Data & Knowledge Engineering 121 (2019) 42–70

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Efficient processing of all-k-nearest-neighbor queries in the MapReduce programming framework

Panagiotis Moutafis [a], George Mavrommatis [a], Michael Vassilakopoulos [a,*], Spyros Sioutas [b]

[a] Data Structuring & Engineering Lab, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
[b] Department of Computer Engineering & Informatics, University of Patras, Patras, Greece

ARTICLE INFO

Keywords:
Spatial query processing
Nearest neighbor query
Plane sweep
Quadtrees
MapReduce
Apache hadoop

ABSTRACT

Numerous modern applications, from social networking to astronomy, need efficient answering of queries on spatial data. One such query is the All k Nearest-Neighbor Query, or k Nearest-Neighbor Join, that takes as input two datasets and, for each object of the first one, returns the k nearest-neighbors from the second one. It is a combination of the k nearest-neighbor and join queries and is computationally demanding. Especially, when the datasets involved fall in the category of Big Data, a single machine cannot efficiently process it. Only in the last few years, papers proposing solutions for distributed computing environments have appeared in the literature. In this paper, we focus on parallel and distributed algorithms using the Apache Hadoop framework. More specifically, we focus on an algorithm that was recently presented in the literature and propose improvements to tackle three major challenges that distributed processing faces: improvement of load balancing (we implement an adaptive partitioning scheme based on Quadtrees), acceleration of local processing (we prune points during calculations by utilizing plane-sweep processing), and reduction of network traffic (we restructure and reduce the output size of the most demanding phase of computation). Moreover, by using real 2D and 3D datasets, we experimentally study the effect of each improvement and their combinations on performance of this literature algorithm. Experiments show that by carefully addressing the three aforementioned issues, one can achieve significantly better performance. Thereby, we conclude to a new scalable algorithm that adapts to the data distribution and significantly outperforms its predecessor. Moreover, we present an experimental comparison of our algorithm against other well-known MapReduce algorithms for the same query and show that these algorithms are also significantly outperformed.

1. Introduction

Numerous modern applications, like Geographical Information Systems (GIS), facilities management, location-based services, smart-cities services, etc., need efficient processing of queries on big spatial data. One such query is the All k Nearest Neighbor Query (AkNNQ, or kNN Join). Given two point datasets, Q (Query) and T (Training), the AkNNQ finds the k nearest neighbors of T for each point of Q, according to a certain distance metric [1].

This query is very useful in practice. For example, travel agencies using a decision support GIS for the design of custom touristic packages may request a list of hotels and their nearest points of interest, depending on the preferences of their customers. Other

* Corresponding author.
E-mail addresses: pmoutafis@uth.gr (P. Moutafis), gmav@uth.gr (G. Mavrommatis), mvasilako@uth.gr (M. Vassilakopoulos), sioutas@ceid.upatras.gr (S. Sioutas).

https://doi.org/10.1016/j.datak.2019.04.003
Received 29 July 2018; Received in revised form 20 March 2019; Accepted 16 April 2019
Available online 20 April 2019

applications of the AkNNQ include classification [2], graph-based computational learning [3], N-body simulations in astronomy [4], etc.

If the datasets involved are relatively small, the AkNNQ can be efficiently answered in a centralized environment. This is generally not the case when dealing with modern spatial data, considering that we live in the era of WWW and mobile computing. There are huge volumes of dynamic spatial data generated by GPS devices, sensor networks, geotagged tweets, scientific devices, etc. [5] and their processing is computationally very demanding, thus the use of a parallel and distributed computing environment is absolutely necessary to get results in a reasonable amount of time.

MapReduce [6–8] is a programming model for distributed computations on very large amounts of data and a framework for large-scale data processing on clusters built from commodity hardware. The MapReduce architecture provides good scalability and fault tolerance mechanisms. MapReduce was originally introduced by Google in 2004 and was based on well-known principles of parallel and distributed processing. It has been widely adopted through Hadoop (an open-source implementation), whose development was led by Yahoo and later became an Apache project [9,10].

To overcome limitations of the MapReduce paradigm and Apache Hadoop (especially regarding iterative algorithms), Apache Spark [11] was developed. This is also an open-source cluster-computing framework based on Resilient Distributed Datasets (RDDs), read-only multisets of data items distributed over the computing nodes. RDDs form a kind of distributed shared memory suitable for the implementation of iterative algorithms. Moreover, specialized spatial frameworks based on Hadoop and Spark, such as SpatialHadoop [12] and LocationSpark [13], respectively, have also been developed.

Hadoop is a shared-nothing framework, meaning that the input data is partitioned and randomly distributed to all computing nodes, which perform calculations on their local data only. This is the major difference from the centralized solutions and the reason why we have to invent a smart, multi-phase algorithm that makes all necessary calculations locally on each node and then rearranges the output results so that the next phases can elaborate on the (possibly incomplete, or partially wrong) results computed locally. Considering the AkNNQ, each node will initially get a set of points contained in the partition block assigned to it and for each query point it will find its k neighbors inside this block. By carefully assigning points to each node, load balancing and global performance is also improved, since all nodes complete their work at approximately the same time. Reducing the cost of processing at each node also contributes to the acceleration of the algorithm. Note that, the neighbor list computed in each node may, or may not be the final one, because the node lacks the location information of other points around, outside its block. Hadoop does not incorporate iteration capabilities, so we had to split the algorithm into phases and the output of each phase becomes the input for the next one. This results in data exchange among nodes and network traffic which costs heavily in performance. Therefore, reduction of the exchange of data between phases can also have a significant positive effect on performance.

In this paper we present a four phases algorithm, originally introduced in [14] and further improved in [2], based on MapReduce framework and implemented in Hadoop. We have added three significant optimizations to that algorithm: plane-sweep reducers (instead of brute force ones), customized Quadtree data partitioning (instead of flat grid) and data output reduction, which is most important, since flooding the network with data is the greatest performance drawback (see experiments section). Plane-sweep calculations in reducers results in efficient pruning of distant points. Quadtree partitioning splits the data into fewer, unequal and approximately similarly populated cells, thus overcoming data skewness and balancing the number of calculations per reducer. Finally, data output reduction practically halves the output of the third phase (performance gains increase as input data grows), which saves a lot of network bandwidth and significantly reduces overall time.

The rest of this paper is organized as follows. In Section 2, we present background material and related work. In Section 3, we briefly present the algorithm [2] (henceforth, referred to as "original" algorithm) which consists of five phases of MapReduce jobs and decomposes space into a grid of small equal sized cells. It will serve as the base for the development of and comparison to our optimized methods. In the next section, we make a detailed presentation of the improvements we have applied to the original algorithm; a new pruning method for determining nearest neighbors, reduction of data traffic on the network and partitioning to variable sized cells. Note that, the original algorithm and its improvements are presented for 2D (extending to 3D is straightforward). In Section 5, we present an extensive experimental study in 2D of the performance of the original algorithm vs. the one resulting from applying the proposed improvements and their combinations. This experimental study reveals that these improvements form a scalable algorithm that achieves high efficiency that self adapts to the data distribution. Experiments in 3D are presented in Section 6. In Section 7, we compare our work to other well known algorithms of the literature. Finally, in the last section, we summarize the contribution of our work and outline our plans for future research.

2. Preliminaries and related work

In this paper we are using the Euclidean distance ($dist$) in 2D and 3D metric space. Extensions utilizing other distance metrics (e.g. Manhattan, maximum distance, etc.) could also be developed.

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

Definition 2 ($AkNNQ$)**.** Given two datasets R and S and an integer k, the result of the All k Nearest Neighbors Query of R from S, denoted as $AkNNQ(R, S)$, is the set of pairs $\{(r, s) : r \in R, s \in kNN(r, S)\}$.

A naive approach to find the k nearest neighbors of two datasets would be to calculate the distances of every point of the one dataset, R, to every point of the other dataset, S, and sort the results by distance. Of course this would be highly inefficient as it would lead to a huge number of calculations, in the order of $O(|R| \times |S|)$.

There have been several attempts to solve this query in a single machine by using sophisticated data structures (such as the B^+-tree and the R-tree [15]) and pruning techniques, like in [1,16–19]. However, here we are interested in parallel and distributed solutions only. MapReduce and Hadoop's key–value programming model and shared-nothing architecture render most of those techniques almost useless, because each computing node "sees" only a part of the whole set and some data must be exchanged across the borders. The approach that most researchers (including ourselves) implement and try to improve is the following: partition the data into disjoint areas and send them to the computing nodes. Each node will calculate and return a kNN list for every query point, based on its local data. Then some additional phases are needed to exchange data among nodes and find possible misses of closer neighboring points, while trying to move as less data as possible between nodes. Our proposed improvements focus on better methods for partitioning, candidate neighbors pruning and network traffic reduction. Performance wise, the network probably plays the most significant role, as shown later in the experiments. Another factor is the load balancing between nodes, meaning that every node must complete its work at approximately the same time. Efficient partitioning plays significant role here, so that data are equally distributed to all nodes.

In [14] the authors present an AkNNQ MapReduce algorithm for 2D spatial data. They propose the decomposition of data space into small equal cells and afterwards the merging of some neighboring cells, always in 2×2 sets, if they do not contain k points, or more in total. This way they create bigger cells so that the initial kNN list of a query point will always be complete. Then they draw a boundary circle around each query point to ensure that this list is the final one (the circle does not overlap other cells), or more checks are needed in next phases.

The authors of [2] have improved this algorithm by replacing the merging step with a circle of increasing radius around the query point, so that the existence of candidate neighbors in nearby cells is checked. This way, the merging step is not needed and the number of distance calculations may be significantly reduced. The authors of [2] have also extended the algorithm to work with more than two dimensions and added a classification step.

The authors of [20] took the previous algorithm and modified the space decomposition technique. Their work is limited to two dimensions. They propose the division of the target space into a large number of equal columns and the merging of some of them to reach the expected average number of points per column. This step is performed on a single machine. Then they divide each merged column again to a large number of equal cells and they repeat the merging step of the cells, for each column. The rest of the algorithm remains the same as in [2]. According to [20], the calculations per (merged) cell are more balanced and they get better performance, compared to [2]. However, their description lacks several details of their algorithm and, thus, a reader cannot implement it. It must be noted here that the single machine merging of columns can seriously delay the whole process and the delay increases with the number of points and the number of columns, as we have noticed by implementing this step ourselves.

In [21], the authors use Voronoi diagram-based partitioning method that exploits pruning rules for distance filtering, and hence reduces both the shuffling and computational costs. It is a three-phase solution, consisting of a preprocessing step (finding Voronoi pivots; several methods are tested for that purpose) executed on a single machine and two MapReduce jobs. The first job is only a Mapper that computes the distance between a pivot and its nearest partition object, while collecting some statistics to calculate pruning distances. In the second and final job the Mapper pairs the partitions of the two datasets and the Reducer performs the kNN join, using the pruning heuristics from the previous phase.

In [22], the authors first perform a centralized hash based partitioning to divide the space into sub areas with approximately the same number of objects. Then each node computes a set of candidate neighbors of the adjacent sub areas to be transferred between them and contribute to the local kNN calculation. Finally each node performs kNN computation on its local data. To implement their algorithm the authors use distributed file system Tachyon and Parallel Java Library for MPI.

In [23], the authors present three AKNN MapReduce algorithms. The first one, H-BNLJ (Hadoop Block Nested Loop Join) splits each dataset into n^2 equal sized blocks. It then puts one block from the first dataset and all blocks from the other into n^2 buckets and computes the neighbors inside the bucket. This is the first phase. In the second phase the neighbors are grouped by distance and the final list is calculated. The second algorithm, H-BRJ (Hadoop Block R-tree Join), is an improvement to HBNLJ by using the R-tree for local indexing of the datasets inside each block. The last algorithm, H-zkNNJ (Hadoop based zkNN Join) is using pruning techniques for fast but approximate AKNN computations, therefore we will not present further details.

