Efficient Multidimensional AkNN Query Processing in the Cloud

Nikolaos Nodarakis[1], Evaggelia Pitoura[2], Spyros Sioutas[3], Athanasios Tsakalidis[1], Dimitrios Tsoumakos[3], and Giannis Tzimas[4]

[1] Computer Engineering and Informatics Department, University of Patras, 26500 Patras, Greece
{nodarakis,tsak}@ceid.upatras.gr
[2] Computer Science Department, University of Ioannina
pitoura@cs.uoi.gr
[3] Department of Informatics, Ionian University, 49100 Corfu, Greece
{sioutas,dtsouma}@ionio.gr
[4] Computer & Informatics Engineering Department, Technological Educational Institute of Western Greece, 26334 Patras, Greece
tzimas@cti.gr

Abstract. A k-nearest neighbor (kNN) query determines the k nearest points, using distance metrics, from a given location. An all k-nearest neighbor (AkNN) query constitutes a variation of a kNN query and retrieves the k nearest points for each point inside a database. Their main usage resonates in spatial databases and they consist the backbone of many location-based applications and not only. In this work, we propose a novel method for classifying multidimensional data using an AkNN algorithm in the MapReduce framework. Our approach exploits space decomposition techniques for processing the classification procedure in a parallel and distributed manner. To our knowledge, we are the first to study the kNN classification of multidimensional objects under this perspective. Through an extensive experimental evaluation we prove that our solution is efficient, robust and scalable in processing the given queries.

Keywords: classification· nearest neighbor· MapReduce· Hadoop· multidimensional data· query processing

1 Introduction

Classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. One of the algorithms for data classification uses the kNN approach [6] as it computes the k nearest neighbors (belonging to the training dataset) of a new object and classifies it to the category that belongs the majority of its neighbors.

Author names appear in alphabetical order

A k-nearest neighbor query [11] computes the k nearest points, using distance metrics, from a specific location and is an operation that is widely used in spatial databases. An all k-nearest neighbor query constitutes a variation of a kNN query and retrieves the k nearest points for each point inside a dataset in a single query process. Although AkNN is a fundamental query type, it is computationally very expensive. As a result, quite a few centralized algorithms and structures (M-trees, R-trees, space-filling curves, etc.) have been developed towards this direction [4], [7], [22]. However, as the volume of datasets grows rapidly even these algorithms cannot cope with the computational burden produced by an AkNN query process. Consequently, high scalable implementations are required. Cloud computing technologies provide tools and infrastructure to create such solutions and manage the input data in a distributed way among multiple servers. The most popular and notably efficient tool is the *MapReduce* [5] programming model, developed by Google, for processing large-scale data.

In this paper, we propose a method for efficient multidimensional data classification using AkNN queries in a single batch-based process in *Hadoop* [14], [16], the open source MapReduce implementation. More specifically, we sum up the technical contributions of our paper as follows:

- We present an implementation of a classification algorithm based on AkNN queries using MapReduce. We apply space decomposition techniques (based on data distribution) in order to bound the amount of distance calculations needed to reckon the k-NN objects before the classification step. The implementation defines the MapReduce jobs with no modifications to the original Hadoop framework.
- We provide an extension for $d > 3$ in Section 5 (d stands for dimensionality).
- We evaluate our solution through an experimental evaluation against large scale data up to 3 dimensions, that studies various parameters that can affect the total computational cost of our method using real and synthetic datasets. The results prove that our solution is efficient, robust and scalable.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the initial idea of the algorithm, our technical contributions and some examples of how the algorithm works. Section 4 presents a detailed analysis of the classification process developed in Hadoop, Section 5 provides an extension for $d > 3$ and Section 6 presents the experiments that where conducted in the context of this work. Finally, Section 7 concludes the paper and presents future steps.

2 Related Work

AkNN queries have been extensively studied in literature. A structure that is popular for answering efficiently to AkNN queries is R-tree [11]. Pruning techniques can be combined with such structures to deliver better results [4], [7]. Moreover, efforts have been made to design low computational cost methods

that execute such queries in spatial databases [18]. The works in [17], [20] propose algorithms to answer kNN join.

The methods proposed above can handle data of small size in one or more dimensions, thus their use is limited in centralized environments only. During the recent years, the researchers have focused on developing approaches that are applicable in distributed environments, like our method, and can manipulate big data in an efficient manner. The MapReduce framework seems to be suitable for processing such queries. For example, in [19] the discussed approach splits the target space in smaller cells and looks into appropriate cells where k-NN objects are located, but applies only in 2-dimensional data. Our method speeds up the naive solution of [19] by eliminating the merging step, as it is a major drawback. We have to denote here that in [19] it is claimed that the computation of the merging step can be performed in one node since we just consider statistic values. But this is not entirely true as we are going to see in the experimental evaluation. In addition, the merging step can produce sizeable groups of points, especially as k increments, that can overload the AkNN process. Moreover, our method applies for more dimensions. Especially, for $d >= 3$ the multidimensional extension is not straightforward at all.

