UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Algorithms for Processing Closest-Pairs and Nearest-Neighbors Queries on Big Spatial Data in Parallel and Distributed Frameworks

Doctoral Thesis

Panagiotis Moutafis

A thesis submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Supervisor

Michael Vassilakopoulos

Associate Professor

April 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Algorithms for Processing Closest-Pairs and Nearest-Neighbors Queries on Big Spatial Data in Parallel and Distributed Frameworks

Doctoral Thesis

Panagiotis Moutafis

A thesis submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Supervisor

Michael Vassilakopoulos

Associate Professor

April 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι Επεξεργασίας Ερωτημάτων Εγγύτερων Ζευγών και Εγγύτερων Γειτόνων επί Χωρικών Δεδομένων Μεγάλου Όγκου σε Παράλληλα και Κατανεμημένα Πλαίσια

Διδακτορική Διατριβή

Παναγιώτης Μουτάφης

Διατριβή η οποία υποβλήθηκε για τη μερική εκπλήρωση
των υποχρεώσεων απόκτησης του Διδακτορικού Διπλώματος

Επιβλέπων

Μιχαήλ Βασιλακόπουλος

Αναπληρωτής Καθηγητής

Απρίλιος 2021

UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Algorithms for Processing Closest-Pairs and Nearest-Neighbors Queries on Big Spatial Data in Parallel and Distributed Frameworks

Doctoral Thesis

Panagiotis Moutafis

Advisory committee

Michael Vassilakopoulos, Associate Professor, University of Thessaly (Supervisor)

Panagiotis Bozanis, Professor, International Hellenic University

Spyros Sioutas, Professor, University of Patras

Examination committee

Michael Vassilakopoulos, Associate Professor, University of Thessaly (Supervisor)

Panagiotis Bozanis, Professor, International Hellenic University

Spyros Sioutas, Professor, University of Patras

Spyros Lalis, Professor, University of Thessaly

Ilias Savvas, Professor, University of Thessaly

Panagiota Tsompanopoulou, Associate Professor, University of Thessaly

Thanasis Loukopoulos, Assistant Professor, University of Thessaly

April 2021

Declaration of Authorship

I declare that this thesis has been composed by myself and that this work has not be submitted for any other degree or professional qualification.

I confirm that the work submitted is my own, except where work which has formed part of publications authored by myself jointly with others has been included. I declare that my contribution concerning all aspects of the work accomplished within these publications (appearing in Appendix A) is substantial, as indicated by the order of the coauthors' names.

I declare that due references have been provided on all supporting literature, resources and work of others and that I have acknowledged all main sources of help.

The declarant

Panagiotis Moutafis
7-4-2021

Acknowledgements

First of all, I would like to thank my supervising professor, Associate Professor Michael Vassilakopoulos, for his trust and the continuous support throughout the duration of my PhD studies. He introduced me into the previously uncharted for me world of spatial queries and the algorithms that process them and guided me through this difficult but interesting journey. If it wasn't for him, I would not have reached this point.

Dr. George Mavrommatis was my first tutor in Computer Science in the Hellenic Open University and he supervised my MSc thesis. He has also constantly and actively supported and helped me improve my PhD thesis work with his knowledge and experience. I am grateful for his support all these years.

I would also like to thank our Spanish friends, professor Antonio Corral, PhD candidate Francisco Garcia-Garcia and professor Luis Iribarne from the University of Almeria, with whom we collaborated many times and produced high quality research papers. They were always very co-operational and helpful, giving new ideas and proposing improvements to our work.

I also want to thank professors Spyros Sioutas and Panagiotis Bozanis for accepting to be members of the Advisory Committee and evaluating my annual work every year through my PhD studies.

In addition, I would like to thank professors Spyros Lalis, Ilias Savvas, Panagiota Tsombanopoulou and Thanasis Loukopoulos for joining the seven-member thesis committee.

Last but not least, I want to thank my mother and my brother for standing by my side all these years and morally supporting me, as they have always done.

Panagiotis Moutafis, April 2021

UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Algorithms for Processing
Closest-Pairs and Nearest-Neighbors Queries
on Big Spatial Data
in Parallel and Distributed Frameworks

Doctoral Thesis

Panagiotis Moutafis

Abstract

Spatial Data refers to data related to the position or geo-location of objects and elements on, below or above the earth's surface. Such data, often termed geospatial data, appear in geography related applications. Currently, numerous applications and sources are creating explosive amounts of data with spatial characteristics or with related geo-located information. Sensors, mobile apps, cars, GPS devices, unmanned aerial vehicles, ships, airplanes, telescopes, medical devices, web apps, social networking and IoT devices are examples of such applications and sources.

Spatial data are harder to handle than data in traditional applications (e.g., names, numbers, dates, etc.) and have higher processing requirements. Furthermore, the big volume of spatial data in modern applications requires the use of multi-node systems for their processing. Among them, shared-nothing parallel and distributed systems based on the MapReduce model and/or Resilient Distributed Datasets (RDDs) are common in research efforts.

Efficient big spatial data management requires efficient processing of computationally demanding spatial query operations. The following demanding queries are applied on two datasets and combine join queries (since all possible combinations formed from these datasets are candidates for the final result) and nearest-neighbor queries (since the final result is formed according to a neighboring criterion).

- The K Closest-Pairs Query (KCPQ): for each possible pair of elements from the two datasets, it discovers K pairs with the smallest distances among their elements.

- The Distance Join Query (DJQ): this is a form of closest-pairs query which, for each possible pair of elements from the two datasets, returns pairs with distances smaller than a given distance.

- The All K Nearest-Neighbor Query (AKNNQ, also termed K Nearest-Neighbor Join): it discovers K nearest neighbors in the one dataset for each element of the other dataset,

- The Group (K) Nearest-Neighbor(s) Query (GKNNQ): it returns K elements of the one dataset with the smallest sum of distances to every element of the other dataset.

Although naive algorithms for the above queries are simple, they suffer from excessive computational, intermediate result storage and network communication cost and low load balancing among computing nodes, especially within a distributed environment. In this thesis, we focus on point data and employ techniques for faster and fewer computations, pruning of unnecessary computations, taking advantage of spatial locality and distribution of data, improving load balancing among computing nodes and optimizing the amount of data transferred between nodes. Based on these techniques,

- we develop the first KCPQ and DJQ algorithms for Apache Spark, a popular parallel and distributed system that has attracted attention due to exploiting in-memory processing capabilities,

- we develop AKNNQ algorithms for Apache Hadoop, the first widely accepted system implementing the MapReduce model,

- we develop the first GKNNQ algorithms for both Apache Hadoop and SpatialHadoop, an extension specifically designed to manage big spatial datasets,

- for each of the above queries, we perform extensive experimental tests to derive the best parameter settings for each algorithm and to compare the efficiency of the several alternative algorithms we developed and ones appearing in the literature (for the cases where such algorithms already existed).

Keywords

Spatial Data, Spatial Query Processing, Closest-pairs Queries, Nearest-Neighbors Queries, Parallel and Distributed Systems, Apache Spark, Apache Hadoop, SpatialHadoop, MapReduce, Resilient Distributed Datasets, Quadtrees, Plane-Sweep.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι Επεξεργασίας
Ερωτημάτων Εγγύτερων Ζευγών και Εγγύτερων Γειτόνων
επί Χωρικών Δεδομένων Μεγάλου Όγκου
σε Παράλληλα και Κατανεμημένα Πλαίσια

Διδακτορική Διατριβή

Παναγιώτης Μουτάφης

Περίληψη

Τα Χωρικά Δεδομένα αναφέρονται σε δεδομένα που σχετίζονται με τη θέση ή τη γεωγραφική τοποθεσία αντικειμένων και στοιχείων υπεράνω, υπό ή επί της επιφάνειας της γης. Τέτοια δεδομένα, συχνά ονομάζονται γεωχωρικά δεδομένα, εμφανίζονται σε εφαρμογές σχετικές με τη γεωγραφία. Καθημερινά, πολυπληθείς εφαρμογές και πηγές δημιουργούν εκρηκτικούς όγκους δεδομένων με χωρικά χαρακτηριστικά ή με σχετική γεωχωρική πληροφορία. Αισθητήρες, εφαρμογές σε κινητά τηλέφωνα, αυτοκίνητα, συσκευές GPS, μη επανδρωμένα εναέρια οχήματα (UAV), πλοία, αεροπλάνα, τηλεσκόπια, ιατρικές συσκευές, διαδικτυακές εφαρμογές, κοινωνικά δίκτυα και συσκευές διαδικτύου των αντικειμένων (IoT) αποτελούν παραδείγματα τέτοιων εφαρμογών και πηγών.

Η επεξεργασία των χωρικών δεδομένων είναι δυσκολότερη σε σχέση με τα δεδομένα των παραδοσιακών εφαρμογών (π.χ. ονόματα, αριθμοί, ημερομηνίες, κλπ.) και έχουν υπολογιστικές υψηλότερες απαιτήσεις. Επιπλέον, ο μεγάλος όγκος των χωρικών δεδομένων στις σύγχρονες εφαρμογές απαιτεί τη χρήση συστημάτων πολλαπλών κόμβων για την επεξεργασία τους. Μεταξύ αυτών, τα παράλληλα και κατανεμημένα συστήματα χωρίς διαμοίραση (shared-nothing) που βασίζονται στο μοντέλο MapReduce και/ή στα Ανθεκτικά Κατανεμημένα Σύνολα Δεδομένων (Resilient Distributed Datasets - RDDs) απαντώνται συχνά στις ερευνητικές προσπάθειες.

Η αποτελεσματική διαχείριση των μεγάλων χωρικών δεδομένων απαιτεί αποτελεσματική επεξεργασία των υπολογιστικά απαιτητικών χωρικών ερωτημάτων. Τα ακόλουθα χω-

ρικά ερωτήματα εφαρμόζονται σε δυο σύνολα δεδομένων και συνδυάζουν ερωτήματα ζεύξης (join queries), καθώς όλοι οι δυνατοί συνδυασμοί που σχηματίζονται από αυτά τα σύνολα δεδομένων είναι υποψήφιοι για το τελικό αποτέλεσμα, και ερωτήματα εγγυτέρων γειτόνων (nearest neighbor queries), καθώς το τελικό αποτέλεσμα διαμορφώνεται σύμφωνα με ένα κριτήριο γειτονικότητας.

1. Το Ερώτημα των K Εγγυτέρων Ζευγών (K Closest-Pairs Query, KCPQ): για κάθε πιθανό ζεύγος στοιχείων από τα δυο σύνολα δεδομένων, ανακαλύπτει τα K ζεύγη με τις μικρότερες αποστάσεις μεταξύ των στοιχείων τους.

2. Το Ερώτημα Ζεύξης Απόστασης (Distance Join Query, DJQ): είναι ένα είδος ερωτήματος εγγυτέρων ζευγών το οποίο, για κάθε πιθανό ζεύγος στοιχείων από τα δυο σύνολα δεδομένων, επιστρέφει τα ζεύγη με αποστάσεις μικρότερες από μια δοσμένη απόσταση.

3. Το Ερώτημα Όλων των K Εγγυτέρων Γειτόνων (All K Nearest Neighbor Query, AKNNQ), που ονομάζεται κσι Ζεύξη K Εγγυτέρων Γειτόνων (K Nearest-Neighbor Join): επιστρέφει τους K εγγύτερους γείτονες στο ένα σύνολο για κάθε στοιχείο του άλλου συνόλου.

4. Το Ερώτημα Ομάδας K Εγγυτέρων Γειτόνων (Group (K) Nearest-Neighbor(s) Query, GKNNQ): επιστρέφει K στοιχεία από το ένα σύνολο με το μικρότερο άθροισμα αποστάσεων προς κάθε στοιχείο του άλλου συνόλου.

Παρόλο που οι αφελείς αλγόριθμοι για τα παραπάνω ερωτήματα είναι απλοί, πάσχουν από υπερβολικό κόστος υπολογισμού, αποθήκευσης ενδιάμεσου αποτελέσματος και δικτυακής επικοινωνίας και χαμηλής εξισορρόπισης φορτίου μεταξύ των υπολογιστικών κόμβων, ιδιαίτερα σε ένα κατανεμημένο περιβάλλον. Σε αυτή τη διατριβή, επικεντρωνόμαστε σε σημειακά δεδομένα και χρησιμοποιούμε τεχνικές για γρηγορότερους και λιγότερους υπολογισμούς, περικοπή των μη αναγκαίων υπολογισμών, εκμετάλλευση της τοπικότητας και της κατανομής των δεδομένων, καλύτερης εξισορρόπησης του φορτίου μεταξύ των υπολογιστικών κόμβων και βελτιστοποίησης της ποσότητας των δεδομένων που διακινούνται μεταξύ των κόμβων. Με αυτά τα εφόδια,

1. αναπτύσσουμε τους πρώτους KCPQ και DJQ αλγορίθμους για το Apache Spark, ένα

δημοφιλές σύστημα παράλληλης και κατανεμημένης επεξεργασίας το οποίο έχει προσελκύσει την προσοχή εξαιτίας των δυνατοτήτων υπολογισμού εντός μνήμης,

2. αναπτύσσουμε AKNNQ αλγορίθμους για το Apache Hadoop, το πρώτο ευρέως αποδεκτό σύστημα που υλοποιεί το μοντέλο MapReduce,

3. αναπτύσσουμε τους πρώτους GKNNQ αλγορίθμους για το Apache Hadoop και το SpatialHadoop, μια επέκταση ειδικά σχεδιασμένη να διαχειρίζεται μεγάλα σύνολα χωρικών δεδομένων,

4. για κάθε ένα από τα παραπάνω ερωτήματα, διενεργούμε εκτεταμένα πειράματα για να εξάγουμε τις καλύτερες ρυθμίσεις των παραμέτρων για κάθε αλγόριθμο και να συγκρίνουμε την αποτελεσματικότητα των διαφόρων εναλλακτικών αλγορίθμων που αναπτύξαμε και εκείνων της βιβλιογραφίας (για τις περιπτώσεις εκείνες όπου τέτοιοι αλγόριθμοι προϋπήρχαν).

Λέξεις Κλειδιά

Χωρικά Δεδομένα, Επεξεργασία Χωρικών Ερωτημάτων, Ερωτήματα Εγγυτέρων Ζευγών, Ερωτήματα Εγγυτέρων Γειτόνων, Παράλληλα και Κατανεμημένα Συστήματα, Apache Spark, Apache Hadoop, SpatialHadoop, MapReduce, Ανθεκτικά Κατανεμημένα Σύνολα Δεδομένων, Τετραδικά Δέντρα, Σάρωση Επιπέδου.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Spatial Data refers to data related to the position or geo-location of objects and elements on, below or above the earth's surface. Such data, often termed geospatial data, appear in geography related application domains, for example, astronomy applications, environmental monitoring, earthquake research, weather forecasting and traffic management which are supported by Geographic Information Systems (GIS) [72], or even in applications from domains beyond the ones directly related to geography, like medicine, or biology.

Spatial data are discriminated in two categories, vector and raster data. For modeling discrete spatial objects, such as rivers, roads, cities or countries, vector data, mainly points, line segments and polygons are used. For modeling spatially continuous phenomena, such as elevation, temperature or air quality, raster data represented by a raster (grid) of equally sized rectangles called cells or pixels are used. Each such cell represents a small area and its value is related to the value of the phenomenon modeled in this area.

Explosive amounts of data with spatial characteristics, or with related geo-located information, are being created from numerous applications and sources. For example, sensors, mobile apps, cars, Global Positioning System (GPS) devices, Unmanned Aerial Vehicles (UAVs), ships, airplanes, telescopes, medical devices, web apps, social networking and IoT devices. According to [38], "Geospatial data has always been big data". For example, personal location data which are created per year fall in the scale of Petabytes. According to the United Nations Initiative on Global Geospatial Information Management (UN-GGIM) [59], "With approximately 2.5 quintillion bytes of data created every day, a significant amount of which will have some kind of location reference, the challenges of data management and data integration will be significant."

1.1 Big spatial data processing

Spatial databases [63] are specialized databases that support storage and querying of spatial data. They are core elements of GIS. Processing of spatial queries can become very demanding if the volume of data on which such a query is applied is large, or if the number of the combinations of data objects that need to be examined for answering such a query is large.

Due to their multidimensional nature, spatial data are harder to handle than data in traditional applications (e.g., names, numbers, dates, etc.) and have higher processing requirements. Furthermore, the big volume of spatial data in modern applications requires specialized systems for their processing.

In stand-alone systems, the exploitation of large amounts of main memory, Solid-state Drives (SSDs), Central Processing Units (CPUs) or Graphics Processing Units (GPUs) with multiple cores can be used for big spatial data management.

As spatial data to be processed gets bigger, the use of multi-node systems is inevitable. Among them, shared-nothing parallel and distributed systems based on the MapReduce model and/or Resilient Distributed Datasets (RDDs) are common in research efforts [18]. Parallel and distributed computing using shared-nothing clusters for big data management is a research trend during last years. MapReduce is a programming paradigm suitable for such clusters and Apache Hadoop [14,31] is a popular open-source software framework implementing this paradigm. Apache Spark [75,90] is another, more recent, open-source cluster-computing framework that uses RDDs. Several spatial extensions of Hadoop (like SpatialHadoop [76]) and Spark (like SIMBA [74] and GeoSpark/Sedona [71]) have appeared during last years. Such systems are usually implemented within a Cloud Computing environment and, beyond processing speed, they provide failure resilience and scalability.

Efficient big spatial data management requires processing of spatial query operations [18], like Point and Range, Nearest-neighbor and Spatial-join queries and execution of computational geometry operations, like Voronoi diagram construction and skyline and convex hull calculation operations.

In general, these operations, are computationally demanding (especially when applied on big data) and efficient algorithms suitable for parallel and distributed systems are needed. The following demanding queries are applied on two datasets and combine join queries (since all possible combinations formed from these datasets are candidates for the final result) and

nearest-neighbor queries (since the final result is formed according to a neighboring criterion).

- The K Closest-Pairs Query (KCPQ), for each possible pair of elements from the two datasets, discovers K pairs with the smallest distances among their elements.
- The Distance-Join Query (DJQ), for each possible pair of elements from the two datasets, returns pairs with distances smaller than a given distance.
- The All K Nearest-Neighbors Query (AKNNQ, also termed K Nearest-Neighbor Join), discovers k nearest neighbors in the one dataset for each element of the other dataset,
- The Group (K) Nearest-Neighbor(s) Query (GKNNQ) returns K elements of the one dataset with the smallest sum of distances to every element of the other dataset.

Naive algorithms for the above queries, although simple, examine all possible combinations and are inefficient. For example, a naive AKNNQ algorithm, for every element of the first dataset would calculate the distance to every element of the second dataset and keep K ones with the smallest distances. While, a naive KCPQ algorithm would calculate the distances of the elements of all possible pairs from the two datasets and keep K pairs with the with the smallest distances.[1] Such naive algorithms, especially when data are stored in a distributed environment, suffer from excessive computational, intermediate result storage and network communication cost and low load balancing among computing nodes.

1.2 Thesis contribution

Motivated by the vast amount of spatial data produced, their exploitation in numerous modern applications and the need to process these data in parallel and distributed environments with efficient algorithms, in this thesis, we develop such algorithms and study their performance.

Since, the spectrum of possible research directions within this discipline is immense, we focus on the above four demanding queries, namely KCPQ, DJQ, AKNNQ and GKNNQ. Moreover, since an algorithm for a parallel and distributed system relies on the characteristics

[1] Note that in both these two queries the answer might not be unique, due to ties of distances.

of the system and the possible system choices are numerous, we develop algorithms for some of the most popular such systems.

In this thesis, we focus on point data and employing of techniques for faster and fewer computations, pruning of unnecessary computations, taking advantage of spatial locality and distribution of data, better load balancing among computing nodes, optimizing the amount of data transferred between nodes,

- we develop the first KCPQ and DJQ algorithms for Apache Spark, a popular parallel and distributed system that has attracted attention due to exploiting in-memory processing capabilities,
- we develop AKNNQ algorithms for Apache Hadoop, the first widely accepted system implementing the MapReduce model,
- we develop the first GKNNQ algorithms for both Apache Hadoop and SpatialHadoop, an extension specifically designed to manage big spatial datasets,
- for each of the above queries, we perform extensive experimental tests to derive the best parameter settings for each algorithm and to compare the efficiency of the several alternative algorithms we developed and ones appearing in the literature (for the cases where such algorithms already existed).

Although, Apache Spark and Apache Hadoop were not designed specifically for spatial data, the algorithms we developed make these systems suitable for processing the spatial queries we study. SpatialHadoop is a spatial aware system and the algorithms we developed for it take advantage of the spatial support it provides to increase performance. Our approach is outlined in the following.

KCPQ and DJQ queries have been thoroughly studied in centralized environments and also DJQ is incorporated as a parallel and distributed out-of-the-box solution in Simba, a derivative of Spark. We presented the first in literature algorithms for KCPQ in Spark and furthermore our own version of DJQ for Spark, which surpassed the performance of DJQ in Simba. We developed several partitioning techniques based on slicing the space into strips, an efficient computation method based on Plane-Sweep technique and utilized a sample based upper bound distance for pruning non-eligible points and strips. All these methods were extensively tested using several real world datasets.

Regarding the AKNNQ, we took a high performing multi-phased MapReduce algorithm from the literature and modified it to perform even better. We used two partitioning methods (Grid and Quadtree), two computational methods (Brute-Force and Plane-Sweep) and a technique that reduces intermediate output data in order to save network bandwidth. We also developed a three-dimensional version of the algorithm using Grid and Octree partitioning. Finally, we compared it to the base algorithm as well as other popular algorithms from the literature, using real world datasets and tested the performance effect of various parameters. Our algorithm was the winner among all the others algorithms tested.

Lastly, we presented the first MapReduce algorithm for the GKNNQ, exploiting and extending techniques from centralized environments. Our algorithm was initially developed for Hadoop, followed by an enhanced version and was also ported to SpatialHadoop. It consists of local and parallel phases and involves two partitioning methods (Grid and Quadtree, or combinations of them in SpatialHadoop version), two computational methods (Brute-Force and Plane-Sweep) and pruning heuristics from the literature. Later on, other enhancements were added, like the Fast Sums technique, which stops calculating distance sums when a particular threshold is reached. SpatialHadoop uses a novel two-level partitioning method and exploits its incorporated spatial filters in order to early prune non-eligible cells and points. The algorithm's performance was thoroughly tested using real world and synthetic datasets. The experiments also included several metrics that showed what happened internally (under the hood) during the operation of the algorithm and which parameters played the most important role. We finally added a pre-partitioning step, which further improved performance by eliminating repeating calls to point location functions. The algorithm was initially developed for Hadoop, was afterwards ported to SpatialHadoop and is now being transferred to Spark, making use of its in-memory computation capabilities.

1.3 Thesis organization

The rest of thesis is organized as follows. In Chapter 2, we present parallel and distributed systems, emphasizing to the ones we used. In the next three chapters, we present the algorithms we developed. More specifically, in Chapter 3, we present the (first in the literature) KCPQ and DJQ Spark-based algorithms and study their performance; in Chapter 4, we present our AKNNQ algorithms, tune their parameters, study their performance and

compare them to existing algorithms in the literature; in Chapter 5, we present our GKNNQ algorithms (the first ones in the literature) for Hadoop and SpatiaHadoop and comparatively study their performance. Last, in Chapter 6, we summarize our work and the conclusions arising from it, while we present future research directions emerging from this thesis.

Chapter 2

Parallel and Distributed Systems

2.1 Introduction

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers." — Grace Hopper

To effectively process any query involving Big Data, we are going to need a distributed computing environment to process the query in parallel and also a suitable algorithm. There are several software frameworks that provide both cluster management and parallel algorithmic models. The most popular ones are the Apache Hadoop [14, 31] and Apache Spark [75, 90]. They both provide effective resources management and monitoring, a programming model and a distributed filesystem.

Apache Spark rightfully holds a reputation for being one of the fastest data processing tools. According to statistics, it's 100 times faster when Apache Spark vs Hadoop are running in-memory settings and ten times faster on disks. Hadoop heavily relies on disks, while Spark processes everything in memory, which allows handling the newly inputted data quickly and provides a stable data stream. This makes Spark perfect for analytics, IoT, machine learning, and community-based sites. Hadoop is a slower framework, but it has its strong suits. During batch processing, RAM tends to go in overload, slowing the entire system down. If you need to process a large number of requests, Hadoop, even being slower, is a more reliable option. Spark is better for smaller but faster apps, whereas Hadoop is chosen for projects where ability and reliability are the key requirements (like healthcare platforms or transportation software).

In recent years many parallel and distributed spatial and spatio-temporal analytics systems have emerged, which are mostly either based on Hadoop, or on Spark. Considering these alternatives, four possible groups of systems are formed: Hadoop or Spark based spatial data systems and Hadoop or Spark based spatio-temporal data systems.

2.2 Apache Hadoop

Hadoop MapReduce [14, 31] is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. It is a shared-nothing framework, meaning that the input data is partitioned and distributed to all computing nodes, which perform calculations on their local data only. Hadoop is a two-stage disk-based MapReduce computation engine, not well suited to repetitive processing tasks.

A task to be performed using the MapReduce framework has to be divided into two phases: the *map* phase, specified by a *map function* which takes input (typically from Hadoop Distributed File System, HDFS, files), performs some computations on this input and produces output results; and the *reduce* phase which processes the map phase results, as specified by a *reduce function*. An important aspect of MapReduce is that both the input and the output of the *map* step are represented as *key-value* pairs, and that the pairs with the same key will be processed as one group by a *reduce* process: $map : (k_1, v_1) \rightarrow list(k_2, v_2)$ and $reduce : k_2, list(v_2) \rightarrow list(v_3)$. Additionally, a *combiner function* can be used to run on the output of the *map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *reduce* phase. Figure 2.1 shows an overview of the MapReduce process when multiple reducers are used.

How does MapReduce scale over a set of servers? The key to how MapReduce works is to take input as, conceptually, a list of records (each single record can be one or more lines of data). Then the input records are split and passed to the many servers in the cluster to be consumed by the *map* function. The result of the *map* computation is a list of key-value pairs. Then the *reduce* function takes each set of values that have the same key and combines them into a single value (or set of values). In other words, the *map* function takes a set of data chunks and produces key-value pairs, and *reduce* merges the output of the data generated by *map*, so that instead of a set of key-value pairs, you get your desired result.

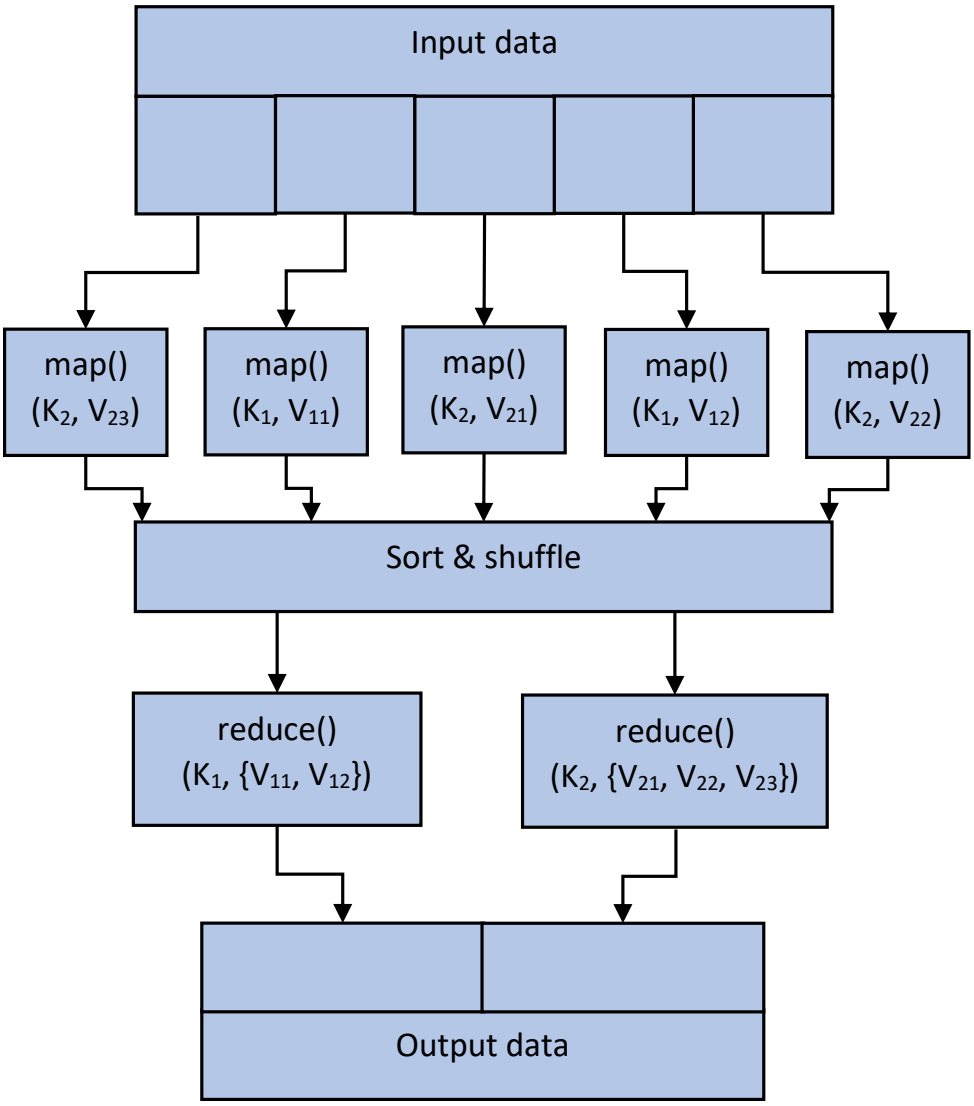

Figure 2.1: MapReduce.

One of the major benefits of MapReduce is its "shared-nothing" data-processing platform. This means that all mappers can work independently, and when mappers complete their tasks, reducers start to work independently (no data or critical region is shared among mappers or reducers; having a critical region will slow distributed computing). This shared-nothing paradigm enables us to write *map* and *reduce* functions easily and improves parallelism effectively and effortlessly.

The files processed by Hadoop are stored into its own distributed filesystem, HDFS. A distributed filesystem is necessary because we deal with Big Data which outgrow the storage capacity of a single machine. HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. HDFS is

built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record. HDFS has a block size of 128MB and all data stored in it are split in block-sized chunks. However, unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage, for example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB. HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks.

An HDFS cluster has two types of nodes operating in a master−worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts. Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to, and they report back to the namenode periodically with lists of blocks that they are storing.

2.3 Apache Spark

To overcome limitations of the MapReduce paradigm and Apache Hadoop (especially regarding iterative algorithms), Apache Spark was developed [75, 90]. This is also an open-source cluster-computing framework based on Resilient Distributed Datasets (RDDs) [89], 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. Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG (Directed Acyclic Graph) scheduler, a query optimizer and a physical execution engine.

The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark's standalone cluster manager, YARN, or Mesos. We then submit Spark Applications to these cluster managers, which will grant resources to our application so that we can complete our work. Spark Applications consist of a *driver* process and a set of *execu-*

tor processes. The driver process runs the *main* function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors. The driver process is absolutely essential; it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application. The executors are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for only two things: executing code assigned to it by the driver, and reporting the state of the computation on that executor back to the driver node.

Figure 2.2 demonstrates how the cluster manager controls physical machines and allocates resources to Spark Applications. At a high level in the Spark architecture, a Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster—the Spark executors and cluster manager—through a Spark Session.

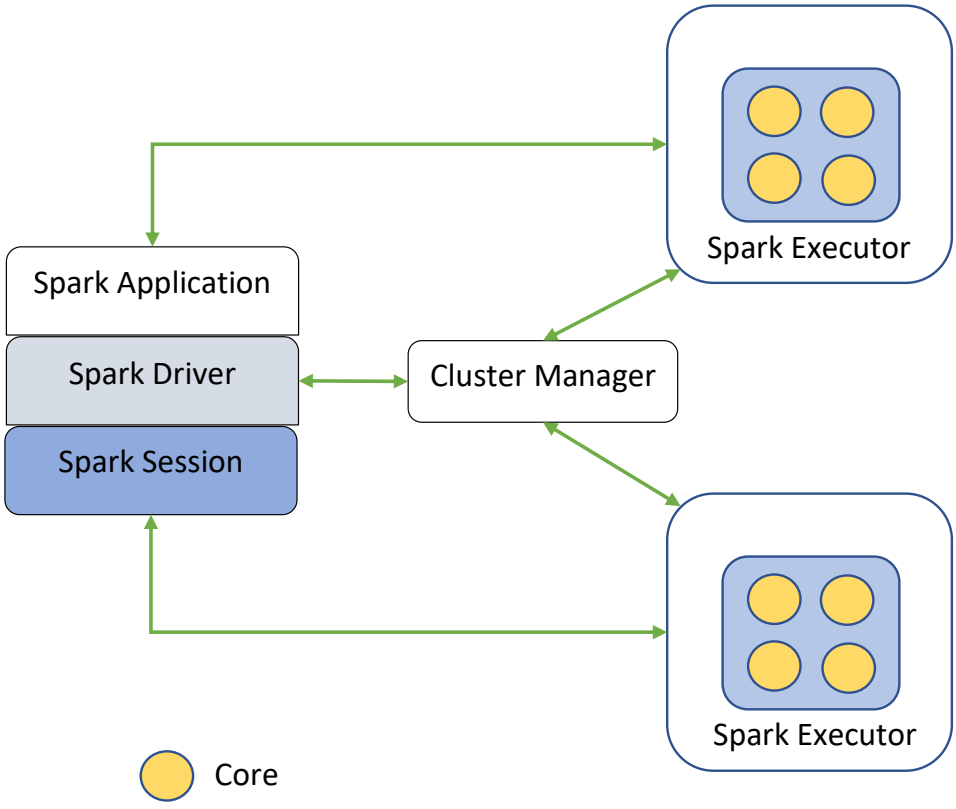

Figure 2.2: The architecture of a Spark application.

Spark revolves around the concept of an RDD, which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in the driver program, or referencing a dataset in an external storage

system, such as a shared filesystem. Once created, the distributed dataset can be operated on in parallel. One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster.

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, *map* is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, *reduce* is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

By default, each transformed RDD may be recomputed each time an action is ran on it. However, we may also *persist* an RDD in memory and/or disk, for much faster access the next time we query it.

Spark also provides two limited types of shared variables for two common usage patterns: broadcast variables and accumulators. Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Accumulators are variables that are only "added" to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters or sums.

2.4 SpatialHadoop

SpatialHadoop[1] [17] is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the *language*, *storage*, *MapReduce*, and *opera-*

[1]During the writing of this thesis, SpatialHadoop developers have announced that they have ceased its development and that they have transferred the project to Apache Spark: `https://github.com/aseldawy/spatialhadoop2`

tions layers. In the language layer, SpatialHadoop adds a simple and expressive high level language for spatial data types and operations. In the storage layer, SpatialHadoop adapts traditional spatial index structures, Grid, R-tree and R+-tree, to form a two-level spatial index. SpatialHadoop enriches the MapReduce layer by two new components, *SpatialFileSplitter* and *SpatialRecordReader*, for efficient and scalable spatial data processing. In the operations layer, SpatialHadoop is already equipped with a dozen of operations, including range query, *K*NN, and spatial join.