The authors in [24] implement a two phase AKNN MapReduce algorithm. First they divide the datasets into equal tiles and then they pack the tiles into buckets, where each bucket represents a MapReduce task. They use the Z curve for packing the tiles and plane sweep for fast pruning, like we do. In their first MapReduce phase they calculate the neighbors for each spatial object in the same tile. They also create "pending files" in HDFS containing potential neighbors from surrounding tiles, that will be processed in the next phase. Our work is similar, but we use two more phases for reasons explained in our algorithm analysis and we use grid and quad tree partitioning instead of the Z curve. Unfortunately the authors only compare their work to Oracle Spatial and not to other AKNN MapReduce solutions.

In [25], among other algorithms for SpatialHadoop and LocationSpark, the authors present the first AkNNQ algorithm in SpatialHadoop. This algorithm takes advantage of SpatialHadoop embedded capabilities, like Grid and Quadtree partitioning. Like the algorithms we present, the algorithm of [25] utilizes processing based on plane-sweep (1st improvement). Unlike the algorithm that we present, the algorithm of [25] handles 2D data only and repartitioning is required in dense areas, since partitioning provided by SpatialHadoop creates partitions close to the underlying filesystem block size and combinations of dense partitions affect the speed of the generation of results. The Quadtree based partitioning that we apply (3rd improvement) avoids such problems, since it is done under the direct control of our algorithm. Nevertheless, unlike [25] we utilize restructuring of phases to reduce network traffic (2nd improvement).

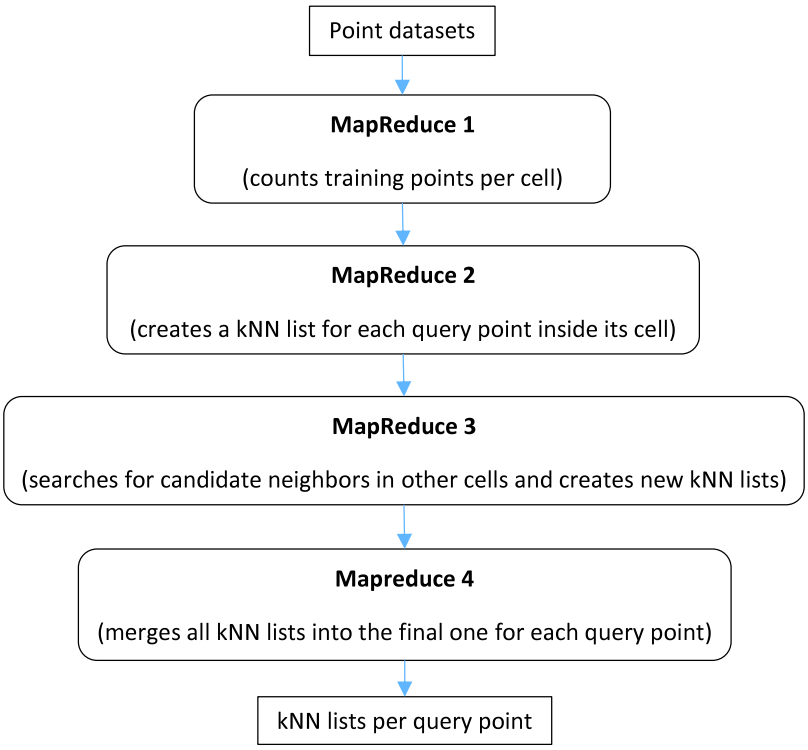

Fig. 1. The four phases overview flowchart.

3. The original algorithm

The algorithm presented in [2] is divided into five MapReduce phases. There are two datasets, named "Input" (or "Query") and "Training" which contain points in the form {point id, x, y}. The 1st phase computes the number of Training points per cell. The target space is decomposed into $N \times N$ equal sized cells in a grid (for the ease of exposition, we consider 2D space). The 2nd phase forms a preliminary k nearest neighbors list for each point of the Input dataset, but only for the Training points inside the same cell. The 3rd phase verifies whether or not the lists from Phase 2 are the final ones. It draws an expanding circle around each Input point to collect more Training points from neighboring cells, if needed. So, it creates additional kNN lists for each point. The 4th phase joins the multiple kNN lists into the final ones. The 5th and final phase does the classification of each Input point, based on the class of its neighbors. The first four phases will be analyzed in detail in the next subsections of this section. We will not implement the fifth phase in this paper, because we want to focus on algorithmic improvements on the first four phases that answer the AkNNQ. The fifth phase is easy to implement and orthogonal to the first four phases.

3.1. Phases

First, we present a diagrammatic overview of the algorithm of [2], without the classification step. Fig. 1 shows the overview of the whole process, which remains unaffected by the improvements we applied. The phases of this algorithm are as follows.

1st MapReduce phase. The target space is decomposed into $N \times N$ equal sized cells (Fig. 2). The Mapper takes points from the Training dataset, locates their cell and outputs the {cell id} as key and {1} as value, for each point. The Reducer then sums all 1's for each cell and outputs the {cell id} as key and the {number of Training points} that it contains as value.

2nd MapReduce phase. The Mappers (Fig. 3) put every point from both datasets to its appropriate cell (separate map functions for each dataset), and the Reducer calculates the distances and forms a preliminary k nearest neighbor list for each point of the Input dataset within the same cell. The list includes Training point id and its distance. The Mapper output key is {cell id} and the value is the list {point id, x, y, "I" or "T"},[1] where "I"/"T" stands for Input/Training. The Reducer output key is {Input point id} and the value is the list {x, y, cell id, kNN list}.

It is obvious that these lists may not be entirely correct.

- There may be cells with less than k (or even not any) Training points.

[1] In the flow diagrams it may be written as "ipoint" or "tpoint", meaning "Input point" or "Training point", accordingly.

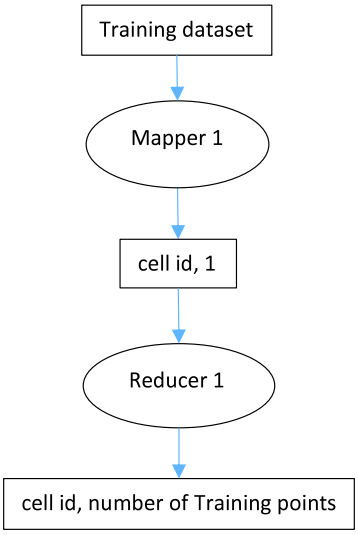

Fig. 2. 1st phase flowchart (original algorithm).

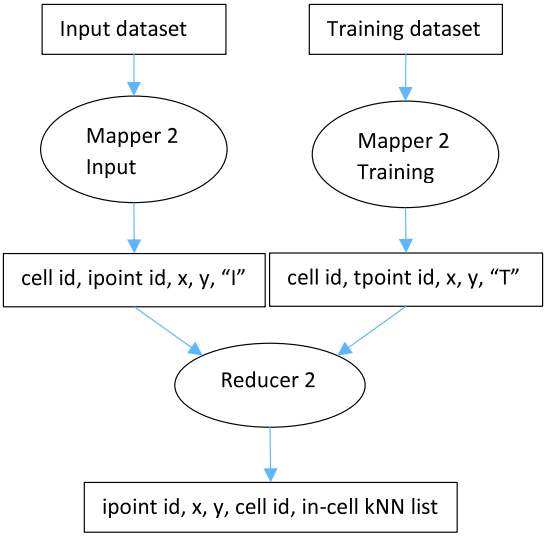

Fig. 3. 2nd phase flowchart (original algorithm).

- There may be Training points from neighboring cells that are closer to the Input point.

In Fig. 4, for $k = 3$, the kNN list for Input point q will be completed from its own cell. But there are better candidates that Phase 2 cannot "see" because they are not in the same cell.

3rd MapReduce phase. The above mentioned problem can be resolved by checking the neighboring cells.

- Case 1: kNN list is complete

We draw a circle, with its center on the Input point and its radius equal to the distance of the furthest neighbor. If the circle lies completely inside the cell (Fig. 5), then this Input point gets a "true" flag and goes to the next phase. This means that its kNN list is final and will not be modified. The Mapper output key is {cell id} and the value is {point id, x, y, kNN list, "true"}.

If the circle intersects one or more cells (Fig. 6) then the point gets a "false" flag and it goes to the next phase, along with each of the overlapped cells, so that additional checks will be made. In this case the Mapper output key is {overlapped cell id} and value {point id, x, y, kNN list, "false"}. The output is repeated for every overlapped cell. Note that we first check the overlapped cells for Training points. If there are none, then the Input point gets a "true" flag as well.

- Case 2: kNN list is not complete

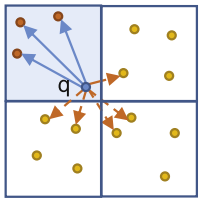

Fig. 4. 2nd phase possible misses.

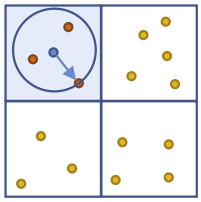

Fig. 5. 3rd phase "true" case (kNN list complete).

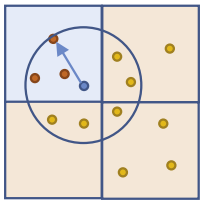

Fig. 6. 3rd phase "false" case (kNN list incomplete).

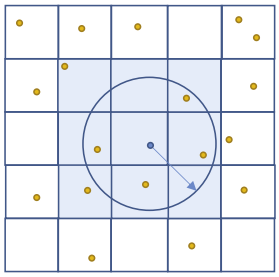

Fig. 7. 3rd phase "false" case (kNN list incomplete).

If the neighbor list is not full, then the circle around the Input point is gradually getting bigger (Fig. 7), until the cells it intersects have at least k points in total (this information comes from Phase 1 output, which is used as input for the Mapper). The output is the same as in the previous case. Suppose that the circle stopped increasing, as shown in Fig. 7, meaning that k total points are found (we are not sure where exactly; we only know in which cells they are located). Looking at Fig. 8, one may notice that some points in non-overlapped cell B may be closer to the Input point than the points in overlapped cell A. We want to be sure that the points in B will be included for further checking as well, so we give the radius a final boost equal to the diagonal of a cell. This guarantees that the final circle will intersect every cell that might contain closer neighbors than the cells intersected by the circle before boosting the radius.

The above process is accomplished by Mapper 3–1, included in Fig. 9. There is also another Mapper in Fig. 9 (Mapper 3–2) that is similar to "Mapper 2 Training" of Phase 2 and feeds the Reducer with Training points' coordinates and cell info.

Reducer 3 in Fig. 9 takes both Mappers output, sorted by cell id, and if there is a "true" flag, it just carries the line to the output. If the flag is "false", it creates a new neighbor list from the Training points of overlapped cells and merges it with the old one.

4th MapReduce phase. In this final phase (Fig. 10) the Mapper takes the output of Phase 3 and keeps only Input point id and the kNN list. The Reducer sorts the merged neighbor lists by distance and keeps only the first k neighbors to form the final kNN list.

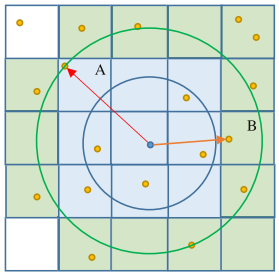

Fig. 8. 3rd phase "false" case (kNN list incomplete, radius boost).