In [13], locality sensitive hashing (LSH) is used together with a MapReduce implementation for processing kNN queries over large multidimensional datasets. This solution suggests an approximate algorithm like the work in [21] (H-zkNNJ) but we focus on exact processing AkNN queries. Furthermore, AkNN queries are utilized along with MapReduce to speed up and optimize the join process over different datasets [1], [10] or support non-equi joins [15]. Moreover, [2] makes use of a R-tree based method to process kNN joins efficiently.

In [3] a minimum spanning tree based classification model is introduced and it can be viewed as an intermediate model between the traditional k-nearest neighbor method and cluster based classification method. Another approach presented in [9] recommends parallel implementation methods of several classification algorithms but does not contemplate the perspective of dimensionality.

In brief, our proposed method implemented in the Hadoop MapReduce framework, extends the traditional kNN classification algorithm and processes exact AkNN queries over massive multidimensional data to classify a huge amount of objects in a single batch-based process. The experimental evaluation considers a wide diversity of factors that can affect the execution time such as the value of k, the granularity of space decomposition, dimensionality and data distribution.

3 Overview of Classification Algorithm

In this section, we first define some notation and provide some definitions used throughout this paper. Table 1 lists the symbols and their meanings. Next, we give a brief review of the method our solution relies on and then we extend it for more dimensions and tackle some performance issues.

Table 1. Symbols and their meanings

n	granularity of space decomposition
k	number of nearest neighbors
d	dimensionality
D	a d-dimensional metric space
$dist(r, s)$	the distance from r to s
$kNN(r, S)$	the k nearest neighbors of r from S
$AkNNC(R, S)$	$\forall r \in R$ classify r based on $kNN(r, S)$
I	input dataset
T	training dataset
c_r	the class of point r
C_T	the set of classes of dataset T
S_I	size of input dataset
S_T	size of training dataset
M	total number of Map tasks
R	total number of Reduce tasks

3.1 Definitions

We consider points in a d-dimensional metric space D. Given two points r and s we define as $dist(r, s)$ the distance between r and s in D. In this paper, we used the distance measure of Euclidean distance

$$(r, s) = \sqrt{\sum_{i=1}^{d} (r[i] - s[i])^2}$$

where $r[i]$ (respectively $s[i]$) denote the value of r (respectively s) along the i-th dimension in D. Without loss of generality, alternative distance measures (i.e. Manhattan distance) can be applied to our solution.

Definition 1. ***kNN:*** *Given a point r, a dataset S and an integer k, the k nearest neighbors of r from S, denoted as $kNN(r, S)$, is a set of k points from S such that $\forall p \in kNN(r, S)$, $\forall q \in \{S - kNN(r, S)\}, dist(p, r) < dist(q, r)$.*

Definition 2. ***AkNN:*** *Given two datasets R, S and an integer k, the all k nearest neighbors of R from S, named $AkNN(R, S)$, is a set of pairs (r, s) such that $AkNN(R, S) = \{(r, s) : r \in R, s \in kNN(r, S)\}$.*

Definition 3. ***AkNN Classification:*** *Given two datasets R, S and a set of classes C_S where points of S belong, the classification process produces a set of pairs (r, c_r), denoted as $AkNNC(R, S)$, such that $AkNNC(R, S) = \{(r, c_r) : r \in R, c_r \in C_S\}$ where c_r is the class where the majority of $kNN(r, S)$ belong $\forall r \in R$.*

3.2 Classification Using Space Decomposition

Consider a training dataset T, an input dataset I and a set of classes C_T where points of T belong. First of all, we define as *target space* the space enclosing

the points of I and T. The parts that occur when we decompose the target space for 1-dimensional objects are called *intervals*. Respectively, we call *cells* and *cubes* the parts in case of 2 and 3-dimensional objects and hypercubes for $d > 3$. For a new 1D point p, we define as *boundary interval* an interval centred at p that covers k-NN elements. Respectively, we define the *boundary circle* and *boundary sphere* for 2D and 3D points and the *boundary hypersphere* for $d > 3$. The notion of hypercube and hypersphere are analyzed further in Section 5. When the boundary ICSH (interval, circle, sphere or hypersphere) centred in an ICCH (interval, cell, cube or hypercube) $icch_1$, intersects the bounds of an other $icch_2$ we say an *overlap* occurs on $icch_2$. Finally, for a point $i \in I$, we define as *updates* of $kNN(i, T)$ the existence of many different instances of $kNN(i, T)$ that need to be unified to a final set.

We place the objects of T on the target space according to their coordinates. The main idea of equal-sized space decomposition is to partition the target space into n^d equal sized ICCHs where n and the size of each ICCH are user defined. Each ICCH contains a number of points of T. Moreover, we define a new layer over the target space according to C_T and $\forall t \in T, c_t \in C_T$. In order to estimate $AkNNC(I, T)$, we investigate $\forall i \in I$ for k-nearest neighbors only in a few ICCHs, thus bounding the number of computations required.