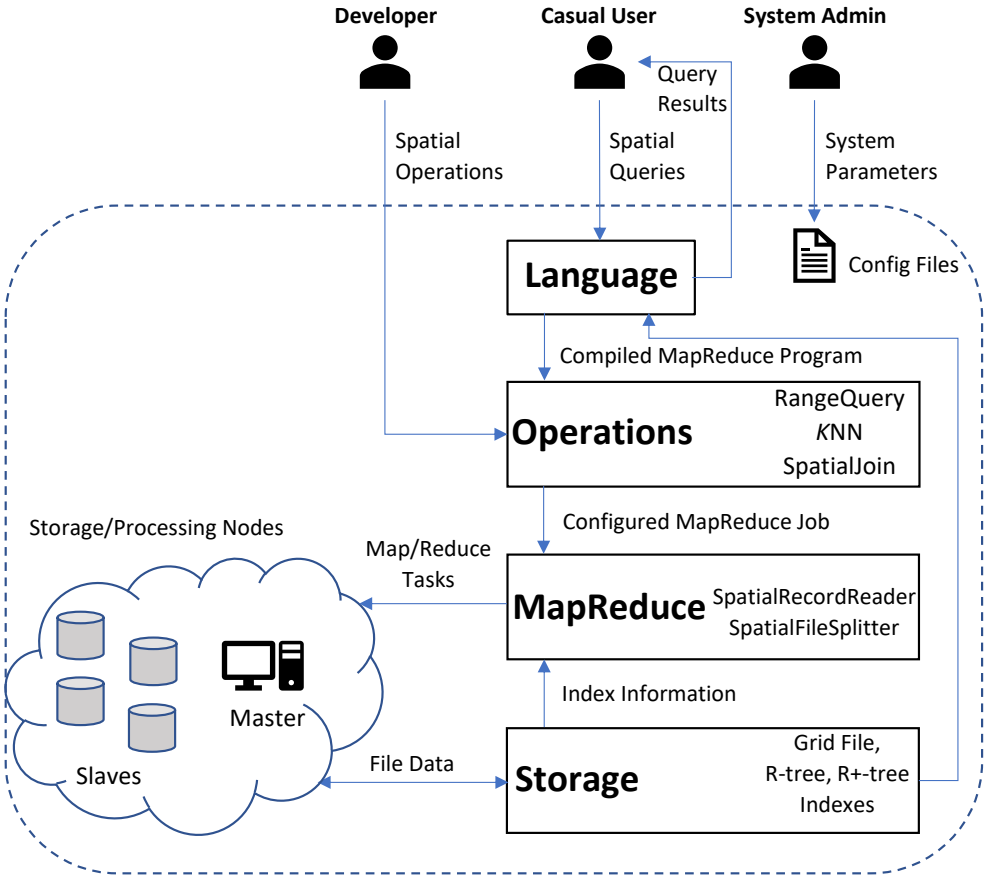

Figure 2.3: SpatialHadoop system architecture.

Fig 2.3 gives the high level architecture of SpatialHadoop. Similar to Hadoop, a SpatialHadoop cluster contains one master node that breaks a MapReduce job into smaller tasks, carried out by slave nodes.

In the storage layer, SpatialHadoop employs a two-level index structure of global and local indexing. The global index partitions data across computation nodes while the local indexes organize data inside each node. SpatialHadoop uses the proposed structure to implement three standard indexes, namely, Grid file, R-tree and R+-tree. SpatialHadoop also adds

two new components to the MapReduce layer to allow MapReduce programs to access the spatial index structures. The SpatialFileSplitter exploits the global index to prune file blocks that do not contribute to answer, while the SpatialRecordReader exploits local indexes to efficiently retrieve a partial answer from each block.

Since input files in Hadoop are non-indexed heap files, the performance is limited as the input has to be scanned. To overcome this limitation, SpatialHadoop employs spatial index structures within Hadoop Distributed File System (HDFS) as a means of efficient retrieval of spatial data. Indexing in SpatialHadoop is the key point in its superior performance over Hadoop. Regardless of the underlying spatial index structure, an index building in Spatial-Hadoop is composed of three main phases, namely, *partitioning*, *local indexing*, and *global indexing*.

The *partitioning* phase spatially partitions the input file into n partitions which satisfy that each partition should fit in a single HDFS block, also that spatially nearby objects are assigned to the same partition, and finally that all partitions should be roughly of the same size. The *local indexing* phase's purpose is to build the requested index structure (e.g., Grid or R-tree) as a local index on the data contents of each physical partition. The *global indexing* phase's purpose is to build the requested index structure (e.g., Grid or R-tree) as a global index that indexes all partitions. The global index is kept in the main memory of the master node all the time.

Fig. 2.4 depicts part of the MapReduce plan in both Hadoop and SpatialHadoop. In Hadoop, the input file goes through a *FileSplitter* that divides it into n splits, where n is set by the the MapReduce program, based on the number of available slave nodes. Then, each split goes through a *RecordReader* that extracts records as key-value pairs which are passed to the map function. SpatialHadoop enriches traditional Hadoop systems by two main components: (1) *SpatialFileSplitter*; an extended splitter that exploits the global index(es) on input file(s) using a filter function provided by the developer, to early prune file blocks not contributing to answer, and (2) *SpatialRecordReader*, which reads a split originating from spatially indexed input file(s) and exploits the local indexes to efficiently process it.

2.5 Spark derivatives

In this section we present the most indicative Spark spatial derivatives.

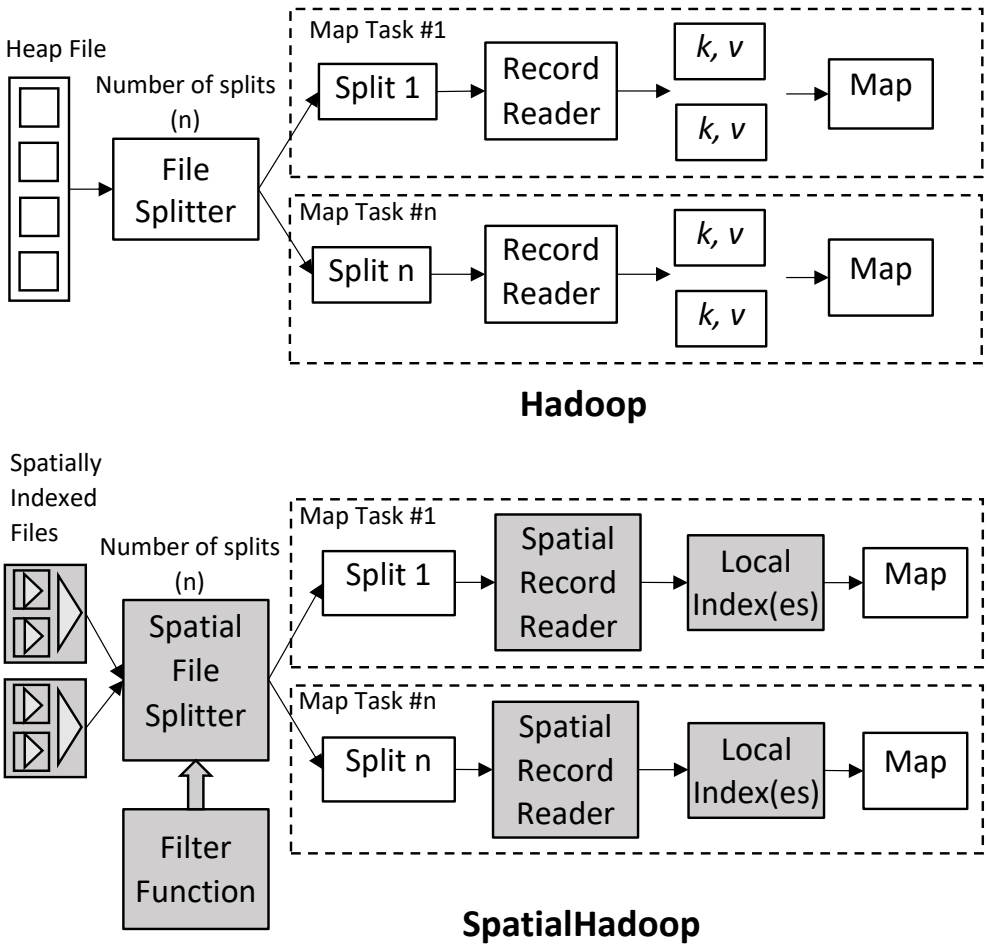

Figure 2.4: Map phase in Hadoop and SpatialHadoop.

Apache Sedona, formerly **GeoSpark** [71, 88], an in-memory cluster computing framework for processing large-scale spatial data. It uses Spark as its base layer and adds two more layers, the Spatial RDD (SRDD) Layer and Spatial Query Processing Layer, thus providing Spark with in-house spatial capabilities. The SRDD layer consists of three newly defined RDDs, PointRDD, RectangleRDD and PolygonRDD. SRDDs support geometrical operations, like Overlap and Minimum Bounding Rectangle. SRDDs are automatically partitioned by using the uniform grid technique, where the global grid file is splitted into a number of equal geographical size grid cells. Elements that intersect with two or more grid cells are being duplicated. Sedona provides spatial indexes like Quad-Tree and R-Tree on a per partition base. The Spatial Query Processing Layer includes spatial range query, spatial join query, spatial KNN query. Sedona relies heavily on the JTS topology suite and therefore conforms to the specifications published by the Open Geospatial Consortium. Experiments, reported by the paper, show that Sedona outperforms its Hadoop-based counterparts (e.g., Spatial-

Hadoop). Mainly because caches the datasets in memory, a functionality that is natively built in the underlying Spark platform.

SpatialSpark [86], that supports indexed spatial joins and range queries. Same as with GeoSpark it utilizes the JTS suite (written in Java). As reported by the authors, JTS seems to be faster than GEOS, a C/C++ port of a subset of JTS and selected functions. Authors report that in some cases of data intensive applications SpatialSpark performs worse on multiple computing nodes than on a single node, thus showing low scalability. This fact is attributed to possible bottlenecks due to communication overheads among computing nodes a factor that is related to the number of partitions, thus rising an interesting research question: "optimizing the number of partitions which represents the tradeoffs between the degrees of parallelisms (the higher the better) and the communication overheads (the lower the better)". Despite all that, the SpatialSpark project seems to have been abandoned by its developers.

LocationSpark [79], an ambitious project, built as a library on top of Spark. It requires no modifications to Spark and provides spatial query APIs on top of the standard operators. It provides Dynamic Spatial Query Execution and operations (Range, KNN, Insert, Delete, Update, Spatial-Join, KNN-Join, Spatio-Textual). The system builds two indexes, a global (grid, quadtree and a Spatial-Bloom Filter) and a local per-worker, user-decided index (grid, rtree, etc). Global index is constructed by sampling the data. Spatial indexes are aiming to tackle unbalanced data partitioning. Additionally, the system contains a query scheduler, aiming to tackle query skew.

Spatial In-Memory Big data Analytics (SIMBA) [83] extends the Spark SQL engine to support spatial queries and analytics through SQL and the DataFrame API. Simba partitions data in a manner that they are of proper and balanced size and gathers records that locate close to the same partition. It builds a local index per partition and a global index by aggregating information from local indexes. It supports range and KNN queries, KNN and distance joins.

Chapter 3

Closest-Pair Queries

3.1 Introduction

Spatial joins, and nearest neighbor queries are typical spatial queries [13]. Spatial Join queries find all pairs of spatial objects from two spatial data sets that satisfy a spatial predicate, like intersects, contains, is enclosed by, etc. Nearest neighbor queries locate the spatial object(s) that is (are) nearest to a query object. The KCPQ discovers the (K) closest pair(s) of object(s) (usually ordered by distance), between two spatial datasets. It combines join and nearest neighbor queries: like a join query, all pairs (combinations) of objects from the two datasets are candidates for the result, and like a nearest neighbor query, the (K) smallest distance(s) is (are) the basis for inclusion in the result (and the final ordering) [10, 11]. The KCPQ can be very demanding if the datasets involved are large, since all the combinations of pairs of objects from the two datasets are candidates for the result.

For example, we can use two spatial datasets that represent the archaeological sites and popular beaches of Greece. A KCPQ (K=10) can discover the 10 closest pairs of archaeological sites and beaches (in increasing order of their distances). The result of this query can be used for planning tourist trips in Greece that combine travelers interest for history / civilization and leisure / enjoyment.

Recently, the utilization of main memory in processing KCPQs on big datasets in centralized systems has been explored [64, 66]. In [49], considering ideas and methods presented in [66] and [64] we presented a Spark based algorithm for computing KCPQs. Moreover, we presented an experimental analysis of the performance of this algorithm, based on large real-world datasets. In [50], we extended [49] by developing three alternative algorithms.

The first algorithm is a simple modification of the method of [49] that is based on single sampling for partitioning data. The other two algorithms are based on more elaborate partitioning techniques. Moreover, through an extensive experimental evaluation, we compared the performance of the three algorithms. Contrary to [49], where we performed experiments using 4 data nodes only, in [50], we performed experiments using 4, as well as, 8 data nodes, to study the scalability of the presented techniques. Furthermore, in [51] we presented another modified version of the work in [49] with alternative partitioning and dataset pairing methods. Furthermore, in [51] we created an algorithm for answering the Distance Join Query (DJQ). This algorithm surpassed in performance built-in methods of other popular distributed frameworks.

3.2 Related work

In [8] Spark is used to compute top-K similarity join in large multidimensional data. Data are being partitioned into buckets so that points that are close to each other are grouped into the same bucket, with high probability. Partitioning is made by means of locality-sensitive hashing and hamming distance computation between every two elements. The method uses Cartesian product, as provided by Spark, to create all possible buckets couples and computes local top-k over each node, then collects the results and combines them to the final solution. Divide & Conquer strategy and pruning is performed at local level.

In [62] Spark is used to perform several computational geometry operations such as Geometry Union, Convex Hull, Closest and Farthest pair, Spatial Range, Join and Aggregation on both small, medium and large data sets. Computation of Farthest Pair is performed by Brute-Force and Closest Pair is reported difficult and time costly to be solved the same way, being efficient solely for small data sets. In order to overcome the problem, computation is being performed in two steps. The first step works per partition and computes the closest point in each subset, plus the points that may still be candidates (found by sorting the x-axis of the points per partition). Local computation is performed by a Divide and Conquer method that splits the local dataset recursively. The second step creates one single partition containing the closest points that were found in step one and all the candidates and once again performs the same Divide and Conquer approach. As authors report “there is a huge jump in execution time for the “large” dataset suggesting algorithm’s effectiveness probably decreases as size

increase" and "the increase in computational resources is offset by the communication cost in the latter case". The latter case refers to the "large" dataset of the experiments, which is about 100MB.

The KCPQ has been actively studied in centralized environments, when both [10, 11, 33, 73, 84], one [29], or none [64, 66] of the two spatial datasets are indexed. Two improvements of the classic Plane-Sweep algorithm and a new Plane-Sweep algorithm, called Reverse Run Plane-Sweep, were proposed in [66] for processing KCPQs when the two datasets are not indexed and reside in main-memory. In [64], it is assumed that the (big) spatial datasets reside on secondary storage and are progressively transferred in main memory, by dividing them in strips, for processing utilizing the methods of [66].

In [49, 50] and [51], we utilized ideas presented in [66] and [64] to develop an algorithm for processing KCPQs in Spark, by separating data in strips and utilizing a Plane-Sweep approach within each strip.

3.3 Closest pair queries in parallel and distributed contexts

In this section we give the formal definitions of the Closest Pair Queries and analyze their implementation in parallel and distributed frameworks as well as the problems arising.

3.3.1 Query definitions

The Spatial Join Query, one of the most frequent queries in spatial database systems, finds all pairs of objects from two spatial datasets that satisfy a predicate ∂ (i.e. distance, intersect, overlap, etc.). In the special case that ∂ is distance we are dealing with distance join queries (DJQ), according to the following definition.

Definition 1 (Distance Join): Given two datasets, P and Q, and a distance threshold $\epsilon > 0$, the distance join between P and Q, denoted as $DJQ(P, Q, \epsilon)$, finds all pairs $(p, q), p \in P$ and $q \in Q$, within distance ϵ:

$$DJQ(P, Q, \epsilon) = \{(p, q) \in P \times Q : dist(p, q) \leqq \epsilon\}$$

Definition 2 (K Closest Pairs): Given two datasets P and Q where $P = \{p_1, p_2, ..., p_n\}$ and $Q = \{q_1, q_2, ..., q_n\}$ and a number $K > 0$, $K \in \mathbb{N}$, the K Closest Pairs Query is an

ordered subset of $P \times Q$ denoted by $KCPQ(P, Q, K)$, where:

$$KCPQ(P,Q,K) = \{(p_1,q_1),...,(p_K,q_K) : (p_i,q_i) \in P \times Q, (p_i,q_i) \neq (p_j,q_j)\, \forall 1 \leq i,j \leq K\}$$

and

$$dist(p_1,q_1) \leq, ..., \leq dist(p_K,q_K) \leq dist(p,q)\ \forall (p,q) \in P \times Q - KCPQ(P,Q,K)$$

The distance function *dist* is a distance metric defined on points in the data space. A very commonly used such distance function is the Euclidean distance, but depending on the application, other functions may be more appropriate.

3.3.2 Data partitioning

An important step, towards answering a query in a parallel and distributed environment, is proper partitioning of the datasets. Data partitioning improves the query performance in two ways [1]:

1. partitioning the data into smaller units enables processing of a query in parallel and
2. I/O can be significantly reduced by only scanning a few partitions that contain relevant data to answer the query.

In the case of the KCPQ, (2) is not applicable; in order to answer the query, we have to search pairs of points from the whole dataset. Therefore, in the case of KCPQ, the most severe obstacle one has to face, in datasets partitioning, is data skewness. In most real-world cases, data is not uniformly distributed in a dataset. Using partitioning techniques such as uniform grid [16] very often leads to partitions that contain much more objects than others, a fact that in a parallel system may prevent proper load balancing and therefore delay the computation of the final result. There are many alternative strategies that can be found in the literature aiming to deal with the skewness problem. Most of them require the construction of a spatial data structure, which allows the queries about spatial relationships of objects to be answered. The simplest spatial data structure is the uniform grid, but, as already said, it very often leads to bad performance in the case of non-uniformly distributed data. This observation has led to more elaborate partition schemes, many of them being generalizations of binary search trees.

A quad-tree [70] is a non-uniform subdivision of area where a region is split into four quadrants by two axisaligned dividing lines. The decomposition proceeds until a certain property is met, i.e., each quadrant contains less than a predefined number of points.

An R-tree [30] is a hierarchical data structure derived from the B-tree [39]. Data objects are represented by their enclosing MBRs, which are grouped into larger nodes hierarchically until the root node of the tree. Each leaf node contains the actual objects, and can store a certain, predefined number of objects.

In most cases within the context of parallel and distributed frameworks, the creation of these hierarchical data structures is based on reading a random sample from the input file and using this sample to partition the whole space.

Splitting into strips [1] is a data partitioning technique used in several applications of parallel systems on various disciplines [5]. Partitioning into strips is lightweight, since it does not need extra computing time to construct and maintain complex data structures that may not be necessary in the case of the query we are dealing with. Most spatial-aware systems based on Spark use a twofold indexing scheme. First a global index is created, then data is shuffled to workers and finally each worker creates its own local index. Albeit this is time consuming, it usually rewards when used, but obviously this depends, among others, on the query itself. In our case, for the KCPQ (like other spatial join queries), it is enough to utilize all cluster resources and achieve a high degree of parallelism, in contrast to other queries, like the range query [16]. This is due to the fact that, in the KCPQ, pairs of the solution are usually dispersed all over the search space and indexing would not accelerate computation.

But, still, we need to slice a dataset into N parts (strips) in a way that they will contain roughly equal number of points. This probably means strips of unequal width corresponding to ascending intervals along one of the dimensions (x axis dimension is assumed, w.l.o.g.) and is achieved by the following procedure: first, we take a sample of size M, then we extract all x-axis values of the sample into an array $xCoord$ and perform quicksort on it. We calculate step as the quotient M/N and finally the split (partitioning) points are extracted as certain x-values of $xCoord$, as it can be seen in Algorithm 1. This partitioning is then applied to the whole dataset. Our experiments have shown that this technique, based on function *takeSample()* of Spark works quite well, and produces strips of variable width, containing number of points that vary within less than 10% between them.

Algorithm 1 $createSplitPoints(dataSet, N)$: find split points for a single dataset

Input: $dataset, N$

1: $sampledData \leftarrow dataSet.takeSample(noReplace, M)$

2: $xCoord \leftarrow sampledData.map(point \Rightarrow point.x)$

3: $quicksort(xCoord)$

4: $step \leftarrow M/N$

5: $splits \leftarrow Array(xCoord(step), xCoord(step * 2), , xCoord(step * (N - 1))$

3.3.3 Plane-Sweep computation of the KCPQ

Regardless of any other detail, one dealing with the KCPQ eventually needs a method to actually compute it. This also holds for parallel systems such as Spark. Some of the most important techniques for the KCPQ computation are the Plane-Sweep family of algorithms. In our implementation we use the classic Plane-Sweep algorithm [64]:

1. sort in increasing order the entries of the two point sets, P and Q, based on the coordinates of one of the axes (e.g. Y).

2. initialize two pointers p and q to point to the first entry for processing of each sorted array of points. Set the $reference$ point be the one with the smallest y-value pointed by one of these two pointers, for example, suppose $reference = p$.

3. pair up the $reference$ point with all the points stored in the other sorted array of points from smaller to larger y-value, satisfying the following condition (δ is the distance of the K-th closest pair found so far):

$$q.y - reference.y < \delta \tag{3.1}$$

4. increase to the next entry the pointer of the array that $reference$ points to (in our example p is increased) and update the $reference$ point with the point of the next smallest y-value pointed by one of the two pointers.

5. repeat steps 3-4 until one of the sorted arrays of points is completely processed.

We have implemented two slight variations of the above algorithm, namely $fy\_bounded$ and fy, depending on whether we already know an upper bound (e.g. from previous computation) or not. Both are being passed as functions to the workers via Spark engine. In both,

we maintain a local $maxHeap$ of type $Tuple3[Distance, Point, Point]$ and with fixed size K. At any point of the computation, the $maxHeap$ stores the best K pairs that have been locally found so far. Initially, $maxHeap$ is empty.

In the case of fy, workers directly store the first K pairs they examine and their corresponding distances. Then, for each consecutive pair it is tested whether the previous inequality holds or not. The value of δ is h, the head of $maxHeap$.

In the case of $fy\_bounded$, the first K pairs are also stored into $maxHeap$. The inequality is tested for every consecutive pair of points, where $\delta = min(h, bnd)$.

Only for pairs of points where the inequality is true, their distance is being fully computed and compared to the distance value stored at the head of $maxHeap$. If the new distance is smaller, the head is extracted (discarded) and the new pair is inserted, otherwise the pair is discarded.

After computation in all workers finishes, all partial results are collected on the driver side and by taking the K ordered with smaller distance pairs, we have the K closest pairs between the datasets that were submitted.

3.4 The Slices algorithm

In order to efficiently compute the k closest pairs query in the Spark context, there are three main tasks, our algorithm has to deal with:

1. Find a good bound for the KCPQ and broadcast it to Workers. This will lead to good pruning criteria, on both Driver and Workers contexts.

2. Partition the data, by setting a proper indexing, and check all pairs of partitions so that all eligible pairs of points from P and Q will be considered, thus preventing any loss of the optimal solution.

3. Use a fast algorithm to compute KCPQ in the Workers context, collect the results and select the top k, having the smallest distance.

The method for answering the KCPQ, as presented in [49] consists of four steps (decribed in the following subsections) that cover all the above mentioned tasks.

3.4.1 Lower bound computation

We initially compute an upper $bound$ for the K-th closest pair. We use the Spark-provided function *sample* to create two RDDs containing samples from each one of the two datasets. We use a sample ratio of $f = 0.001$ on each dataset.

In order to obtain a good upper bound, we partition the sampled RDDs in a manner so that points with close x-axis values fall into the same partition.

Partitioning each of the two sampled datasets into n strips of unequal width is done by calculating the border (separation) x-axis points, separately for each sampled dataset. Both samples are collected to the Driver and their x-values are extracted and stored in two sorted arrays sP and sQ. The predefined number n and the sizes of the two arrays obtain the indices of each array that contain the separation x-points $PSep$ and $QSep$. Value $stepP$ is $sP.size/n$ and value $stepQ$ is $sQ.size/n$. The two arrays $PSep$ and $QSep$ are merged into a sorted array $PQSep = PSep + +QSep$ that contains the separation x-axis points applicable on both sampled RDDs. This array is passed to Spark and all points in both the sampled RDDs are being assigned the proper keys.

A $join$ is performed between the two keyed RDDs, creating an RDD of type $(Int, (Point, Point))$ that is mapped to an RDD[$Double, (Point, Point)$] where the Double presents the distance (Euclidean in our case) of every pair of points in the joined RDD. By using the *takeOrdered* function of Spark, we select the k pairs with smaller distance. The K-th distance is our upper $bound$. This means that in the following steps we do not need to seek for pairs that have their distance greater than this $bound$ and consequently we do not need to examine pairs that have their x-axis (y-axis can also be used) distance greater than $bound$.

3.4.2 Datasets partitioning

After the bound computation from samples, both datasets are separately divided into a, user defined, number of n strips. Partitioning each of the two datasets into strips of unequal width is done by calculating the border (separation) points from samples, separately for each dataset by the same procedure shown in the previous step.

The sample size m is set to $dataset.size/n$ where n is the number of desired partitions. Each point from the sample is mapped to its x-axis coordinate and all 1D points are collected to the Master node as an Array A[m]. Sorting is performed on the array and the number of predefined strips n determines the $step = m/n$ on the indices of the array that contain

the separation x-points. The actual x-values, depicted in Fig. 3.1, are $X1 = A[step], X2 = A[2 \cdot step], ..., Xn = A[(n-1) \cdot step]$.

Each subinterval contains approximately m/n points and therefore the projection of this upon the whole dataset creates strips with approx. equal size of points. The separation points, shown in gray in Fig. 3.1, are being used on the whole dataset, to split it into n strips. The partitioning is being done a function *xPartitionSpace*, which assembles the envelopes of the strips

$$Env[minX : X1, minY : maxY],$$

$$Env[X1 : X2, minY : maxY], ...,$$

$$Env[Xn - 1 : maxX, minY : maxY]$$

The function returns an array of type (Int, Env), by assigning consecutive integers to each partition presented as Envelope. Actual partitioning is done by passing a function to Spark with parameter the array of Envelopes and each Worker scans the dataset and assigns the proper key to each point.

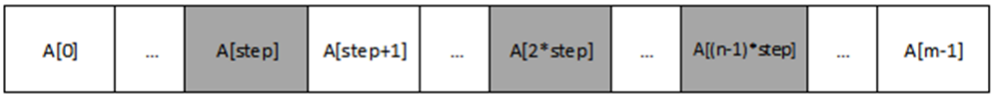

Figure 3.1: Selection of splitting points.

3.4.3 Classification of strips

The third step of the algorithm uses $bound$, the distance of the K-th closest pair, which was computed from the sample as described in step 1, to classify all pairs of strips from the two datasets into two categories, *Eligible* and *non-Eligible*.

This is accomplished by first finding the relative position between each pair of strips. The criterion used to derive the relative position is based on the relation of the minimum and maximum value of the x-coordinates of the strips. In Fig. 3.2 all possible cases are being depicted.

In the case of overlapping pairs, as it happens with W and B, the expression

$$(W.x1 < B.x2 \ \&\& \ W.x2 > B.x1)$$

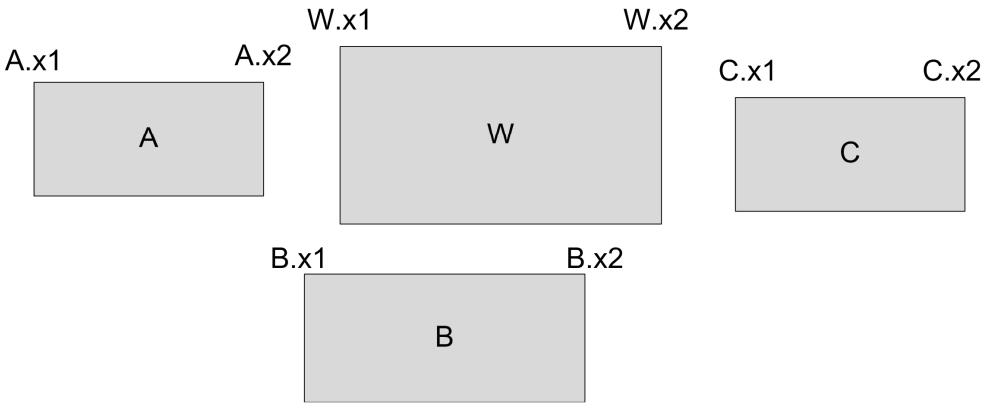

Figure 3.2: Relative position of strips.

evaluates to *true*.

If it evaluates to *false*, then there are two cases, either strip is on the left of W (strip A), and $W.x1 > A.x2$, or strip is on the right of W (strip C) and $W.x2 < C.x1$.

Eligible and *non-Eligible* categories are defined as follows:

1. *Eligible* pairs consist of all pairs of strips containing points that may contribute to the final query answer. The eligible pairs may be:

   i Pairs that overlap. As seen in Fig. 3.3, strip P1 from P overlaps with strips Q1 and Q2 from Q.

   ii Pairs that do not overlap, but have their x-axis distance smaller than $bound$. In Fig. 3.3 the x-distance between P1 and Q3 is $d1 < bound$.

2. *non-Eligible* pairs consist of all pairs of strips that do not overlap and have their x-axis distance greater than $bound$. The contained points cannot contribute to the final query answer. Such a pair is P1 and Q4, two strips that have their x-distance $d2 > bound$, as shown in Fig. 3.3. The same holds for every consecutive Q-strip after Q4.

In the case of non-overlapping but yet eligible pairs (case 1-ii, above), not all points from both strips need to be considered in the forthcoming step, since pruning can be performed to reduce both strips to these points that their x-axis distance from each other is smaller than $bound$.

For example, in the case of pair P1, Q3, we use the *filter* function of Spark to reduce Q3 to these points (Q3.x, Q3.y) such that $Q3.x - P1.max < bound$ and also reduce P1 to these points (P1.x, P1.y) such that $Q3.xmin - P1.x < bound$.

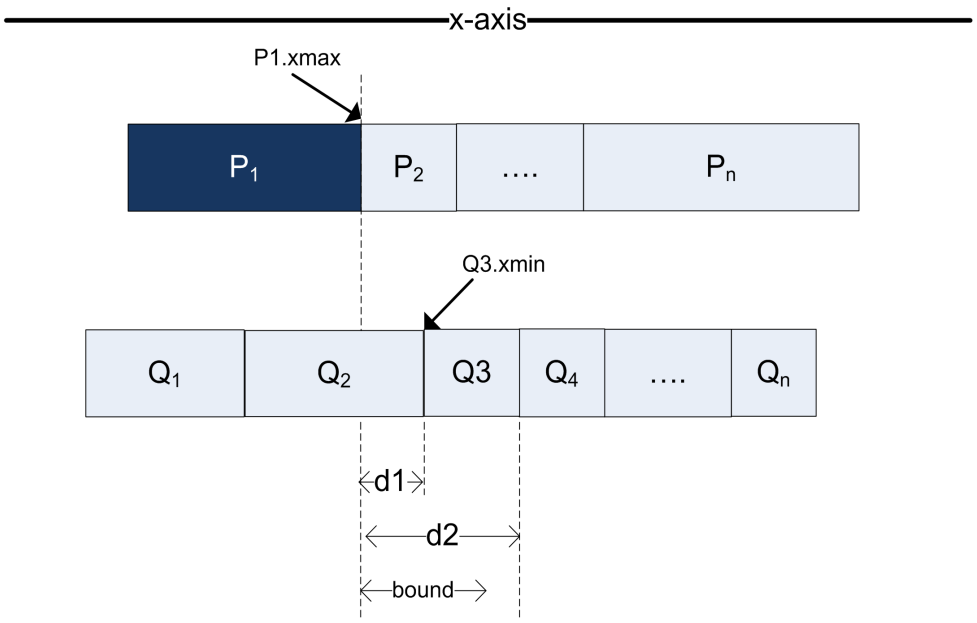

Figure 3.3: Eligible strips and filtering of non overlapping strips.

3.4.4 KCPQ computation

Having located all the eligible pairs, we create two new RDDs, with all possible pairs of strips containing points that may contribute to the answer of the query. This is done by duplicating, as needed, the points from strips of each dataset that have to be accounted with points from strips from the other dataset (a union) and assigning proper keys.

In the final step, a Plane-sweep algorithm is applied within each eligible (and filtered) pair of strips from P and Q for calculating K Closest Pairs and storing the result in a, separately for each partition, maximum binary heap (max-Heap). Previously, the $bound$ (computed in step 1) has been broadcasted to all workers to use it as stop condition for the Plane-Sweep algorithm. Taking the first (sorted on distance) K tuples with the smaller distances, yields the final (and exact) solution.

3.5 Experimental evaluation of the Slices algorithm

To evaluate the performance of our algorithm, we used the following three big real 2d datasets from OpenStreetMap [17]: $WATER$ resources consisting of 5,836,360 line segments, $PARKS$ (or green areas) consisting of 11,504,035 polygons and $BUILDINGS$ of the world consisting of 114,736,611 polygons. To create sets of points, we used the centers of the Minimum Bounding Rectangles (MBRs) of the line-segments from $WATER$ and the

centroids of polygons from $PARKS$ and $BUILDINGS$.

All experiments were conducted on a cluster of 5 nodes. Each node has 4 vCPUs running at 2.1GHz, with a total of 16GB of main memory per node, running Ubuntu Linux 16.04 operating system. Spark 2.0.2 running on Hadoop 2.7.2 Distributed File System (HDFS) was used as our parallel computing system. The block size of HDFS was 128 MB. Of the 5 computing nodes, one was running the NameNodes for Hadoop and Master for Spark, while the remaining four (4 nodes x 4 vCPUs = 16 vCPUs) were used as HDFS DataNodes and Spark Worker nodes. Java openjdk ver. 1.8.0 and Scala code runner ver. 2.11 were used.

All datasets are text files stored in HDFS. Each line contains an index and a pair of coordinates. We used the *textFile()* function of Spark to import the data, and set the numPartitions parameter to 4. Typically, Spark creates one partition for each block. We can increase the number of partitions by passing a larger value but it is not possible to have fewer partitions than the blocks of each file.

We measured total execution time (i.e., response time) in seconds (sec) that expresses the overall CPU, I/O and communication time needed for the execution of each query.

We varied sample fraction (values used: 0.01, 0.001, 0.0001), the number of closest pairs K (values used: 1, 10, 100, 1000, 10000) and the number of strips per dataset (values used: 16, 32, 64, 80). We tested all possible combinations between the three datasets ($PARKS \times WATER$, $BUILDINGS \times WATER$, $BUILDINGS \times PARKS$). In the following, we present a representative portion of the results.