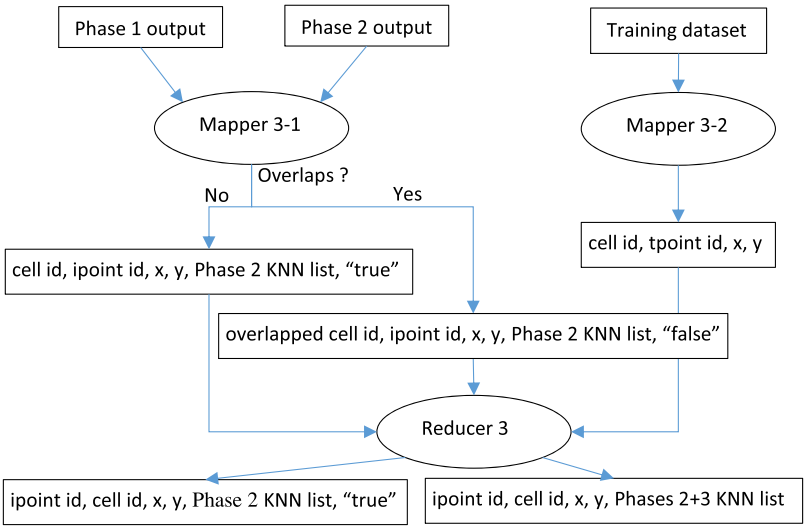

Fig. 9. 3rd phase flowchart (original algorithm).

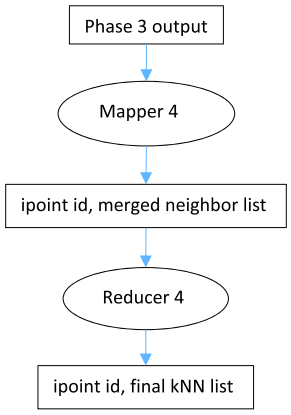

Fig. 10. 4th phase flowchart (original algorithm).

In the approach of [2], the Reducers in Phases 2 and 3 calculate distances by combining all Input points with all Training points within a cell (two nested loops). If the number of Input points in a cell is N_q and the Training points is N_t, then the number of distance calculations is $N_q \times N_t$, which may be in the order of billions. Considering that the Euclidean distance formula involves power and square root calculations, this can lead to major slowdowns and even failures (due to Hadoop timeout setting, in case of no output). Later we will follow another approach that prunes many points before calculating. We made our own implementation of the original algorithm and present it in pseudocode in a more detailed version than [2] in Figs. 11–14.

```
function MAP1 (Training dataset id, x, y)
        input n
        read x, y
        calculate cell_id from x, y, n
        output <cell_id, 1>
end

function REDUCE1 (MAP1 output)
        read cell_id, 1
        sum = 0
        for all c ∈ cell_id do
                sum += 1
        output <cell_id, sum>
end
```

Fig. 11. 1st phase pseudocode (original algorithm).

```
function MAP2_1 (Input dataset id, x, y)
        input n
        read x, y
        calculate cell_id from x, y, n
        output <point_id, x, y , "I">
end

function MAP2_2 (Training dataset id, x, y)
        input n
        read x, y
        calculate cell_id from x, y, n
        output <point_id, x, y , "T">
end

function REDUCE2 (MAP2_1 output, MAP2_2 output)
        input k;
        read point_id, x, y and put them in two lists (separate from "I" or "T")
        for all Input points in cell_id do
                L = maxHeap (k)
                for all Training points in cell_id do
                        calculate d = distance (ipoint, tpoint)
                        if size (L) < k then
                                L.push (tpoint_id, d)
                        else
                                dmax = L.getMax (distance)
                                if d < dmax then
                                        L.pop ( )
                                        L.push (tpoint_id, d)
                output < ipoint_id, x, y, cell_id, in-cell kNN list>
end
```

Fig. 12. 2nd phase pseudocode (original algorithm).

4. The new algorithm

So far we have presented the original algorithm [2]. In this section we will show in detail the improvements we have applied to it. A synopsis of the abbreviations we used in presenting our improvements follows:

- GD, or $Grid$: data partitioning used in the original algorithm,
- BF, or $BruteForce$: processing in Reducers used in the original algorithm,
- PS, or $PlaneSweep$: processing in Reducers used in the 1st improvement,
- LD, or $LessData$: Phase 3 that produces less output data (2nd improvement),
- QT, or $Quadtree$: 2D data partitioning used by Mappers (3rd improvement).
- OT, or $Octree$: 3D data partitioning used by Mappers (3rd improvement).

These abbreviations will also be used in combinations expressing the combined application of techniques, like $QT + PS + LD$ (application of Quadtree Mapper, plane-sweep Reducers and less output data from Phase 3).

4.1. 1st improvement — plane-sweep Reducers (PS)

The first improvement we made was to replace the brute force distance calculations in Reducers 2, 3 with a faster method, plane sweep, which is a computational geometry method that saves calculations by taking advantage of data projections on one of the axes [26].

```
function MAP3_1 (MAP1 output, REDUCE2 output)
        input n, k
        read ipoint_id, x, y, cell_id, kNN list
        get num = number of Training points in this cell_id from MAP1 output
        R = distance of furthest neighbor in kNN list
        if num >= k then
                overlaps = List { }
                overlaps.add (overlapping neighboring cells with circle (ipoint, R) )
                if overlaps = null then
                        output <cell_id, ipoint_id, x, y, KNN list, "true">
                else
                        for all ov_cell Є overlaps do
                                output <ov_cell, ipoint_id, x, y, KNN list, "false">
        else
                total_points = 0
                while (total_points < k) do
                        increase R
                        check for overlapping cells with circle (ipoint, R)
                        total_points += points from overlaps
                        overlaps.add (overlapping cells with circle (ipoint, R) )
                for all ov_cell Є overlaps do
                        output <ov_cell, ipoint_id, x, y, KNN list, "false">
end

function MAP3_2 (Training point id, x, y)
        (same as MAP2_2 without "T" in the output)
end

function REDUCER3 (MAP3_1 output, MAP3_2 output)
        (reads both input streams and calculates a new neighbor list, as in REDUCE2,
        then merges it with the old kNN list)
        output < ipoint_id, cell_id, x, y, merged kNN list>
        or
        output < ipoint_id, cell_id, x, y, kNN list, "true">
end
```

Fig. 13. 3rd phase pseudocode (original algorithm).

```
function MAP4 (REDUCER3 output)
        read ipoint_id, neighbors list
        output < ipoint_id, neighbors list>
end

function REDUCE4 (MAP4 output)
        input k
        read ipoint_id, neighbors list
        sort neighbors list by distance ascending
        output < ipoint_id, kNN list>
end
```

Fig. 14. 4th phase pseudocode (original algorithm).

First (Fig. 15) we sort the Training points by the first axis (let x) in ascending order, in each cell. Then, for every Input point, we find its place among the Training points by interpolating its x coordinate. Next, we create a max heap with k places and begin two scans, one to the right and another one to the left of the Input point, by x ascending or descending, accordingly. The first k Training points along with their Euclidean distances are inserted into the heap. After this (the heap holds k points), we check the x-distance between the Input point and a candidate Training point. If it is smaller than the maximum distance in the heap, then we calculate the distance between the Input point and the candidate Training point and if it is also smaller, we pop the point in the heap root and push this Training point. If this x-distance in a scan direction gets bigger than the maximum distance in the heap, we stop checking the rest of the Training points in this direction (since, the Euclidean distance between two points is not smaller than the x-distance between them, meaning that it is not possible to find closer neighbors in this scan direction than the neighbor at the top of the heap), thus saving a lot of unnecessary calculations. Plane-sweep method really shines when there are many points inside a cell.

The modified Reducer 2 using plane sweep is illustrated in Fig. 16. Reducer 3 is similar.

4.2. 2nd improvement — less output data from phase 3 (LD)

During experimentation with the above code, both in plane sweep and brute force, we noticed the huge size of the output file in Phase 3 (4+ times bigger than the other phases output). This phase was also the slowest to complete because of the network traffic. We had to reduce the output file size somehow.

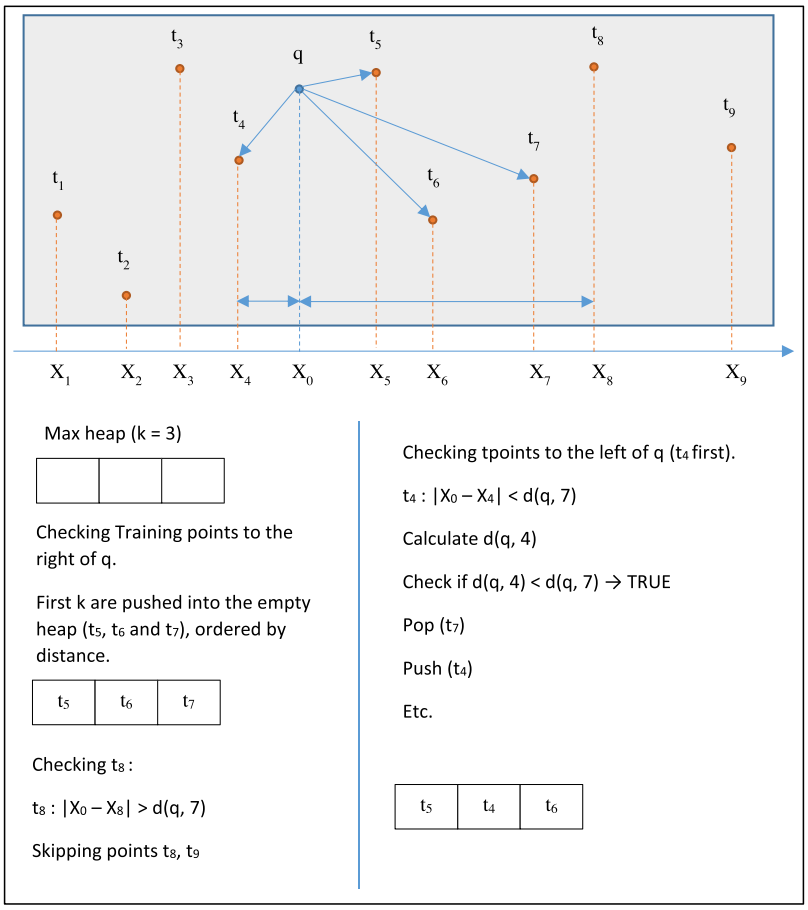

Fig. 15. Plane-sweep technique illustration.

In the original algorithm, every Input point coming out from Mapper 3–1 has a long tail (x, y, Phase 2 kNN list, "true" or "false"), the biggest part of which is the kNN list. This list is just passed through here to end up in Phase 4. It is not used in Phase 3 at all, yet it adds a significant load to the network, especially for large k values. We decided to cut the kNN list from the tail and inject it directly into Phase 4, where the lists will be merged. We also cut the x and y coordinates if the flag is "true".

The improved Phases 3 & 4 are depicted in Figs. 17 and 18 (Phases 1 & 2 remain unchanged). If you compare them with Figs. 9 and 10 you can see that Phase 2 kNN list is now completely absent from the output of Mapper 3–1 and Reducer 3, as well as the coordinates of the Input point in "true" case. Phase 2 kNN list is injected into Mapper 4 (Fig. 18) with an added "false" flag.

4.3. 3rd improvement — Quadtree Mappers (*QT*)

Both previous improvements along with the original version of the algorithm have one thing in common: they are applied to equal sized cells on a grid. This space decomposition method has a serious disadvantage. The datasets may be highly skewed, which means that in some areas there will be a very large number of points, while others will be completely empty. Consider the analogy of population distribution among various geographic areas including cities, deserts and oceans. This algorithm makes calculations cell-wise, which means that if a cell contains millions of points from the two datasets, the calculations will be in the order of billions or more, considerably slowing down the whole process. One solution is to cut space into many and very small cells, but this is a performance killer, as shown later in the experiments section.