3.3 Previous Work

A very preliminary study of naive AkNN solutions is presented in [19] and uses a simple cell decomposition technique to process AkNN queries on two different datasets, i.e. I and T. The elements of both datasets are placed on the target space, which comprises of $2^n \times 2^n$ equal-sized cells, according to their coordinate vector and a cell decomposition is applied. $\forall i \in I$ it is expected that its $kNN(i, T)$ will be located in a close range area defined by nearby cells. At first, we look for candidate k-NN points inside the cell (cl) that i belongs in the first place. If we find at least k elements we draw the boundary circle. In case any neighboring cells are overlapped we need to investigate for possible k-NN objects inside them. If no overlap occurs, the k-NN list of i is complete. The algorithm outputs an instance of the k-NN list for every overlapped cell. These instances need to be unified into a final k-NN list.

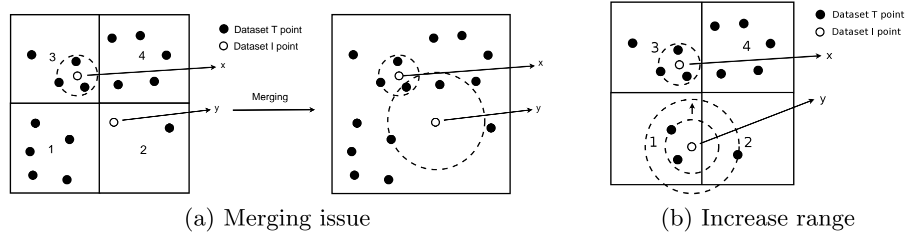
This approach, as described above, fails to draw the boundary circle if cl contains less than k points. To overcome this issue, before starting calculating $kNN(i, T)$, we need to estimate the number of points that fall into every cell and merge neighboring cells (according to the principles of hierarchical space decomposition used in quad-trees [12]) to assure that all will contain at least k objects. This preprocessing phase induces additional cost to the total computation and the merging step can lead to a bad algorithmic behavior.

3.4 Technical Contributions

In this subsection, we extend the previous method for more dimensions and adapt it to the needs of the classification problem. Moreover, we analyze some

drawbacks of the method studied in [19] and propose a mechanism to make the algorithm more efficient. Firstly, we have a training dataset T, an input dataset I and a set of classes C_T where points of T belong. The points in the training dataset have two attributes, the coordinate vector and the class they belong. In order to compute $AkNNC(I,T)$, a classification step is executed after the construction of the k-NN lists. Furthermore, we extend the solution presented in [19] for more dimensions, and now the space is decomposed in 2^{dn} ICCHs.

Figure 1(a) depicts a situation where the merging step of the original method in [19] can significantly increase the total cost of the algorithm. Consider two points x and y entering cells 3 and 2 respectively and $k = 3$. We can draw point's x boundary circle since cell 3 includes at least k elements. On the contrary, we cannot draw the boundary circle of point y, so we need to unify cells 1 through 4 into one bigger cell. Now point y can draw its boundary circle but we overload point's x k-NN list construction with redundant computations and this would happen for all points that would join cells 1,3 and 4 in the first place.



(a) Merging issue (b) Increase range

Fig. 1. Issue of the merging step before the kNN process and way to avoid it ($k = 3$)

In order to avoid a scenario like above, we introduce a mechanism where only points that cannot find at least k-nearest neighbors in the ICCH in the first place proceed to further actions. Let a point p joining an ICCH $icch$ that encloses $l < k$ neighbors. We draw the boundary ICSH based on these l neighbors and then check if the boundary ICSH overlaps any neighboring ICCHs. In case it does, if the boundary ICSH covers at least k elements in total, then we are able to build the final k-NN list of the point. In case the boundary ICSH does not cover at least k objects in total or does not overlap any ICCHs then we gradually increase its search range (by a fraction of the size of the ICCH each time) until the prerequisites are fulfilled.

Figure 1(b) explains this issue. Consider two points x and y entering cells 3 and 1 respectively and $k = 3$. We observe that cell 3 contains 4 neighbors and point x can draw its boundary circle that covers k-NN elements. However, the boundary circle centred at y does not cover k-NN elements. Consequently, we gradually increase its search range until the boundary circle encloses at least k-NN points. By eliminating the merging step, we also relax the condition of decomposing the target space into 2^{dn} equal-sized splits and generalize it to n^d.

Summing up, our solution can be implemented as a series of MapReduce jobs as shown below. Note, that the first MapReduce job acts as a preprocessing step and its results are provided as additional input in MapReduce Job 3 and that the preprocessing step is executed only once for T.