In Figure 3.4, we present the results for the $PARKS \times WATER$ combination, for K = 10, using different combinations of n (number of strips) and f (sample fraction). As one can see, there is a tradeoff between total execution time and the time taken in order to sample the datasets and compute the value of $bound$. If we take a small fraction of the datasets as sample, the bound we compute is not tight enough, therefore leading to increased KCPQ computation time. The larger the fraction of dataset we sample, the better (lower) is the upper bound we obtain. But if we surpass a certain fraction, then the computation of $bound$ in the sample dominates the total computation time.

Studying the results of the above experiment leads us to the observation that a fraction of 0.001 is a good selection for the rest of our experiments.

In Figure 3.5, we present the results for the $PARKS \times WATER$ combination, for all K values, using 16, 32, 64 and 80 strips per each dataset. Initially, we ran each experiment

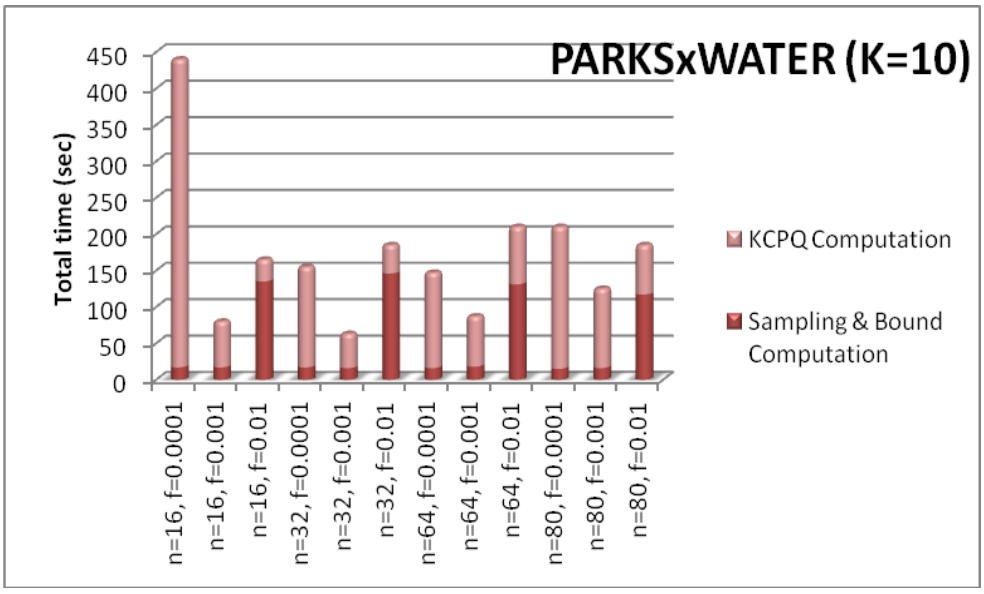

Figure 3.4: Effect of sample fraction.

independently from the others. We faced a problem, though. Phase two (the KCPQ computation) relies on the value of $bound$ that is computed in phase one. Since phase one uses a randomly selected sample, $bound$ is likely to be different in each experiment. In order to be able to extract better and comparable results, we used the following procedure for our second experiment: having taken into consideration that phase one is independent from phase two, we conducted the first phase of the experiment (K=1, n=16, $fraction$=0.001) and saved the calculated value of $bound$. In all consecutive phases of the experiment, the bound was computed as usual, but we used the value we found in the first phase of the experiment instead.

As we observe, $n = 32$ strips seems to be the optimal partitioning size for PARKS and WATER datasets, although $n = 16$ gives similar results. As K increases from 1 to 10,000, execution time is hardly affected, in some cases showing a tendency to increase slightly, as expected.

We conducted our third experiment in order to see to what extent the value of $bound$ affects the running time of the algorithm. We used a value for $bound$ with an order of magnitude 10 times greater than the one previously used. Time for sampling and bound computation was taken into account when counting total running time. Figure 3.6 presents the running times compared to the ones that were measured in the previous experiment.

From the above comparison, we conclude that the value of $bound$ is more significant than the number of strips and the number of partitions provided to Spark as well.

In Figure 3.7, we present the results for the $BULDINGS \times WATER$ combination, for

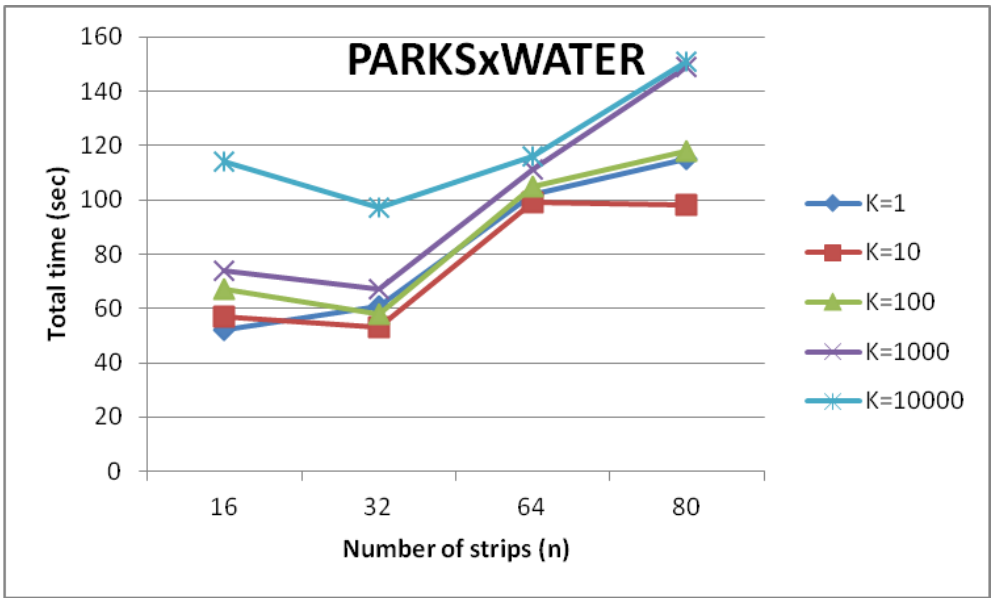

Figure 3.5: KCPQ ($PARKS \times WATER$).

several K values using 8, 16, 32 and 64 strips per each dataset (once again *bound* was set to a constant value for all cases, to an order of e-05). We observe than in the case of $BULDINGS$ the algorithm gives better results for a lower number of strips than in the case of $PARKS$.

We believe that this has to do with a combination of the characteristics of the multi-parametric system we study (hardware, HDFS, Spark, our algorithm). The combination of available cores, starting partitions, Spark partitioning procedures, number of strips that lead

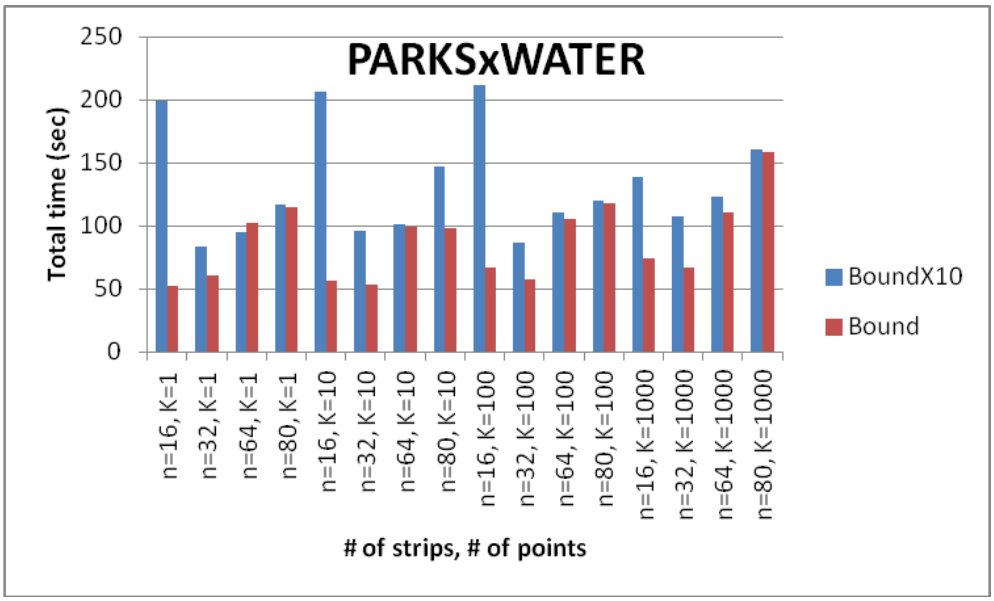

Figure 3.6: Effect of lower bound.

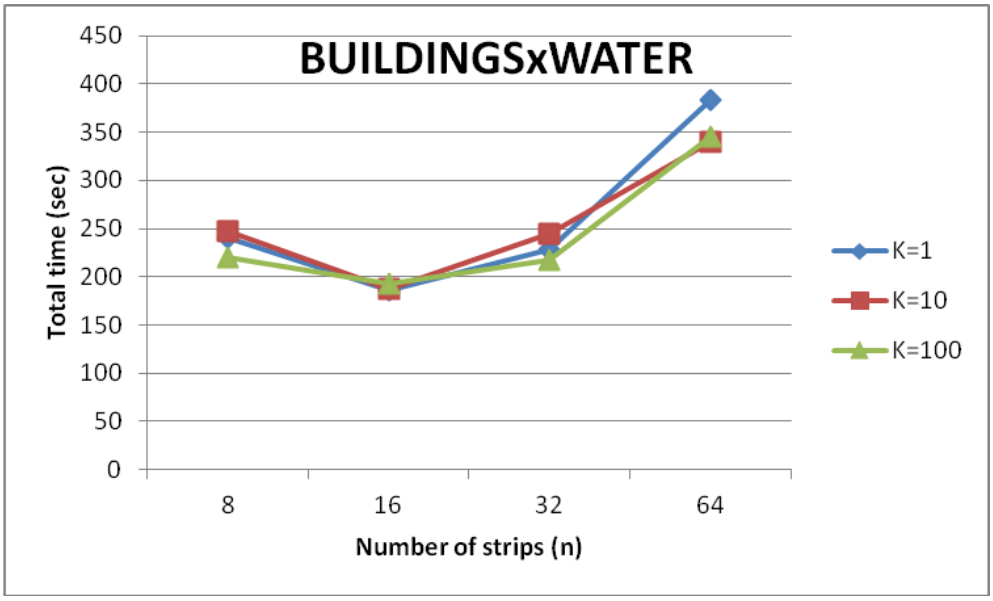

Figure 3.7: KCPQ ($BUILDINGS \times WATER$).

to a number of eligible pairs, results to an increased number of partitions that in the cases of larger n (strips) overwhelms the computing cluster.

In all previously described experiments, both datasets are being sliced into strips along x-axis (y-axis can also be used). Then, within each partition created by the eligible pairs of points from P and Q, Plane-Sweep is applied along the other axis, in our case the y-axis. It is possible to slice the strips and sweep along the same axis (Figure 3.8).

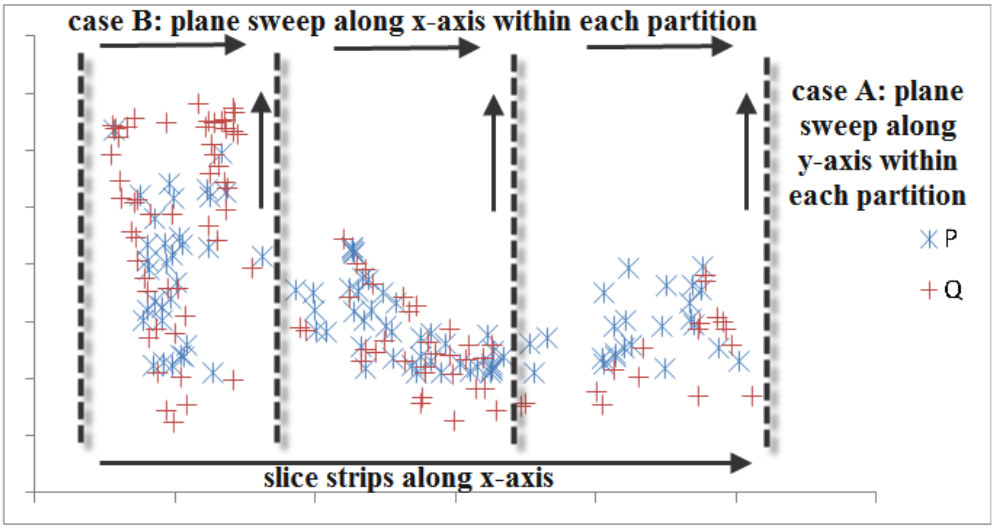

Figure 3.8: Strips Slice & Plane-Sweep cases.

In order to check which choice is better (slicing and sweeping along the same or different axes), we conducted our next experiment. We used the $BULDINGS \times PARKS$ combina-

tion with $n = 8, 16, 32$, $K = 10$ and fraction $f = 0.001$.

We ran each combination three times, used the average time and the results are being presented in Figure 3.9.

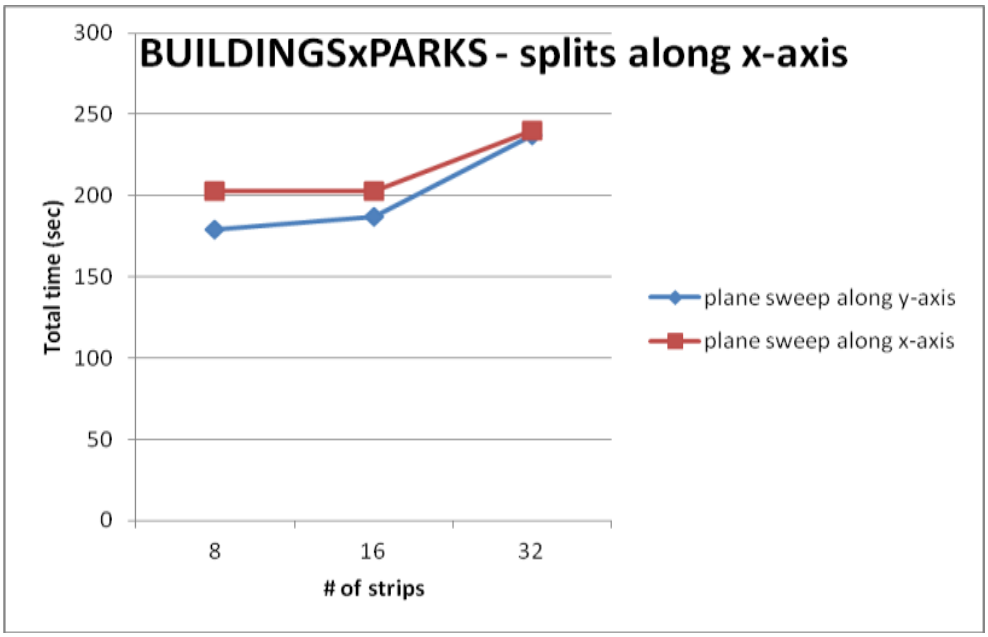

Figure 3.9: Split axis vs Plane-Sweep axis.

The results seem to lead us to the conclusion that "crossing" the axes for slicing and sweeping is more efficient than working on the same axis. This observation is clearer in the cases of smaller strips number, when the algorithm gives the best results.

Although this is consistent with other observations we have made during our experiments, we believe that it needs further investigation, an action we plan to take in the near future.

3.6 The BST slices algorithm

As already described, the method of [49] (Slices algorithm) uses a lightweight partitioning scheme that splits the datasets into strips. In [50] we maintained the fundamental concept of partitioning into strips, but implemented two additional partitioning schemes, influenced by the Quad-tree and R-tree principles. Since the partitioning derives from Quad-tree and R-tree, we call them Q-split and R-split partitionings, respectively.

3.6.1 Outline of the partitioning procedure

As in the Slices method [49], partitioning of datasets is performed along an axis, which in our case is the x-axis. Both Q-split and R-split partitions use a Binary Search Tree (BST) to store the splitting points (therefore, the resulting variation of the Slices algorithm is called "BST slices algorithm"). A *class* BST extends Scala collection *mutable.Map* and is used to implement the Binary Search Tree. Each Node of the BST is defined as a Scala class of type *class Node(key, value, left: Node, right: Node)*.

Parameter key is of type $Double$ and is the actual splitting x-coordinate. Parameter $value$, in our case is of type $(Int, Double)$ where the Int in $value$ is showing the level of decomposition and the $Double$ is the splitting x-point. Parameter $value$ can store any kind of information and we intend to use it in further experimentation, for example, one can store sizes of every child area and use it to make decisions regarding the computation. The BST class is equipped with a function *compare* that is used to compare the keys of the points to be added.

Every new node is being added by a function $+$ that recursively scans the BST, locates its correct position and adds it to the tree.

By sampling the dataset with ratio f, we map the points to their x-coordinates and collect the results to the Driver program, in an array DS. The "middle" point selection depends on the criterion used to partition the array and afterwards the dataset.

In the case of R-split, $middle$ is selected as the point that leaves equal number of points on the two sub-intervals, while in the case of Q-split $middle$ is the point that splits the interval into two sub-intervals with the same x-axis width. This means that in the first case $middle$ is an actual x-point from the dataset, while in the latter it may be not since it is computed as the median of each interval.

Each dataset is partitioned separately, as happens in the Slices method. In both partitioning schemes, we set two parameters $PCapacity$ and $QCapacity$, which represent the maximum number of points that a node can store. In contrast to the Slices method, no additional sampling is performed, as partitioning is done by using the samples taken for upper bound computation.

3.6.2 R-split

We perform Quicksort on array DS (the full sample) and locate the first splitting point of the array, named $middle$. In the case of R-split, initial value of $middle$ is set as the quotient

$DS.length/2$. The value of capacity, idx (an integer counting the level of decomposition), TR (an empty BST), DS (the array with x-points from sample) and the splitting point $middle$ are passed to a function *partitionEqualSize* that uses recursion to create and return the BST with the splitting points. Algorithm 2 outlines the procedure.

Algorithm 2 R-SPLIT

```
1:  function PARTITIONEQUALSIZE(capacity, idx, TR, DS, middle): BST
2:      function PS(capacity,idx, TR, DS, middle): Int
3:          id ← idx + 1
4:          TR+ = middle → (id, middle)
5:          FPLeft ← DS.filter(x ⇒ x < middle)
6:          FPRight ← DS.filter(x ⇒ x >= middle)
7:          FPLeftSize ← FPLeft.length
8:          FPRightSize ← FPRight.length
9:          FL ← FPLeftSize/f                          ▷ f is the sampling ratio
10:         FR = FPRightSize/f
11:         if FL > capacity then
12:             ml ← FPLeft(FPLeftSize/2)
13:             id ← ps(capacity, id, TR, FPLeft, ml)
14:         if FR > capacity then
15:             mr ← FPRight(FPRightSize/2)
16:             id ← ps(capacity, id, TR, FPRight, mr)
17:         return id
18:     ps(capacity, idx, TR, DS, middle)
19:     return TR
```

3.6.3 Q-split

The initial value of $middle$ is computed as the median of the maximum and minimum x-values of each dataset. For example, in the case of dataset P, middle is the quotient $(P.maxX + P.minX)/2$. Function *partitionEqualwidth* is similar to *partitionEqualSize*, with two differences. Line 12 is replaced with $ml = (FPLeft.maxX + FPLeft.minX)/2$ and Line 15 is $mr = (FPRight.maxX + FPRight.minX)/2$.

In both R-split and Q-split, the splitting points are retrieved from the BST by using an

inorder traversal, being utilized by means of a properly designed iterator.

3.7 Experimental evaluation of the BST slices algorithm

To evaluate the performance of our methods, we used three large real 2d datasets from OpenStreetMap [17]: $WATER$ resources consisting of 5,836,360 line segments, $PARKS$ (or green areas) consisting of 11,504,035 polygons and $BUILDINGS$ of the world consisting of 114,736,611 polygons. To create sets of points, we used the centers of the Minimum Bounding Rectangles (MBRs) of the line-segments from $WATER$ and the centroids of polygons from $PARK$ and $BUILDINGS$.

All experiments were conducted on a cluster of 9 nodes. Each node has 4 vCPUs running at 2.1GHz, with a total of 16GB of main memory per node, running Ubuntu Linux 16.04 operating system. Spark 2.1.1 running on Hadoop 2.7.2 Distributed File System (HDFS) was used as our parallel computing system. The block size of HDFS was 128 MB. Of the 9 computing nodes, one was running the NameNodes for Hadoop and Master for Spark, while the remaining eight (8 nodes x 4 vCPUs = 32 vCPUs) were used as HDFS DataNodes and Spark Worker nodes. Java openjdk ver. 1.8.0 and Scala code runner ver. 2.11 were used.

In all experiments, the data sets P and Q are in the form of text files formatted in columns with a separator (in our case a tab), one point per line (x, y coordinates). We also make the assumption that the two data files have already been stored in the HDFS, at an earlier phase without being subject to any kind of processing (e.g., sorting, indexing, and so forth). The datasets are read by using the *textFile* function of Spark. Each dataset is presented as an RDD[$Point$], where $Point$ is of type $Tuple2[Double, Double]$. In all experiments the number of closest pairs is set to $K = 10$.

3.7.1 Speedup of the Slices algorithm

First we measure the total computing time for 4 and 8 computing nodes in the case of $BUILDINGS \times PARKS$ (Fig. 3.10) and $PARKS \times WATER$ (Fig. 3.11) of the Slices method [49], varying the number of the preset partitions.

We measured total execution time (i.e., response time) in seconds (sec) that expresses the overall CPU, I/O and communication time needed for the execution of each query.

Let $T4$ be the execution time for four nodes and $T8$ be the execution time for eight nodes.

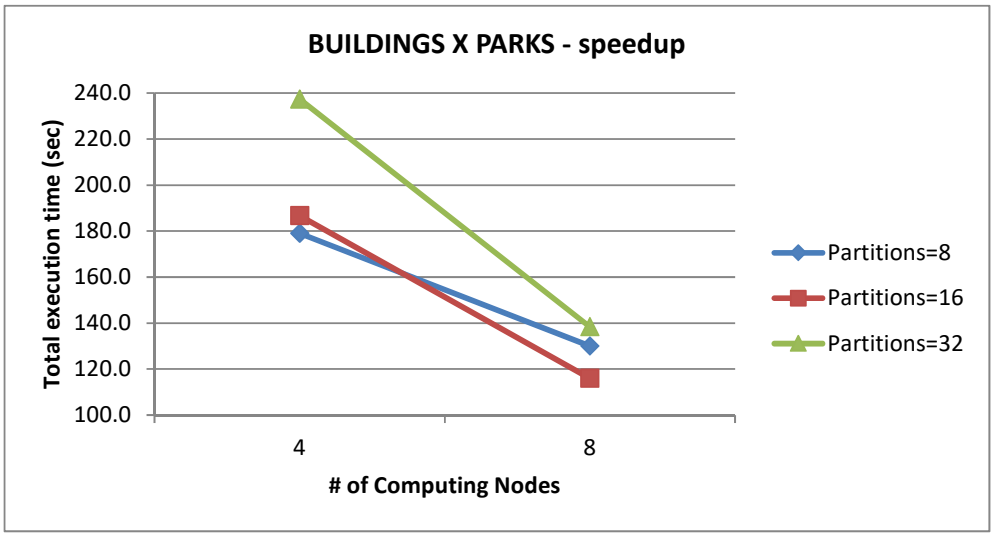

Figure 3.10: KCPQ($BUILDINGS \times PARKS$), Nodes = 4, 8.

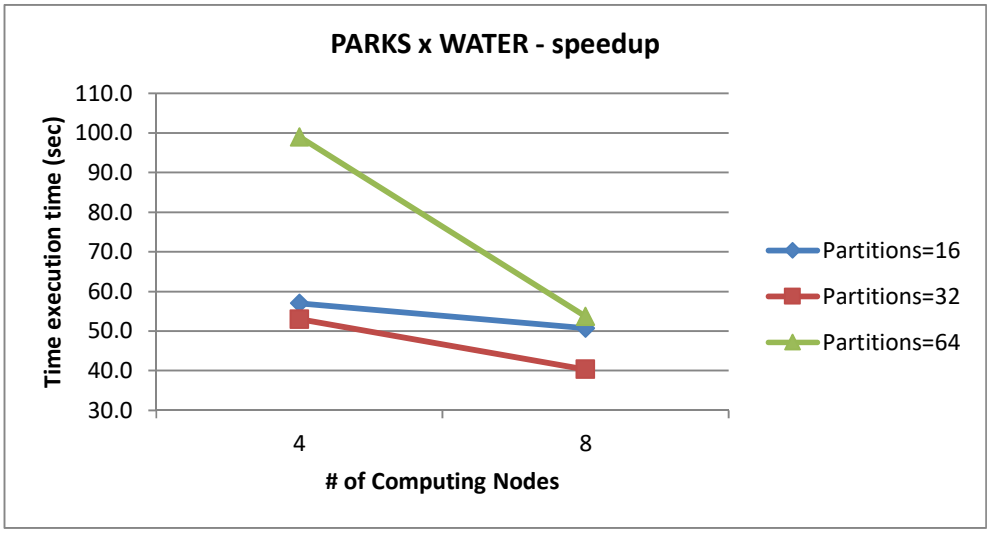

Figure 3.11: KCPQ($PARKS \times WATER$), Nodes = 4, 8.

The speedup is defined by $T4/T8$. We observe that execution time decreases more rapidly in the cases of larger number of partitions. This is expected since a larger number of partitions leads to more Spark jobs being more efficiently managed when a larger number of computing nodes is available. In the case of $BUILDINGS \times PARKS$, best speedup (1.7) is measured at 32 partitions. Best response time is measured with 16 partitions, where the speedup is about 1.6. In the case of $PARKS \times WATER$, best speedup (1.84) is measured at 64 partitions. Best response time comes when number of partitions is set to 32, and the speedup is 1.3.

3.7.2 Performance of the improved Slices method

The second experiment deals with an improvement we have made to the Slices algorithm. By refactoring the code, we have removed the second sampling that is used by the Slices method in order to partition the datasets. Instead, we use the sample already taken for the upper bound computation. The results are being presented in Fig. 3.12 and Fig. 3.13.

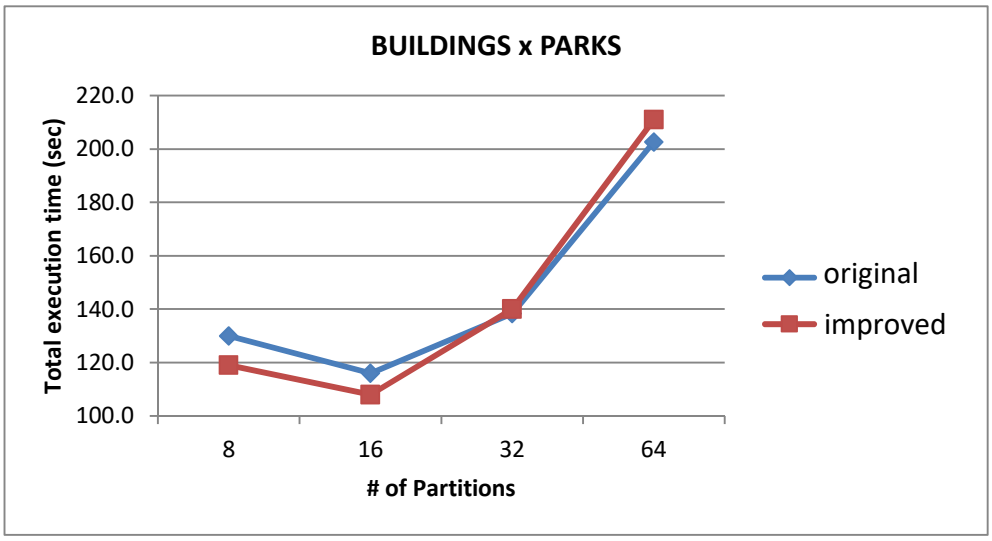

Figure 3.12: $BUILDINGS \times PARKS$. Original vs improved Slices method.

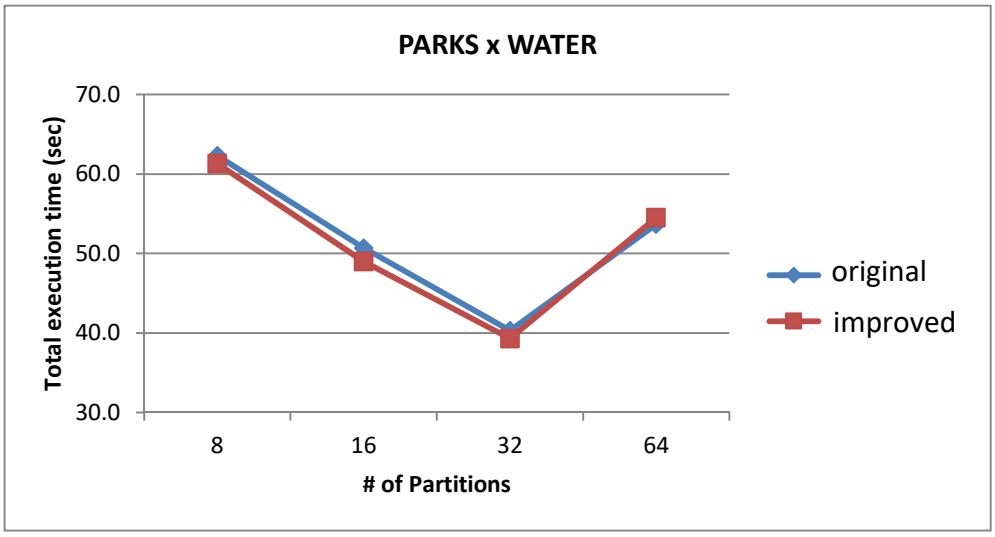

Figure 3.13: $PARKS \times WATER$. Original vs improved Slices method.

It was observed that a sample ratio of $f = 0.001$ is adequate to efficiently partition the datasets. In general, the system runs faster, mostly in the case of larger datasets where a relatively small fixed number of partitions are used. This is reasonable, since in the Slices method

the sample size is computed as the quotient of dataset size and number of partitions; therefore as partitions number decreases it results in an increase of the sample size and therefore an increase in upper bound computation time.

We use this strategy (a single sampling per dataset) in all following experiments, and use only one very small sample, with ratio 0.001 for computing both upper bound and partitioning x-axis points.

3.7.3 R-split and Q-split performance

In the third experiment we test the performance of the method in the case of both R-split and Q-split. We have run several experiments with various combinations of capacity for each dataset. Table 3.1 presents the results of the experiment regarding R-split for $BUILDINGS \times PARKS$ and $PARKS \times WATER$ datasets. As it can be observed, the new partitioning scheme provides much more freedom in fine tuning the system, since we now don't need to preset the number of partitions, but rather set capacities and have the system compute the number of partitions that meet the settings.

In Fig. 3.14 and Fig. 3.15 we compare the results measured for the improved version of [49], as described above, with the best obtained by using the R-split, subject to the same number of partitions between the two methods. It can be deduced that an R-split achieves better running times compared to both the original and the improved version of the Slices method especially when dealing with larger datasets.

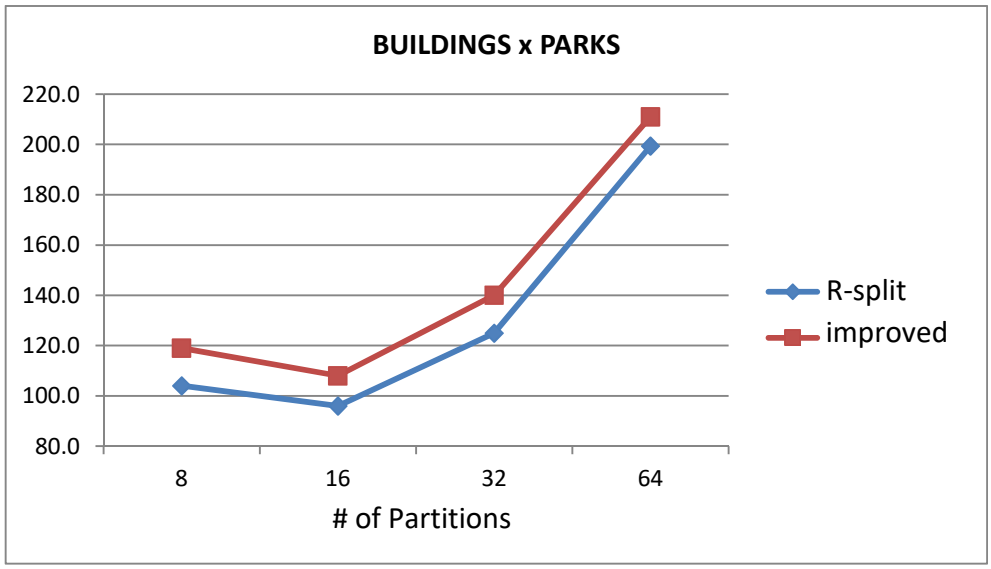

Figure 3.14: $BUILDINGS \times PARKS$. R-split vs improved.

Table 3.1: R-Split results for the KCPQ.

capacity (millions of points)		Derived partitions #		Eligible	Time
BUILDINGS	PARKS	BUILDINGS	PARKS	pairs #	(sec)
23	2.3	8	8	15	111.3
14.5	1.5	8	8	15	104
14.5	1.4	8	16	23	126
14	1.4	16	16	31	104.3
12	1.4	16	16	31	101.3
10	1.4	16	16	31	96
10	0.75	16	16	31	98
10	0.7	16	32	47	138
8	1.4	16	16	31	99
8	0.75	16	16	31	95.7
8	0.7	16	32	47	138.7
6	1.4	32	16	47	104.3
6	0.7	32	32	63-64	125
4	1.4	32	16	47	109
2	1.4	64	16	79	131.3
2	0.35	64	64	127-128	199.3
capacity (millions of points)		Derived partitions #		Eligible	Time
PARKS	WATER	PARKS	WATER	pairs #	(sec)
1.4	0.7	16	16	31	49.7
1.4	0.35	16	32	47	39
2.8	1.4	8	8	15	61
0.7	0.35	32	32	63	40.3
0.7	0.7	32	16	47	40.7
0.35	0.175	64	64	127-128	53

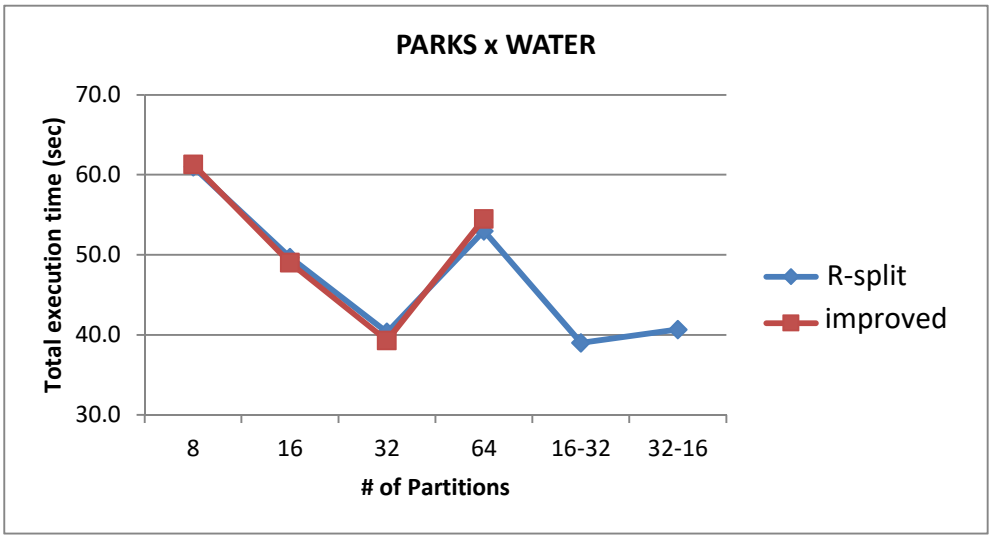

Figure 3.15: $PARKS \times WATER$. R-split vs improved.