A better method for space decomposition is the use of a Quadtree-like (regular) decomposition that divides space recursively into four equal quadrants until no quadrant is overpopulated, ending up to non-equal cells (Fig. 19). We have developed an algorithm that implements this decomposition by taking into account the desired maximum capacity of points per cell. If a cell's contained points are more than the allowed capacity, this cell is divided again and again. This way, we can control the total number of points per cell and consequently the maximum number of calculations per cell. This improvement is denoted by *QT*, or *Quadtree*.

Before starting Phase 1, a sample from the Training dataset is taken (which is read directly from HDFS) and a Quadtree is constructed locally, on a single machine. This sampled tree is stored in HDFS so that all phases can access it. The rest of the algorithm

```
function REDUCE2 (MAP2_1 output, MAP2_2 output)
        input k
        read point_id, x, y and put them in two lists (by "I" or "T")
        sort tpoints list by x ascending
        for all Input points in cell_id do
                interpolate Input point into sorted Training points list
                L = maxHeap (k)
                for all Training points in cell_id to the right of Input point do
                        if size (L) < k then
                                calculate d = distance (ipoint, tpoint)
                                L.push (tpoint_id, d)
                        else
                                dmax = L.getMax (distance)
                                if |xi – xt|> dmax then
                                        break
                                else
                                        calculate d = distance (ipoint, tpoint)
                                        if d < dmax then
                                            L.pop ( )
                                            L.push (tpoint_id, d)
                for all Training points in cell_id to the left of Input point do
                        (same as above)
                output <ipoint_id, x, y, cell_id, in-cell kNN list>
end
```

Fig. 16. Plane-sweep Reducer 2.

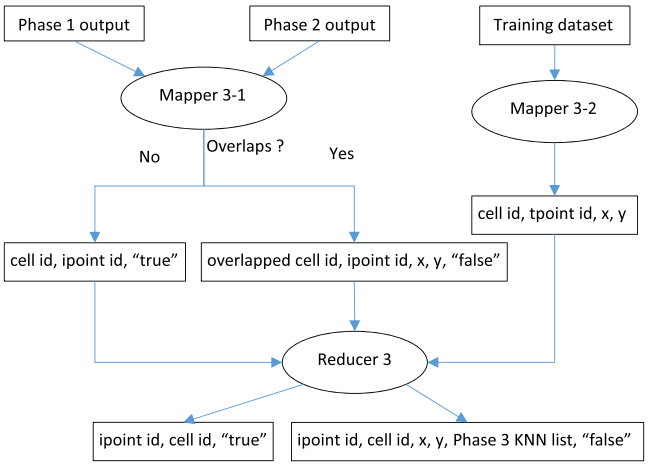

Fig. 17. 2nd improvement: Phase 3 with less output data.

remains pretty much the same. Therefore, we were able to implement BruteForce, PlaneSweep, and LessData in combination with Quadtree.

A clarification is needed regarding Mapper 3–1 radius boost that is shown in Figs. 7 and 8 for the original algorithm. Grid has equal sized cells, so radius boost is the same for every cell. Quadtree however has vastly different sized cells and a fixed radius boost may not work as expected. For example, when an Input point lies into a big cell, its radius may be as long as the side of a root-child quadrant and this cell will cover a lot of unnecessary space, flooding the output of the phases of the algorithm with data. On the contrary, when this cell is very small, we may miss several candidate neighbors due to a small radius boost. For this reason, first, we calculate an initial radius, based on the current cell size and k and increase it until it encompasses at least k Training points. Then, we calculate the distance of the most distant (to the query point) vertex among all intersected cells and use it as the final radius. By this way, no solution is lost. The method is shown in Fig. 19 (the vector heading to the top-left depicts the final radius calculated).

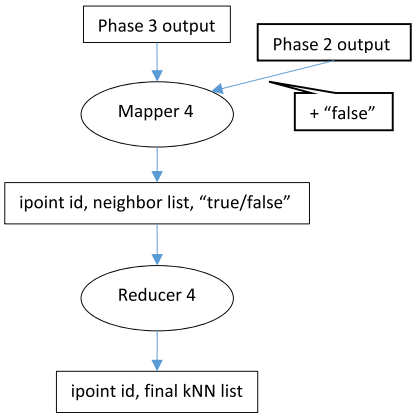

Fig. 18. 2nd improvement: Phase 4.

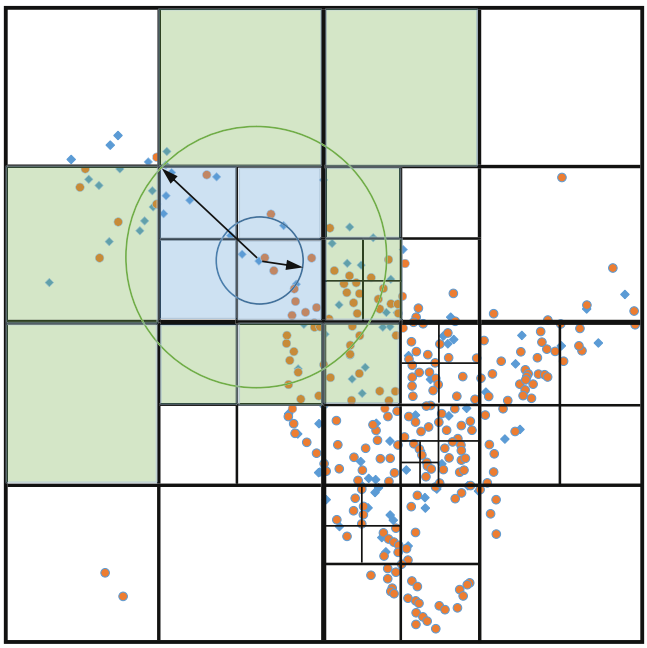

Fig. 19. Quadtree space decomposition and radius boost.

5. Experimental evaluation in 2D

In this section we will present the experiments we conducted for every improvement described above, as well as for their combinations. The effect of the number of neighbors, in relation to the granularity of Grid space decomposition and Quadtree node-capacity will be studied to find the best setting for each method.

We have set up a cluster of 9 virtual machines (1 NameNode and 8 DataNodes) running Ubuntu Linux 16.04 64-bit. Each machine is equipped with a Xeon quad core at 2.1 GHz and 16 GB RAM, connected to a 10 Gbit/sec network. Hadoop version is 2.8.0, the replication factor is set to 1, HDFS chunk size is 128 MB and the virtual memory for each Map and Reduce task is set to 4 GB.

We used two real world datasets from OpenStreetMap [12], one consisting of 11.504.035 points (the coordinates of parks around the world) that weighs 373 MB and another consisting of 11.473.662 points (this is a 10% subset of a dataset that contains the coordinates of buildings around the world) that weighs 383 MB. The first one is used as Input and the second one as Training dataset.

We only use a 10% subset of the buildings dataset to make the execution of more experiments possible and the study of alternative algorithms and parameter values more detailed. However, we also perform and present limited experiments with the complete buildings dataset (that weighs about 4 GB and contains over 110 million points) to verify our findings at larger scale. We did not use far bigger datasets mainly because these ones and other datasets of these sizes are the most often used in the literature, so

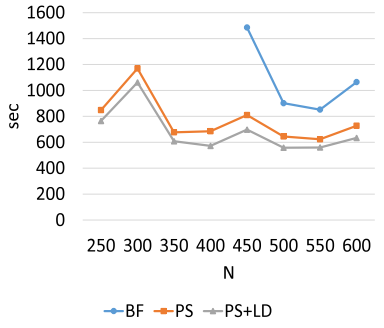

Fig. 20. GD: exec. time of BF, PS and PS+LD, for $k = 5$.

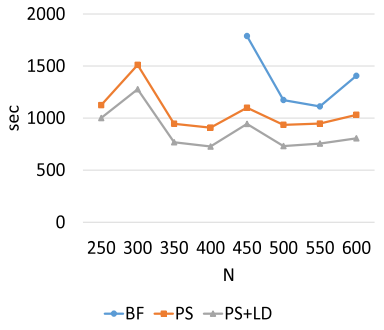

Fig. 21. GD: exec. time of BF, PS and PS+LD, for $k = 10$.

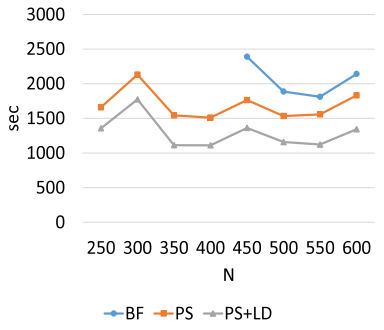

Fig. 22. GD: exec. time of BF, PS and PS+LD, for $k = 20$.

the results are immediately comparable. Moreover, the size of these datasets is enough to highlight the performance gains of the proposed improvements.

5.1. PlaneSweep

In this experiment, we used Grid partitioning and studied the performance of the original algorithm against two PlaneSweep versions, for $k = 5$, 10 and 20. In Figs. 20–22, we show the execution time of BruteForce (original algorithm), PlaneSweep and PlaneSweep+LessData, as a function of N ($N = 250$ to 600 in steps of 50 and space is decomposed in $N \times N$ cells).

As shown in these figures, the results for Bruteforce start at $N = 450$, since this algorithm did not produce results at all for N smaller than 250 (Hadoop killed the processes, after there was no Phase 2 output for 600 s, which is its default setting). For N larger than 600, the execution time rose significantly, since Phase 3 output got very big, because of too many cells.

It is also obvious that PlaneSweep always finishes before BruteForce, because it prunes many points. Therefore, we will only use the PlaneSweep method in the rest of the experiments. Performance improvement of PlaneSweep over BruteForce, when $N = 500$, is 28% for $k = 5$, 20% for $k = 10$ and 19% for $k = 20$. As k grows, performance gain is dropping, because the dominant factor becomes the increasing size of output data that floods the network. This is where the combined application of PlaneSweep and LessData shows its value, since the size of the output of Phase 3 is reduced. This effect will be studied in more detail in the next subsection.

Table 1
Grid Training points distribution.

N	Number of cells containing Training points					
	<50k	50k–100k	100k–200k	200k–300k	Total	Non empty
250	5 384	34	22	2	62 500	5442 (8.71%)
300	6 918	41	13	1	90 000	6973 (7.75%)
350	8 546	45	8	0	122 500	8599 (7.02%)
400	10 391	30	3	1	160 000	10 425 (6.52%)
450	11 749	24	1	1	202 500	11 775 (5.81%)
500	13 358	15	3	0	250 000	13 376 (5.35%)
550	15 048	16	1	0	302 500	15 065 (4.98%)
600	16 605	14	1	0	360 000	16 620 (4.62%)

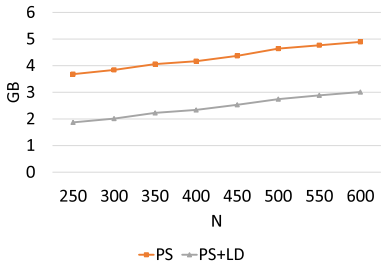

Fig. 23. GD: Phase 3 size of PS and PS+LD, for $k = 5$.

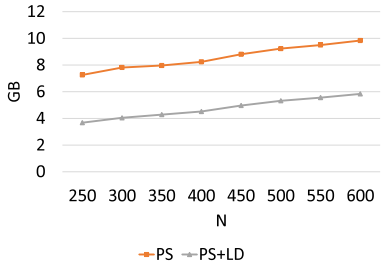

Fig. 24. GD: Phase 3 size of PS and PS+LD, for $k = 10$.