1. **Distribution Information**. Count the number of points of T that fall into each ICCH. The output of this job is utilized by the third MapReduce job to help determine how much we need to increase the boundary ICSH.
2. **Primitive Computation Phase**. Calculate possible k-NN points $\forall i \in I$ from T in the same ICCH.
3. **Update Lists**. Draw the boundary ICSH $\forall i \in I$ and increase it, if needed, until it covers at least k-NN points of T. Check for overlaps of neighboring ICCHs and derive updates of k-NN lists.
4. **Unify Lists**. Unify the updates of every k-NN list into one final k-NN list $\forall i \in I$.
5. **Classification**. Classify all points of I.

4 Detailed Analysis of Classification Procedure

In this section, we present a detailed description of the classification process as implemented in the Hadoop framework. The records in T have the format <point_id, coordinate_vector, class> and in I have the format <point_id, coordinate_vector>. Furthermore, parameters n and k are defined by the user. In the following subsections, we describe each MapReduce job separately and analyze the Map and Reduce functions that take place in each one of them[1]. Also, we proceed in time and space complexity analysis.

4.1 Getting Distribution Information of Training Dataset

This MapReduce job is a preprocessing step required by subsequent MapReduce jobs that receive its output as additional data. In this step, we decompose the entire target space and count the number of points of T that fall in each ICCH.

The *Map* function takes as input records with the training dataset format, estimates the ICCH id for each point based on its coordinates and outputs a key-value pair where the key is ICCH id and the value is number 1. The *Reduce* function receives the key-value pairs from the Map function and for each ICCH id it outputs the number of points of T that belong to it.

Each Map task needs $O(S_T/M)$ time to run. Each Reduce task needs $O\left(n^d/R\right)$ time to run as the total number of ICCHs is n^d. So, the size of the output will be $O\left(n^d \cdot c_{\mathrm{si}}\right)$, where c_{si} is the size of sum and icch_id for an output record.

[1] Due to space limitations we do not quote pseudo-code for Map and Reduce functions. Pseudo-code and more details of the current work are available in a technical report in `http://arxiv.org/abs/1402.7063`

4.2 Estimating Primitive Phase Neighbors of A*k*NN Query

In this stage, we concentrate all training (L_T) and input (L_I) records for each ICCH and compute possible k-NN points for each item in L_I from L_T inside the ICCH. Below, we condense the Map and Reduce functions. We use two Map functions in this job, one for each dataset.

For each point $t \in T$, *Map1* outputs a new key-value pair in which the ICCH id, where t belongs, is the key and the value consists of the id, coordinate vector and class of t. Similarly, for each point $i \in I$, *Map2* outputs a new key-value pair in which the ICCH id where i belongs is the key and the value consists of the id and coordinate vector of i. The *Reduce* function receives a set of records from both Map functions with the same ICCH ids and separates points of T from points of I into two lists, L_T and L_I respectively. Then, the Reduce function calculates the distance for each point in L_I from L_T, estimates the k-NN points and forms a list L with the format $< p_1, d_1, c_1 \colon \ldots \colon p_k, d_k, c_k >$, where p_i is the i-th NN point, d_i is its distance and c_i is its class. Finally, for each $p \in L_I$, *Reduce* outputs a new key-value pair in which the key is the id of p and the values comprises of the coordinate vector, ICCH id and list L of p.

Each Map1 task needs $O\left(S_T/M\right)$ time and each Map2 task needs $O\left(S_I/M\right)$ time to run. Suppose u_i and t_i the number of input and training points that are enclosed in an ICCH in the i-th execution of a Reduce function and $1 \leq i \leq n^d/R$. Each Reduce task needs $O\left(\sum_i u_i \cdot t_i\right)$. Let L_{s} to be the size of k-NN list and ICCH id $\forall i \in I$. The output size is $O\left(S_I \cdot L_{\mathrm{s}}\right)$, which is $O\left(S_I\right)$.

4.3 Checking for Overlaps and Updating *k*-NN Lists

In this step, at first we gradually increase the boundary ICSH (how much depends on information from the first MapReduce job), where necessary, until it includes at least k points. Then, we check for overlaps between neighboring ICCHs and derive updates of the k-NN lists. The Map and Reduce functions of this job are outlined next (again, we have two Map functions).

The *Map1* function is exactly the same as *Map1* function in the previous job. For each point $i \in I$, function *Map2* computes the overlaps with neighboring ICCHs. If no overlap occurs, it does not need to perform any additional steps and outputs a key-value pair in which ICCH id is the key and the value consists of id, coordinate vector and list L of i and a flag *true* which implies that no further process is required. Otherwise, for every overlapped ICCH it outputs a new record where ICCH id′ (id of an overlapped ICCH) is the key and the value consists of id, coordinate vector and list L of i and a flag *false* that indicates we need to search for possible k-NN objects inside the overlapped ICCHs. The *Reduce* function receives a set of points with the same ICCH ids and separates the points of T from points of I into two lists, L_T and L_I respectively. After that, the Reduce function performs extra distance calculations using the points in L_T and updates k-NN lists for the records in L_I. Finally, for each $p \in L_I$ it generates a record in which the key is the id of p and the values comprises of the coordinate vector, ICCH id and list L of p.