Table 3.2 presents the results of the experiment regarding Q-split for $BUILDINGS \times PARKS$ and $PARKS \times WATER$ datasets. In contrast to R-split, using Q-split leads to a, more or less, worse performance in almost every case when compared to both R-split and the improved method in [49]. Another observation is that Q-split leads to a number of derived partitions that varies more widely than in the case of R-split, where the number of partitions is mostly constant.

The fourth experiment aims to check the quality of R-split and Q-split partitions. As it can be seen, R-split partitioning results in strips with uniformly allocated number of points (Fig. 3.16 and Fig. 3.17). Furthermore, the selection of a small sample with ratio 0.001 is proved to be sufficient for this purpose.

On the other hand, a Q-split partition results in strips with unequal number of points (Fig. 3.18 and Fig. 3.19).

The fifth experiment aims to enlighten the differences measured in execution times regarding different $PCapacity$ and $QCapacity$ settings, for both R-split and Q-split partitions.

As already mentioned, after locating all the eligible pairs of strips, two RDDs are derived containing all possible pairs of strips with points that may contribute to the answer of the query. These RDDs are created as a union of properly keyed points from strips of each dataset that have to be accounted with points from strips from the other dataset.

In the (usual) case that a strip from P has to be combined to more than one strip from Q, then P is being duplicated (and filtered in the case of non overlapping strips) as needed and proper keys are being assigned. This means that the derived RDDs upon which the actual

Table 3.2: Q-Split results for the KCPQ.

capacity (millions of points)		Derived partitions #		Eligible	Time
BUILDINGS	PARKS	BUILDINGS	PARKS	pairs #	(sec)
23	2.3	8	8	17	141.7
14.5	1.5	15	11-14	25-28	122.3
14.5	1.4	14-15	12-15	25-29	121.3
14	1.4	15-16	14-15	29-30	118.3
12	1.4	16	15-16	30-31	120
10	1.4	18-19	14-15	32-33	116.7
10	0.75	18-19	22-24	41-42	132.3
10	0.7	19	24	42	135.7
8	1.4	22	15	36	118.3
8	0.75	22-23	22-24	44-45	127.3
8	0.7	22-23	24-25	45-47	136
6	1.4	28	14	41	116
6	0.7	28	23-27	50-54	127.7
4	1.4	46	14-15	59-60	130.7
2	1.4	84-86	14-15	98-99	205.3
2	0.35	86-87	47-48	134	230

capacity (millions of points)		Derived partitions #		Eligible	Time
PARKS	WATER	PARKS	WATER	pairs #	(sec)
1.4	0.7	14-15	15-16	28-30	35
1.4	0.35	14-15	26-27	40-41	39.3
2.8	1.4	8	8-9	16-15-15	59.7
0.7	0.35	24-25	27-28	50-52	41
0.7	0.7	24-25	16-17	39-41	33.3
0.35	0.175	48-49	55-56	102-104	49

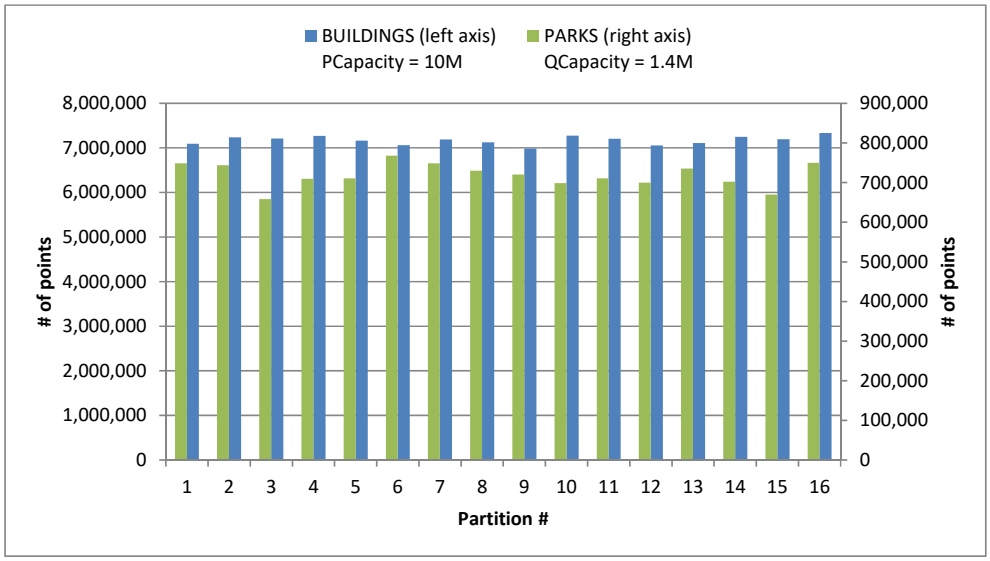

Figure 3.16: R-split: # of points per partition. Capacity = (10M, 1.4M).

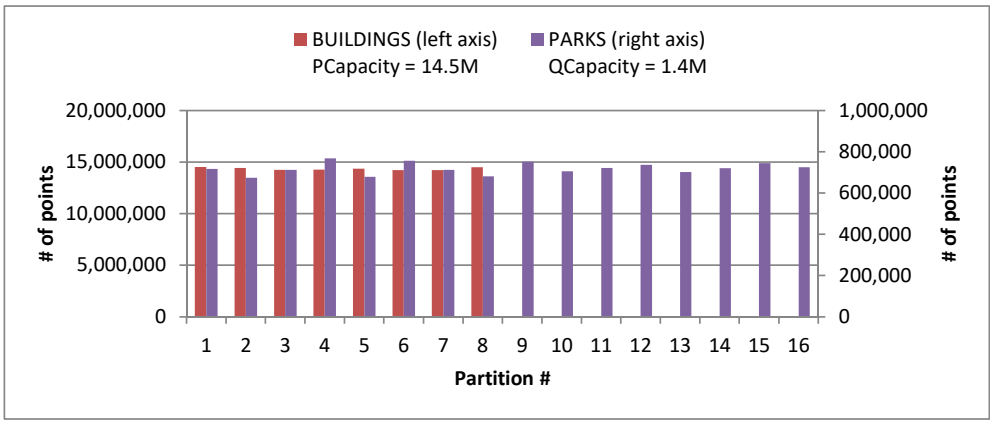

Figure 3.17: R-split: # of points per partition. Capacity = (14.5M, 1.4M).

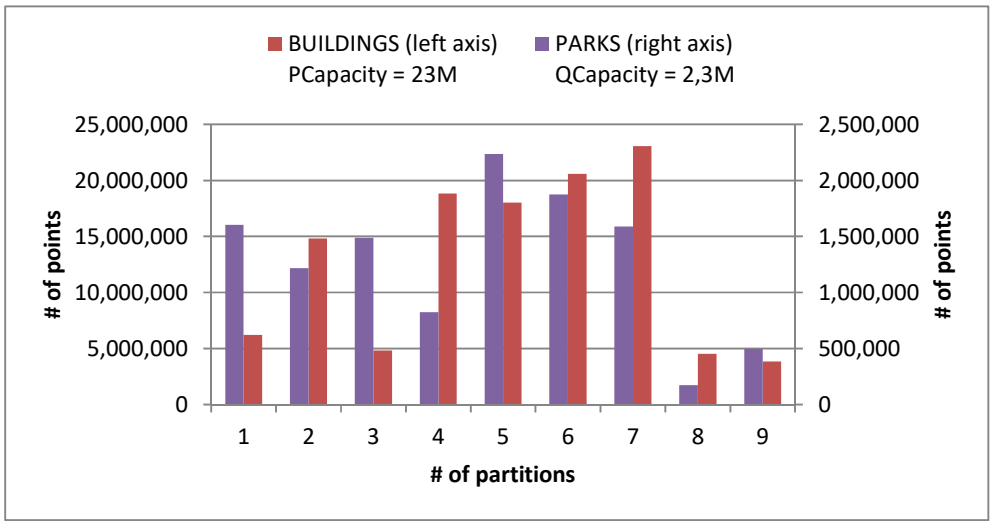

Figure 3.18: Q-split: # of points per partition. Capacity = (23M, 2.3M).

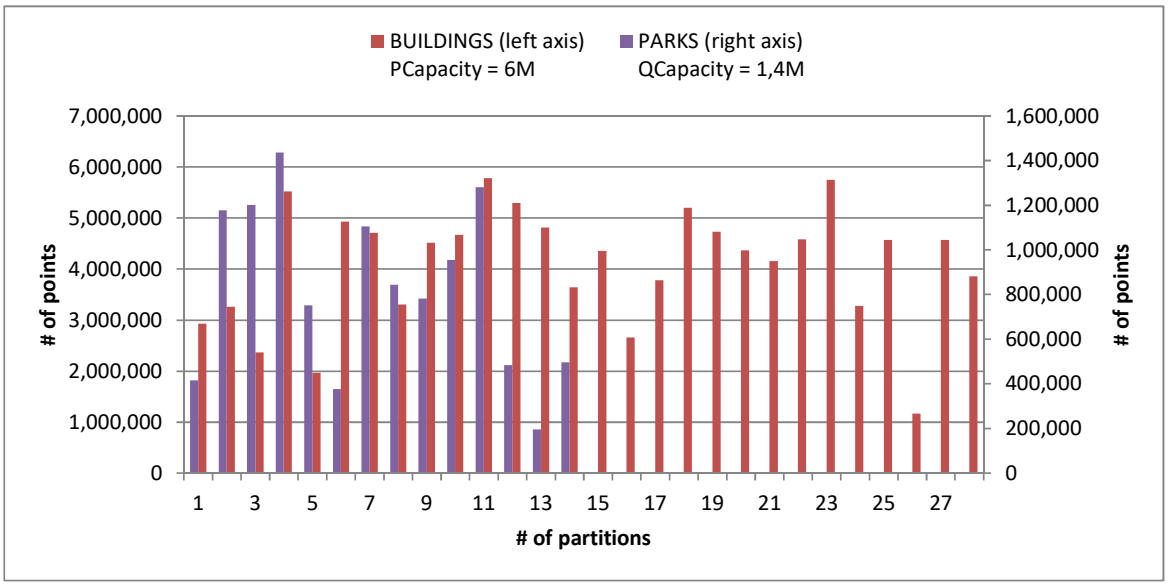

Figure 3.19: Q-split: # of points per partition. Capacity = (6M, 1.4M).

computation is being performed contain replicated points from both datasets.

In Fig. 3.20 and Fig. 3.21 we present the derived datasets in two different cases of partitioning by using R-split and Q-split respectively. In both figures, the first pair of columns presents the sizes of the original datasets and the rest two pairs present the sizes of the derived datasets (that are actually used for KCPQ computation) for two pairs of capacity settings. Using larger capacities leads to a smaller number of partitions (as shown in Table 3.1 and Table 3.2) but this leads to an increased number of eligible points, a fact that influences the total execution time (also shown in figures).

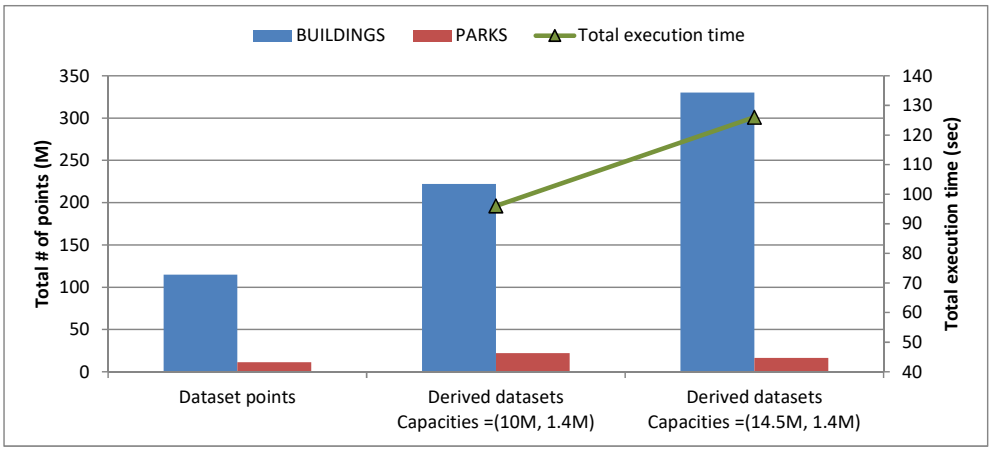

Figure 3.20: R-split: Replicated points and execution time.

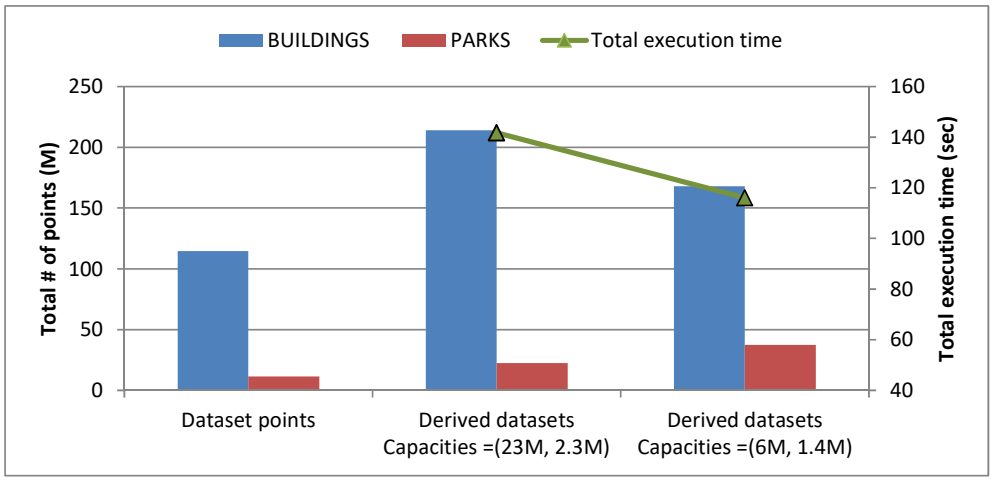

Figure 3.21: Q-split: Replicated points and execution time.

3.7.4 Testing with larger datasets

In order to test our method on even larger pairs of datasets, we used CLUS_LAKES[1], a new big quasi-real dataset derived from a real one. To create this dataset, for each point of LAKES, p, 15 new points gathered around p (i.e., the center of the cluster) are generated according to a Gaussian distribution with mean = 0.0 and standard deviation = 0.2. The dataset CLUS_LAKES contains around 126M of points. We computed the KCPQ on the pair P = BUILDINGS and Q = CLUS_LAKES for several combinations of $PCapacity$ and $QCapacity$ and the total execution time (averaged) is shown in Table 3.3 (R-split has been used for the partitioning of the whole datasets).

Table 3.3: R-Split results for the KCPQ($BUILDINGS \times CLUS\_LAKES$).

R-split					
capacity (millions of points)		Derived partitions #		Eligible pairs #	Time (sec)
BUILD-INGS	CLUS_ LAKES	BUILD-INGS	CLUS_ LAKES		
2	2	64	64	127-129	433.7
6	6	32	32	63	316.0
6	8	32	16	47	287.7
8	6	16	32	48	548.7
8	8	16	16	31	385.0

[1]Kindly provided by Antonio Corral and Francisco Garcia-Garcia, University of Almeria, Spain.

3.8 The SliceNBound algorithm

In [51] we expanded the work of [49] and presented a new algorithm for the KCPQ in Spark, called SliceNBound (SnB). The contributions are summarized as follows:

1. A novel, four-phased, iterative algorithm in "plain" Apache Spark to perform efficient parallel KCPQ processing on big spatial datasets. It is based on a simple and, therefore, not very computationally demanding partitioning scheme that enables the two datasets to share a common partitioning. Additionally, it only exploits built-in functions of Spark, thus making it easy to be imported in any spatial-oriented, or general, Spark-based parallel system.

2. A couple of fast heuristic methods that use a two-stage sampling technique to compute good upper bounds of the distance value of the K-th closest pair. These methods can be used as a preprocessing phase for any technique that uses preprocessing.

3. A version of SnB algorithm for Distance Join Queries (DJQs). As already mentioned, apart from [49], there was no other research paper on KCPQ processing in Spark at that time, so the SnB technique was compared to this work. However, in order to compare the performance of our method against other solutions, we have implemented an extension of our algorithm to answer DJQs. In [83], it has been reported that Simba yields superior performance compared against other spatial analytics system (GeoSpark, SpatialSpark, SpatialHadoop, Hadoop GIS etc). Simba does not support KCPQs, but does support DJQs. Results indicate that, in the case of the DJQ, our method is faster than the one embedded in Simba.

4. The execution of an extensive set of experiments using big real-world points datasets that demonstrate the efficiency and scalability of our proposal.

As it can be inferred from its name, our method slices the plane into strips and uses sampling to bound the solution space. The bound is used as a pruning criterion in several phases of the computation. The algorithm, iteratively and incrementally, approximates the solution in three major phases, until the exact solution is derived in the fourth Phase.

3.8.1 Pair-partitioning of datasets

Upon loading the two datasets, we calculate their max and min values for each dimension, thus creating an $Envelope = Env[minX, maxX, minY, maxY]$ for each one. We use two partioning variants, $parent-child$ and $common-merged$, both ending to split both datasets into the same strips, N in the first case and $2N + 1$ in the latter one.

Parent-child partition. We consider one of the two datasets as the $parent$ and the other as the $child$. Then $createSplitPoints$ (Algorithm 1) is used on parent to partition it into a user defined number of N strips (Figure 3.22, left). Afterwards, child dataset is being partitioned by using the split points of the parent. In Figure 3.22 parent is P and child is Q. Obviously, some partitions of child Q may be empty, or skewed. The existence of empty partitions hardly affects computing time, since no records are to be examined and skewness of child dataset is not a problem in the case of small datasets (e.g. samples). Note that this partition is applied only on the samples, while the next one is applied on the full datasets.

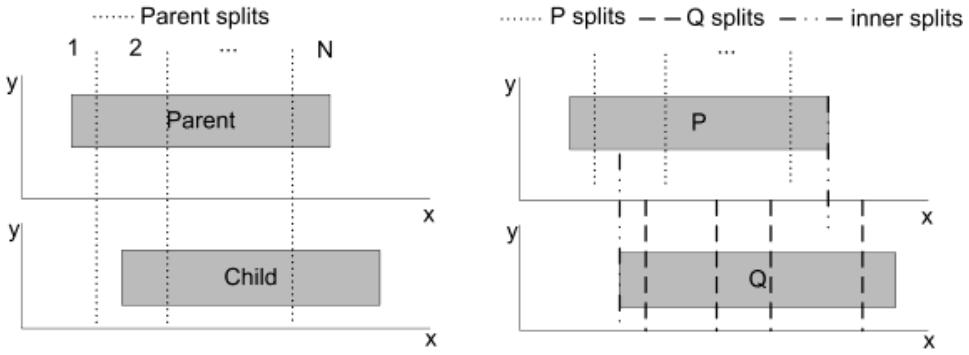

Figure 3.22: Pair-partitioning: parent-child (left), common-merged (right).

Common-merged partition. $createSplitPoints$ is used on each dataset to compute the split x-coordinates. Then we merge them into an array $allPoints$, together with the $minX$, $maxX$ values of both (Figure 3.22, right). The array of length $2N + 2$ is being sorted. The first and the last element of this array contain the min and max x-coordinates of all points, and are being removed, thus leaving $2N$ split x-coordinates $allSplits$, from which we create the $2N + 1$ strips, with a minimum width e, for both datasets. Algorithm 3 describes the procedure.

In both alternatives, the common Envelopes of the strips from both datasets are formed and broadcasted to the workers. Next, workers assign a proper key to each locally stored point, according to the Envelope it belongs and a shuffling is executed to create the N or

Algorithm 3 $mergeSplitPoints(dataSet1, dataSet2, N, e)$

Input: $dataset1, dataset2, N, e$

1: $Psplits \leftarrow createSplitPoints(dataSet1, N)$
2: $Qsplits \leftarrow createSplitPoints(dataSet2, N)$
3: $allPoints \leftarrow Psplits \cup Qsplits \cup \{PmaxX, PminX, QmaxX, QminX\}$
4: $quicksort(allPoints)$
5: $left = allPoints(0)$
6: $right = allPoints(1)$
7: **for** $i \leftarrow 1$ *to* N **do**
8: **if** $(right - left) < e$ **then**
9: $allPoints(i) \leftarrow allPoints(i-1) + e$ ▷ $e = sBound$
10: $left \leftarrow allPoints(i)$
11: $right \leftarrow allPoints(i+1)$
12: $splits \leftarrow allPoints.drop(first\ and\ last\ elements)$

$2N + 1$ partitions (according to the alternative used) for each dataset. Thus, points in both datasets that belong to the same strip get the same key.

3.8.2 Approximating the KCPQ

By cogrouping the two, pair-partitioned datasets, a new RDD is being created. This new, cogrouped, RDD contains items in the form $(stripNo, (Iterable[Point], Iterable[Point]))$ where $stripNo$ is the number of the strip, the first iterable contains points from P and the second points from Q, that belong to the same strip, therefore are close to each other. Each partition in this RDD is submitted for computation. Then all results are collected and the top-k among them form an approximation to the solution. Note that this solution is probably not accurate, since there may be points from P and Q in different strips that belong to the result. Still, the larger distance of the pairs found by this approach, is a good upper bound for the final KCPQ.

3.8.3 Cross-border computations

Suppose that we have finished the computation between pairs of strips and have found an upper bound bnd, as the distance of the K-th closest pair. Also, suppose that X_i is one splitting

point (Figure 3.23, left). In order to find the exact solution, we need to check whether points from each dataset belonging to different strips have their distance less than bnd as it has been computed in the previous step. We have already searched pairs of points in the strip on the left of X_i and pairs on the right of X_i as well. A pair of points (p, q) is a candidate pair only if $dist(p, q) < bnd$ and therefore, one of the following conditions may hold:

1. p belongs to Strip 1 and q belongs to Strip 4, or
2. p belongs to Strip 2 and q belongs to Strip 3

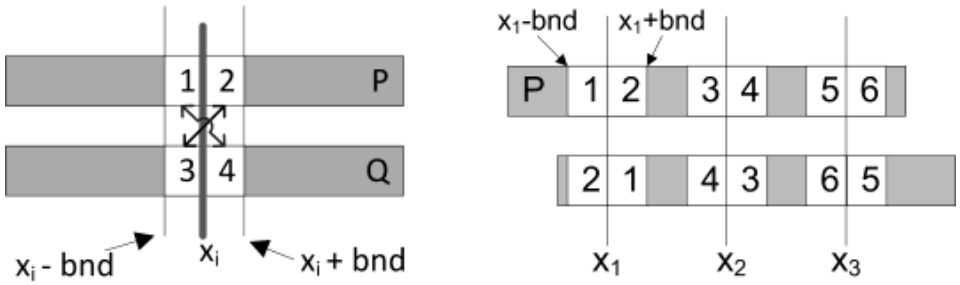

Figure 3.23: Cross-border eligible strips (left), cross-border indexing (right).

The bound bnd is broadcasted to the workers. Then, on a per-partition basis, data in each dataset is being filtered so that only points between $X_i - bnd$ and $X_i + bnd$ are being included in the newly created RDDs. Additionally, during the procedure, each worker assigns the proper indexes to the points, in a cross-border manner as shown in Figure 3.23 (right). The two newly created RDDs are being joined in order to create all candidate cross-border pairs. In the case that one of the corresponding (sub)partitions is empty, then no pairs will be created. As in previous step, a local $maxHeap$ is being utilized during processing. The first K pairs and their corresponding distances are being stored in the $maxHeap$. Then, for each consecutive pair, it is tested whether Plane-Sweep Inequality 3.1 holds or not, where in this case $\delta = h$, the distance of the head of the (local) maxHeap.

3.8.4 Phases of SliceNBound for the exact KCPQ

As in many demanding problems, knowing a good bound for the solution in question accelerates the total execution, since this bound helps in pruning the search space. Sampling for upper bound computation has been used in both [25] and [49] and reported that this increases efficiency. During the computations, sampling is utilized in two cases, one in order to derive

the splitting points and two in order to derive an upper bound of the KCPQ. The algorithm proceeds in four major phases, as being depicted in Algorithm 4.

Algorithm 4 $SliceNBound(P, Q, K, N, fraction)$

Input: $P, Q, K, N, fraction$

1: $sP \leftarrow P.sample(noReplace, fraction)$
2: $sQ \leftarrow Q.sample(noReplace, fraction)$
3: $parent - child\ Partition\ (parent\ sP, child\ sQ)$
4: $sampledKApproxDist \leftarrow KCPQ\_approximation(sP, sQ, K)$ ▷ Plane-Sweep
5: $Broadcast\ sBound = max\_distance\ \{sampledKApproxDist\}$
6: $createSplitPoints(P, N)$
7: $createSplitPoints(Q, N)$
8: $mergeSplitPoints(P, Q, N, sBound)$
9: $common - merged\ Partition(P, Q)$
10: $approxKDist \leftarrow KCPQ\_approximation(P, Q, K)$ ▷ Plane-Sweep
11: $Broadcast\ bnd = max\_distance\ \{approxKDist\}$
12: $crossDist \leftarrow crossBorderComputation(P, Q, K)$
13: $allResults \leftarrow Merge\_array(approxKDist, crossKDist)$
14: $KCPQ(P, Q, K) \leftarrow allResults.sortByDistance.take(K)$

Phase one consists of steps 1 to 5 and is used to quickly compute a first upper bound of the KCPQ. Phase two, consisting of steps 6 to 11, computes an approximation of the KCPQ over the whole datasets. Phase three, in step 12, performs the cross-border computation as described in Subsection 3.8.3 and, finally, Phase four consisting of steps 13 and 14 collects partial results from Phases two and three and merges them in an array from where the final result is extracted as the top-K (smaller distances). Due to the nature of the partitioning, no pairs are possible to be included in the result twice, so there are no duplicates in the final solution.

Based on our experimental results, we are using parent-child partitioning for computing the KCPQ over the relatively small sample (Step 3 of the above algorithm) and common-merged partitioning for the computation on the whole datasets (Step 9 of the above algorithm), a combination shown to work effectively. Notice that, in case the common-merged partitioning procedure is used as part of the exact KCPQ computation in Phase two, special care has to be taken so that the width of every strip does not fall under the value of $e = bnd$. If such case is found, then the split point is moved rightwards so that width is greater than bnd.

3.8.5 SliceNBound for the DJQ

In the case of the DJQ, the upper bound is already known. Following this observation, there are three major differences compared to the KCP Query.

First, there is no need for Phase one to compute an initial upper bound via sampling. Second, there is no need for $maxHeap$ to store (part of) the K best pairs, since all pairs (p, q) such that $dist(p, q) \leq e$ are part of the solution. Instead, a local list is being utilized. And, third, we don't have to collect results from Phase two before proceeding to Phase three. The algorithm proceeds by performing Phase two and Phase three and each one creates an intermediate RDD. The two RDDs contain all pairs that form the exact result, which can be collected or stored.

3.9 Experimental evaluation of the SliceNBound algorithm

We have used 2d point datasets to test our KCPQ/DJQ algorithms in Spark derived from OpenStreetMap [17]: $WATER$ resources (5,836,360 line segments), $PARKS$ (11,504,035 polygons) and $BUILDINGS$ of the world consisting (114,736,611 polygons) have been used to create sets of 2D points. Experiments run on a cluster of 5 nodes, each one having 4 vCPUs at 2.1GHz, with a total of 16GB of main memory per node (12GB for Spark), running Ubuntu Linux 16.04 operating system, Spark 2.0.2 on Hadoop 2.7.2 Distributed File System (HDFS with 128 MB block size). Four nodes (4 nodes x 4 vCPUs = 16 vCPUs) were used as HDFS DataNodes and Spark Worker nodes. Java openjdk ver. 1.8.0 and Scala code runner ver. 2.11 were used. We performed each experiment several times and averaged the total response time that expresses the overall CPU, I/O and communication time needed for the execution of each query from startup until count of the results. In all experiments, sample $fraction$ for upper bound computation in Phase one was set to 0.01 and sample size for split points computation was set to $noOfRecords/N$ (N the initial number of partitions).

Our first experiment measures the efficiency of SnB for the KCPQ as compared to the Slices algorithm [49]. It also examines the effect of the number of partitions and the increase of the K value. In the case of SnB, each dataset starts with N partitions (Phase one) and ends with $2N + 1$ (Phases two and three). In Fig. 3.24 experiments were conducted for both methods, using the $PARKS \times WATER$ datasets combination. The total height of each vertical bar presents time measured for the Slices algorithm and the bottom part presents

time measured for SnB. The upper part of each vertical bar presents the difference (gain) of SnB compared to the Slices algorithm. It can be seen that the improvement varies around 30%-60%, mainly due to the elaborate partitioning schemes. In Fig. 3.25 experiments were conducted solely for SnB and are presented versus the best results recorded in [49] (using the same infrastructure and settings) using the $BUILDINGS \times WATER$ datasets combination. There is also a substantial improvement in total response time that exceeds 30%. The total execution time grows moderately as the number of results to be obtained increases. It may be concluded that there is no significant impact on the execution time but, in general, as K increases, pruning gets less effective. Results verify the recommendation of having a number of partitions that equals 2 to 4 times the number of cores. Larger number of partitions overwhelms the computing cluster.

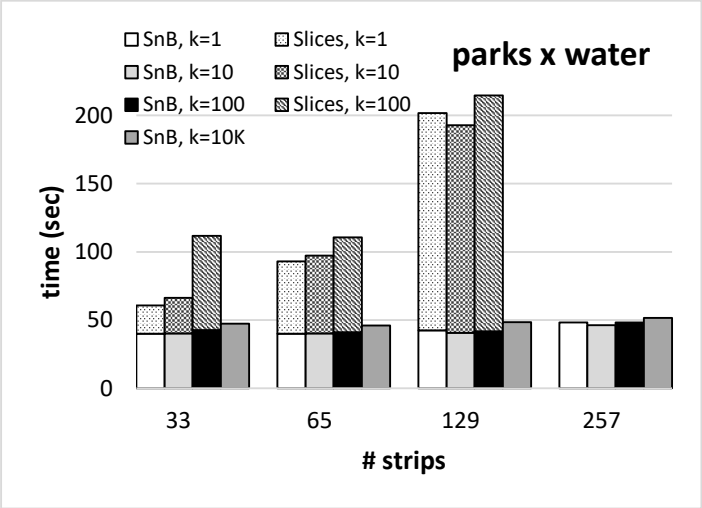

Figure 3.24: SnB vs Slices algorithm for the KCPQ.

The second experiment measures the computing time for the case of $BUILDINGS \times PARKS$ (line with diamonds, left vertical axis) and the speedup (right vertical axis) of SnB for the KCPQ, varying the number of computing nodes (Fig. 3.26). The dotted line shows the ideal (theoretical) speedup. It could be inferred that the performance of SnB will increase if more computing nodes are added.

The third experiment presents the performance of the SnB algorithm variant for the Distance Join Query, compared to Simba [83], for the case of the $PARKS \times WATER$ dataset combination (Fig. 3.27). We have used branch simba-spark-1.6 and the Distance Join example, as provided by the project with very few modifications in order to best fit to our cluster. In both methods, the number of partitions was set to 65, which means starting with 32 for

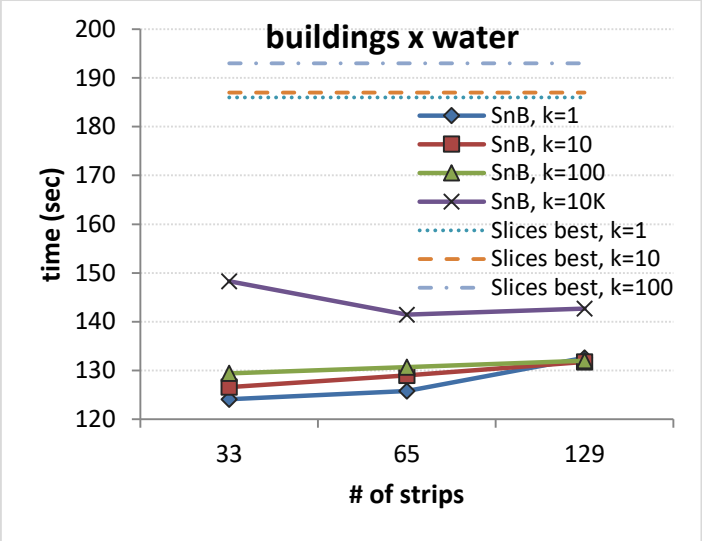

Figure 3.25: SnB vs Slices algorithm for the KCPQ.

SnB, finally leading to 2N + 1 = 65. The value of epsilon distance varied from 0 (solution contains no points) to 6E-4. In the latter, the cardinality of the solution set is more than half a billion points, as shown on the horizontal axis. The diamond-marked line presents time measured for Simba and the square-marked line shows time measured for SnB (denoted as SnB PS1). We have observed, though, that in contrast to the KCPQ variant of our algorithm where Phase two dominates the computing time, in the case of the DJQ, Phase three also takes a large amount of time, since we are dealing with large sets of eligible cross-border points. To tackle this, we used the Plane-Sweep technique in Phase three as well. As shown by the

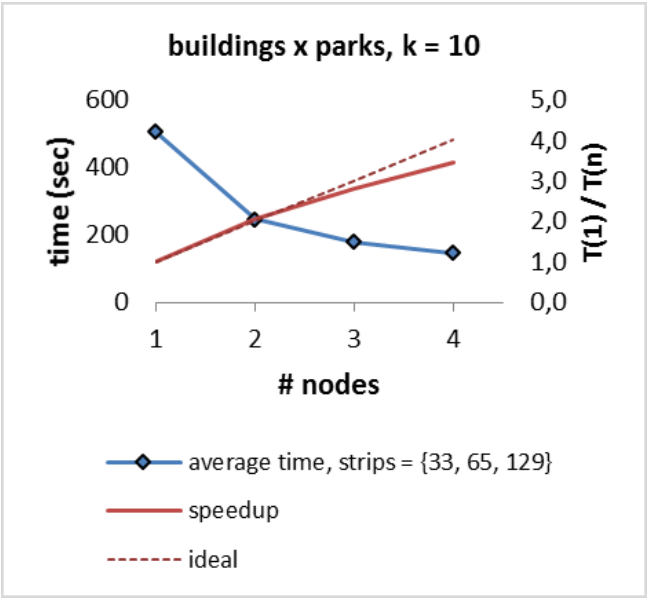

Figure 3.26: Total response time and speedup of SnB for the KCPQ.

triangle-marked line (denoted as SnB PS2), there is a considerable increase in the efficiency of SnB. However, either with or without Plane-Sweep during the cross-border computation, SnB performs significantly better in our cluster than the method provided by Simba, in almost all cases taken into consideration.