One of Grid's disadvantages is the percentage of empty space, which is the number of cells that contain few, or no points and need not be further divided in the first place. Another disadvantage is that several cells with excessive number of points may be created. To study the bad performance of Grid, we have analyzed the output data of Phase 1 (Training points per cell distribution) and present the results in Table 1. For several values of N, we depict the number of cells containing [1..50k), [50k..100k), [100k..200k) and [200k–300k) points, the total number of cells (including the empty ones) and the number/percentage of non-empty cells. The conclusion arising from this table is that 90%–95% of the cells are empty and the remaining 5%–10% contain hundreds of thousands of points which take part in distance and other calculations (in Phase 2, mainly). This is the reason that for N's below 250 BruteForce cannot finish: if there are many cells with more than 300k points, the processes handling these cells will probably fail (unless the default Hadoop timeout parameter is modified).

5.2. PlaneSweep+LessData

Figs. 20–22 also depict the PlaneSweep+LessData improvement that prunes a lot of Phase 3's output data, thus saving network traffic. The performance difference becomes clearer as k grows. As Figs. 20–22 show, when $N = 500$, performance gain of PlaneSweep+LessData, for $k = 5$, against PlaneSweep is 13% and against BruteForce is 38%, for $k = 10$, against PlaneSweep is 22% and against BruteForce is 38%, and, for $k = 20$, against PlaneSweep is 24% and against BruteForce is 38%. So there is a constant $\approx 40\%$ improvement against the original algorithm (BruteForce).

To study the reasons for this effect, in the next set of graphs (Figs. 23–25) we depict the Phase 3 output-data size of PlaneSweep and PlaneSweep+LessData, as a function of N (for the same N range as in the previous figures). The data-size difference between the two methods is about 2 GB for $k = 5$, 4 GB for $k = 10$ and 8 GB for $k = 20$. It is growing linearly with k, as expected, and remains constant in relation to N. This is a 40%–50% of reduction.

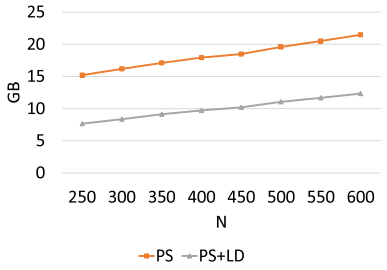

Fig. 25. GD: Phase 3 size of PS and PS+LD, for $k = 20$.

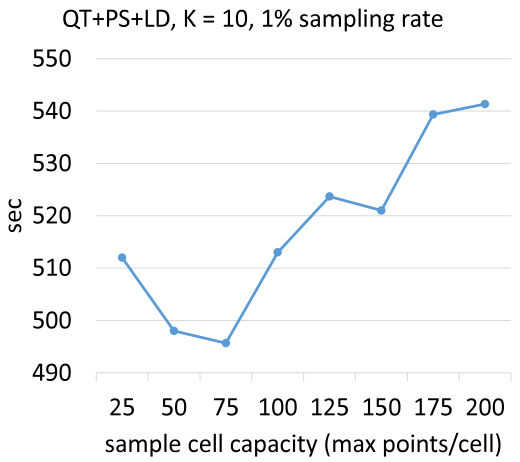

Fig. 26. Exec. time of QT using PS+LD, to determine the best cell capacity (1% sampling rate).

5.3. Quadtree

In this section, we first seek for a good cell capacity for Quadtree creation (Section 5.3.1), before we compare the performance of algorithmic variations with partitioning based on Quadtree against Grid, using PlaneSweep (Section 5.3.2) and using PlaneSweep+LessData (Section 5.3.3).

5.3.1. Tuning cell capacity

To fine tune the cell capacity (how many points from the Training dataset a cell will contain) for Quadtree creation, we keep $k = 10$ (since, results were not significantly affected by k) and sampling rate equal to 1% (since this rate proved adequate for effective partitioning). In the graph of Fig. 26 we present the execution time of the Quadtree+PlaneSweep+LessData algorithm for various sample cell capacities. Too many points (bigger cells) mean a lot of calculations within a cell, which affects performance negatively. The best capacity seems to be about 75 points. For lower cell capacity than this, the performance drops (few points per cell mean many small cells which lead to a lot of output data). Maximum cell capacity of 75 points for the 1% sample dataset corresponds to maximum cell capacity of 7.500 points for the complete dataset. Local sample-tree creation time remained constant at about 30 s.

It is generally difficult to obtain a "universal" capacity for all datasets, because this depends on skewness and subsequently the number of cells created. The optimal capacity should balance the total number of cells (the less the better) with the average number of points per cell (too many means lots of calculations). Later in the scalability experiments we will show that the difference between optimized and non-optimized trees is not that great, regarding the change of capacity.

We claim that Quadtree really "cuts" space where necessary. There is no wasted space here, unlike Grid, and there are much fewer cells in total (compared to Grid) which all contain a number of Training points less than or equal to the maximum capacity. Table 2 (where for several capacities, we depict the number of cells containing <50k and ≥ 50k points, the total number of cells – including the empty ones – and the number/percentage of non-empty cells) justifies this claim. The best Grid for $k = 10$ resulted from $N = 400$, which means $400 \times 400 = 160{,}000$ cells, while the best Quadtree that resulted at sample capacity of 75 creates only 4507 cells. Its Grid analog should be $N = \sqrt{4507} \approx 67$, only.

5.3.2. Quadtree vs. Grid, using PlaneSweep

In this subsection, we will compare the performance resulting from Quadtree against Grid partitioning, using PlaneSweep, in both cases. In Figs. 27–29, we depict execution time of the three top Grids and Quadtree, for $k = 5$, 10 and 20, respectively. The

Table 2
Quadtree Training points distribution.

Capacity	Number of cells containing Training points			
	<50k	≥50k	Total	Non empty
25	12 625	0	13 414	12 625 (94.12%)
50	6 449	0	6 871	6449 (93.86%)
75	4 195	0	4 507	4195 (93.08%)
100	3 246	0	3 484	3246 (93.17%)
125	2 582	0	2 776	2582 (93.01%)
150	2 129	0	2 284	2129 (93.21%)
175	1 835	0	1 969	1835 (93.19%)
200	1 623	0	1 732	1623 (93.71%)

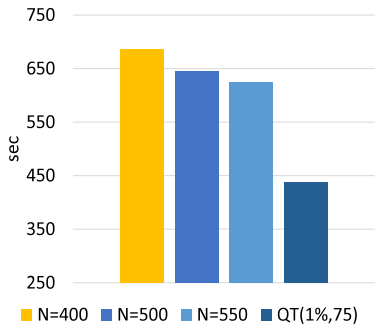

Fig. 27. Exec. time of QT and 3 versions of GD, using PS, for $k = 5$.

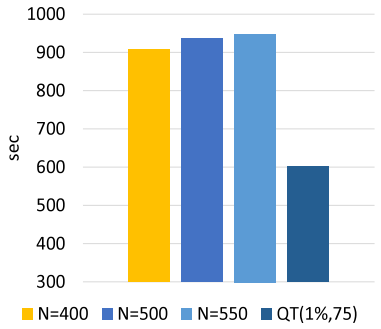

Fig. 28. Exec. time of QT and 3 versions of GD, using PS, for $k = 10$.

bar-charts of these figures show a clear win for the Quadtree by about 30%. We will analyze the performance gains in the next subsection.

5.3.3. Quadtree vs. Grid, using PlaneSweep+LessData

The set of graphs in Figs. 30–32 present an analogous comparison to Figs. 27–29, using PlaneSweep+LessData, instead of only PlaneSweep. Quadtree again wins in all cases by 30%.

The explanation behind these numbers lies in the graphs of Figs. 33–35, where we depict the output sizes of Phases 2 and 3 for Grid with $N = 400$ and Quadtree with capacity = 75 (both using PlaneSweep+LessData), for the usual three k values. While Grid has a slightly smaller Phase 2 output size, Quadtree's Phase 3 output is impressively smaller than Grid's. For $k = 20$ there is almost 7.5 GB difference, or 77% smaller size. This happens because Grid creates many small cells in sparse areas that may contain very few points each. Eventually, all of them are carried into the output (this is how the algorithm works). Quadtree creates only few large cells in sparse areas, so the output is quite smaller.

Per phase time performance (execution time) is shown in Figs. 36–38, for the same algorithmic settings of Figs. 33–35. Phase 1 was deliberately left out, because it always only takes 40–50 s, regardless of partitioning method, or k.

Quadtree dominates Grid in all phases, even in Phase 2, where its output was slightly bigger. This can be explained by the uniform points distribution across its cells, compared to the Grid. Every cell in Phase 2 has a number of points that is not greatly different from the others, so the calculations are more balanced among nodes and, when executed in parallel, finish faster. We saw in Table 1 that some Grid cells may have hundreds of thousands of points, which leads to major slowdowns.

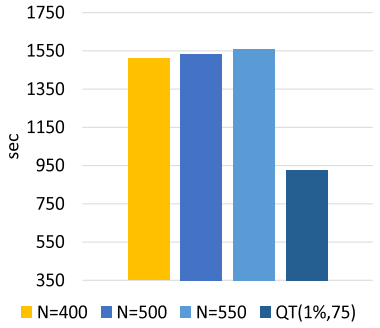

Fig. 29. Exec. time of QT and 3 versions of GD, using PS, for $k = 20$.

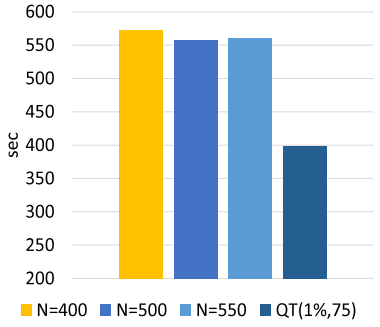

Fig. 30. Exec. time of QT and 3 versions of GD, using PS+LD, for $k = 5$.

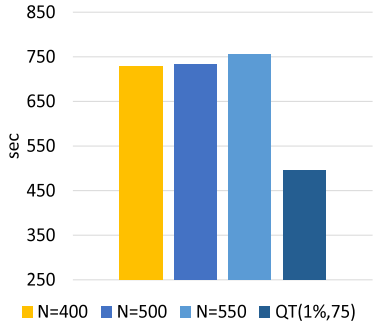

Fig. 31. Exec. time of QT and 3 versions of GD, using PS+LD, for $k = 10$.

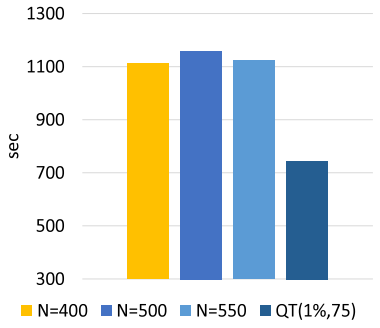

Fig. 32. Exec. time of QT and 3 versions of GD, using PS+LD, for $k = 20$.

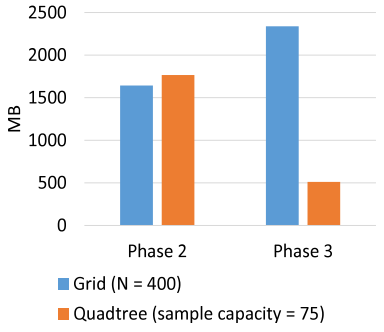

Fig. 33. Phases 2 & 3 size of QT and GD, using PS+LD, for $k = 5$.

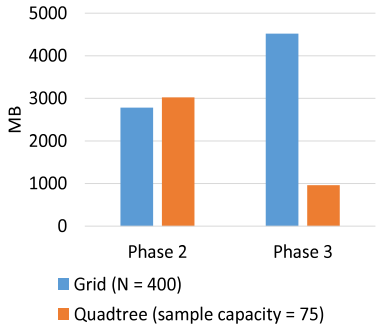

Fig. 34. Phases 2 & 3 size of QT and GD, using PS+LD, for $k = 10$.