Each Map1 task needs $O(S_T/M)$ time to run. Consider an unclassified point p initially belonging to an ICCH *icch*. Let r be the number of times we increase the search range for p and *icchov* the number of ICCHs that may be overlapped for p. For each Map2 task the i-th execution of the Map function performs $icchov_i + r_i$ steps, where $1 \leq i \leq S_I/M$. So, each Map2 task runs in $O\left(\sum_i (icchov_i + r_i)\right)$ time. Suppose u_i and t_i the number of points of I and T respectively that are enclosed in an ICCH in the i-th execution of a Reduce function and $1 \leq i \leq n^d/R$. Each Reduce task needs $O\left(\sum_i u_i \cdot t_i\right)$. The size of updated records is a fraction of S_I. So, the size of the output is also $O(S_I)$.

4.4 Unifying Multiple k-NN Lists

During the previous step it is possible that multiple updates of a point's k-NN list might occur. This MapReduce job tackles this problem and unifies possible multiple lists into one final k-NN list for each point $i \in I$.

The *Map* function receives the records of the previous step and extracts the k-NN list for each point. For each point $i \in I$, it outputs a key-value pair in which the key is the id of i and the value is the list L. The *Reduce* function receives as input key-value pairs with the same key and computes $kNN(i,T), \forall i \in I$. The key of an output record is again the id of i and the value consists of $kNN(i,T)$.

Each Map task runs in $O(S_I/M)$. For each Reduce task, assume $updates_i$ the number of updates for the k-NN list of an unclassified point in the i-th execution of a Reduce function, where $1 \leq i \leq |N_I|/R$ and $|N_I|$ the number of points in input dataset. Then, each Reduce task needs $O\left(\sum_i updates_i\right)$ to run. Let, I_{id} the size of ids of all points in I and L_{final} is the size of the final k-NN list $\forall i \in I$. The size of L_{final} is constant and I_{id} is $O(S_I)$. Consequently, the size of the output is $O(S_I)$.

4.5 Classifying Points

This is the final job of the whole classification process. It is a Map-only job that classifies the input points based on the class membership of their k-NN points. The Map function receives as input records from the previous job and outputs $AkNNC(I,T)$. Each Map task runs in $O(S_I/M)$ time and output size is $O(S_I)$.

5 Extension for $d > 3$

Here we provide the extension of our method for $d > 3$. In geometry, a hypercube is a n-dimensional analogue of a square ($n = 2$) and a cube ($n = 3$) and is also called a n-cube (i.e. 0-cube is a hypercube of dimension zero and represents a point). It is a closed, compact and convex figure that consists of groups of opposite parallel line segments aligned in each of the space's dimensions, perpendicular to each other and of the same length. Figure 2 displays how to create a hypercube for $d = 4$ (4-cube) from a cube for $d = 3$. Respectively, an n-sphere is a generalization of the surface of an ordinary sphere to a n-dimensional space.

Spheres of dimension $n > 2$ are called hyperspheres. For any natural number n, an n-sphere of radius r is defined as a set of points in $(n+1)$-dimensional Euclidean space which are at distance r from a central point and r may be any positive real number. So, the n-sphere centred at the origin is defined by:

$$S^n = \{x \in \Re^{n+1} : \| x \| = r\}$$

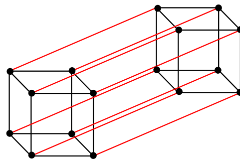


Fig. 2. Creating a 4-cube from a 3-cube

6 Experimental Evaluation

In this section, we conduct a series of experiments to evaluate the performance of our method under many different perspectives such as the value of k, the granularity of space decomposition, dimensionality and data distribution.

Our cluster includes 32 computing nodes (VMs), each one of which has four 2.1 GHz CPU processors, 4 GB of memory, 40 GB hard disk and the nodes are connected by 1 gigabit Ethernet. On each node, we install Ubuntu 12.04 operating system, Java 1.7.0_40 with a 64-bit Server VM, and Hadoop 1.0.4. To adapt the Hadoop environment to our application, we apply the following changes to the default Hadoop configurations: the replication factor is set to 1; the maximum number of Map and Reduce tasks in each node is set to 3, the DFS chunk size is 256 MB and the size of virtual memory for each Map and Reduce task is set to 512 MB. We evaluate the following approaches in the experiments: a) *kdANN*, which is the solution proposed in [19] along with the extension (which invented and implemented by us) for more dimensions, in order to be able to compare it with our solution and b) *kdANN+*, which is our solution for d-dimensional points without the merging step as described in Section 3.