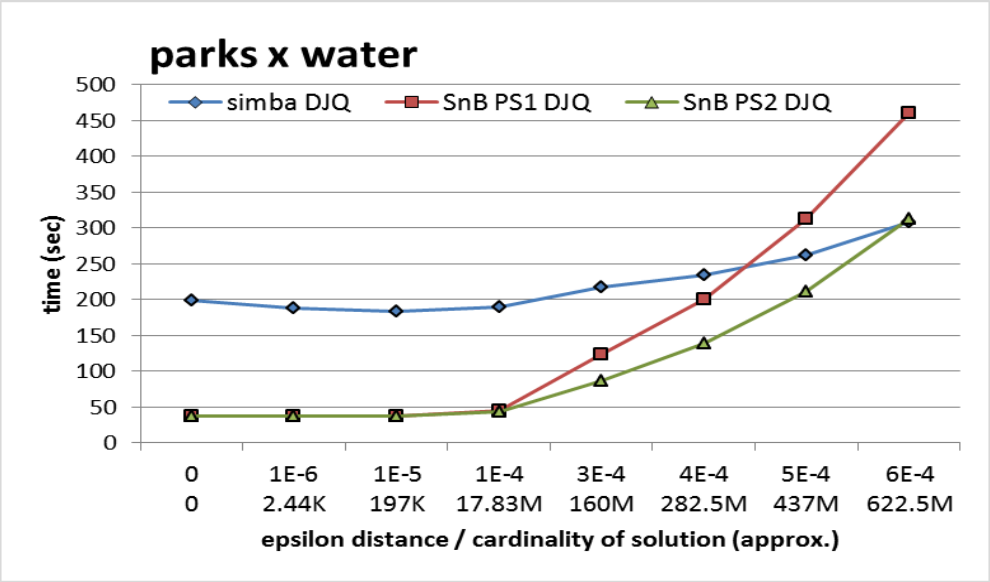

Figure 3.27: Total response time of SnB for the DJQ compared to the DJQ in [83]

3.10 Conclusions

In [49] we presented the first algorithm (called "Slices" in this thesis) for processing KCPQs in Spark. This algorithm utilizes sampling for achieving an upper bound of the distance of the -th closest pair between the two datasets, partitions independently each of the two datasets into strips and processes eligible (in relation to the calculated bound) pairs of strips, utilizing Plane-sweep. In [49, 50] was significantly improved. Only one sampling phase was used and two additional, BST based, partitioning schemes, influenced by the Quad-tree and R-tree principles, were introduced (Q-split and R-split, respectively). The resulting algorithm (called "BST slices" in this thesis) significantly improved performance.

In [51] the problem of answering the KCPQ as well as DJQ in Spark is studied and the SnB algorithm, utilizing parent-child and common-merged strip partitioning, local/global bounding and the Plane-Sweep technique is proposed. The performance of the new method has been evaluated with big real-world points datasets. Response times for the KCPQ were compared to the ones in [49] and shown to significantly excel (the improvement over [49] is

quite larger than the one of [50]). SnB for the DJQ was also compared to the results derived by using Simba [83]. The presented technique has been shown to be of superior performance.

SnB shares some common parts with the "Slices" method family presented in [49] and [50], but also incorporates quite important improvements leading to the substantial reduction of response time and increase in scalability. Concerning the similarities, both methods use Plane-Sweep to compute the KCPQ. Also, they both utilize sampling to find the partition points for a single dataset and the same technique to strip-partition a single dataset. Both use bounds for the solution, one in the case of [49] and [50], where it is computed from a sample and used to find all the eligible pairs of strips and two in the case of SnB.

The major improvements in SnB are as follows: [49] and [50] separately partition each dataset to strips and then perform the computation on the eligible pairs of strips that are extracted by utilizing a bound. SnB proceeds in a completely different manner. It actually creates the overlapping pairs by sharing the same partition points between the two datasets. Disjoint pairs are not considered for computation in the first place, but are taken into account in Phase three, now heavily pruned. Initially, sampling is used in combination with parent-child partition to find a good upper bound, since parent-child partition creates pairs of strips with points more or less close to each other. In contrast, upper bound in [49] and [50] is being computed with a more naive and time consuming method, leading to the need of selecting of smaller sample fraction and consequently to significant fluctuation in the quality of the bound. Phase two of SnB is preceded by a common-merged partitioning scheme that generates pairs of strips while ensuring that data points are equally shared among them and that all strips have width greater than the bound found so far. In SnB the usage of the bound is twofold: first it serves as a measure for the minimum width of each strip in Phase two and secondly it is used to prune the search space. In Phase three SnB performs the cross border computation on strips that are bounded by the solution found in Phase two, a solution that is already quite accurate and minimizes the need for extra computation.

Chapter 4

The All K Nearest-Neighbor Query

4.1 Introduction

Given two point datasets, Q (Query) and T (Training), the All K Nearest Neighbor Query (AKNNQ, or KNN Join) finds the K nearest neighbors of T for each point of Q, according to a certain distance metric [4]. 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 applications of the AKNNQ include classification [58], graph-based computational learning [37], N-body simulations in astronomy [15], 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. [19] 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.

In [54] we presented a four phases algorithm, which improves significantly the algorithm originally introduced in [85] and further improved in [58], 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.

4.2 Preliminaries and related work

In [54] 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 points from S such that, $\forall r \in KNN(p, S), \forall q \in S - kNN(p, S),\ \ dist(p, r) \leq dist(p, q)$

Definition (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 [87]) and pruning techniques, like in [4,9,21,68,82]. 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 [85] 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 don't 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 [58] 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 [58] have also extended the algorithm to work with more than two dimensions and added a classification step.

The authors of [6] 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 [58]. According to [6], the calculations per (merged) cell are more balanced and they get better performance, compared to [58]. 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 [47], 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 [7], 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 [91], 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 techiques for fast but approximate AKNN computations, therefore we will not present further details.

The authors in [81] 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 Quadtree partitioning instead of the Z curve. Unfortunately the authors only compare their work to Oracle Spatial and not to other AKNN MapReduce solutions.

In [27], 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 [27] utilizes processing based on Plane-Sweep (1st improvement). Unlike the algorithm that we present, the algorithm of [27] 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 [27] we utilize restructuring of phases to reduce network traffic (2nd improvement).

4.3 Presentation of the algorithm

The base algorithm presented in [58] 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 won’t 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.

4.3.1 Phases

First, we present a diagrammatic overview of the algorithm of [58], without the classification step. Figure 4.1 shows the overview of the whole process, which remains unaffected by the improvements we applied. The phases of this algorithm follow.

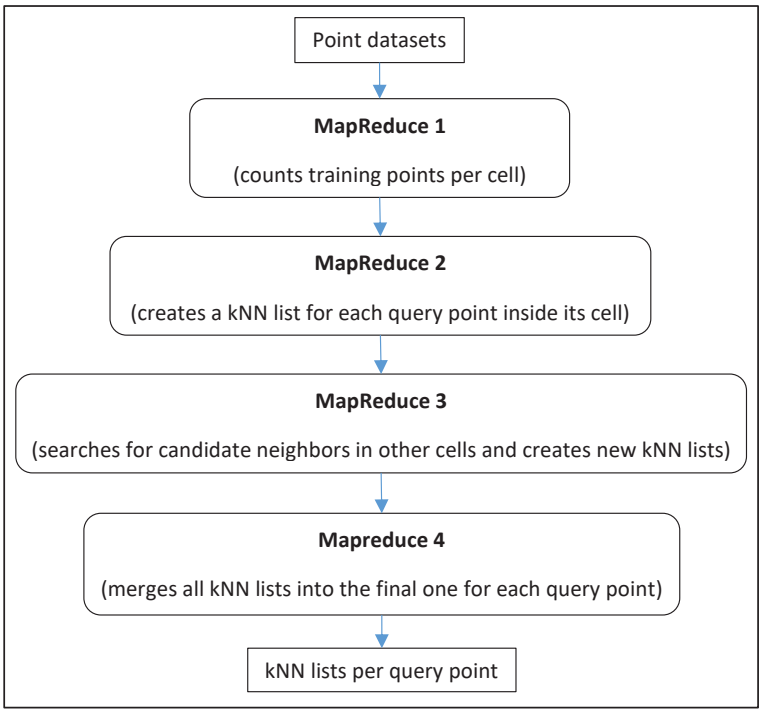

Figure 4.1: The four phases overview flowchart.

1st MapReduce phase The target space is decomposed into $N \times N$ equal sized cells (Figure 4.2). 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 (Figure 4.3) 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

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

for Input / Training. The Reducer output key is {Input point id} and the value is the list {x, y, cell id, KNN list}.

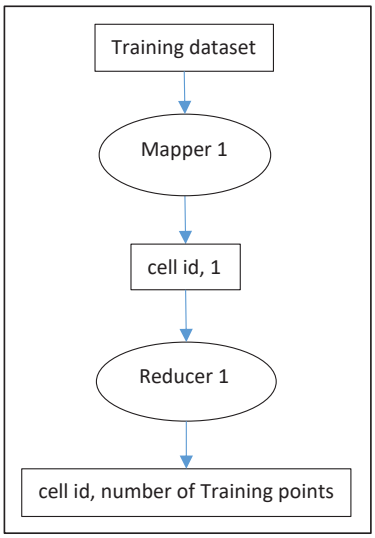

Figure 4.2: 1st phase flowchart (base algorithm).

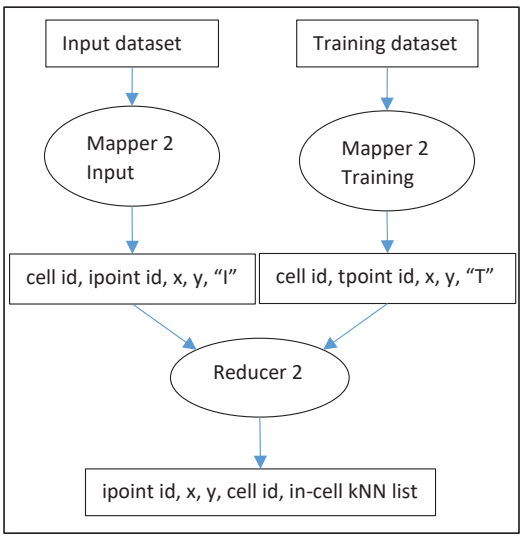

Figure 4.3: 2nd phase flowchart (base algorithm).

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

In Figure 4.4, 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 (Figure 4.5), 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 (Figure 4.6) 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.

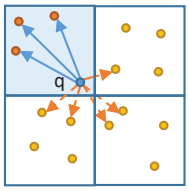

Figure 4.4: 2nd phase possible misses.

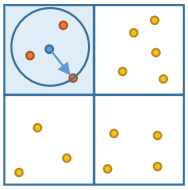

Figure 4.5: 3rd phase "true" case.

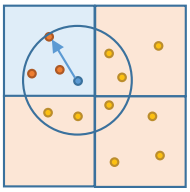

Figure 4.6: 3rd phase "false" case (KNN list complete).

- Case 2: KNN list is not complete

If the neighbor list is not full, then the circle around the Input point is gradually getting bigger (Figure 4.7), 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 Figure 4.7, meaning that K total points are found (we are not sure where exactly; we only know in which cells they are located). Looking at Figure 4.8, 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 Figure 4.9. There is also another Mapper in Figure 4.9 (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 Figure 4.9 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.

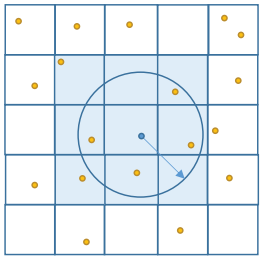

Figure 4.7: 3rd phase "false" case (KNN list incomplete).

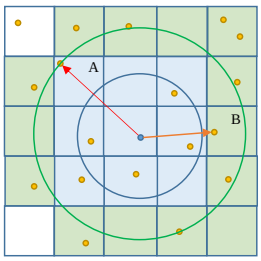

Figure 4.8: 3rd phase "false" case (KNN list incomplete, radius boost).

4th MapReduce phase In this final phase (Figure 4.10) 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.

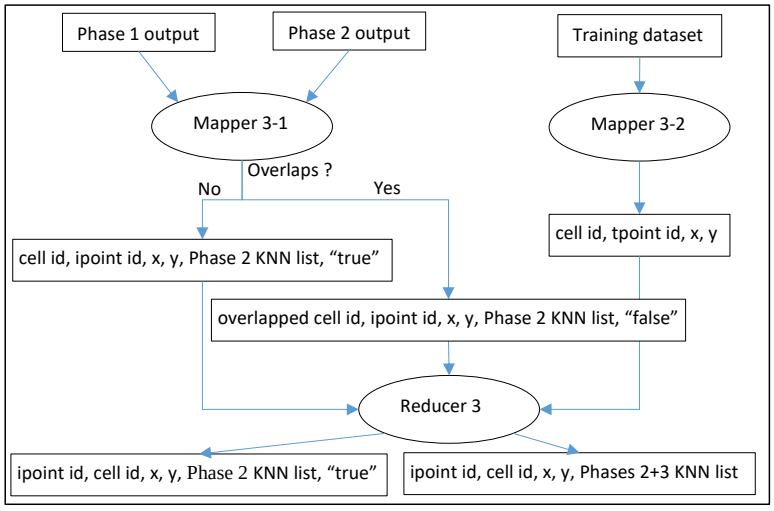

Figure 4.9: 3rd phase flowchart (base algorithm).

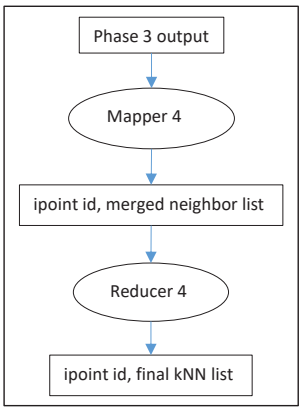

Figure 4.10: 4th phase flowchart (base algorithm).

In the approach of [58], 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 and Training points in a cell are N_q and N_t respectively, 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 base algorithm and present it in pseudocode in a more detailed version than [58] in Figures 4.11-4.14.

```
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
```

Figure 4.11: 1st phase pseudocode (base 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
```

Figure 4.12: 2nd phase pseudocode (base algorithm).

4.3.2 Algorithmic improvements

So far we have presented the base algorithm ([58]). 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 base algorithm,
- BF, or $Brute - Force$: processing in Reducers used in the base algorithm,
- PS, or $Plane - Sweep$: 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).

```
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
```

Figure 4.13: 3rd phase pseudocode (base 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
```

Figure 4.14: 4th phase pseudocode (base algorithm).

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 ([64]).

First (Figure 4.15) 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's 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's 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.

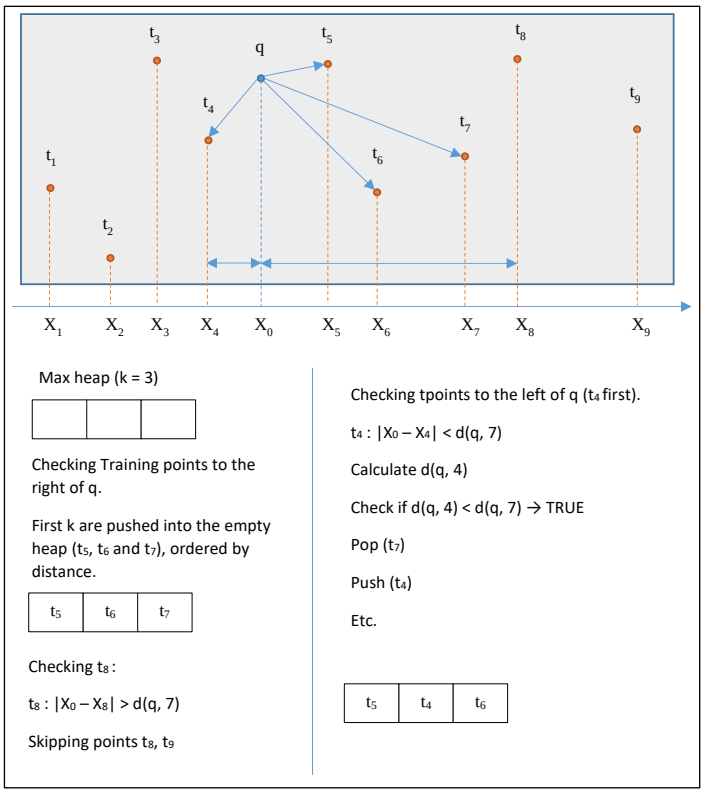

Figure 4.15: Plane-Sweep technique illustration.

The modified Reducer 2 using Plane-Sweep is illustrated in Figure 4.16. Reducer 3 is similar.

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.

In the base 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

```
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
```

Figure 4.16: Plane-Sweep Reducer 2.

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 Figures 4.17 and 4.18 (Phases 1 & 2 remain unchanged). If you compare them with Figures 4.13 and 4.14 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 (Figure 4.18) with an added "false" flag.

3rd Improvement: Quadtree Mappers (QT)

Both previous improvements along with the base 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

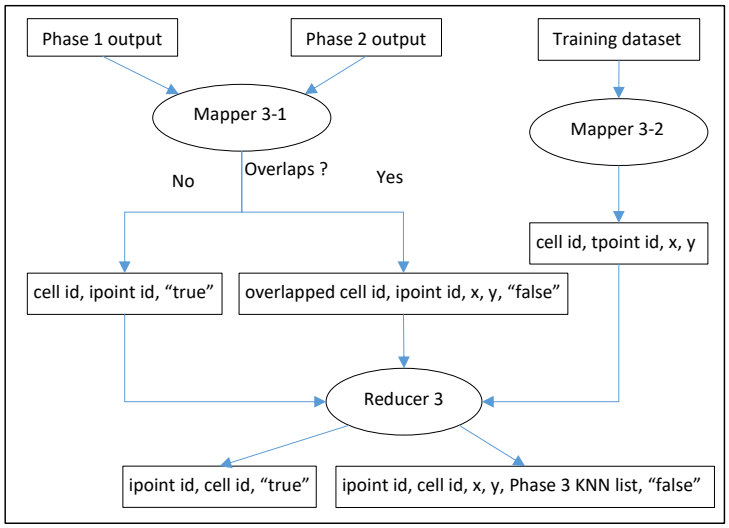

Figure 4.17: 2nd improvement: Phase 3 with less output data.

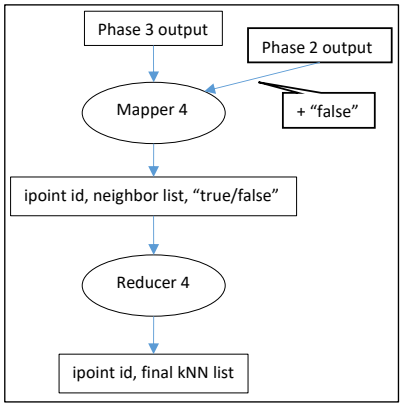

Figure 4.18: 2nd improvement: Phase 4.

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 (Figure 4.19). 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 remains pretty much the same. Therefore, we were able to implement Brute-Force, Plane-Sweep, and LessData in combination with Quadtree.

A clarification is needed regarding Mapper 3-1 radius boost that is shown in Figures 4.7 and 4.8 for the base 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 Figure 4.19 (the vector heading to the top-left depicts the final radius calculated).

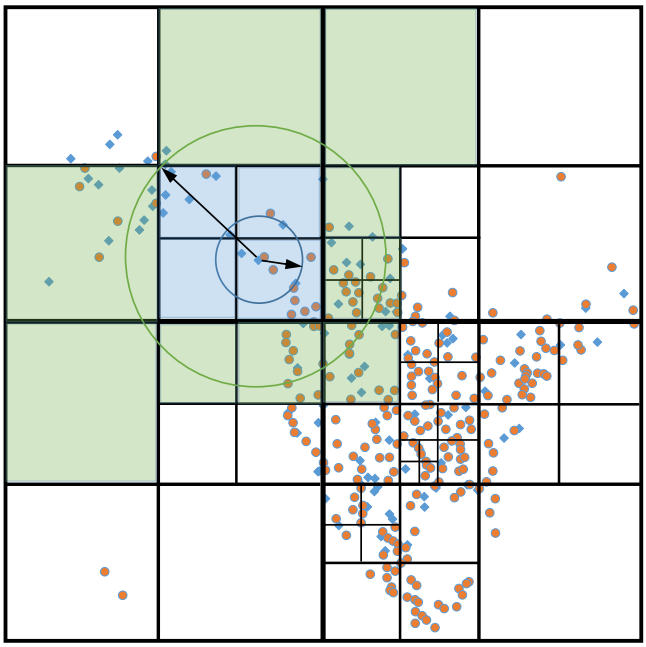

Figure 4.19: Quadtree space decomposition and radius boost.

4.4 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 [17], 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 performed 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 the results are immediately comparable. Moreover, the size of these datasets is enough to highlight the performance gains of the proposed improvements.

4.4.1 Plane-Sweep

In this experiment, we used Grid partitioning and studied the performance of the base algorithm against two Plane-Sweep versions, for K = 5, 10 and 20. In Figures 4.20-4.22, we show the execution time of Brute-Force (base algorithm), Plane-Sweep 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).

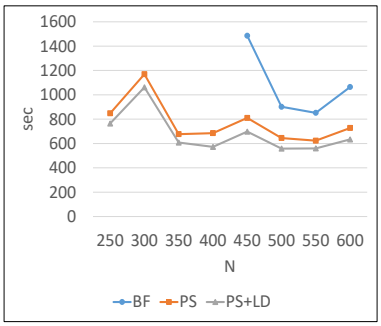

Figure 4.20: GD: exec. time of BF, PS and PS+LD, for $K = 5$.

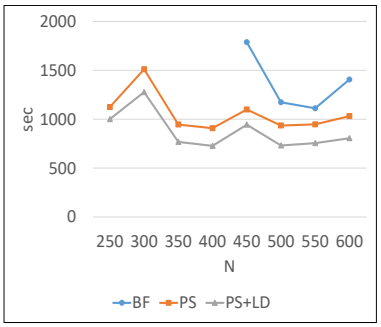

Figure 4.21: GD: exec. time of BF, PS and PS+LD, for $K = 10$.

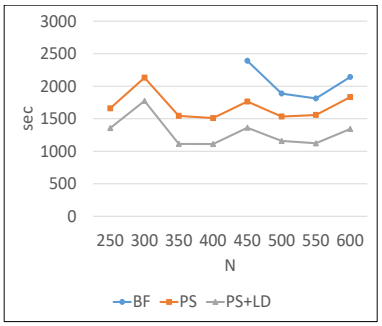

Figure 4.22: GD: exec. time of BF, PS and PS+LD, for $K = 20$.

As shown in these figures, the results for Brute-force 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 sec, 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 Plane-Sweep always finishes before Brute-Force, because it prunes many points. Therefore, we will only use the Plane-Sweep method in the rest of the experiments. Performance improvement of Plane-Sweep over Brute-Force, 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 Plane-Sweep and LessData shows its strength, since the size of the output of Phase 3 is reduced. This effect will be studied in more detail in the next subsection.

One of Grid's disadvantages is the percentage of empty space, which is the number of cells that contain few, or no points and needn't 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 4.1. 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 Brute-Force 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).

4.4.2 Plane-Sweep + LessData

Figures 4.20-4.22 also depict the Plane-Sweep + 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 Figures 4.20-4.22 show, when $N = 500$, performance gain of Plane-Sweep + LessData, for $K = 5$, against Plane-Sweep is 13% and against Brute-Force is 38%, for $K = 10$, against Plane-Sweep is 22% and against Brute-Force is 38%, and, for $K = 20$, against Plane-Sweep is 24% and against Brute-Force is 38%. So there is a constant $\approx 40\%$ improvement over the base algorithm (Brute-Force).

Table 4.1: Grid Training points distribution

	number of cells containing Training points					
N	**<50K**	**50K-100K**	**100K-200K**	**200K-300K**	**total**	**non empty**
250	5384	34	22	2	62500	5442 (8.71%)
300	6918	41	13	1	90000	6973 (7.75%)
350	8546	45	8	0	122500	8599 (7.02%)
400	10391	30	3	1	160000	10425 (6.52%)
450	11749	24	1	1	202500	11775 (5.81%)
500	13358	15	3	0	250000	13376 (5.35%)
550	15048	16	1	0	302500	15065 (4.98%)
600	16605	14	1	0	360000	16620 (4.62%)

To study the reasons for this effect, in the next set of graphs (Figures 4.23-4.25) we depict the Phase 3 output-data size of Plane-Sweep and Plane-Sweep + 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.

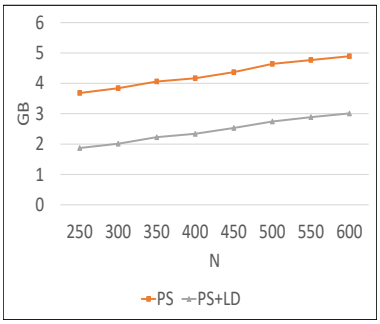

Figure 4.23: GD: Phase 3 size of PS and PS+LD, for $K = 5$.

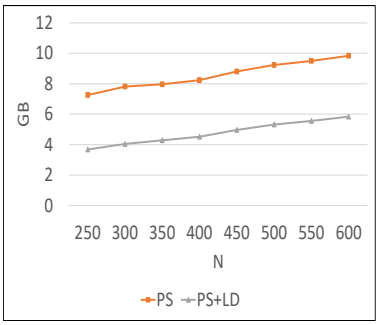

Figure 4.24: GD: Phase 3 size of PS and PS+LD, for $K = 10$.

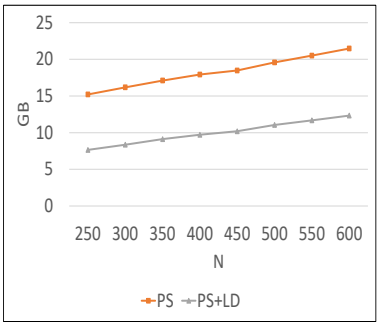

Figure 4.25: GD: Phase 3 size of PS and PS+LD, for $K = 20$.

4.4.3 Quadtree

In this section, we first seek for a good cell capacity for Quadtree creation (Subsection 4.4.3), before we compare the performance of algorithmic variations with partitioning based on Quadtree against Grid, using Plane-Sweep (Subsection 4.4.3) and using Plane-Sweep + LessData (Subsection 4.4.3).

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 Figure 4.26 we present the execution time of the Quadtree + Plane-Sweep + 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 seconds.

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 4.2 (where for several capacities, we depict the number of cells containing $< 50K$ and $\geq 50K$ 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.

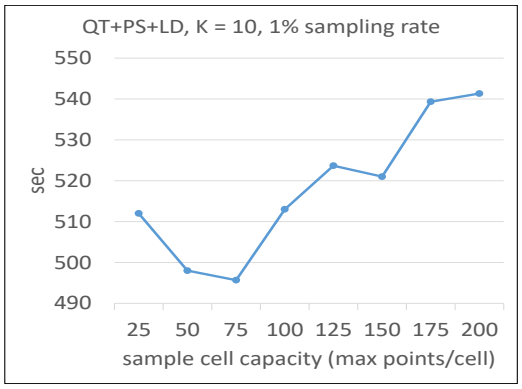

Figure 4.26: Exec. time of QT using PS+LD, to determine the best cell capacity (1% sampling rate).

Table 4.2: Quadtree Training points distribution

	number of cells containing Training points			
capacity	$< 50K$	$\geq 50K$	**total**	**non empty**
25	12625	0	13414	12625 (94.12%)
50	6449	0	6871	6449 (93.86%)
75	4195	0	4507	4195 (93.08%)
100	3246	0	3484	3246 (93.17%)
125	2582	0	2776	2582 (93.01%)
150	2129	0	2284	2129 (93.21%)
175	1835	0	1969	1835 (93.19%)
200	1623	0	1732	1623 (93.71%)

Quadtree vs. Grid, using Plane-Sweep

In this subsection, we will compare the performance resulting from Quadtree against Grid partitioning, using Plane-Sweep, in both cases. In Figures 4.27, 4.28 and 4.29, we depict execution time of the three top Grids and Quadtree, for $K = 5, 10$ and 20, respectively. The 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.

Quadtree vs. Grid, using Plane-Sweep + LessData

The set of graphs in Figures 4.30-4.32 present an analogous comparison to Figures 4.27-4.29, using Plane-Sweep + LessData, instead of only Plane-Sweep. Quadtree again wins in all cases by 30%.

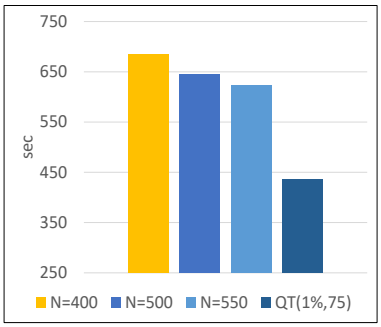

Figure 4.27: Exec. time of QT and 3 versions of GD, using PS, for $K = 5$.

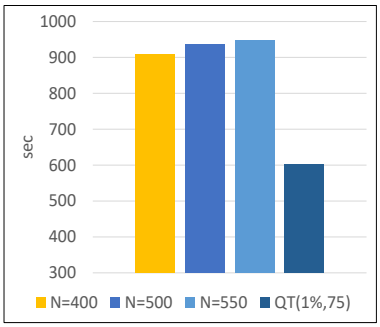

Figure 4.28: Exec. time of QT and 3 versions of GD, using PS, for $K = 10$.

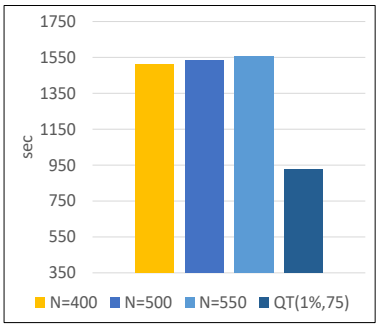

Figure 4.29: Exec. time of QT and 3 versions of GD, using PS, for $K = 20$.

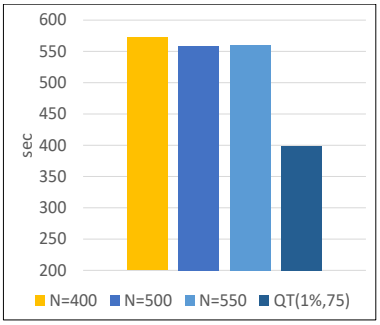

Figure 4.30: Exec. time of QT and 3 versions of GD, using PS+LD, for $K = 5$.

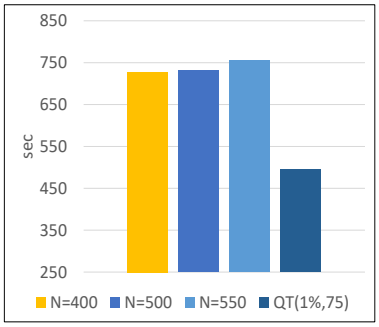

Figure 4.31: Exec. time of QT and 3 versions of GD, using PS+LD, for $K = 10$.

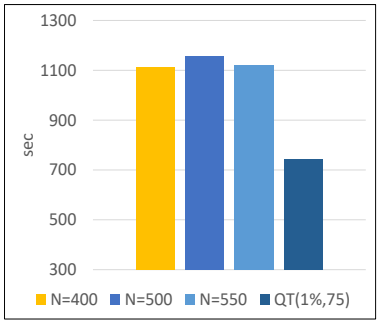

Figure 4.32: Exec. time of QT and 3 versions of GD, using PS+LD, for $K = 20$.

The explanation behind these numbers lies in the graphs of Figures 4.33-4.35, where we depict the output sizes of Phases 2 and 3 for Grid with $N = 400$ and Quadtree with capacity = 75 (both using Plane-Sweep + 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 Figures 4.36-4.38, for the same algorithmic settings of Figures 4.33-4.35. Phase 1 was deliberately left out, because it always only takes 40-50 sec, 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 4.1 that some Grid cells may have hundreds of thousands of points, which leads to major slowdowns.

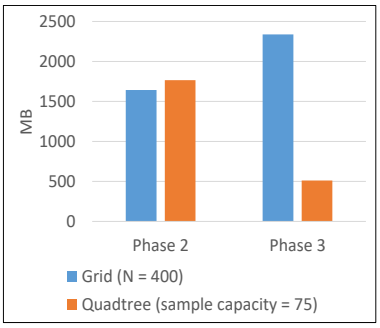

Figure 4.33: Phases 2 & 3 size of QT and GD, using PS+LD, for $K = 5$.

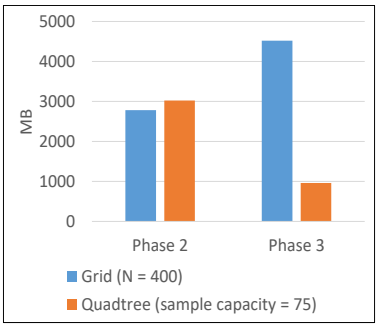

Figure 4.34: Phases 2 & 3 size of QT and GD, using PS+LD, for $K = 10$.

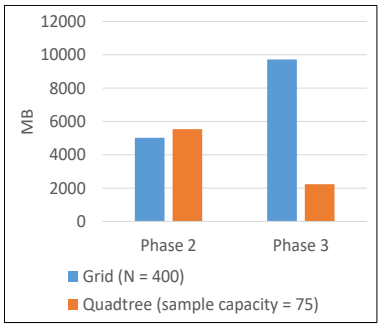

Figure 4.35: Phases 2 & 3 size of QT and GD, using PS+LD, for $K = 20$.

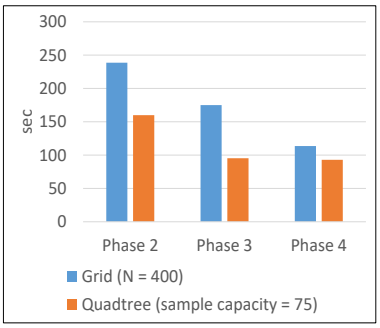

Figure 4.36: Phases 2-4 exec. time of QT and GD, using PS+LD, for $K = 5$.

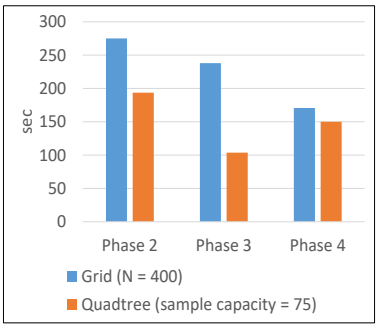

Figure 4.37: Phases 2-4 exec. time of QT and GD, using PS+LD, for $K = 10$.

