computation if the cache size is small. Caching entire nodes in pages is wasteful since children that are far with respect to the query value may never be visited. Thus, rather than using the available memory to cache entire pages, it may be better to use it to store candidates. Once a page is read, we could generate all possible candidates from it and add them to the candidate pool. By using the available memory to store the most promising candidates, we could potentially avoid revisiting a page. This would lead to a big saving in the IO cost and make it feasible for low density datasets as well. Another direction for further study is to handle attributes with large number of distinct values. For attributes that are from an ordered domain such as $\Re$ and take many distinct values, we could bucketize the attribute values into a smaller number of buckets. This will effectively increase the data density, leading to better performance of the AL Tree at the cost of query accuracy.

9. REFERENCES

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

[2] 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.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[4] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[5] W. Chung, Gray and Horst. Windows 2000 disk io performance. *Microsoft Research Technical Report, MSTR-2000-55*, June 2000.

[6] C. B. D.J. Newman, S. Hettich and C. Merz. UCI repository of machine learning databases, 1998.

[7] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl.*, 21(1):9–33, 2003.

[8] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226. ACM Press, 1996.

[9] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

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

[11] K. Goh, B. Li, and E. Chang. Dyndex: A dynamic and nonmetric space indexer, 2002.

[12] U. Guntzer, W.-T. Balke, and W. Kiesling. Towards efficient multi-feature queries in heterogeneous environments. *itcc*, 00:0622, 2001.

[13] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. Software Eng.*, 9(5):631–634, 1983.

[14] K. Kummamuru, R. Krishnapuram, and R. Agrawal. On learning asymmetric dissimilarity measures. In *ICDM*, pages 697–700. IEEE Computer Society, 2005.

[15] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *ICDE*, page 72. IEEE Computer Society, 2006.

[16] T. Mandl. Learning similarity functions in information retrieval. In *EUFIT*, pages 771–775, 1998.

[17] A. Marian, N. Bruno, and L. Gravano. Evaluating top-queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[18] T. Skopal. On fast non-metric similarity search by metric access methods. In *EDBT*, pages 718–736, 2006.

[19] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 648–659. Morgan Kaufmann, 2004.

[20] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[21] E. Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15(1):1–7, 1994.

[22] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD Conference*, pages 103–114, 2007.

[23] P. Zesula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*. Springer, 1978.