We evaluate our solution using both real[2] and synthetic datasets. We create 1D and 2D datasets from the real dataset keeping the x and the (x, y) coordinates respectively. We process the dataset to fit into our solution (i.e. normalization) and we end up with 1D, 2D and 3D datasets that consist of approximately 19,000,000 points and follow a power law like distribution. Respectively, we create 1, 2 and 3-dimensional datasets containing 19,000,000 uniformly distributed points. From each dataset, we extract a fraction of points (10%) that are used as

[2] The real dataset is part of the Canadian Planetary Emulation Terrain 3D Mapping Dataset and is available in `http://asrl.utias.utoronto.ca/datasets/3dmap/`

a training dataset. For each point in a training dataset we assign a class based on its coordinate vector. The file sizes (in MB) of real datasets are a) Input: {(1D, 309.5), (2D, 403.5), (3D, 523.7)} and b) Training: {(1D, 35), (2D, 44.2), (3D, 56.2)}. The file sizes (in MB) of synthetic datasets are a) Input: {(1D, 300.7), (2D, 359.2), (3D, 478.5)} and b) Training: {(1D, 33.9), (2D, 39.8), (3D, 51.7)}.

One major aspect in the performance of the algorithm is the tuning of granularity parameter n. Each time the target space is decomposed into 2^{dn} equal parts in order for kdANN to be able to perform the merging step, as described in Section 3. The values of n that were chosen for the rest of the experiments are: a) Real dataset: (1D, 18), (2D, 9), (3D, 7) and b) Synthetic dataset: (1D, 16), (2D, 7), (3D, 5). The procedure that was carried out in order to end up with these values is described in the aforementioned technical report.

6.1 Effect of k and Effect of Dimensionality

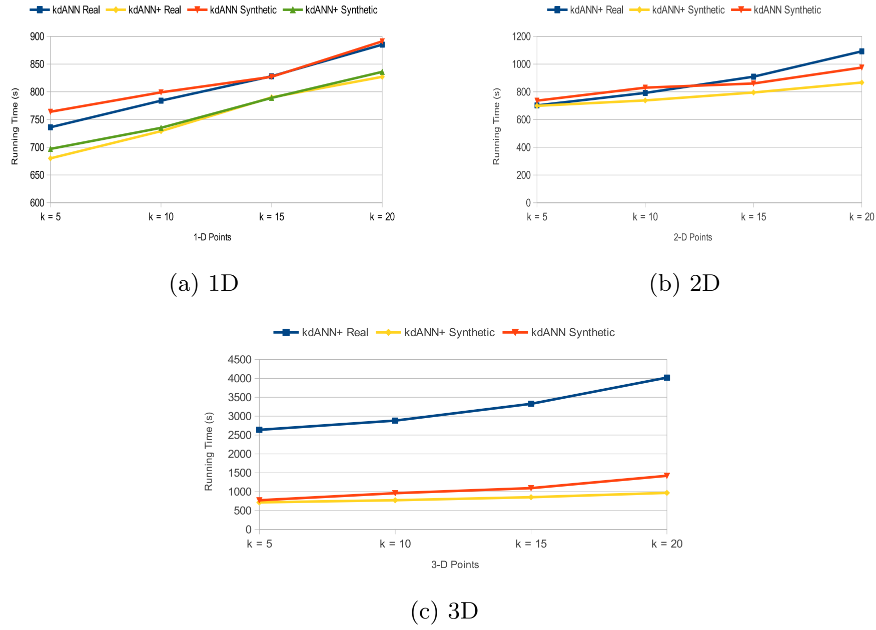
In this experiment, we evaluate both methods using real and synthetic datasets and record the execution time as k increases for each dimension. Then, we study the effect of dimensionality on the performance of kdANN and kdANN+.

Figure 3(a) presents the results for kdANN and kdANN+ by varying k from 5 to 20 on 1D real and synthetic datasets. In terms of running time, kdANN+ always perform best, followed by kdANN and each method behave in the same way for both datasets, real and synthetic.

In Fig. 3(b), we demonstrate the outcome of the experimental procedure for 2D points when we alter k value from 5 to 20. No results of kdANN are included for the real dataset since the method only produced results for $k = 5$ and needed more than 4 hours. Beyond this, the merging step of kdANN derived extremely sizeable cells leading to a bottleneck to some nodes that strangled their resources, thus preventing them to derive any results. Overall, in the case of power law distribution, kdANN+ behaves much better than kdANN since the last one fails to process an AkNN query as k increases. Also, kdANN+ is faster and in case of synthetic dataset, especially as k grows.

Figure 3(c) displays the results generated from kdANN and kdANN+ for 3D points when we increase k value from 5 to 20. Once again, kdANN could not produce any results for any value of k in the case of real dataset. Table 2 is pretty illustrative in the way the merging step affects the AkNN process. The computational cost is far from negligible if performed in a node (in contrary with the claim of the authors in [19]). Apart from this, the largest merged cube consists of 32,768 and 262,144 initial cubes for $k = 5$ and $k > 5$ respectively. In the case of kdANN+ for the real dataset, it is obvious that the total computational cost is much larger compared to the one shown in Figs. 3(a) and 3(b). Finally, kdANN+ outperforms kdANN, in the case of synthetic dataset, and the gap between the curves of running time tends to be bigger as k increases.