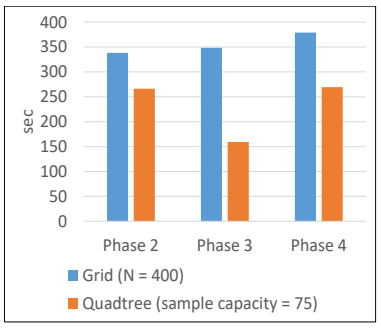

Figure 4.38: Phases 2-4 exec. time of QT and GD, using PS+LD, for $K = 20$.

4.4.4 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.

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 Plane-Sweep + LessData) in 2, 4, 6

and 8 nodes, for all three K values. Results are depicted in Figures 4.39-4.40.

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 $4 \rightarrow 8$ nodes. This is almost linear. In Quadtree, the $2 \rightarrow 4$ nodes performance increase is 39% and from $4 \rightarrow 8$ nodes it is 46%.

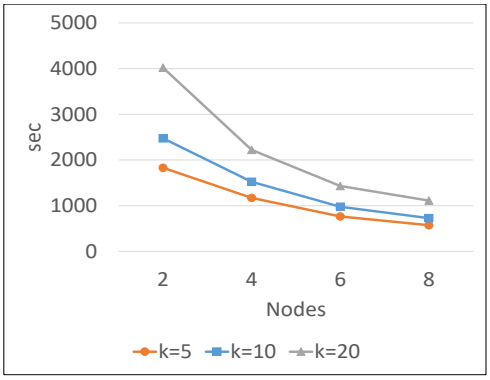

Figure 4.39: Exec. time of GD+PS+LD, N = 400.

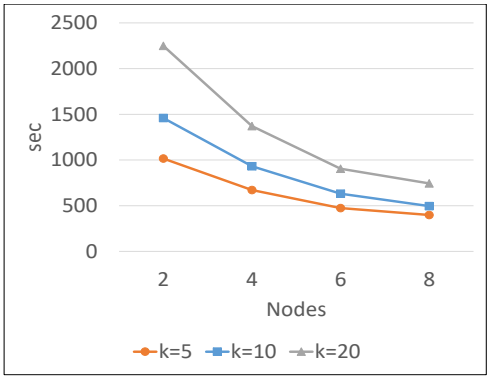

Figure 4.40: Exec. time of QT+PS+LD, 1% sampling rate, capacity = 75.

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 4.4) 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 Plane-Sweep + LessData),

The results are shown in Figures 4.41-4.42. 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

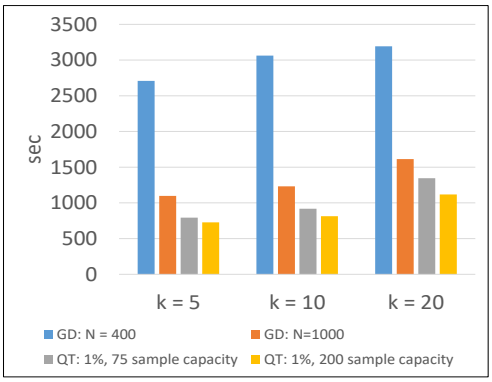

Figure 4.41: Exec. time of QT and GD, using PS+LD.

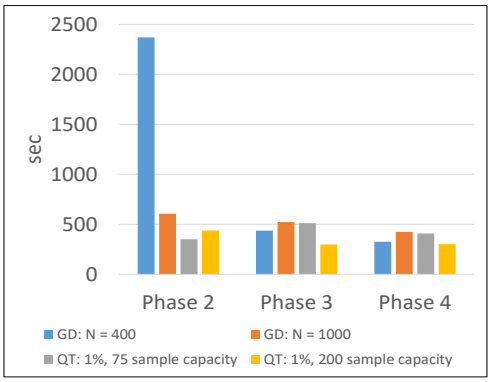

Figure 4.42: Phases 2-4 exec. time of QT and GD, using PS+LD, $K = 20$.

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. Figure 4.42 shows the huge Phase 2 time of the old best Grid, which is explained by the Training points distribution in Table 4.3. 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.

Table 4.3: Grid Training points distribution

	number of cells containing Training points	
number of 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	11701	39292
total	160000	1000000
non empty	12135 (7.58%)	39881 (3.99%)

Quadtree achieves a better points distribution (even the old best one). In Table 4.4, for

capacity = 75 and 200, we depict the number of cells containing $< 50K$ and $\geq 50K$ 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.

Table 4.4: Quadtree Training points distribution

	number of cells containing Training points			
capacity	$< 50K$	$\geq 50K$	**total**	**non empty**
75	42439	0	44749	42439 (94.84%)
200	16032	0	16816	16032 (95.34%)

4.5 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 Plane-Sweep + LessData to both partitioning methods, since this combination has clearly achieved the best results in our experiments so far.

4.5.1 3D Grid

In Figure 4.43, we depict the execution time of 3D Grid using Plane-Sweep + LessData for several values of N and $K = 10$ (results were analogous for other values of K). The best performance is achieved for $N = 25$.

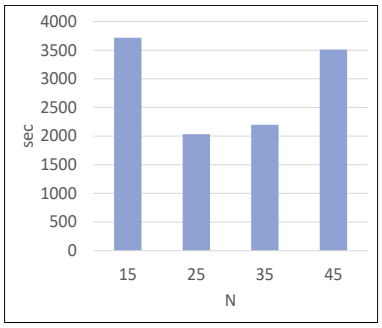

Figure 4.43: Exec. time of 3D GD, using PS+LD, $K = 10$.

4.5.2 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 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 Plane-Sweep + LessData), for $K = 10$. The results (execution time as a function of cell capacity) are shown in Figure 4.44. 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.

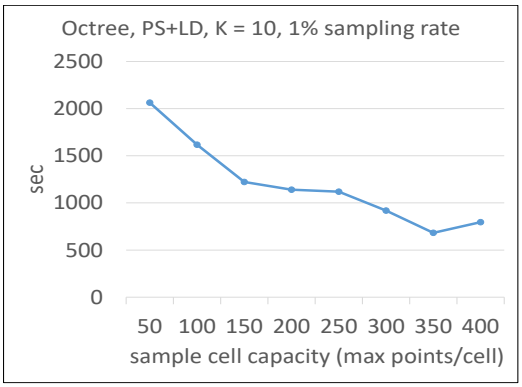

Figure 4.44: Exec. time of OT using PS+LD, to determine the best cell capacity (1% sampling rate).

4.5.3 Octree vs 3D Grid, using Plane-Sweep + LessData

In Figures 4.45-4.47, 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.

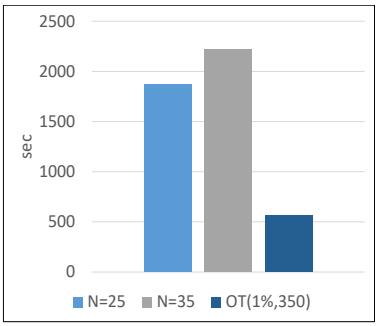

Figure 4.45: Exec. time of OT and 2 versions of GD, using PS+LD, for $K = 5$.

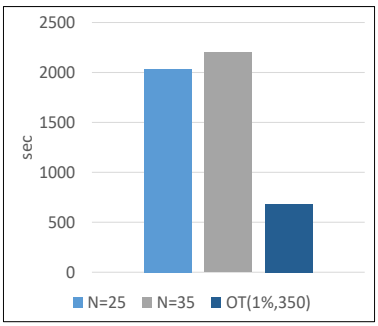

Figure 4.46: Exec. time of OT and 2 versions of GD, using PS+LD, for $K = 10$.

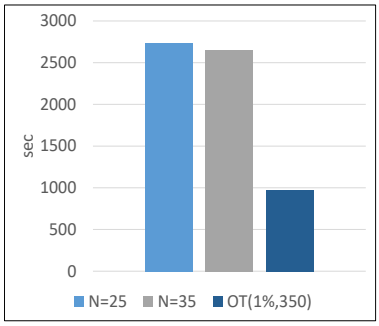

Figure 4.47: Exec. time of OT and 2 versions of GD, using PS+LD, for $K = 20$.

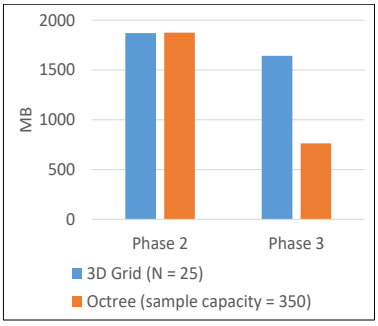

Figure 4.48: Phases 2 & 3 size of OT and GD, using PS+LD, for $K = 5$.

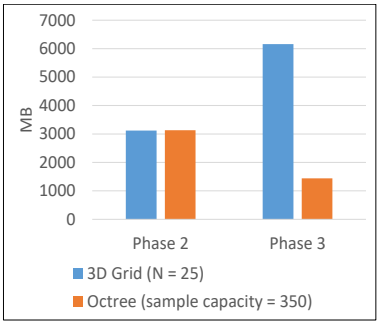

Figure 4.49: Phases 2 & 3 size of OT and GD, using PS+LD, for $K = 10$.

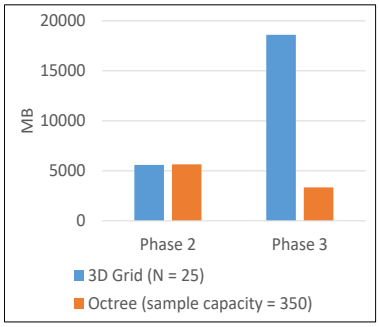

Figure 4.50: Phases 2 & 3 size of OT and GD, using PS+LD, for $K = 20$.

The explanation behind these numbers lies in the graphs of Figures 4.48-4.50,

where we depict the output sizes of Phases 2 and 3 for Grid with $N = 25$ and Octree with capacity = 350 (both using Plane-Sweep + 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 Figures 4.51-4.53, for the same algorithmic settings of Figures 4.48-4.50. 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.

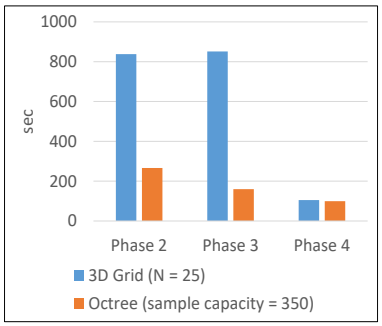

Figure 4.51: Phases 2-4 exec. time of OT and GD, using PS+LD, for $K = 5$.

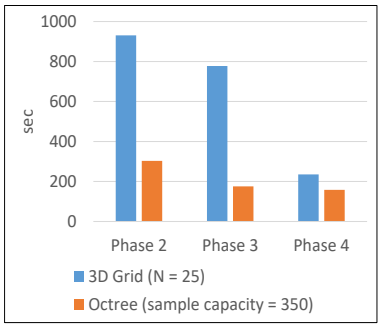

Figure 4.52: Phases 2-4 exec. time of OT and GD, using PS+LD, for $K = 10$.

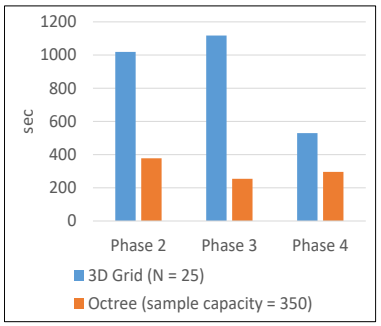

Figure 4.53: Phases 2-4 exec. time of OT and GD, using PS+LD, for $K = 20$.

4.5.4 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.

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 Figures 4.54-4.55.

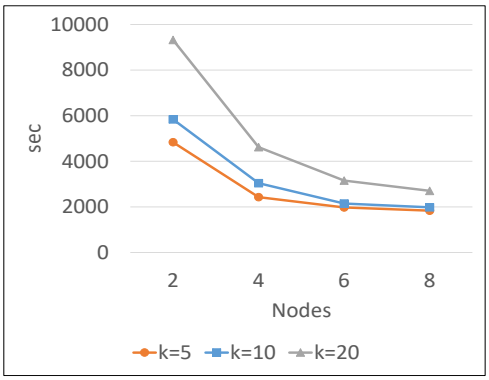

Figure 4.54: 3D GD: PS+LD, N = 25.

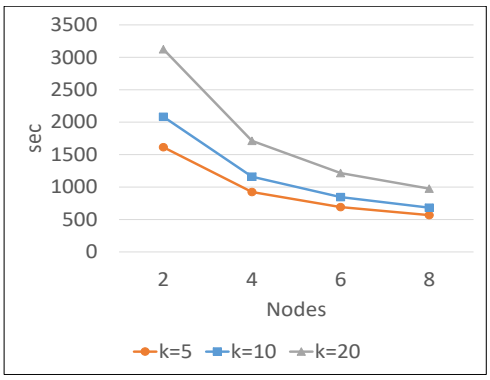

Figure 4.55: OT: PS+LD, 1% sampling rate, sample cell capacity = 350.

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

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 Figures 4.56-4.57. 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).

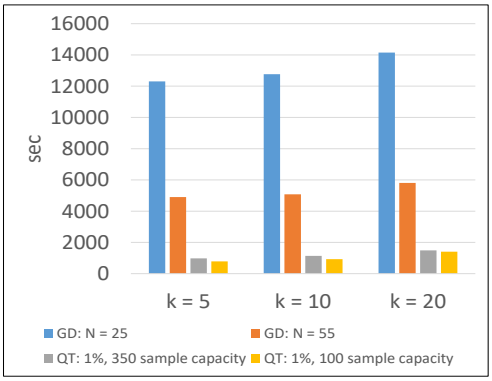

Figure 4.56: Exec. time of OT and 3D GD, using PS+LD.

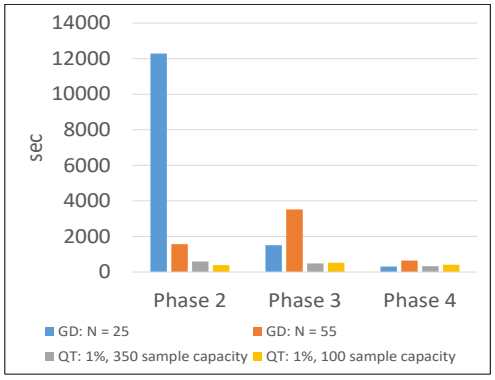

Figure 4.57: Phases 2-4 exec. time of QT and 3D GD, using PS+LD, $K = 20$.

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%. Figure 4.57 shows the huge Phase 2 time of the old best Grid, which is explained by the Training points distribution in Table 4.5 (which is analogous to Table 4.3). There are many cells with hundreds of thousands of points, causing delay to the new best Grid Phase 3. Octree achieves a better points distribution (even the old best one) as shown in Table 4.6 (which is analogous to Table 4.4), where all cells contain limited number of Training points.

4.6 Comparison to existing algorithms

In this section we test our work against other well-known algorithms of the literature. Specifically, we compare to HBRJ [91], PGBJ [47] and also our implementation of the base

Table 4.5: 3D Grid Training points distribution

	number of cells containing Training points	
number of 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	29639
total	15625	166375
non empty	4558 (29.17%)	30079 (18.08%)

Table 4.6: Octree Training points distribution

	number of cells containing Training points			
capacity	**<50K**	**>50K**	**total**	**non empty**
350	12086	0	12482	12086 (96.83%)
100	45649	0	48378	45649 (94.36%)

algorithm [58] (for the sake of completeness of this comparison). The source code and pre-compiled 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 4.4 and 4.5. 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 4.4 and 4.5 for the base and the new algorithm, respectively, and the recommended parameters mentioned in [91] for HBRJ and in [47] for PGBJ.

In Figure 4.58, we can see how the four algorithms compare to each other in 2D, using

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

$11M$ datasets. Obviously, the slowest, by far, is HBRJ. The fastest is our Quad tree, combined with Plane-Sweep and LessData. PGBJ and the base algorithm take the 3rd and 4th places. Our algorithm is 3 - 4 times faster than PGBJ.

In Figure 4.59, where 3D datasets are used, the last one to finish is once more HBRJ, followed by the base 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 base algorithm in 3D previously in this paper.

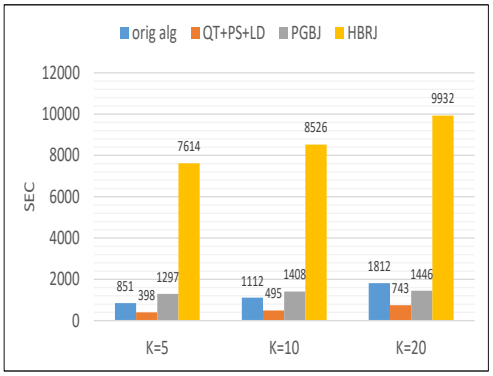

Figure 4.58: 2D comparison, $11M \times 11M$ datasets.

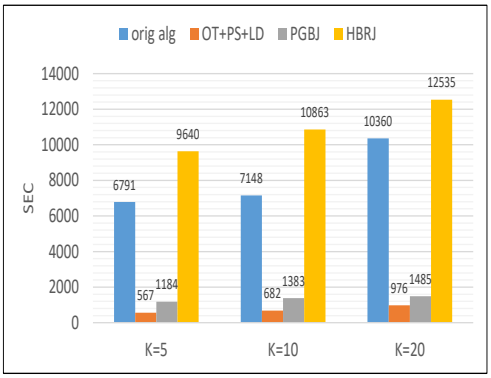

Figure 4.59: 3D comparison, $11M \times 11M$ datasets.

When using almost half of the dataset sizes (Figures 4.60 and 4.61), 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 ourperformed by PGBJ). The base 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 Plane-Sweep and Quad/Oct tree partitioning.

4.7 Conclusions

In [54], we have presented a set of improvements that can be used to accelerate the performance of the algorithm that was presented in [58] 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),

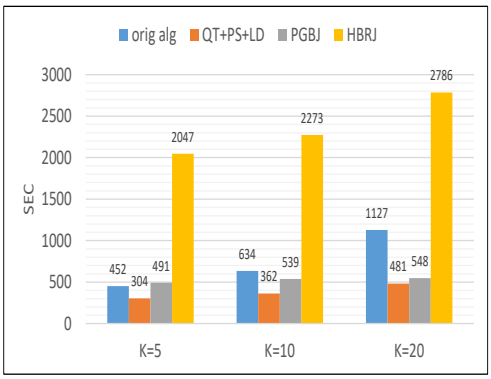

Figure 4.60: 2D comparison, $5M \times 5M$ datasets.

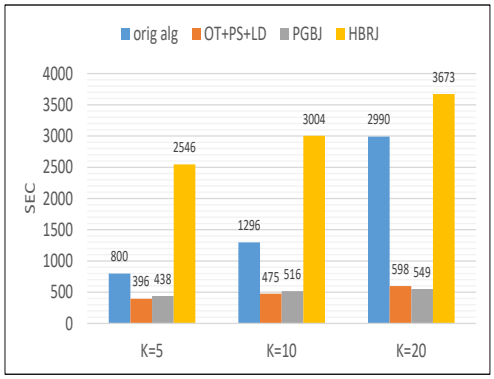

Figure 4.61: 3D comparison, $5M \times 5M$ datasets.

- 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 base algorithm),
- smaller Phase 3 output gives another significant boost (about 40% improvement over the base 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 base and improved algorithms by varying the number of computing nodes and dataset size. The performance improvement was 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 base algorithm were very effective, especially in 3D and for bigger K's, where the base algorithm is inefficient.

Chapter 5

The K Group Nearest-Neighbor Query

5.1 Introduction

The Group (K) Nearest-Neighbor (GKNN) query [60] belongs to the big family of Nearest-Neighbor searches, which has been widely studied in Computer Science and has numerous modern applications, like GIS [61], mobile computing [57], clustering [34], outlier detection [45], facilities management [60], etc. Between two datasets of points, this query retrieves the K points of one dataset (called Training) with the smallest sum of distances to every point of the other dataset (called Query). The Training is considered a static dataset queried by multiple Query datasets. Consider a group of friends (Query points), residing in different areas of a city, arranging for a meeting to a public place (Training point), e.g. restaurant, café, etc. They would make a list of K such places with the smallest sum of distances to all friends (as a group) and then choose among these places the one that best suits most of them. Without effective pruning and calculation techniques this can be a very demanding query, because it involves distance calculations that may be in the order of millions or more.

There have been several attempts to solve GKNN queries effectively in the past, however all of them regarded data that fitted on a single machine and focused on geometric methods to accelerate calculations and prune distant points. In a series of publications, we have introduced, for the first time in literature, a parallel and distributed algorithm that efficiently solves this query using datasets that fall into the Big Data category. More specifically, the Query is supposed to be small enough to fit into the memory of a single machine, while the Training is very large and will be partitioned into the distributed file system. The algorithm consists of seven phases, four local and three distributed, executed in an interleaved way

(each distributed phase follows a local one). Distributed phases perform CPU intensive calculations on big amounts of data that can be processed independently of each other, contrary to the local phases that process rather small amounts of data which need to be processed as a whole. The algorithm features several different partitioning, refining, pruning and computation techniques, some of which are adapted from the literature, while others are introduced for the first time.

The algorithm was first presented in [52] and was implemented in the Apache Hadoop framework. Later in [53] an improved version of the algorithm was presented, along with a SpatialHadoop implementation and a study of what happens internally during the operation of the algorithm, using specialized cluster wide metrics. In [55] a prepartitioning method was introduced for the Training, which proved to be very effective. All these versions of the algorithm were extensively tested using a large variety of configuration parameters and spatial datasets.

5.2 Related work

Nowadays, researchers, developers and practitioners worldwide are benefiting from cluster-based systems supporting large-scale data processing. There are a lot of works on specific spatial queries using Hadoop-MapReduce frameworks [26, 27]. Moreover, the GKNN query [60, 61] has received considerable attention from the spatial database community, due to its importance in numerous applications, and it has been actively studied in centralized environments. However, it has not attracted similar attention for parallel and distributed frameworks.

GKNN queries were introduced in [60]. Given two sets of points T and Q, a GKNN query retrieves the K point(s) of T with the smallest sum of distances to all points in Q. In [60], the authors designed three different methods, MQM (multiple query method), SPM (single point method) and MBM (minimum bounding method), to evaluate a GNN query that minimizes the total distance from a set of Query points to a training point. MQM performs incremental nearest-neighbor (NN) queries for each point in the Query set and combines their results using a *threshold algorithm* [22]. MQM will terminate the procedure of search if all the retrieved data objects so far are below a threshold distance (i.e. K-th minimum aggregate distance). MQM traverses an R-tree indexing data [30] multiple times and it can access the

same Training point more than once. SPM calculates the centroid of the Query set, which is a point in space with a small sum of distances to all query points. The centroid is used to prune the search space by exploring the triangular inequality. MBM treats Query points as a whole, i.e., MBR of Query points, and prunes unqualified data nodes as early as possible. SPM and MBM, find the GNNs in a single traversal of the R-tree. MBM dominates MQM and SPM in both I/O and CPU costs. In particular, MBM is better than SPM in terms of node access and CPU cost, while MQM is the worst.

In [61], the methods proposed in [60] have been extended to minimize the minimum and maximum distance in addition to the total distance with respect to a set of Query points. All these methods assume that the data points are indexed using an R-tree [30] and can be implemented using both depth-first and best-first tree-search algorithms. In [41], the authors propose two pruning strategies for GKNN queries which take into account the distribution of Query points. Such methods employ an ellipse to approximate the extent of multiple Query points, and then derive a distance or minimum bounding rectangle using that ellipse to prune intermediate nodes in a depth-first search via an R*-tree [3]. These methods are also applicable to the best-first traversal. The experimental results show that the proposed pruning strategies are more efficient than the methods presented in [60]. In addition, [40] explores new geometric insights, such as the minimum enclosing ball, the convex hull and the furthest Voronoi diagram to prune more intermediate nodes.

A new method to evaluate a GKNN query for non-indexed data points using projection-based pruning strategies was presented in [48]. Two points projecting-based ANN queries algorithms were proposed, which can efficiently prune the data points without indexing. This new method projects the Query points into a special line, on which their distribution is analyzed, for pruning the search space. In [56], a new property in vector space was proposed and, based on it some efficient bound estimations were developed for two most popular types of ANN queries (sum and maximum). Taking into account these bounds, indexed and non-indexed ANN algorithms were designed. From the experimental point of view, the proposed algorithms showed interesting results, especially for high dimensional queries.

Other related contributions in this research line have been proposed in the literature. In [32] an efficient algorithm for GKNN query considering privacy preserving was proposed, and the GKNN algorithms [61] for point locations were extended to regions in order to preserve user privacy. In [94], the GKNN query in road networks based on network voronoi

diagram was solved. In [35], the reverse top-K GNN query is presented. In [92], the K Nearest-Neighbor (KNN) and GKNN queries are extended to get a new type of query, called K Nearest Group (KNG) query. It retrieves closest elements from multiple data sources, and finds K groups of elements that are closest to a given Query point, with each group containing one object from each data source. For uncertain databases, probabilistic GKNN query was studied in [42, 44].

To emphasize even more the importance of the GKNN query in different fields of data processing, we highlight that [20] addresses the problem of GNN query monitoring for moving objects in Euclidean space. [43] discusses GNN queries with moving query points. In [78], the authors study the GNN queries in presence of obstacles. [46] examines GNN queries on graphs. [69] deals with GKNN queries in spatial network databases, and in [28], the problem of GNN search on road networks that incorporate cohesive social relationships is presented.

Recently, in [67] the GKNN query is studied, considering non-indexed data sets, since this case is frequent in practical applications. Two (RAM-based) Plane-Sweep algorithms were presented, that apply optimizations emerging from the geometric properties of the problem. In addition, an extensive experimentation process was executed, using real and synthetic data sets, highlighting the most efficient algorithm by reducing the number of points involved in query processing and the number of distance computations.

SpatialHadoop [17] is a comprehensive extension to Hadoop-MapReduce that injects spatial data awareness in each Hadoop layer. In particular, the operation layer enables the efficient implementation of spatial operations, considering the combination of the spatial indexing in the storage layer with the new spatial functionality in the MapReduce layer. Many spatial operations have been implemented in SpatialHadoop as range, K nearest-neighbors and spatial join queries [17]; K closest pairs, K nearest-neighbors join, ϵ distance join queries [26,27]; reverse K nearest-neighbors query [23], skyline queries [36], etc. These research papers indicate that SpatialHadoop is an excellent spatial processing MapReduce framework to implement distributed versions (MapReduce algorithms) of different spatial queries.

Contrary to previous methods that are all based on centralized systems, in [52] we proposed the first MapReduce algorithm to effectively process the GKNN query in a parallel and distributed environment. Utilizing ideas and elements from previous work (query definition and heuristics [60], processing without indexes and PS heuristics [65, 67], repartitioning data [24, 27]), in this paper we present an algorithm consisting of seven phases, local or

distributed, and exploit pruning heuristics, as well as two partitioning (Grid and Quadtree) and two computation techniques (Brute-Force and Plane-Sweep). All these are tested using several datasets, both real and synthetic ones.

The algorithm in [53] utilized ideas and elements from previous works, namely query definition and heuristics [60], processing without indexes and PS heuristics [65, 67], repartitioning of data [24, 27]. A new high performance point/cell filtering method was introduced as well as several other calculation improvements and output data minimization techniques, which greatly improved total algorithm performance. A SpatialHadoop version was also presented, utilising a novel two level partitioning technique and enhanced pruning, using spatial filter functions. We also presented what happens internally (under the hood) during the operation of the algorithm, using special metrics that highlight the effect of each of the techniques used. All these were tested using several datasets, both real and synthetic ones. The winner configuration, in most cases, was the combination of Grid partitioning with Brute-Force reducers, against Quadtree partitioning and Plane-Sweep reducers.

In [55] we presented a modified version of the algorithm which utilizes a prepartitioning phase that transforms the Training to a new dataset. This dataset contains all cells together with their contained Training points and it replaces the Training as input in all phases. The implementation regarded the Hadoop version with Grid partitioning and Brute-Force reducers. The experiments showed a great performance increase, compared to the previous version.

5.3 Algorithms presentation

The algorithm presentation in this section is a unified approach of the ones presented in [52] and [53] and their differences will be noted when necessary. The algorithm modification in [55] will be presented separately, because it is quite different. We start with two spatial datasets of points, Q or Query and P or Training. They are in the form {id, x, y}. Query is small enough to fit in memory of a single machine, while Training is considered a very large one and it will be partitioned into smaller parts.

An overview of the algorithm is as follows:

1. **Preliminary step.** Local calculation of a sample-based Quadtree, the sorted list of Query points, Query MBR and centroid coordinates, the sum of distances from centroid to Q. These are needed by most of the pruning heuristics.

2. **Phase 1.** Distributed computation of the number of Training points per cell (needed by Phase 1.5).

3. **Phase 1.5.** Local discovery of a group of cells that contain at least K Training points in total. Two different approaches that have vast performance differences will be presented.

4. **Phase 2.** Distributed computation of GKNN lists, one per intersected cell. Pruning heuristics are applied.

5. **Phase 2.5.** Local merging of the GKNN lists into one with the best points found so far.

6. **Phase 3.** Distributed computation of GKNN lists for the non-intersected cells. Heuristics are applied to prune distant cells and save unnecessary calculations. All new candidate neighbors are checked against the best ones from Phase 2.5.

7. **Phase 3.5.** Local phase (final) that merges the list of Phase 2.5 with lists from Phase 3 into the final one.

A more detailed per phase analysis, including partitioning and computational methods, as well as description of the pruning heuristics used, follows.

5.3.1 Pruning heuristics

Some heuristics from [60, 65, 67] used in Phases 2 and 3 for fast pruning of intermediate nodes and Training points will be presented.

Figure 5.1 shows the Training MBR, containing an arbitrary node D (from either Grid or Quadtree partitioning) and the Query MBR M with its centroid. Table 5.1 contains some basic symbols used in this section.

- **Heuristic 1 [60]:** Node D can be pruned if:

$$mindist(D, c) \geq \frac{best\_dist + sumdist(c, Q)}{|Q|}$$

- **Heuristic 2 [60]:** Node D cannot contain qualified points if:

$$mindist(D, M) \geq \frac{best\_dist}{|Q|}$$

- **Heuristic 3 [60]:** Node D can be pruned if:

$$\sum_{q \in Q} mindist(D, q) \geq best\_dist$$

- **Heuristic 4 [60]:** Training point p can be pruned if:

$$|Q| \cdot dist(p, c) \geq best\_dist + sumdist(c, Q)$$

- **Heuristic 5 (Plane-Sweep only) [65,67]:** When the sweep line is outside M and checks Training point p:

$$sumdx(p, Q) = |Q| \cdot |p.x - c.x|$$

Also, for every Training point p:

$$sumdx(p, Q) \leq sumdist(p, Q)$$

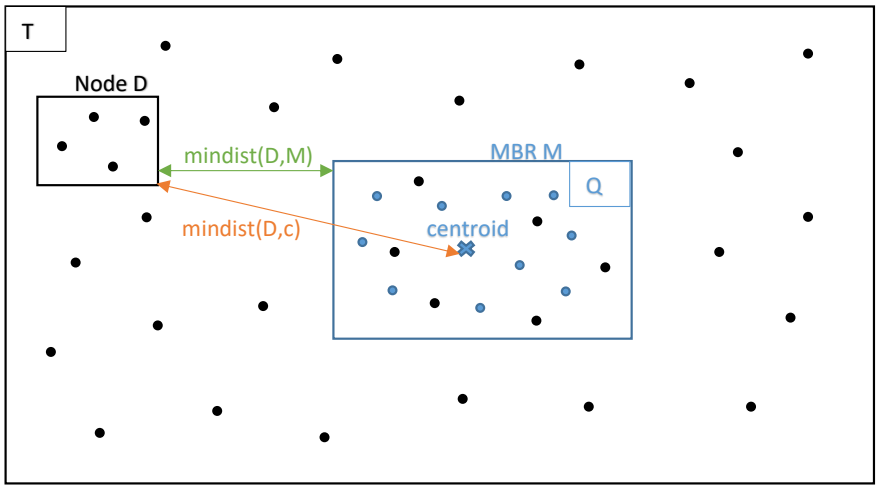

Figure 5.1: Query MBR and arbitrary node.

Heuristics 1, 2, 3 easily prune most of intermediate cells when used in Mapper 3_1, while heuristics 4, 5 are extensively used inside Reducers 2, 3. Heuristic 4 is used both in Brute-Force and Plane-Sweep reducers, while heuristic 5 is applied in Plane-Sweep only. Without heuristic 4, $sumdist(p, Q)$ would have to be calculated for every point p, while heuristic 5 gives us $sumdx(p, Q)$ with just a single x-distance calculation.

A visualization of Heuristics 1, 2 and 4 can be seen in Figure 5.2. The geometric data comes from the datasets described in Figure 5.29 in the experimental section. The circle has its center on the Query centroid and its radius is calculated from the right member of Heuristic

Table 5.1: Symbols

Symbol	Description
$\|Q\|$	cardinality of Q
c	centroid of Q
M	MBR of Q
D	rectangle of node (cell) D
$sumdist(p, Q)$	sum of distances from point p to all points of Q
$sumdx(p, Q)$	sum of x-distances from point p to all points of Q
$best\_dist$	K-th nearest-neighbor distance
$mindist(D, p)$	min distance between D and point p
$mindist(D, M)$	min distance between D and M

1 inequality. The rounded rectangle's perimeter has a distance from the Query MBR equal to the right member of Heuristic 2 inequality.

Any cell from any partitioning method that intersects with the circle, will not be pruned by Heuristic 1. Any cell from any partition method that intersects with the rounded rectangle will not be pruned by Heuristic 2. All cells outside these two shapes, are automatically pruned. The cells that passed Heuristics 1 and 2 will be checked by Heuristic 3, which is more costly (not

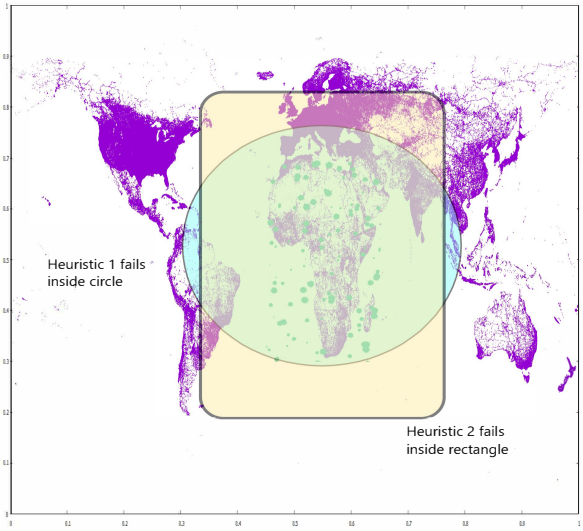

Figure 5.2: Heuristics 1, 2 and 4 visualization.

depicted here) and those cells which will not be pruned at all, will be checked for neighbors, inside Phase 3. The circle also applies to Heuristic 4, every single Training point outside the circle will be pruned.

5.3.2 Space partitioning techniques in Mappers

Each of the distributed Phases 1, 2 and 3 consists of two sub phases: *Map* and *Reduce*. The Mappers deal with the geometry, which essentially means the eligibility of the cells and/or points to be processed, while the Reducers perform the calculations on the eligible cells and points.