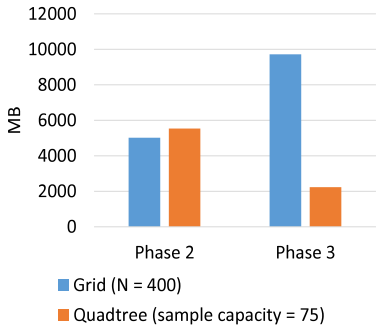

Fig. 35. Phases 2 & 3 size of QT and GD, using PS+LD, for $k = 20$.

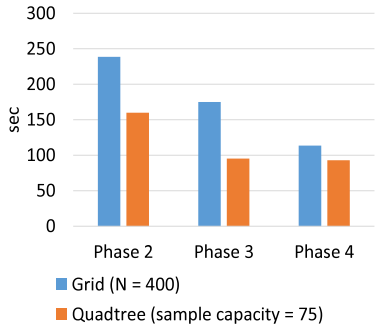

Fig. 36. Phases 2–4 exec. time of QT and GD, using PS+LD, for $k = 5$.

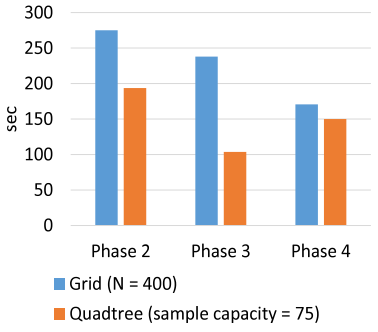

Fig. 37. Phases 2–4 exec. time of QT and GD, using PS+LD, for $k = 10$.

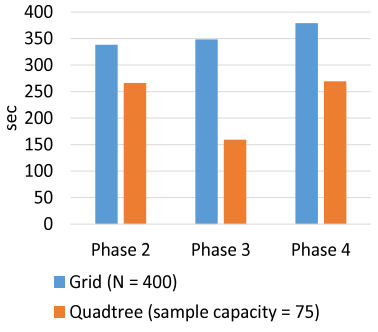

Fig. 38. Phases 2–4 exec. time of QT and GD, using PS+LD, for $k = 20$.

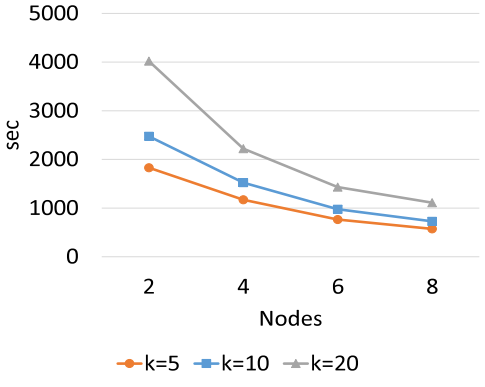

Fig. 39. Exec. time of GD+PS+LD, N = 400.

5.4. Scalability experiments in 2D

Our final experiments for 2D regard performance improvement in relation to the number of computing nodes and an order of magnitude bigger dataset.

5.4.1. Computing-nodes scalability in 2D

We have tested the execution time of the best Grid and Quadtree versions (Grid with $N = 400$ and Quadtree with capacity = 75, both using PlaneSweep+LessData) in 2, 4, 6 and 8 nodes, for all three k values. Results are depicted in Figs. 39–40.

Performance improvement is almost linear in relation to the number of nodes. For $k = 20$, in Grid, there is a 45% performance increase from 2→4 nodes and another 50% increase from 4→8 nodes. This is almost linear. In Quadtree, the 2→4 nodes performance increase is 39% and from 4→8 nodes it is 46%.

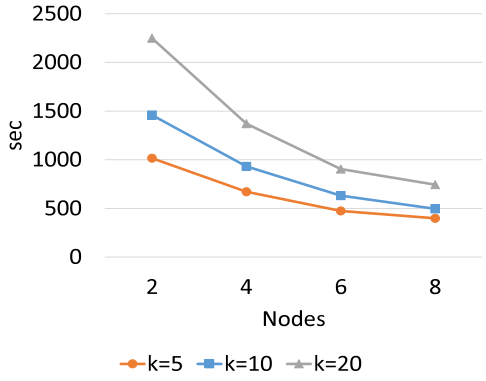

Fig. 40. Exec. time of QT+PS+LD, 1% sampling rate, capacity = 75.

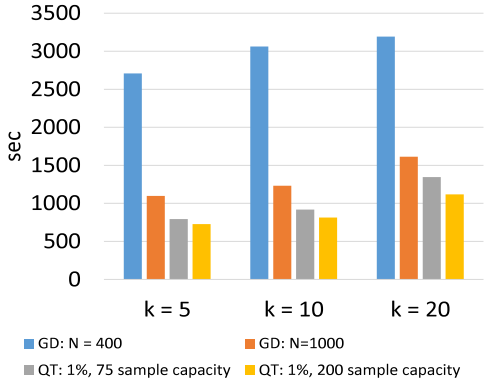

Fig. 41. Exec. time of QT and GD, using PS+LD.

5.4.2. Dataset scalability in 2D

We will now test how a bigger (by an order of magnitude) dataset affects performance. The new Training dataset will be the full buildings, consisting of 114,736,611 points and weighing 3.8 GB, which is of course tenfold its 10% subset we used until now. The Input dataset remains the same.

Both Grid and Quadtree were tested again to find the best performers. Now, the best Quadtree resulted from a capacity of 200 (using capacity 75 creates three times more cells, thus gives a much bigger data output, see Table 4) and the best Grid from $N = 1000$. We will test them against each other and compare them with the previous best performers (Grid with $N = 400$ and Quadtree with capacity = 75, both using PlaneSweep+LessData),

The results are shown in Figs. 41–42. The first one shows the total execution time, for all three k values, while the second one shows the distribution of time per phase, for $k = 20$ (for the sake of the clarity of this figure only one k value was used, although, the results were analogous for the other ones).

For $k = 20$, we can see that the new best Quadtree performs about 17% better than the old one (which is best for the smaller dataset size). But the difference between the two Grids is triple (50%). Even the old best Quadtree beats the new best Grid by 17%, while the new best Quadtree has a 30% performance gain over the new best Grid. Fig. 42 shows the huge Phase 2 time of the old best Grid, which is explained by the Training points distribution in Table 3. In this table, for $N = 400$ and 800 and for a partition of the cardinalities ranges of Training points per cell, we depict the number of cells of Grid containing each cardinality range, the total number of cells (including the empty ones) and the number/percentage of non-empty cells. It is obvious that for $N = 400$, there are many cells with hundreds of thousands of points.

Quadtree achieves a better points distribution (even the old best one). In Table 4, for capacity = 75 and 200, we depict the number of cells containing <50k and ≥ 50k Training points, the total number of cells (including the empty ones) and the number/percentage of non-empty cells. Note that all non-empty cells contain a limited number of Training points.

6. Experimental evaluation in 3D

We have artificially expanded our 2D datasets to the 3 dimensions by randomizing z. In our 3D experiments, we will apply the combination PlaneSweep+LessData to both partitioning methods, since this combination has clearly achieved the best results in our experiments so far.

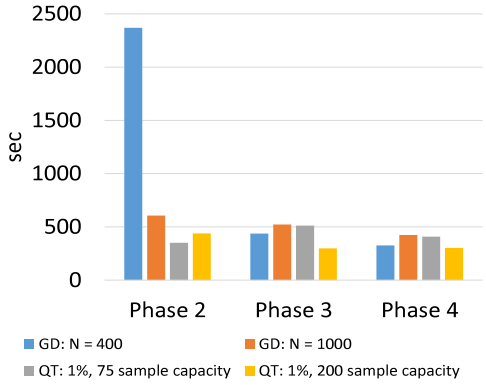

Fig. 42. Phases 2–4 exec. time of QT and GD, using PS+LD, $k = 20$.

Table 3
Grid Training points distribution.

Number of Training points	Number of cells containing Training points	
	N = 400	N = 1000
>500k	35	4
400k–500k	24	5
300k–400k	42	6
200k–300k	52	22
100k–200k	123	165
50k–100k	158	387
<50k	11 701	39 292
Total	160 000	1 000 000
Non empty	12 135 (7.58%)	39 881 (3.99%)

Table 4
Quadtree Training points distribution.

Capacity	Number of cells containing Training points			
	<50k	≥50k	Total	Non empty
75	42 439	0	44 749	42 439 (94.84%)
200	16 032	0	16 816	16 032 (95.34%)

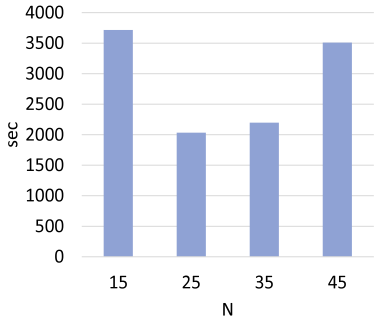

Fig. 43. Exec. time of 3D GD, using PS+LD, $k = 10$.

6.1. 3D Grid

In Fig. 43, we depict the execution time of 3D Grid using PlaneSweep+LessData for several values of N and $k = 10$ (results were analogous for other values of k). The best performance is achieved for $N = 25$.

6.2. Octree

In this subsection, we are going to seek for a good capacity for the 3D version of the Quadtree (the Octree). The Octree (OT) works in an analogous to the Quadtree way: the more dense 3D space is, the deeper it cuts. It divides the root cube in all three

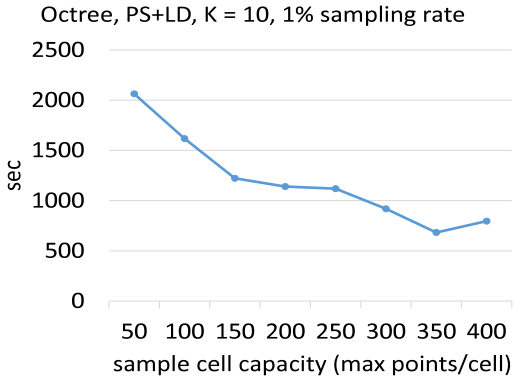

Fig. 44. Exec. time of OT using PS+LD, to determine the best cell capacity (1% sampling rate).

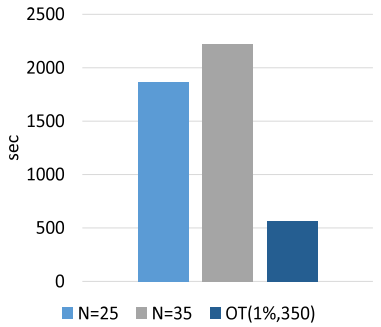

Fig. 45. Exec. time of OT and 2 versions of GD, using PS+LD, for $k = 5$.

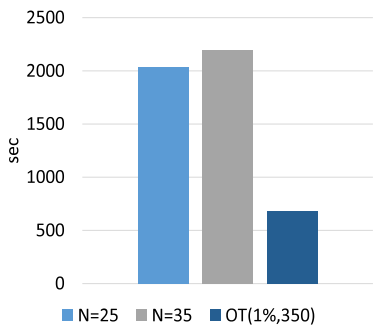

Fig. 46. Exec. time of OT and 2 versions of GD, using PS+LD, for $k = 10$.

dimensions recursively, thus cutting it in eight equal sub-cubes, until the capacity requirements for each sub-cube are met. Once again, a sample-based Octree is constructed locally from 1% sampling of the Training dataset and by defining a maximum sample capacity. We tested various capacities to find the best performing tree (that uses PlaneSweep+LessData), for $k = 10$. The results (execution tome as a function of cell capacity) are shown in Fig. 44. The best capacity in 3D is 350 points per sampled-data cell, which means 35,000 points (max) per complete-data cell. It is quite bigger than 2D's 75 points per sampled-data cell (7500 per complete-data cell). An explanation to this is the number of cells created: the new optimized capacity creates 6–7 times less cells than the old one, but it also greatly depends on both datasets distribution.