Overall, looking at Figs. 3(a), 3(b) and 3(c), we observe that as k increases the execution time augments. This occurs because we need to perform more distance calculations and the size of intermediate records becomes larger respectively as the value of k rises.



(a) 1D

(b) 2D

(c) 3D

Fig. 3. Effect of k for $d \in \{1, 2, 3\}$

Now, we evaluate the effect of dimensionality for both real and synthetic datasets. Figure 4 presents the running time for $k = 20$ by varying the number of dimensions from 1 to 3. From the outcome, we observe that kdANN is more sensitive to the number of dimensions than kdANN+ when we provide a dataset with uniform distribution as input. In particular, when the number of dimensions varies from 2 to 3 the divergence between the two curves starts growing faster. In the case of power law distribution, we notice that the execution time of kdANN+ increases exponentially when the number of dimensions varies from 2 to 3. This results from the curse of dimensionality. As the number of dimensions increases, the number of distance computations as well as the number of searches in neighboring ICCHs increases exponentially. Nevertheless, kdANN+ can still process the AkNN query in a reasonable amount of time in contrast to kdANN.

Table 2. Statistics of merging step for kdANN

	$k = 5$	$k = 10$	$k = 15$	$k = 20$
Time (s)	271	675	962	1,528
# of merged cubes	798,032	859,944	866,808	870,784
% of total cubes	38%	41%	41.3%	41.5%
Max merged cubes	32,768	262,144	262,144	262,144

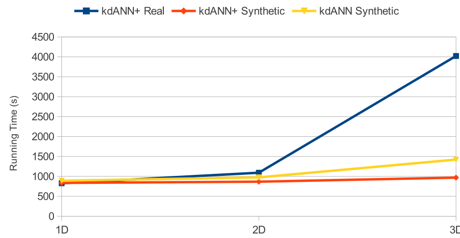


Fig. 4. Effect of dimensionality for $k = 20$

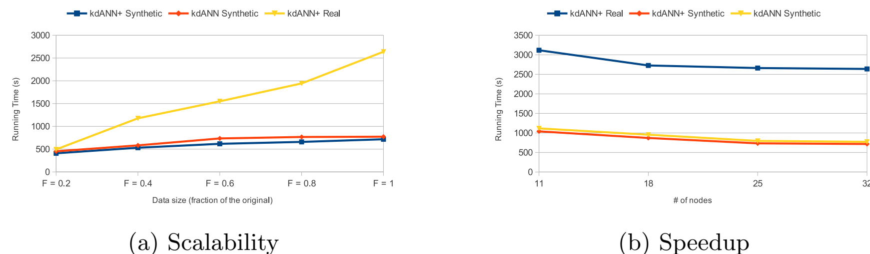
6.2 Scalability and Speedup

In this subsection, we investigate the scalability and speedup of the two approaches. In the scalability experiment we utilize the 3D datasets, since their size is bigger than the others, and create new chunks smaller in size that are a fraction F of the original datasets, where $F \in \{0.2, 0.4, 0.6, 0.8\}$. Moreover, we set the value of k to 5. Figure 5(a) presents the scalability results for real and synthetic datasets. In the case of power law distribution, the results display that kdANN+ scales almost linearly as the data size increases. In contrast, kdANN fails to generate any results even for very small datasets since the merging step continues to be an inhibitor factor in its performance. In addition, we can see that kdANN+ scales better than kdANN in the case of synthetic dataset and the running time increases almost linearly as in the case of power law distribution. Regarding kdANN, the curve of execution time is steeper until $F = 0.6$ and after that it increases more smoothly.

In our last experiment, we measure the effect of the number of computing nodes. We test four different cluster configurations and the cluster consist of $N \in \{11, 18, 25, 32\}$ nodes each time. As before, we use the 3D datasets when $k = 5$. Figure 5(b), displays that total running time of kdANN+, in the case of power law distribution, tends to decrease as we add more nodes to the cluster. Due to the increment of number of computing nodes, the amount of distance calculations and update steps on k-NN lists that undertakes each node decreases respectively. Moreover, it is obvious that kdANN will fail to produce any results when $N < 32$. This explains the absence of kdANN's curve from Fig. 5(a). In the case of synthetic dataset, we observe that both kdANN and kdANN+ achieve almost the same speedup; still kdANN+ performs betters than kdANN. Observing Fig. 5(b) we deduce that the increment of computing nodes has a greater effect on the running time of both approaches when the datasets follow a uniform distribution due to better load balancing.

7 Conclusions and Future Work

In the context of this work, we presented a novel method for classifying multidimensional data using AkNN queries in a single batch-based process in Hadoop.



(a) Scalability (b) Speedup

Fig. 5. Scalability and speedup results

To our knowledge, it is the first time a MapReduce approach for classifying multidimensional data is discussed. By exploiting equal-sized space decomposition techniques we bound the number of distance calculations needed to compute $kNN(i, S), \forall i \in I$. We conduct a variety of experiments using real and synthetic datasets and prove that our system is efficient, robust and scalable.