Two different partitioning techniques are used, *Grid* and *Quadtree*. Only the Training is partitioned (the Query fits into memory and does not need partitioning).

In Grid partitioning (Figure 5.3) the space is divided into $N \times N$ equal square cells. N is a user defined parameter and it plays a major role in the algorithm's performance. When changing N, we are modifying each cell's size (bigger N means more, smaller cells, smaller N means fewer, bigger cells) and thus its ability to contain more or less points. When a cell contains many points, the number of calculations inside it will grow quadratically. Few but big cells will give their Reducers a lot of calculations to perform. Many but small cells will result in faster computations, but will generate many Reducer processes, which is also undesired.

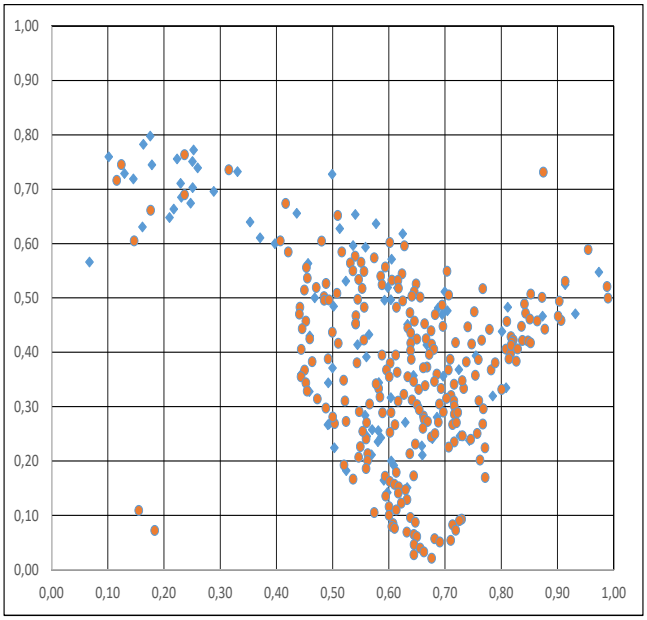

Figure 5.3: Grid partitioning.

Grid's advantage is fast point and cell location. If we know N, we are just a few easy algebraic calculations away of knowing where each cell and point is located. Its disadvantage is that we cannot control the number of points inside each cell, it just applies blind cuts. Some cells may have few or no points, while others may have thousands or more. This obviously leads to unbalanced Reducer loads and may cause serious slowdowns.

To counter this disadvantage, we are also testing Quadtree partitioning. Quadtree divides a square area into four equal quadrants recursively (Figure 5.4). The user defines a maximum Training points capacity per cell, which is used to end recursion. Large, empty cells will not be further divided. But if they contain more Training points than the allowed capacity, they will be partitioned into smaller quadrants. This way we are creating a tree structure with more equally balanced cells. Like Grid's N, capacity also plays significant role in cell size and its number of contained points, thus affecting performance. Quadtree's disadvantages are its creation process (local sampling of the Training and computation) and its point location procedure, which begins from the root node and traverses the tree top to bottom.

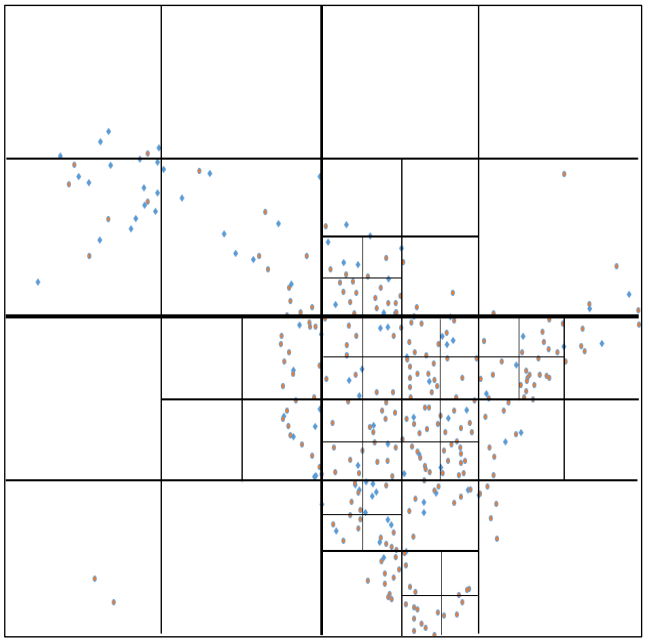

Figure 5.4: Quadtree partitioning.

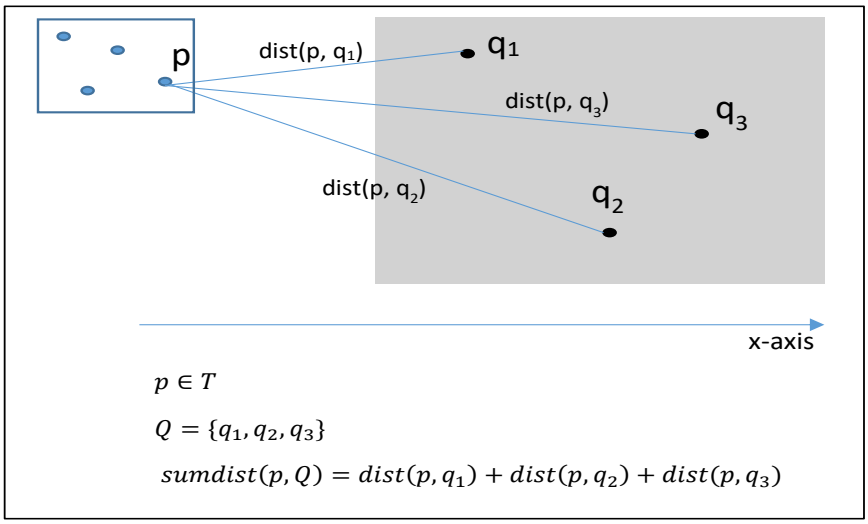

Figure 5.5: Brute-Force.

5.3.3 Computational methods in Reducers

Two different computational techniques in Reducers of Phases 2 and 3[1]: *Brute-Force* and *Plane-Sweep* are applied.

Brute-Force approach is the simplest one (Figure 5.5). Imagine a single cell containing several Training points, from which we want to find the best K ones. We start calculating the sum of distances from every Training point to all Query points. The first K points are inserted into a max heap, along with their sum of distances. We continue to check every other Training point in the cell and if we find a smaller sum of distances, we replace the top of the heap with this point, and so on. Only Heuristic 4 (Section 5.3.1) is applied here.

Plane-Sweep approach is more sophisticated. Again, imagine a single cell (Figure 5.6). First, we sort the Query Points and the cell's Training points by ascending x-coordinate into two lists. We set a vertical sweep line to the Training point that is closest to the median Query point (q_2 in the figure) and start moving it to the opposite direction. The first K Training points that it meets are placed into a max heap, along with their sum of distances, like previously. But for every other point swept by the sweep line, we only calculate the sum of its x-distances from every Query point. We compare this sum to the heap's top and if it's bigger, it means that the sum of the point's Euclidean distances will also be bigger and it is not necessary to be calculated. Furthermore, all the other points on this side of the sweep line obviously have a bigger sum of x-distances, so they are all skipped, saving us a lot of unnecessary

[1]Phase 1 Reducer only performs a simple summation

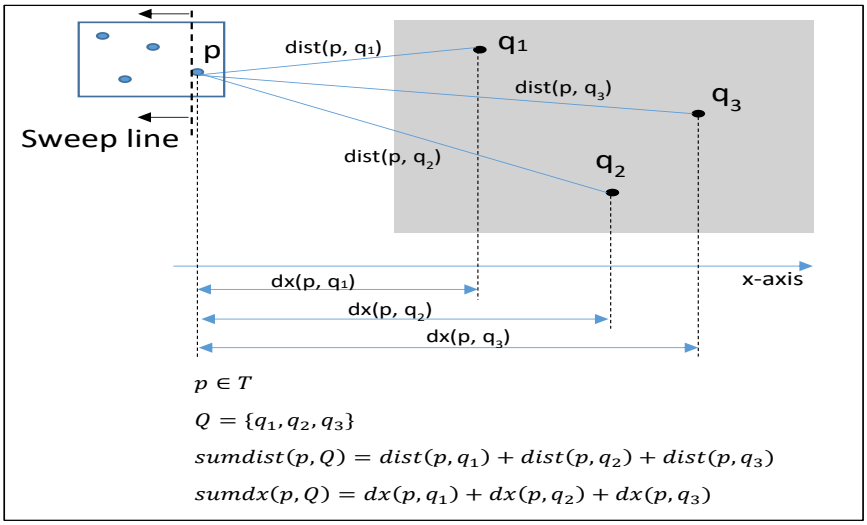

Figure 5.6: Plane-Sweep.

calculations. If, however, the sum of a point's x-distances is smaller than the heap's top, we must also calculate the sum of its Euclidean distances and compare it to the heap's top and replace it, if it's smaller too. Heuristic 4 is again used here and also Heuristic 5 (Section 5.3.1), which gives us a fast computation of x-distances sum outside the Query MBR.

Plane-Sweep's performance advantage is obvious, it may prune lots of points and it calculates mostly sums of x-distances, which are computationally cheaper than Euclidean distances. Its disadvantages are the double calling of summing functions (x-distances and Euclidean distances, when the first one fails) and the sorting of the points lists.

To avoid sorting the Query points repeatedly when checking each cell, we sort them only once in the preliminary step and store the result in HDFS, so that every phase can use it. So now we only have to sort the Training points in each cell under process.

The x-distance check fail (and the subsequent Euclidean sum calculation) is encountered very often in cells closely around the Query MBR, because the distances are smaller. In distant cells the x-distance check is almost always successful. This observation was the key to improve our initial Phase 1.5 methodology.

We introduced the "Fast Sums" technique in [53] and used it in both approaches, when iteratively calculating the sums of Euclidean or x-distances to compare them with the heap top. Instead of waiting for the whole iteration on the Query to complete and then compare the result with the heap top (as in the literature), we are checking the heap top distance in every iteration step, and if the partial sum is bigger, the loop exits. This can obviously save hundreds or thousands of distance calculations and boost the performance. We also apply this

technique in Heuristic 3 (Section 5.3.1). In the experiments we use the full sums calculations, unless otherwise stated and we make the comparison with "Fast Sums" later in a special chart.

Algorithms 5 and 6 present the pseudocode for these two methods.

Algorithm 5 Get neighbors list using Brute-Force method

Input: K, Query MBR coordinates, centroid coordinates, sum of distances from centroid to all Query points, Query points list, Training points list in this cell

```
 1: initialize empty max heap "Neighbors" (Phase 2) or
    import "Neighbors" from Phase 2.5 (Phase 3)
 2: for all tpoint in Training points list do
 3:     if Neighbors.size < K then
 4:         calculate sum of distances from tpoint to all Query points
 5:         insert {tpoint, sum of distances} into "Neighbors"
 6:     else
 7:         get best_dist = K-th neighbor's sum of distances
 8:         calculate {centroid, tpoint} distance
 9:         if Heuristic 4 is true then
10:             prune this tpoint
11:         else
12:             calculate sum of distances from tpoint to all Query points
13:             if sum of distances < best_dist then
14:                 remove K-th neighbor (heap top) from "Neighbors"
15:                 insert {tpoint, sum of distances} into "Neighbors"
16: if Neighbors is changed then
17:     return "Neighbors"
18: else
19:     return an empty max heap
```

Notes on Algorithm 5:

- "Neighbors" is a priority queue that takes elements of type {point id (int), sum of distances (float)} and is ordered by the sum of distances.
- While "Neighbors" heap is not full, insert every new point encountered (lines 2 - 5).
- If is full, check new candidate points using Heuristic 4 (lines 9 - 10) and after that check their sum of distances (lines 12 - 15) before insertion.

- Lines 16 - 19 return only lists with new neighbors to avoid duplicate lists and save bandwith.

Notes on Algorithm 6:

- Lines 2 - 4 sort the Training points in each cell and define the starting Training point.
- Lines 5 - 22 deal with the Training points of this cell inside the Query MBR.
- Lines 23 - 41 deal with the Training points of this cell outside the Query MBR.
- We separate these two areas to make use of Heuristic 5 (line 30), which easily calculates the sum of x-distances, when outside of the Query MBR.
- Lines 15 - 17 and 30 - 32 are the essence of Plane-Sweep method, which massively prunes Training points, before calculating sums of Euclidean distances.
- Heuristic 4 is also used here (lines 12, 13 and 35, 36). Note that inside MBR it precedes the sum of x-distances check, while outside MBR it succeeds it. The reason is Heuristic 5 in the second case.
- Lines 42 - 45 return only lists with new neighbors to avoid duplicate lists and save bandwith.

5.3.4 Preliminary step (local)

The first phase (Figure 5.7) is a local preliminary step that takes the two datasets as input. From the Query it calculates its MBR and centroid coordinates and the sum of distances from all Query points to its centroid. All these are needed by the pruning heuristics. If we want to use the Plane-Sweep reducers, it also sorts the Query points by x-distance. When using Grid partitioning, only the Query is needed. In Quadtree version, the Training is also used to create a local, cluster-wide Quadtree.

In Quadtree partitioning, we take a small sample (usually about 1%) of the Training and we give a user defined integer as the maximum capacity of Training points per sample cell. The algorithm then recursively splits the root cell into four equal sub cells, until the capacity requirements are met for every cell. The sample-based tree is stored into HDFS for cluster wide availability. Note that sampling is done in the name node by selecting points directly from a file stored in HDFS (in distributed storage). Since only a very small portion is stored

Algorithm 6 Get neighbors list using Plane-Sweep method

Input: K, Query MBR coordinates, centroid coordinates, sum of distances from centroid to all Query points, sorted Query points list, Training points list in this cell

```
 1: initialize empty max heap "Neighbors" (Phase 2) or
    import "Neighbors" from Phase 2.5 (Phase 3)
 2: sort Training points list in x-ascending order
 3: find tpoint from Training points that is closest to the median Query point
 4: scan all this cell's Training points using the sweep line, starting from the one we found in line 3
    and moving to the opposite direction(s) from the median Query point
 5: if tpoint inside Query MBR then
 6:     if Neighbors.size < K then
 7:         calculate sum of distances from tpoint to all Query points
 8:         insert {tpoint, sum of distances} into "Neighbors"
 9:     else
10:         get best_dist = K-th neighbor's sum of distances
11:         calculate {centroid, tpoint} distance
12:         if Heuristic 4 is true then
13:             prune this tpoint
14:         else
15:             calculate sum of x-distances from tpoint to all Query points
16:             if sum of x-distances > best_dist then
17:                 prune this and all the remaining Training points in this cell in this direction
18:             else
19:                 calculate sum of distances from tpoint to all Query points
20:                 if sum of distances < best_dist then
21:                     remove K-th neighbor (heap top) from "Neighbors"
22:                     insert {tpoint, sum of distances} into "Neighbors"
23: else                                                    ▷ point outside Query MBR
24:     if Neighbors.size < K then
25:         calculate sum of distances from tpoint to all Query points
26:         insert {tpoint, sum of distances} into "Neighbors"
27:     else
28:         get best_dist = K-th neighbor's sum of distances
29:         calculate {centroid, tpoint} x-distance
30:         calculate sum of x-distances from tpoint to all Query points using Heuristic 5
31:         if sum of x-distances > best_dist then
32:             prune this and all the remaining Training points in this cell in this direction
33:         else
34:             calculate {centroid, tpoint} distance
35:             if Heuristic 4 is true then
36:                 prune this tpoint
37:             else
38:                 calculate sum of distances from tpoint to all Query points
39:                 if sum of distances < best_dist then
40:                     remove K-th neighbor (heap top) from "Neighbors"
41:                     insert {tpoint, sum of distances} into "Neighbors"
42: if Neighbors is changed then
43:     return "Neighbors"
44: else
45:     return an empty max heap
```

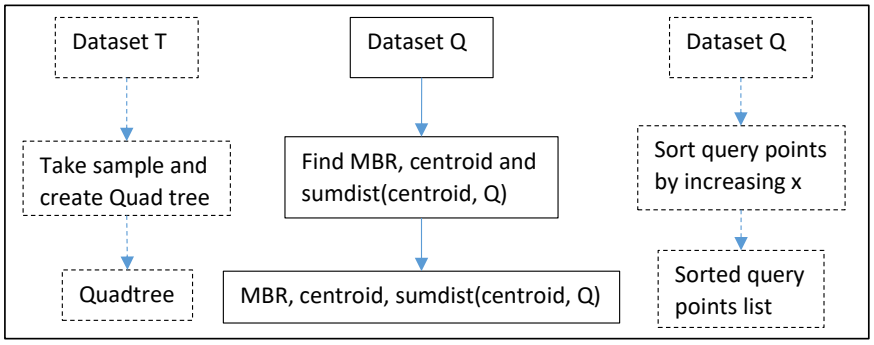

Figure 5.7: Preliminary step (local).

in the name node, a very large dataset can be handled. If the dataset is too large, it could be performed in a distributed manner (easy transformation).

These two partitioning parameters, N and capacity, are greatly affecting performance and will be studied thoroughly in experiments.

The centroid, regarding the GKNN query, is defined as the point with the minimum sum of distances from all Query points [60]. It's generally close to, but different from the centroid in Mathematics or Mechanics, which is defined as:

$$(x_c, y_c) = (\frac{\sum x_i}{|Q|}, \frac{\sum y_i}{|Q|})$$

where $x_i, y_i \in Q$ and $|Q|$ is the cardinality of the Query.

To locate the centroid, we must find the (x, y) that minimize the function

$$dist(x, y) = \sum_i \sqrt{(x - x_i)^2 + (y - y_i)^2}, (x_i, y_i) \in Q$$

which represents the sum of distances from centroid's coordinates (x, y) to all Query points (x_i, y_i).

This problem cannot be solved analytically, so we make use of the well-known numerical method *gradient descent*, as suggested in [60]. In short words, we calculate the gradient[2] of the above function at a starting point, for example the centroid as defined in Mechanics. The gradient vector always points to the direction of the fastest increase of the function and the opposite direction is towards the fastest decrease, pointing to the local minima. We get the next best point iteratively using a small step, like this:

$$(x, y)_{next} = (x, y)_{current} - step \cdot \nabla dist(x, y)_{current}$$

[2] $gradf(x, y)$ or $\nabla f(x, y) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$

After that we calculate the sum of distances from all Query points to the centroid (i.e. the previous distance formula). All these results are stored into HDFS, so that every node can access them when needed.

5.3.5 Phase 1 (distributed)

The next phase is the first distributed one. It counts the Training points inside each cell using the Training as input (Figure 5.8). The Quadtree is also used as input in Quadtree version. The Mapper emits a *{cell_id, 1}* for each point encountered, while the Reducer just sums up the 1's. The result is a list of key-value pairs in the form *{cell_id, number of contained points}* and it is stored in HDFS. Next phase depends on this result.

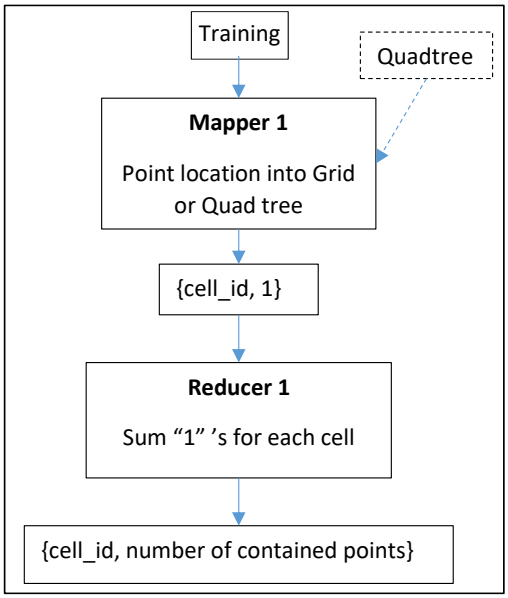

Figure 5.8: Phase 1 (distributed).

5.3.6 Phase 1.5 (local), the MBR and the centroid approaches

In this local phase, we seek for a first estimation of the final GKNN query, that would be as close as possible to optimal. This may be achieved by discovering a (small as possible) "target" group of cells that are more likely to obtain this good estimation. Needless to say that these cells must contain at least K Training points in total. In Phase 1 we have counted the points within each cell (Figure 5.9), so we can use this result.

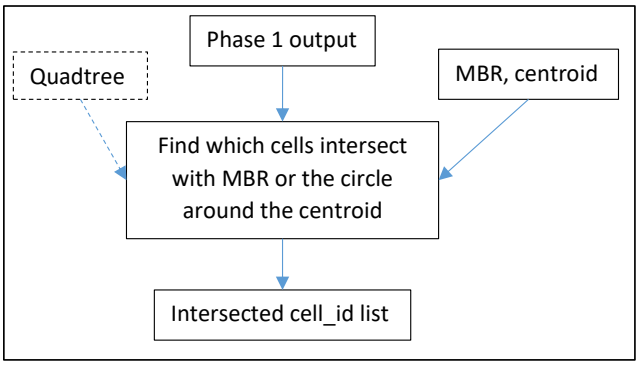

Figure 5.9: Phase 1.5 (local).

In [52] we used all the cells that intersect with the Query MBR (Figure 5.10). It is obvious that the GKNN candidates will most likely be found inside or closely around the MBR than far away from it, while the farthest cells will probably be pruned in Phase 3. In the case that these cells did not contain K points or more, the MBR was gradually expanded until they did. In order to locate these cells, we used the output of Phase 1, the MBR and centroid coordinates derived in the Preliminary step and (in Quadtree version) the Quadtree. This phase outputs a list of the intersecting target cells and takes only a few seconds to complete. The result list is loaded into HDFS.

Extensive experimentation on the algorithm [52] has shown that the subsequent Phase 2, was by far the hardest one to complete. This happens because the use of MBR yields a rather large number of intersecting cells that results to too many calculations. Furthermore, the pruning heuristics inside Reducers (especially when using the Plane-Sweep algorithm), do not work efficiently because of the short distances.

In order to improve Phase 2 performance, we had to reduce the number of eligible cells and this was the greatest improvement introduced in [53]. After several experiments we ended using the centroid as the center of a circle which is expanding itself to intersect cells with at least K Training points in total (Figure 5.11). Intuitively, the centroid seems an ideal choice, since, by definition, it has the smallest sum of distances to all Query points, so other Training points around it should be the first ones to check. It turned out that the number of overlapped target cells is drastically reduced and the calculations in Phase 2 finish about 90% faster. Moreover, we have found that the GKNN list produced in Phase 2.5 (Section 5.3.8) is in many cases identical to the final one, which justifies that the centroid circle method is an extremely fast and reliable choice.

As shown in the experiments section, the number of eligible cells was reduced from a few

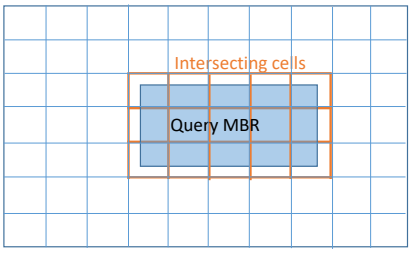

Figure 5.10: Cells overlapping with MBR.

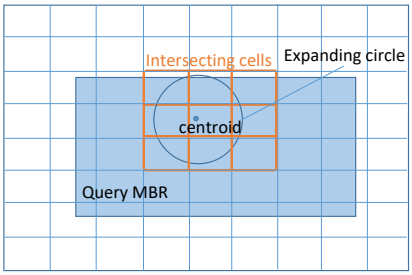

Figure 5.11: Cells overlapping with centroid's circle.

thousands, when using the first approach, to less than ten, without compromising the quality of the GKNN estimation.

Algorithms 7 and 8 present the pseudo code for these two methods.

Algorithm 7 Get overlaps using MBR method (Phase 1.5)

Input: number of Training points per cell from Phase 1, Query MBR coordinates, K, N (Grid) or Quadtree

```
1:  initialize empty list "Overlaps"
2:  overlaps_points = 0
3:  while overlaps_points < K do
4:      check every cell from Phase 1 output (it contains only non-empty cells)
5:      if cell overlaps with Query MBR then
6:          add its Training points number to overlaps_points
7:          add cell to "Overlaps"
8:      if all cells overlapping with Query MBR are checked and still overlaps_points < K then
9:          expand Query MBR by a small percentage and check again
10: return "Overlaps"
```

Notes on Algorithm 7:

- If Query MBR contains less than K Training points, lines 8, 9 ensure that the method will not fail.
- N and Quadtree are used in the geometric calculations (cell and point location, overlapping).

Notes on Algorithm 8:

Algorithm 8 Get overlaps using Centroid Circle method (Phase 1.5)

Input: number of Training points per cell from Phase 1, Query MBR & centroid coordinates, K, N (Grid) or Quadtree

```
1:  initialize empty list "Overlaps"
2:  overlaps_points = 0
3:  Locate centroid cell and get its Training points (from Phase 1 output)
4:  Get centroid cell width: ds
5:  Initialize circle with centroid as its center and starting radius: R = 0.5*ds
6:  Set radius increase step: dr = 0.5*R
7:  overlaps_points += centroid cell points
8:  while overlaps_points < K do
9:      check all cells that intersect with centroid circle
10:     if cell contains any Training points (crosscheck with Phase 1 output) then
11:         add its Training points to overlaps_points
12:         add cell to "Overlaps"
13:     increase radius: R += dr
14: return "Overlaps"
```

- The circle around centroid is initialized using a small radius, we used half the cell width (line 5).
- A small radius increase step is also initialized (line 6) and is added to R (line 13) until the cells intersecting the circle contain at least K Training points.
- N and Quadtree are used in the geometric calculations (cell and point location, overlapping).

5.3.7 Phase 2 (distributed)

Phase 2 is distributed (Figure 5.12) and computes GKNN lists, one for each intersected cell from Phase 1.5 (Figure 5.10 or 5.11). The Mapper takes the Training and overlaps lists (Phase 1.5 output) and emits each intersected cell along with all its contained Training points. The Quadtree is also used as input in Quadtree version. The Reducer receives Mapper output and additionally the Query (in Brute-Force version) or the sorted query points list (in Plane-Sweep version) and additionally MBR and centroid coordinates from Preliminary step and

finds the K GNNs from overlapped cells, while applying heuristics 4 and 5 (Section 5.3.1). The output consists of multiple lists with *null* as key and the K points in this cell with the smallest sum of distances from Q as value, sorted by ascending distance.

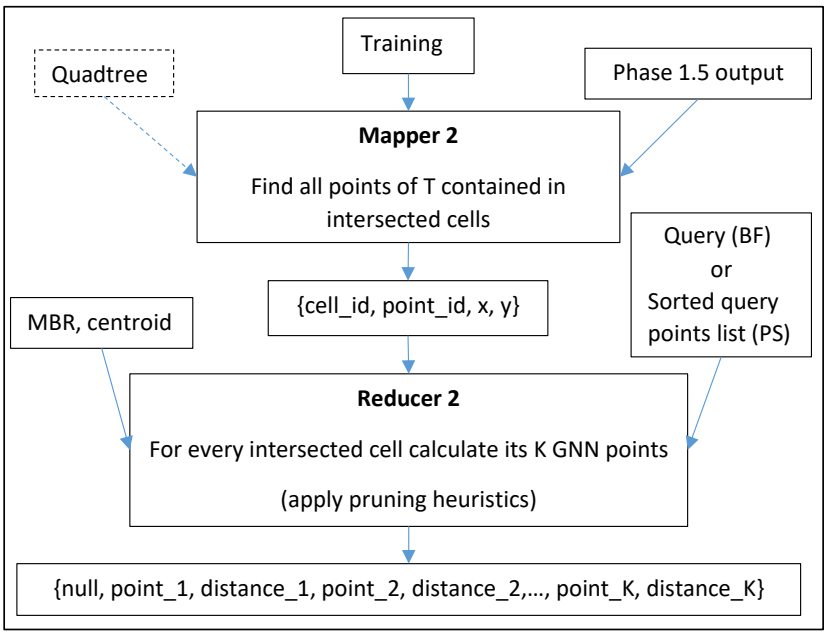

Figure 5.12: Phase 2 (distributed).

5.3.8 Phase 2.5 (local)

In [52] we used cell id as the output key, but it wasn't needed in subsequent phases after all, so we decided to remove it in [53] and replaced it with *null* to save network bandwidth.

Algorithms 9 and 10 present the pseudocode for the Mapper and the Reducer.

Algorithm 9 Mapper 2

Input: Training points dataset (partitioned), overlaps list from Phase 1.5, N (Grid) or Quadtree

```
1: for all tpoint in Training points (from partition) do
2:     get tpoint's cell using N or Quadtree
3:     if cell in overlaps then
4:         return {cell id, Training point id, x, y}
```

Notes on Algorithm 10

- Output contains only value, key is "null".

Local Phase 2.5 (Figure 5.13) reads all GKNN lists from Phase 2 output and merges them into a single GKNN list containing only the best K points along with their sum of distances,

Algorithm 10 Reducer 2

Input: Mapper's output, Query MBR & centroid coordinates, Query points list sorted (PS) or non-sorted (BF), K

1: read Mappers' output and create a list of the Training points per cell

2: call Brute-Force or Plane-Sweep method to get neighbors list

3: **return** {point1, sumDist1, point2, sumDist2, ..., pointK, sumDistK}

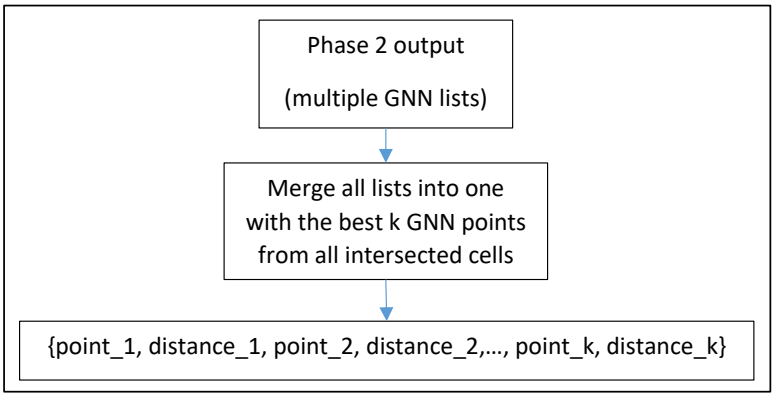

Figure 5.13: Phase 2.5 (local).

ordered by increasing distance. The list is loaded into HDFS and is the final one in most cases, since Phase 3 will prune most if not all of the distant cells, especially when using the MBR intersection method of Phase 1.5.

5.3.9 Phase 3 (distributed)

This distributed phase (Figure 5.14) works very much like Phase 2, for the rest of the cells that do not intersect with the Query MBR or the circle around the centroid (all except for the "Intersecting cells" in Figures 5.10 and 5.11). The first Mapper applies heuristics 1, 2, 3 (Section 5.3.1) to easily prune most of the distant cells and outputs the non-pruned ones' id's as keys and *"true"* as value. In many cases no cells pass these heuristics, meaning that GNN list of Phase 2.5 is the final one. The second Mapper feeds the Reducer with the points within the non-intersected cells. If some cells pass through, the Reducer works like Phase 2 and prepares a GKNN list for every non-pruned cell. If we did not make use of these heuristics, we would have to perform expensive distance computations for thousands of cells and millions of points, see the comparison in Table 5.8 in the experiments.

The first Mapper needs multiple inputs, such as Phase 1 output (contains non-empty cells), Phase 1.5 output (to get the non-intersected cells), Phase 2.5 output (to get the k-th best

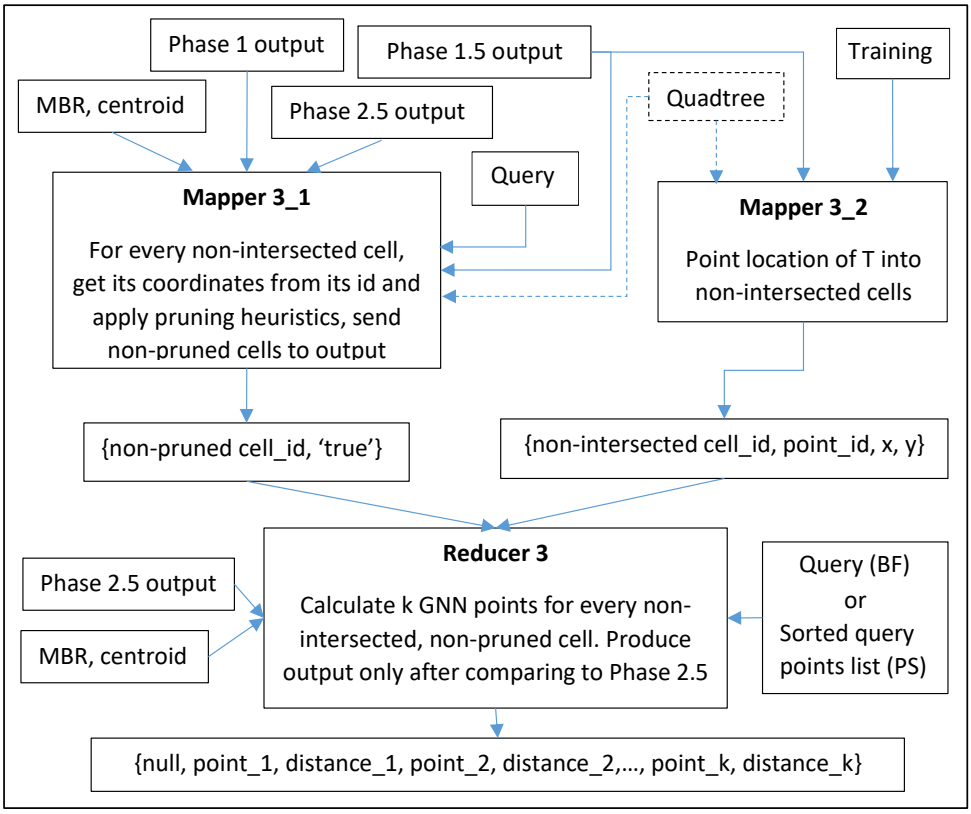

Figure 5.14: Phase 3 (distributed).

distance found so far, needed by heuristics 1, 2, 3), the Query (needed by heuristic 3), the Quadtree (in Quadtree version), the coordinates of the Query MBR and centroid, and the sum of distances from all Query points to the centroid (for heuristic 1), from Preliminary step.

The second Mapper needs the list of intersected cells (from Phase 1.5), the Quadtree and the Training.

The Reducer receives both Mappers' outputs, which are the non-pruned, non-intersected cells (those that have been tagged "true" by the first Mapper) and their Training points and also the Query (or the sorted Query points list, in Plane-Sweep version), MBR and centroid coordinates and Phase 2.5 output. The last one is used to load the best list so far and use it as a base to compare each new candidate point before inserting. To save network bandwidth, the Reducer will output a list only if new points are inserted. In [52] we used cell id as output key, which we have replaced with *null* in [53], as in Phase 2, and also the Reducer used to create a new list for every non-pruned cell, without comparing it to Phase 2.5 list first, flooding the network with mostly useless data (neighbors lists that will probably be rejected by the next phase).