6.3. Octree vs. 3D Grid, using PlaneSweep+LessData

In Figs. 45–47, we depict execution time of the two top Grids and Octree, for all k values. The results and conclusions are pretty much the same as in 2D, while the differences are enlarged now, in favor of the Octree. 3D space is more demanding in calculations and good partitioning plays a more significant role. Octree is constantly 65%–70% better than the best 3D Grid.

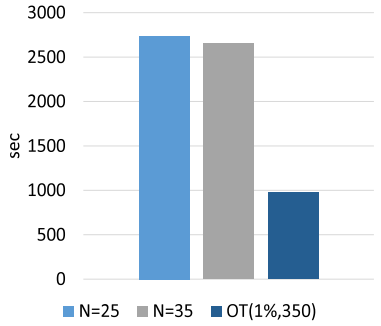

Fig. 47. Exec. time of OT and 2 versions of GD, using PS+LD, for $k = 20$.

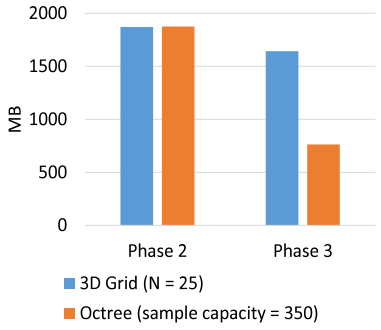

Fig. 48. Phases 2 & 3 size of OT and GD, using PS+LD, for $k = 5$.

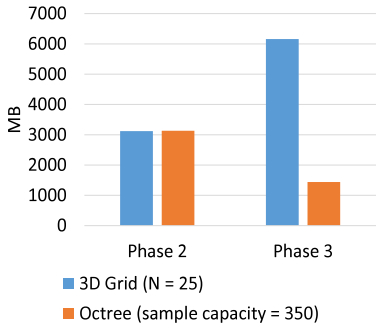

Fig. 49. Phases 2 & 3 size of OT and GD, using PS+LD, for $k = 10$.

The explanation behind these numbers lies in the graphs of Figs. 48–50, where we depict the output sizes of Phases 2 and 3 for Grid with $N = 25$ and Octree with capacity = 350 (both using PlaneSweep+LessData), for the usual three k values. While the Phase 2 output size of the two algorithms is almost the same, Octree's Phase 3 output is impressively smaller than Grid's.

Per phase time performance (execution time) is shown in Figs. 51–53, for the same algorithmic settings of Figs. 48–50. Phase 1 is not presented, since its execution time is limited. Octree is the winner in all phases, even in Phase 2. This can be explained (in 3D too) by the uniform points distribution among its cells, compared to the Grid.

6.4. Scalability experiments in 3D

Our final experiments for 3D study performance improvement in relation to the number of computing nodes and a significantly bigger dataset.

6.4.1. Computing-nodes scalability in 3D

The execution times of the best 3D Grid and Octree versions were tested in 2, 4, 6 and 8 nodes, for all three k values. Results are depicted in Figs. 54–55.

For $k = 20$, 3D Grid performance is improved by 50% from 2→4 nodes and by 41% from 4→8 nodes. In the Octree graph, performance is improved by 45% from 2→4 nodes and by 43% from 4→8 nodes.

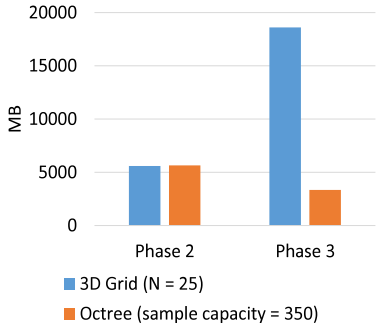

Fig. 50. Phases 2 & 3 size of OT and GD, using PS+LD, for $k = 20$.

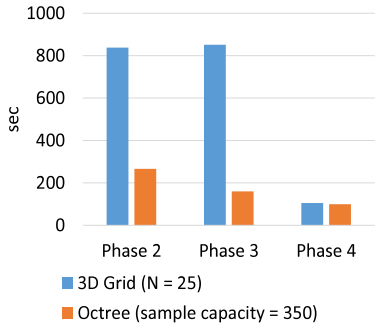

Fig. 51. Phases 2–4 exec. time of OT and GD, using PS+LD, for $k = 5$.

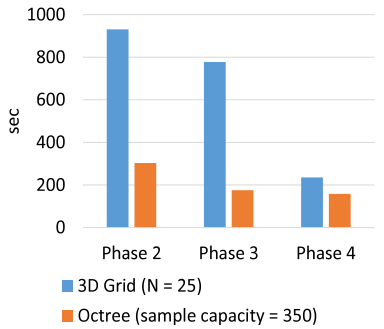

Fig. 52. Phases 2–4 exec. time of OT and GD, using PS+LD, for $k = 10$.

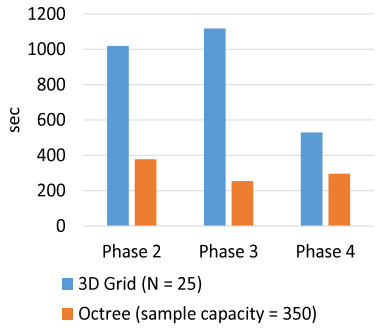

Fig. 53. Phases 2–4 exec. time of OT and GD, using PS+LD, for $k = 20$.

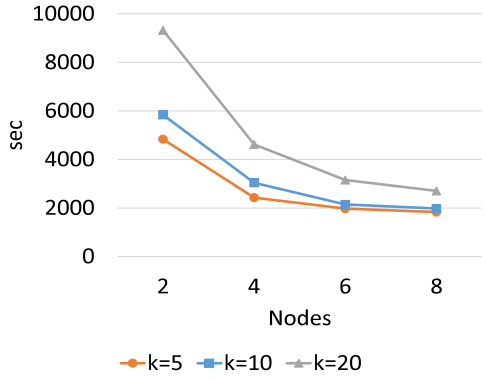

Fig. 54. 3D GD: PS+LD, N = 25.

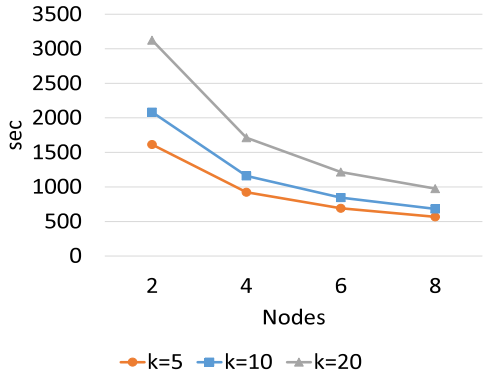

Fig. 55. OT: PS+LD, 1% sampling rate, sample cell capacity = 350.

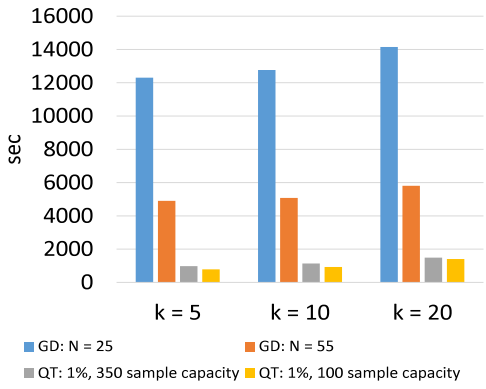

Fig. 56. Exec. time of OT and 3D GD, using PS+LD.

6.4.2. Dataset scalability in 3D

Similarly to 2D, the new, bigger Training dataset will be the 3D version of the full buildings dataset. We made some experiments to determine the best Grid and Octree for this new dataset. The new best Grid has $N = 55$ and the new best Octree has sample-dataset capacity 100. The results of the performance comparison of the new and old best Octree and the new and old best Grid are shown in Figs. 56–57. The first one shows the total execution time, for all three k values, while the second one shows the distribution of time per phase, for $k = 20$ (again, for the sake of the clarity of the figure, only one k value was used, although, the results were analogous for the other ones).

The new best Octree performs 5%–18% better than the old one. But the difference between the two Grids is once again very large (60%). Even the old best Octree beats the new best Grid by a vast 80%, while the new best Octree gains another 5%–10%. Fig. 57 shows the huge Phase 2 time of the old best Grid, which is explained by the Training points distribution in Table 5 (which is analogous to Table 3). There are many cells with hundreds of thousands of points, causing delay to the new best Grid Phase 3.

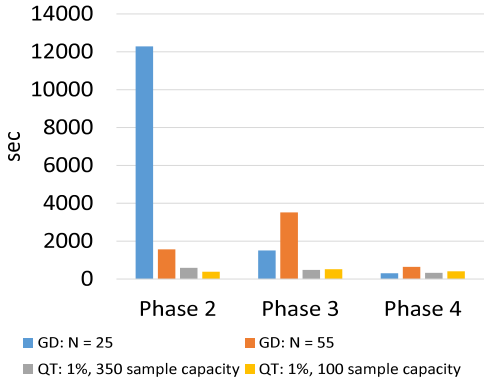

Fig. 57. Phases 2–4 exec. time of QT and 3D GD, using PS+LD, $k = 20$.

Table 5
3D Grid Training points distribution.

Number of Training points	Number of cells containing Training points	
	N = 25	N = 55
>500k	50	0
400k–500k	0	0
300k–400k	0	55
200k–300k	25	55
100k–200k	75	220
50k–100k	50	110
<50k	4358	29 639
Total	15 625	166 375
Non empty	4558 (29.17%)	30 079 (18.08%)

Table 6
Octree Training points distribution.

Capacity	Number of cells containing Training points			
	<50k	>50k	Total	Non empty
350	12 086	0	12 482	12 086 (96.83%)
100	45 649	0	48 378	45 649 (94.36%)

Octree achieves a better points distribution (even the old best one) as shown in Table 6 (which is analogous to Table 4), where all cells contain limited number of Training points.

7. Comparison to other algorithms

In this section we test our work against other well-known algorithms of the literature. Specifically, we compare to HBRJ [23], PGBJ [21] and also our implementation of the original algorithm [2] (for the sake of completeness of this comparison). The source code and precompiled classes, kindly provided by the authors of PGBJ,[2] were used for both HBRJ and PGBJ.

We ran all algorithms in our 9 node cluster, using 2 different dataset combinations, both in 2D and 3D. The first combination uses the same datasets as in the experiments of Sections 5 and 6. Each dataset contains approximately 11.5 million points, so we will call this combination $11M \times 11M$, for short. We also took the first 5 million lines of each dataset and created a $5M \times 5M$ combination, to see how the algorithms perform with smaller datasets.

We used the best-performing grid parameter N and Quad/Oct tree capacity from Sections 5 and 6 for the original and the new algorithm, respectively, and the recommended parameters mentioned in [23] for HBRJ and in [21] for PGBJ.

In Fig. 58, we can see how the four algorithms compare to each other in $2D$, using $11M$ datasets. Obviously, the slowest, by far, is HBRJ. The fastest is our Quad tree, combined with PlaneSweep and LessData. PGBJ and the original algorithm take the 3rd and 4th places. Our algorithm is 3–4 times faster than PGBJ.