In the near future, we plan to extend and improve our system in order to become more efficient and flexible. At first, we have in mind to implement a technique that will allow us to have unequal splits that will contain approximately the same number of points. In this way we will achieve distribution independence and better load balancing between the nodes. In addition, we intend to apply a mechanism in order for the cluster to be used in a more elastic way, by adding/removing nodes as the number of dimensions increase/decrease. Finally, we plan to use indexes in order to prune any points that are redundant and cumber additional cost to the method.

Acknowledgements. This work was partially supported by Thales Project entitled “Cloud9: A multidisciplinary, holistic approach to internet-scale cloud computing”. For more details see the following URL:
`https://sites.google.com/site/thaliscloud9/home`

References

1. Afrati, F.N., Ullman, J.D.: Optimizing Joins in a Map-Reduce Environment. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 99–110. ACM, New York, NY, USA (2010)
2. Böhm, C., Krebs, F.: The k-Nearest Neighbour Join: Turbo Charging the KDD Process. Knowl. Inf. Syst. 6, 728–749 (2004)
3. Chang, J., Luo, J., Huang, J.Z., Feng, S., Fan, J.: Minimum Spanning Tree Based Classification Model for Massive Data with MapReduce Implementation. In: Proceedings of the 10th IEEE International Conference on Data Mining Workshop, pp. 129–137. IEEE Computer Society, Washington, DC, USA (2010)
4. Chen, Y., Patel, J.M.: Efficient Evaluation of All-Nearest-Neighbor Queries. In: Proceedings of the 23rd IEEE International Conference on Data Engineering, pp. 1056–1065. IEEE Computer Society, Washington, DC, USA (2007)

5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 137–150. USENIX Association, Berkeley, CA, USA (2004)
6. Dunham, M.H.: Data Mining, Introductory and Advanced Topics. Prentice Hall, Upper Saddle River, NJ, USA (2002)
7. Emrich, T., Graf, F., Kriegel, H.-P., Schubert, M., Thoma, M.: Optimizing All-Nearest-Neighbor Queries with Trigonometric Pruning. In: Scientific and Statistical Database Management. LNCS, vol. 6187, pp. 501–518. Springer-Verlag, Berlin, Heidelberg (2010)
8. Gkoulalas-Divanis, A., Verykios, V.S., Bozanis, P.: A Network Aware Privacy Model for Online Requests in Trajectory Data. Data Knowl. Eng. 68, 431–452 (2009)
9. He, Q., Zhuang, F., Li, J., Shi, Z.: Parallel implementation of classification algorithms based on MapReduce. In: Proceedings of the 5th International Conference on Rough Set and Knowledge Technology, pp. 655–662. Springer-Verlag, Berlin, Heidelberg (2010)
10. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient Processing of k Nearest Neighbor Joins using MapReduce. Proc. VLDB Endow. 5, 1016–1027 (2012)
11. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest Neighbor Queries. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 71–79. ACM, New York, NY, USA (1995)
12. Samet, H.: The QuadTree and Related Hierarchical Data Structures. ACM Comput. Surv. 16, 187–260 (1984)
13. Stupar, A., Michel, S., Schenkel, R.: RankReduce - Processing K-Nearest Neighbor Queries on Top of MapReduce. In: Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval, pp. 13–18. (2010)
14. The apache software foundation: Hadoop homepage, `http://hadoop.apache.org/`
15. Vernica, R., Carey, M.J., Li, C.: Efficient Parallel Set-Similarity Joins Using MapReduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 495–506. ACM, New York, NY, USA (2010)
16. White, T.: Hadoop: The Definitive Guide, 3rd Edition. O'Reilly Media / Yahoo Press (2012)
17. Xia, C., Lu, H., Chin, B., Hu, O.J.: Gorder: An efficient method for knn join processing. In: VLDB, pp. 756–767. VLDB Endowment (2004)
18. Yao, B., Li, F., Kumar, P.: K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free. In: Proceedings of the 26th International Conference on Data Engineering, pp. 4–15. IEEE Computer Society, Washington, DC, USA (2010)
19. Yokoyama, T., Ishikawa, Y., Suzuki, Y.: Processing All k-Nearest Neighbor Queries in Hadoop. In: Proceedings of the 13th International Conference on Web-Age Information Management. LNCS, vol. 7418, pp. 346–351 (2012)
20. Yu, C., Cui, B., Wang, S., Su, J.: Efficient index-based KNN join processing for high-dimensional data. Information & Software Technology 49, 332-344 (2007)
21. Zhang, C., Li, F., Jestes, J.: Efficient Parallel kNN Joins for Large Data in MapReduce. In: Proceedings of the 15th International Conference on Extending Database Technology, pp. 38–49. ACM, New York, NY, USA (2012)
22. Zhang, J., Mamoulis, N., Papadias, D., Tao, Y.: All-Nearest-Neighbors Queries in Spatial Databases. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management, pp. 297–306. IEEE Computer Society, Washington, DC, USA (2004)