In Fig. 59, where $3D$ datasets are used, the last one to finish is once more HBRJ, followed by the original algorithm, who shows its weakness in $3D$ calculations. Our algorithm is the clear winner, followed by PGBJ, which is almost 2 times slower. Please note that we have not presented results for the original algorithm in $3D$ previously in this paper.

[2] https://www.comp.nus.edu.sg/~dbsystem/source.html.

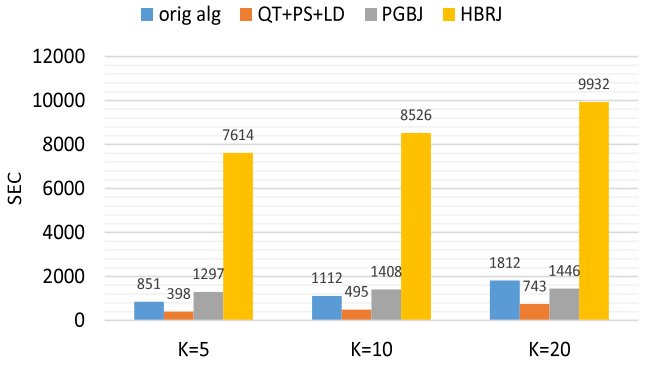

Fig. 58. $2D$ comparison, $11M \times 11M$ datasets.

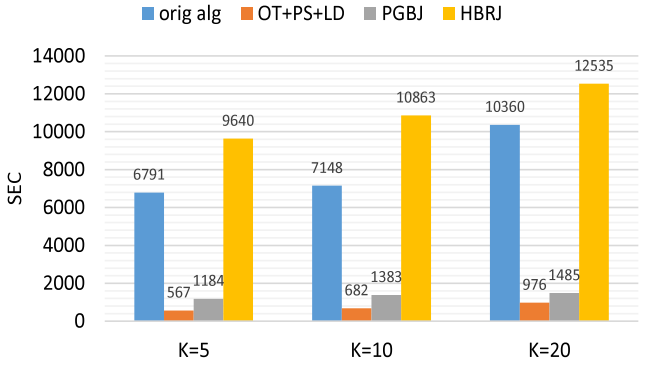

Fig. 59. $3D$ comparison, $11M \times 11M$ datasets.

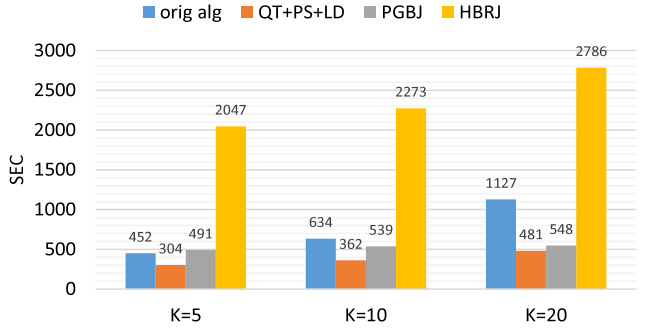

Fig. 60. $2D$ comparison, $5M \times 5M$ datasets.

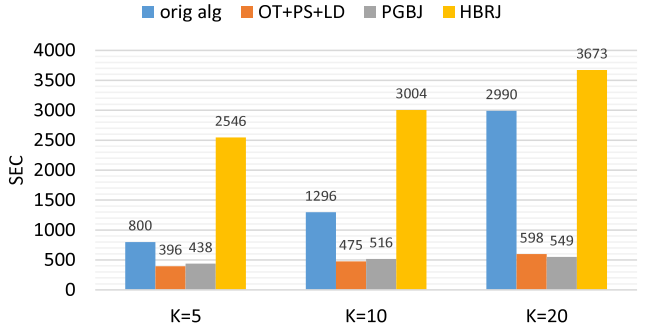

Fig. 61. $3D$ comparison, $5M \times 5M$ datasets.

When using almost half of the dataset sizes (Figs. 60 and 61), the four algorithms maintain the same places at the finish line. HBRJ is still the slowest and our own is still the fastest (except for $k = 20$ in 3D, where it is slightly outperformed by PGBJ). The original algorithm is once more very ineffective in $3D$, especially as k grows, which proves again that our LessData technique was a necessary improvement, combined of course with PlaneSweep and Quad/Oct tree partitioning.

8. Conclusions and future plans

In this paper, we have presented a set of improvements that can be used to accelerate the performance of the algorithm that was presented in [2] for processing the AkNNQ in a parallel and distributed environment (using the Apache Hadoop infrastructure). These improvements include

- pruning candidate points during neighbors' calculation more efficiently (using plane sweep),
- partitioning space into unequal cells containing almost equal numbers of points (Quadtree partitioning),
- restructuring the input and output between the phases of the algorithm, minimizing network traffic (as shown by experimentation, even a limited reduction of the data exchanged between phases can lead to significant acceleration of execution).

By extensive experimentation on 2D and 3D real data, we examined the effect of these improvements on performance. Our experiments showed that

- plane sweep has a major impact on performance (20%–30% improvement over the original algorithm),
- smaller Phase 3 output gives another significant boost (about 40% improvement over the original algorithm) and saved about 40% of Phase 3's output data size,
- the Quadtree/Octree space decomposition leads to about 35%/70% performance improvement over the best Grid space decomposition in 2D/3D, since it adapts to the data distribution.
- Quadtree/Octree space decomposition leads to an algorithm that is less sensitive (regarding performance) to the change of the partitioning parameter (tree node capacity) than a Grid based algorithm (such an algorithm is significantly affected by the partitioning parameter, the number of Grid cells).

We have also tested the scalability of the original and improved algorithms by varying the number of computing nodes and dataset size. The performance improvement proved to be roughly linear to the number of nodes, while the larger dataset showed that Grid partitioning needs major calibration and still performs 60%–80% worse than a non-calibrated Quadtree/Octree.

Finally, a comparison to other well known algorithms of the literature, such as HBRJ and the state-of-the-art PGBJ, showed that our algorithm is the clear winner, proving that the optimizations made on the original algorithm were very effective, especially in $3D$ and for bigger k's, where the original algorithm is inefficient.

This work was based on Apache Hadoop because our purpose was to improve the work of [2] on the same infrastructure and propose alternative computational methods. Hadoop, despite its age, it is still widely used for a variety of applications and its lack of integrated spatial capabilities give the developer the ability to create solutions tailored closely to the query studied, without depending on generic spatial extensions with possible overheads. For example, even the feature rich SpatialHadoop does not incorporate more than two dimensions processing out of the box. This work is one of the several steps we plan to follow for the creation and study of AkNNQ processing in parallel and distributed environments. In the future, we plan to implement the improvements presented here into specialized spatial frameworks, such as SpatialHadoop [12] and LocationSpark [13], as well as into plain Apache Spark [11] and compare their efficiency to the algorithms studied in [25]. In this way, we will contrast the performance of our methods on plain Hadoop vs. systems with specialized integrated spatial functions and indexing (SpatialHadoop and LocationSpark) and also I/O vs. in-memory processing systems (Spark). Finally, we plan to apply the ideas and techniques presented here to processing of other queries, like reverse and group nearest neighbor queries.

References

[1] C. Böhm, F. Krebs, The k-nearest neighbour join: Turbo charging the KDD process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.
[2] N. Nodarakis, E. Pitoura, S. Sioutas, A.K. Tsakalidis, D. Tsoumakos, G. Tzimas, Kdann+: A rapid aknn classifier for big data, Trans. Large-Scale Data-Knowl.-Centered Syst. 24 (2016) 139–168.
[3] S. Koenig, Y.V. Smirnov, Graph learning with a nearest neighbor approach, in: Proceedings of the Ninth Annual Conference on Computational Learning Theory, COLT 1996, Desenzano Del Garda, Italy, June 28-July 1, 1996, pp. 19–28.
[4] D.J. Eisenstein, P. Hut, HOP: A new group-finding algorithm for n-body simulations, Astrophys. J. 498 (1) (1998) 137–142.
[5] A. Eldawy, M.F. Mokbel, The era of big spatial data, PVLDB 10 (12) (2017) 1992–1995.
[6] D. Jiang, B.C. Ooi, L. Shi, S. Wu, The performance of mapreduce: An in-depth study, PVLDB 3 (1) (2010) 472–483.
[7] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[8] S. Zhang, J. Han, Z. Liu, K. Wang, S. Feng, Spatial queries evaluation with mapreduce, in: Eighth International Conference on Grid and Cooperative Computing, GCC 2009, Lanzhou, Gansu, China, August 27-29, 2009, pp. 287–292.
[9] Apache Software Foundation: Hadoop homepage https://hadoop.apache.org/.
[10] T. White, Hadoop: The Definitive Guide, fourth ed., O'Reilly Media, Inc., 2015.
[11] Apache Software Foundation: Spark homepage https://spark.apache.org/.
[12] A. Eldawy, M.F. Mokbel, Spatialhadoop: A mapreduce framework for spatial data, in: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, pp. 1352–1363.
[13] M. Tang, Y. Yu, Q.M. Malluhi, M. Ouzzani, W.G. Aref, Locationspark: A distributed in-memory data management system for big spatial data, PVLDB 9 (13) (2016) 1565–1568.
[14] T. Yokoyama, Y. Ishikawa, Y. Suzuki, Processing all k-nearest neighbor queries in hadoop, in: Web-Age Information Management - 13th International Conference, WAIM 2012, Harbin, China, August 18-20, Proceedings, 2012, pp. 346–351.
[15] C. Yu, B. Cui, S. Wang, J. Su, Efficient index-based KNN join processing for high-dimensional data, Inf. Softw. Technol. 49 (4) (2007) 332–344.

[16] Y. Chen, J.M. Patel, Efficient evaluation of all-nearest-neighbor queries, in: Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, the Marmara Hotel, Istanbul, Turkey, April 15-20, 2007, pp. 1056–1065.
[17] T. Emrich, F. Graf, H. Kriegel, M. Schubert, M. Thoma, Optimizing all-nearest-neighbor queries with trigonometric pruning, in: Scientific and Statistical Database Management, 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30 - July 2, Proceedings, 2010, pp. 501–518.
[18] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, pp. 71–79.
[19] C. Xia, H. Lu, B.C. Ooi, J. Hu, Gorder: An efficient method for KNN join processing, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases, in: VLDB '04, 2004, pp. 756–767.
[20] H.V.L. Cao, T.N. Phan, M.Q. Tran, T.L. Hong, M.N.Q. Truong, Processing all k-nearest neighbor query on large multidimensional data, in: 2016 International Conference on Advanced Computing and Applications, ACOMP 2016, Can Tho City, Vietnam, November 23-25, 2016, pp. 11–17.
[21] W. Lu, Y. Shen, S. Chen, B.C. Ooi, Efficient processing of k nearest neighbor joins using mapreduce, PVLDB 5 (10) (2012) 1016–1027.
[22] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W. Lee, E. Pitoura, Distributed in-memory processing of all k nearest neighbor queries, IEEE Trans. Knowl. Data Eng. 28 (4) (2016) 925–938.
[23] C. Zhang, F. Li, J. Jestes, Efficient parallel knn joins for large data in mapreduce, in: 15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, Proceedings, 2012, pp. 38–49.
[24] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, X. Song, Accelerating spatial data processing with mapreduce, in: 16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010, pp. 229–236.
[25] F. Garcia-Garcia, A. Corral, L. Iribarne, M. Vassilakopoulos, Y. Manolopoulos, Distance Join Query Processing in Distributed Spatial Data Management Systems, 2018 (Submitted).
[26] G. Roumelis, A. Corral, M. Vassilakopoulos, Y. Manolopoulos, New plane-sweep algorithms for distance-based join queries in spatial databases, Geoinformatica 20 (4) (2016) 571–628.