During our experiments we have found that in some cases the list from Phase 2.5 was wrong or incomplete, when using the "centroid method" in Phase 1.5. This means that the cells provided to Phase 2 by Phase 1.5 were not the best candidate neighbors. In these cases, the pruning heuristics in Phase 3 Mapper let the appropriate cells pass through, so the correct neighbors inside them were found and replaced the incorrect ones in the final list (Phase 3.5).

Algorithm 11 presents the pseudocode for Mapper 3_1. Mapper 3_2 pseudocode is similar to Mapper 2 (Algorithm 9) if you rephrase line 3's if condition as "cell NOT in overlaps". Also Reducer 3 is similar to Reducer 2 (Algorithm 10) and it takes as input both Mappers' outputs and also the best neighbors list from Phase 2.5 to compare with each new candidate point. Reducer 3 creates a list (as in line 1) of the Training points that carry the "true" tag, only, and then proceeds as in lines 2 and 3.

Algorithm 11 Mapper 3_1

Input: number of Training points per cell from Phase 1, Query MBR & centroid coordinates, overlaps list from Phase 1.5, Query points list, best neighbors list from Phase 2.5, N (Grid) or Quadtree

```
1: for all cell from Phase 1 output (it contains only non-empty cells) do
2:     if cell in overlaps then
3:         prune it
4:     if cell does not pass Heuristics 1, 2, 3 then
5:         prune it
6:     return {non-pruned cell id, "true"}
```

Notes on Algorithm 11:

- Cells that are not in overlaps (they were checked in Phase 2) and have passed cell pruning heuristics 1, 2, 3 get "true" tag and go to the output.

- Both Mappers' output key is a cell id, so that the values that will be paired (Training points with coordinates and with or without "true" tag) can be easily separated in the Reducer and keep only the Training points in non-overlapped and non-pruned cells (those that carry the "true" tag).

- N and Quadtree are used in the geometric calculations in heuristics (cell and point location) together with the best list from Phase 2.5.

5.3.10 Phase 3.5 (local)

This phase is the last one, it takes the outputs of Phases 2.5 and 3, which are all the available GKNN lists, and merges them into the final one (Figure 5.15). This is a local phase.

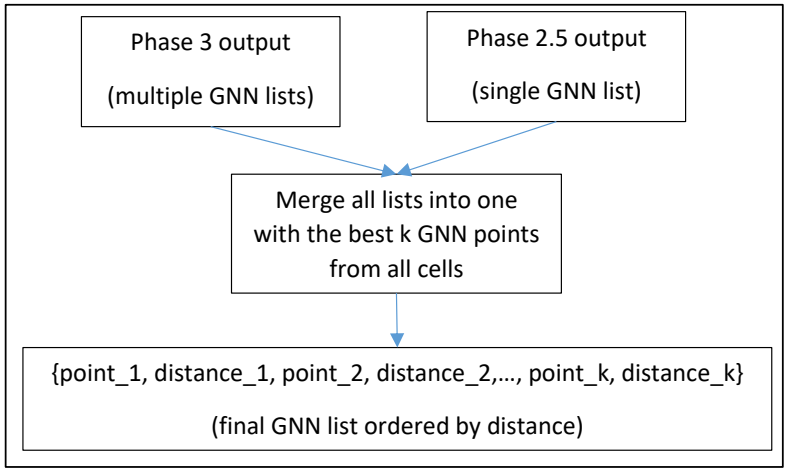

Figure 5.15: Phase 3.5 (local).

5.4 Algorithm porting to SpatialHadoop

SpatialHadoop is a MapReduce framework that provides support for spatial data management in Hadoop. This provides native management of spatial structures and indexing mechanisms that allow selective access to specific regions of spatial data to achieve more efficient query processing algorithms. Moreover, SpatialHadoop uses a two-level index structure. At a global level, spatial data is partitioned and indexed by using a known partitioning technique (Grid, STR, Quadtree, etc.), and then distributed to cluster nodes so that spatial proximity data is in the same partition/node. Also for each partition, you can create a local index (e.g. R-tree) that facilitates the design of efficient query algorithms that access only part of the data. For the use of these indexes, there are two components: SpatialFileSplitter and SpatialRecordReader. On one hand, the SpatialFileSplitter uses the global index and a filter function to return only those partitions that the query needs. On the other hand, the SpatialRecordReader is responsible for reading the spatial structures (e.g. points) and local index of each partition to provide them as input for the Map tasks. For instance, we can process the elements of a partition as a whole instead of having to read element by element.

The main challenges / opportunities when using SpatialHadoop can be summarized as

follows:

- The use of spatial index structures.
- The use of the two-level index structure and pruning mechanisms.
- Handling of spatial data distribution, considering spatial data proximity.
- Handling of skewed spatial data.

This section presents our proposal for the transformation of the GKNN query algorithm to SpatialHadoop. We present a general overview of the algorithm with the considerations taken into account and the changes made in each phase or step of the algorithm for Hadoop.

5.4.1 Algorithm description

An overview of the algorithm is as follows:

1. **Preliminary step.** Local calculation of the sorted list of Query points, the Query MBR and centroid coordinates, the sum of distances from centroid to Q. These are needed by most of the pruning heuristics.
2. **Partitioning step.** The Training is partitioned by any of the available partition techniques in SpatialHadoop: Grid, STR, Quadtree, Hilbert, etc.
3. **Phase 1.** Distributed Grid repartitioning of each partition by a fixed cell size and computation of the number of Training points per cell. Needed by Phase 1.5.
4. **Phase 1.5.** Local discovery of a group of cells that contain at least K Training points in total. By using MBR or centroid circle methods.
5. **Phase 2.** Distributed computation of GKNN lists, one per intersected cell. Pruning heuristics are applied.
6. **Phase 2.5.** Local merging of the GKNN lists into one with the best points found so far.
7. **Phase 3.** Distributed computation of GKNN lists or the non-intersected cells. Heuristics are applied to prune distant cells and save unnecessary calculations. All new candidate neighbors are checked against the best ones from Phase 2.5.

8. **Phase 3.5.** Local phase (final) that merges the list of Phase 2.5 with lists from Phase 3 into the final one.

In general, the GKNN query algorithm in SpatialHadoop takes advantage of the spatial features that SpatialHadoop provides to Hadoop while dealing with the challenges of the unique characteristics of this query. Therefore, it embeds some modifications over the original algorithm: A partitioning step using a built-in partition technique from SpatialHadoop, a repartitioning step to better redistribute the workload and the use of spatial indexes and filter functions in the different phases of the original algorithm.

5.4.2 Partitioning step (distributed)

To work with big spatial datasets in SpatialHadoop, it is mandatory to have them partitioned by some partitioning technique (Grid, STR, Quadtree). For instance, this process will divide the Training into several files, one for each partition, based on the default HDFS block size and then, distribute and replicate them to different nodes. Moreover, a master index is created with the MBR and number of Training points inside each partition/file that can be used to easily access only certain elements based on a given query, e.g. “Give me the partition where a point is located”.

5.4.3 Phase 1 (distributed)

For instance, with the repartitioning technique each top-level partition made by SpatialHadoop is split into cells with a smaller number of elements allowing a better distribution of the workload from the mappers to the reducers. Moreover, this new partitioning process can be fine-tuned to the query’s parameters, while not modifying the original partitions that work well with other built-in queries of SpatialHadoop. Finally, this technique allows us to apply the heuristics at a more fine-grain level, that is, global filtering is made by pruning top-level partitions and later, local filtering is applied on the cells of the non-pruned partitions. For instance, each top-level partition made by SpatialHadoop is split into cells with a smaller number of elements allowing a better distribution of the workload from the mappers to the reducers. Furthermore, it allows us to apply the heuristics at a more fine-grain level, that is, global filtering is made by pruning top-level partitions and later, local filtering is applied on the cells of the non-pruned partitions.

In this step, the Mapper emits a *{cell_id, 1}* for each point encountered on a cell of a partition, while the Reducer just sums up the 1's. The result is a list of key-value pairs in the form *{cell_id, number of contained points}* and it is stored in HDFS. This will be needed by following phases.

Cell size

Another important aspect that has been taken into consideration to get a fair comparison between plain Hadoop and SpatialHadoop is the cell size. For plain Hadoop, the dataset is divided by $N \times N$ Grid in equally sized cells. The problem is that if the same parameter is used for SpatialHadoop, the size of the cells is different from plain Hadoop (smaller or larger). For instance, SpatialHadoop partitions the Training in P partitions based on the HDFS block size, so if we want $N \times N$ cells after the repartitioning phase we need to divide each partition in $N' = N/P$. However, as shown in Figure 5.4 which depicts a Quadtree partitioning, the partitions are not equally sized and if each of them is repartitioned by an $N' \times N'$ Grid, the resulting cells won't be either. Moreover, some of them could be very small and the workload would not be well distributed.

So, the solution is to divide each partition by the same cell's width / height that is used on plain Hadoop. For instance, if we have a Training with an MBR of $(x_{min}, y_{min})(x_{max}, y_{max})$ and the number of cells N, cell's width cw can be calculated as:

$$cw = \frac{x_{max} - x_{min}}{N}$$

and we can find the number of cells for a partition P_i as N_{P_i}:

$$N' = \frac{P_i.x_{max} - P_i.x_{min}}{cw}$$

5.4.4 Phase 1.5 (local)

In the same way as for Hadoop, in this local phase, we try to find a small group of cells that will give us a first estimation of the final GKNN list. These cells must contain at least K Training points in total, so we need the result of Phase 1 for the counting and the index. Therefore, the main difference with the Hadoop version is precisely the availability of this index that allows both approaches to be carried out more efficiently.

The first approach based on the intersection of the cells with the Query MBR, is done in two steps: first, a global filtering is done by selecting the partitions with which it intersects and then refining, finally obtaining those cells that intersect with the MBR. As for the second approach based on the centroid, this is modified similarly, first looking for the partition in which it is located, to then locate the specific cell in which it is located. If the number of Training points in the cell were not K, contiguous cells would be added.

5.4.5 Phase 2 (distributed)

The algorithm for GKNN query in SpatialHadoop exploits the indexes to prune partitions that cannot contribute to the different GKNN lists of Phases 2. Before the map phase begins, a filter function will select the partitions that intersect (Phase 1.5 output) and are not pruned by heuristics in order to reduce the number of points to process. Furthermore, the files associated to the pruned partitions are not read, that is, there is no need to read all the distributed files of the Training.

The Mapper and Reducer of Phase 2 of the GKNN algorithm for SpatialHadoop are very similar to those of the Hadoop version with the only difference of the Mapper input. For instance, each Mapper task receives as input the data of a partition and an iterable of the Training points located in it. The rest of the phase remains the same: Points that are not in overlapped cells are pruned, and each selected point is sent to the Reducer of its corresponding cell where a Brute-Force or Plane-Sweep GKNN query algorithm is applied.

5.4.6 Phase 3 (distributed)

Phase 3 takes advantage of the same features of SpatialHadoop to reduce the reading and processing of distributed files that are part of the Training Dataset. In this case, a filter function prunes those partitions that are not part of the final solution by applying heuristics 1, 2 and 3. As for Mapper 3_1 and 3_2, these are combined into one that has as input the partition data, along with the points found in it. Therefore, we can eliminate those cells within a partition that have already been processed (output of Phase 1.5) and those that do not meet heuristics 1 and 2. Note that this could not be done before because the points of the same partition were not processed in the same Mapper task and the heuristics were calculated more than once. The application of heuristic 3 has been discarded in the Mapper, since its processing is very demanding and the number of tasks did not facilitate its parallelization. However, we have

a larger number of Reducer tasks, that is, we can calculate heuristic 3 at the beginning of the Reducer task and prune the corresponding cell. Moreover, we can apply heuristics 1, 2 and 3 with an updated threshold as the GKNN list of the current Reducer is updated, that is, we can prune cells that had not been previously eliminated if only the distance of the K-th nearest-neighbor distance obtained in Phase 2 is used.

5.5 Prepartitioning to improve performance

During the numerous experiments conducted on the previous algorithms, we noticed that in every distributed phase there was a repeated reading of the Training (reading all points, line to line) just to create a list of the cells with their contained points. The Mappers read every line, corresponding to a Training point, and converted it to a {cell, point} couple which was extensively used. This repeated conversion is of course CPU consuming, but more importantly it produces random cells in each machine, which are emitted by the Mappers and grouped during shuffling to reach the Reducers, travelling through HDFS from node to node. This is an unnecessary I/O costly operation, which we can tackle by prepartitioning the Training and creating a {cell, contained Training points} collection for every partitioned cell, only once, in the beginning. This collection is the output of the prepartitioning phase, it is stored in HDFS, and will be used as input in the next phases.

This algorithm only regards the Hadoop version. SpatialHadoop already incorporates a lengthy but effective prepartitioning process, which is different from this one.

A comparative description of the modified algorithm follows. Only the distributed phases are modified, the local ones remain unchanged and will not be presented again.

Phase 0: The Training dataset is read line to line (each line contains one point), the Mapper emits a {cell_id, point_id, point_x, point_y} for each point (using the Grid parameter, N) and the Reducer groups the points that belong to a certain cell together in the form: {cell_id, point1_id, point1_x, point1_y, point2_id, point2_x, point2_y,...}, where “cell_id” is the key and points data are the value of this MapReduce phase. As N grows bigger (Grid partitioning creates $N \times N$ equal sized cells), there will be more cells and less points inside each. The generated file contains all the information of the original Training dataset (points data) plus the cell ids. Each point is associated to only one cell. This phase is completely I/O bound, its running time grows proportionally to the size of the Training dataset and the output is of

similar size (Figure 5.16).

Preliminary step: Local calculation of Query MBR and centroid coordinates, the sum of distances from centroid to Q. These are needed by most of the pruning heuristics (Figure 5.17).

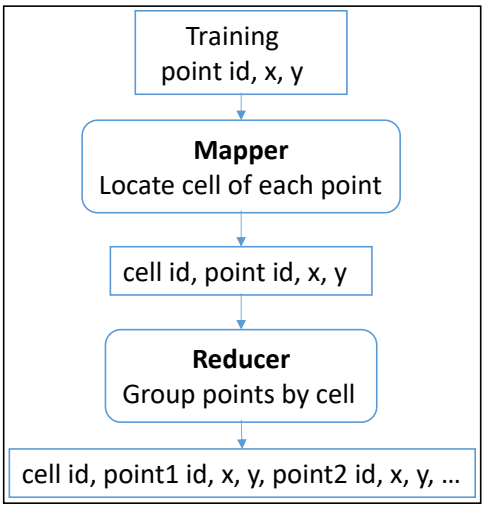

Figure 5.16: Phase 0 (prepartitioning).

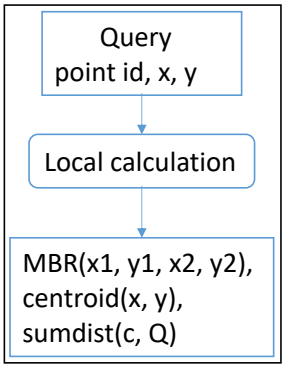

Figure 5.17: Preliminary phase (local).

Phase 1: In base algorithm, the Mapper reads every line (point) of the Training dataset, calculates its cell (using the Grid parameter, N) and emits a {cell_id, 1} couple in the output. The Reducer sums the 1's for each cell, which gives the number of Training points it contains (Figure 5.18). Now only a Mapper is needed, it reads the prepartitioned file from Phase 0 and just counts the Training points, which are already grouped by cell. The Reducer is omitted (Figure 5.19).

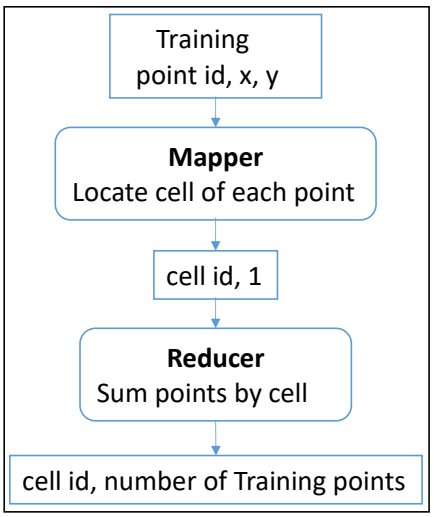

Figure 5.18: Phase 1 (base algorithm).

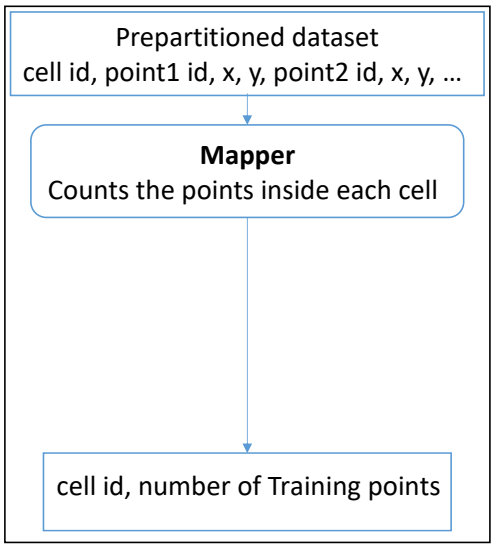

Figure 5.19: Phase 1 (new algorithm).

Phase 2: Distributed computation of GKNN lists, one per intersected cell. In base algorithm, the Mapper read every line (point) of the Training dataset, calculated its cell (using the Grid parameter, N), grouped their points and then let only the overlapped cells and their contained points pass to the Reducer. A lot of unnecessary cell calculations were performed here by the Mapper, i.e. the distant, non-overlapped cells, which will be computed again in the next phase (Figure 5.20). Now, the Mapper reads the prepartitioned file and the overlapped cells are found by just checking the key. The overlapped cell's points are the value and the whole line will pass to the Reducer (Figure 5.21).

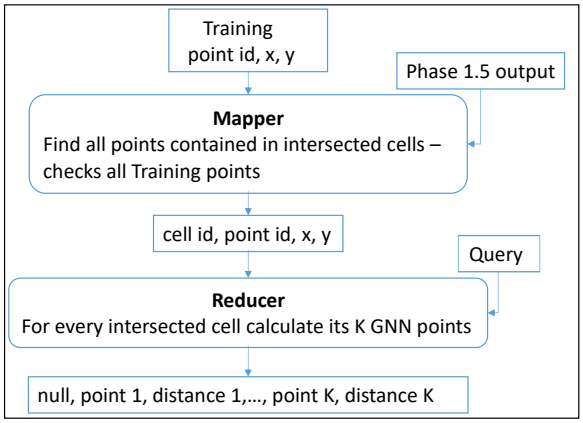

Figure 5.20: Phase 2 (base algorithm).

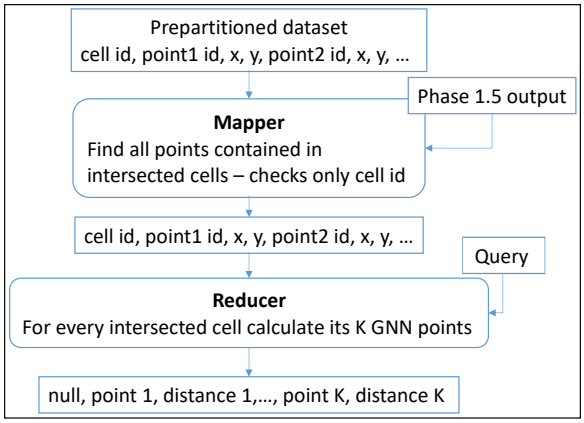

Figure 5.21: Phase 2 (new algorithm).

Phase 3: Distributed computation of GKNN lists for the non-intersected cells. Heuristics are applied to prune distant cells and save unnecessary calculations. All new candidate neighbors are checked against the best ones from Phase 2.5. In base algorithm, there were two Mappers here. The first one read Phase 1 output (non-empty cells) and pruned the distant cells, using heuristics, attaching a flag to the non-pruned, while the second one read the Training file again and emitted a list of cells with their contained Training points (the ones that were not checked by Phase 2). Both their outputs went to the Reducer, which only performed calculations on the non-pruned cells. A lot of shuffling was needed in this phase, to match both Mappers' outputs, and a lot of unnecessary cell calculations were performed, i.e. the overlapped cells, that were computed again in Phase 2 (Figure 5.22). Now, only the prepartitioned file is needed, instead of both the Training file and Phase 1 output, and only one Mapper to process it, while the second Mapper is now omitted (Figure 5.23).

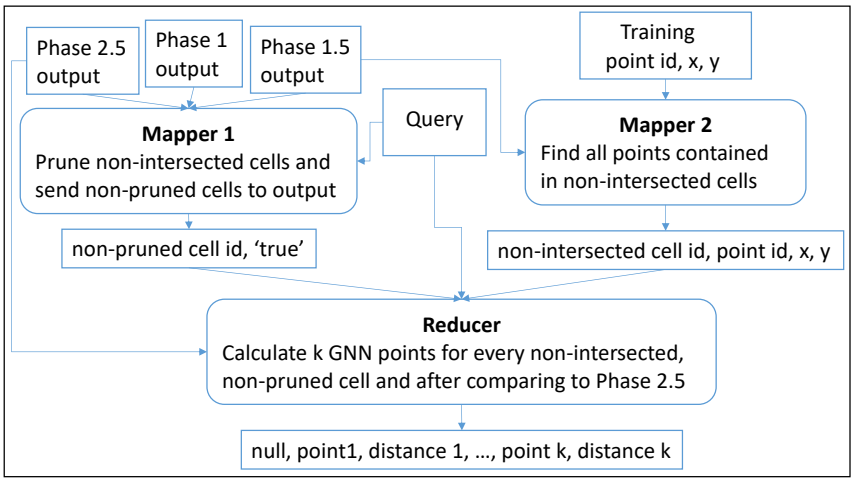

Figure 5.22: Phase 3 (base algorithm).

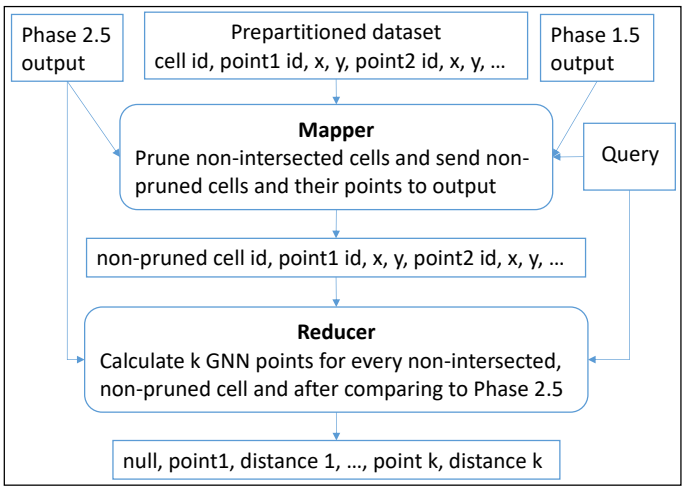

Figure 5.23: Phase 3 (new algorithm).

5.6 Experimental evaluation of the MBR algorithm

This is the very first version presented in [52]. Its differences from the consequent versions are:

1. No prepartitioning phase.
2. Only MBR intersection in Phase 1.5 (see Figure 5.10).
3. No Fast Sums technique in the Reducers (as described in 5.3.3).
4. Phases 2 & 3 output contains cell_id as key, not *null* (see Figures 5.12, 5.14).
5. Every cell checked in Phase 3 will emit a neighbor list, without comparing it first with Phase 2.5 list (as described in 5.3.9).

Partitioning methods used are Grid and Quadtree and calculation methods in Reducers are Brute-Force and Plane-Sweep.

In order to evaluate the behaviour of the proposed algorithms, we have used a real dataset as Training and six artificial datasets as Query (Table 5.2). The real dataset is a 10% sample of the one that contains the coordinates of buildings around the world from OpenStreetMap [17] and has 11,473,662 points. Its coordinates are normalized in $[0, 1]$ range. Inside it we have placed six artificial Query datasets, each containing 900,000 points (about 8% of the Training dataset), evenly distributed in their MBRs (covering 8% of the Training dataset MBR surface). Figure 5.24 shows the datasets on the plane. The Query MBRs are two squares (1 and 5), two horizontal rectangles (3 and 4) and two vertical rectangles (2 and 6). While the Query points contained are the same in number and distribution, the Training points in them are drastically different, both in number and distribution.

Table 5.2: Datasets

Dataset	**Num. of pts**	**MBR** (x_{min}, x_{max}) (y_{min}, y_{max})	**disk size**
10% buildings	11.473M	(0.0023, 0.9969) (0.0086, 0.4916)	383 MB
Query 1	900K	(0.1, 0.3), (0.3, 0.5)	29 MB
Query 2	900K	(0.4, 0.5), (0.1, 0.5)	29 MB
Query 3	900K	(0.6, 1.0), (0.3, 0.4)	29 MB
Query 4	900K	(0.1, 0.5), (0.0, 0.1)	29 MB
Query 5	900K	(0.6, 0.8), (0.0, 0.2)	29 MB
Query 6	900K	(0.6, 0.7), (0.1, 0.5)	29 MB

We presented experimentation results for the three worst case Querys, 1, 2 and 4. The results for the other three Querys are similar. This way we covered all shapes. We ran our algorithm in four different combinations for each Query MBR: Grid or Quadtree partitioning with Brute-Force or Plane-Sweep Reducers. We presented results of the average total time of ten measurements in seconds, as a function of the space partitioning parameter, which is N for Grid ($N \times N$ cells) and capacity for Quadtree (maximum capacity of Training points in the sample). In Grid it is obvious that the number of cells increases (quadratically) with

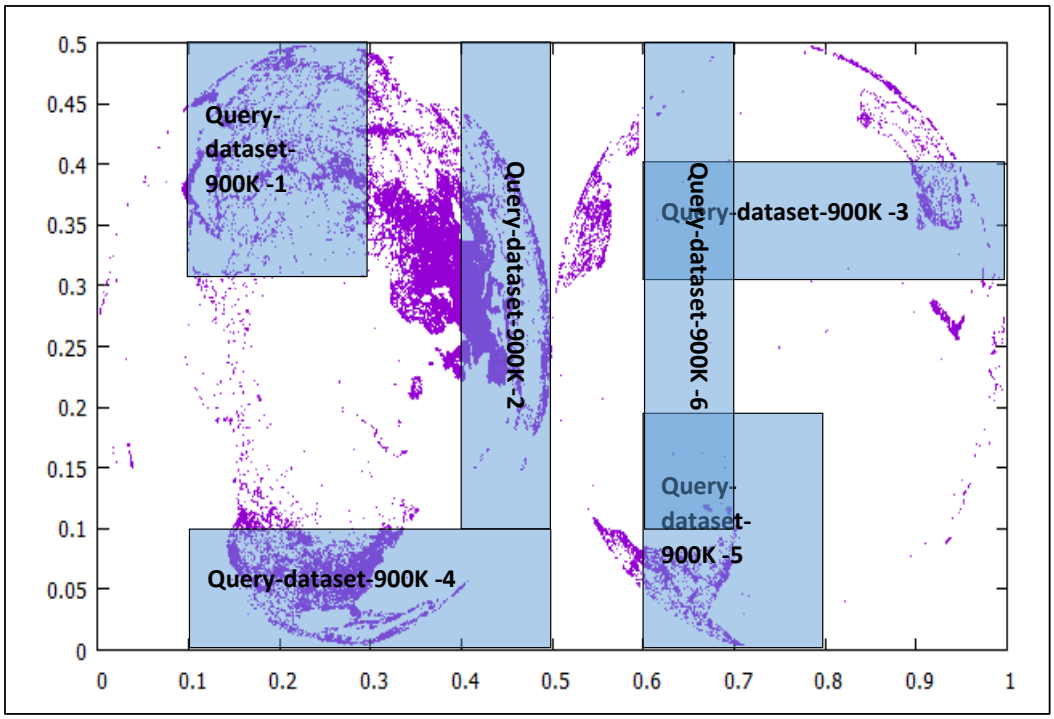

Figure 5.24: Query and Training datasets.

N, while in Quadtree it decreases with capacity (bigger capacities mean more points per cell, thus fewer and bigger cells).

A synopsis of the abbreviations we used in presenting our algorithm follows:

- GD, or $Grid$: Grid space partitioning used by Mappers,
- BF, or $BruteForce$: Brute-Force processing in Reducers,
- PS, or $PlaneSweep$: Plane-Sweep processing in Reducers,
- QT, or $Quadtree$: Quadtree space partitioning used by Mappers.

These abbreviations will also be used in combinations expressing the combined application of techniques, like $QT + PS$ (application of Quadtree Mapper and Plane-Sweep Reducer).

In Figure 5.25, we can see the performance comparison of all combinations for Querys 1, 2 and 4; the upper (lower) diagrams refer to Grid (Quadtree) partitioning, while each diagram displays the performance of Brute-Force and Plane-Sweep Reducers. Figure 5.26 shows a comparative chart of the best (regarding N and Reducer processing technique) Grid vs the best (regarding Capacity and Reducer processing technique) Quadtree, for each Query.

We draw some useful conclusions from these charts:

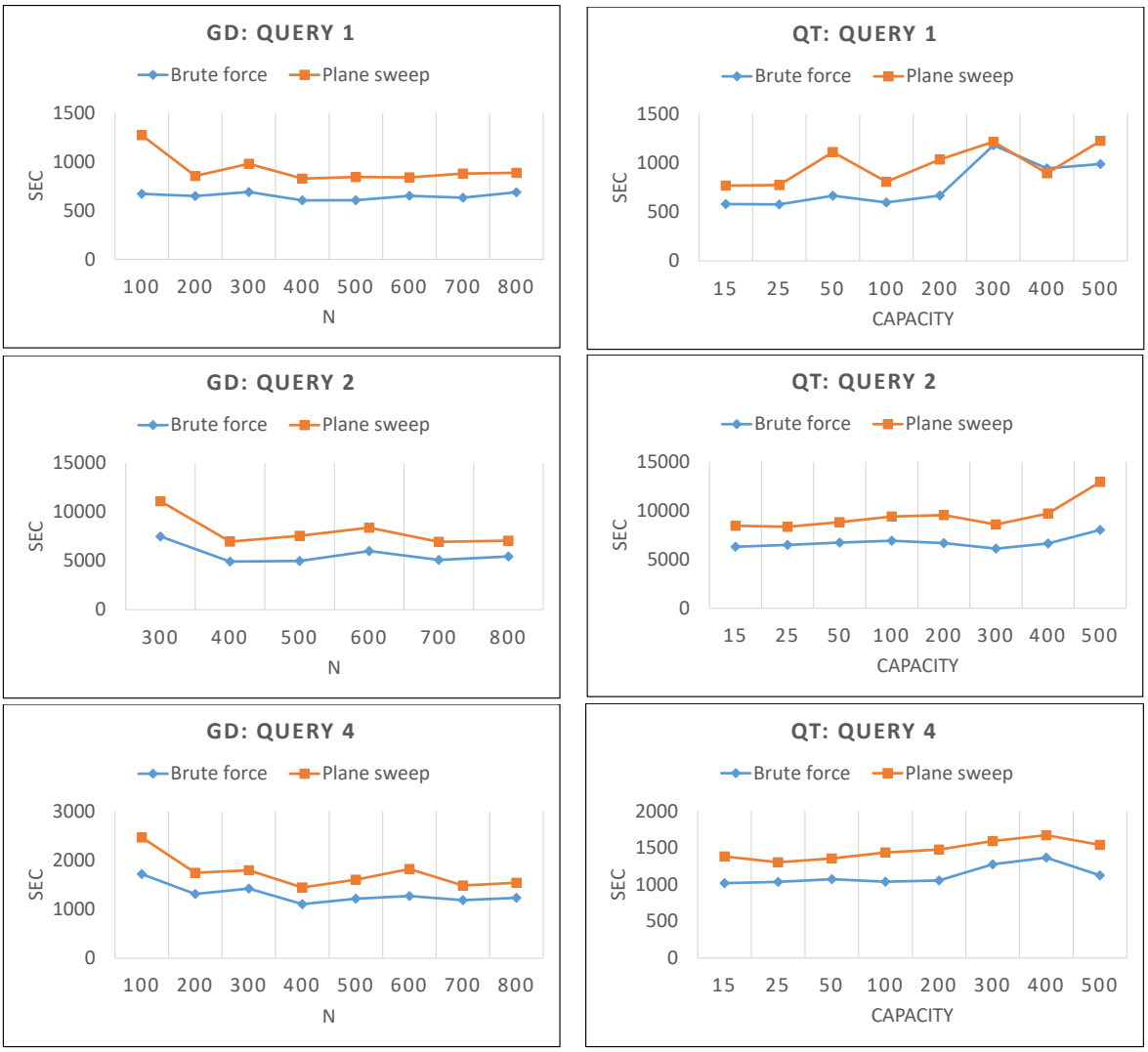

Figure 5.25: Algorithm performance on Query datasets 1, 2 and 4.

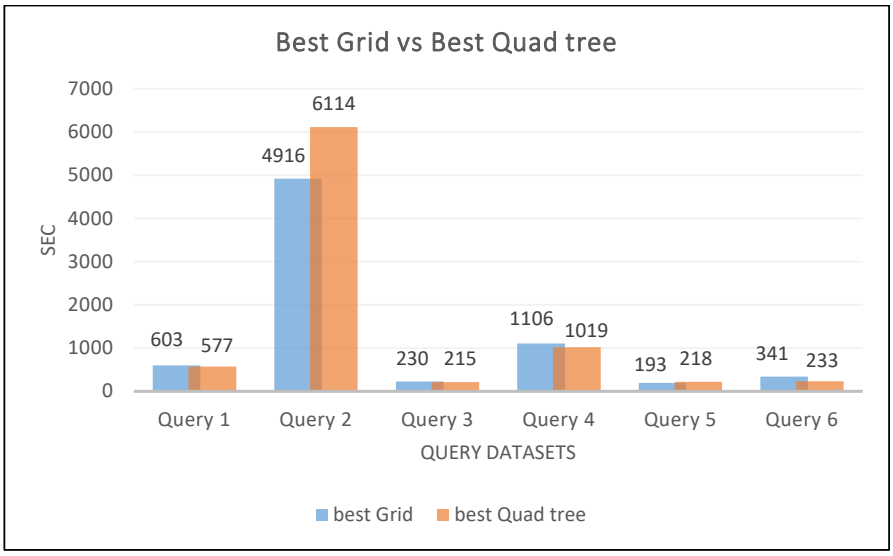

Figure 5.26: Best GD vs best QT.

- Query 2 is the slowest by far, because it corresponds to the biggest number of Training points, which are dense and quite close to the Query MBR centroid, meaning that the pruning heuristics will not work efficiently. Most of the points will be both checked for distances and x-distances and only few will be pruned. Query 4 follows and then comes Query 1. Querys 3, 5 and 6 (not presented here) are obviously much easier to compute.

- Plane-Sweep performance always comes second to the Brute-Force one and they maintain a steady distance between them. The reason behind this is the frequent failure of x-distance check in Plane-Sweep Reducers (meaning that the distance computations are almost equal to the x-distance ones) and the extra cost of sorting of Training points in each cell. Further analysis of the number of calls to the distance computation functions and their respective times (they were not presented, for the sake of space, but will be presented in the next version's experiments) has shown that for almost every x-distance call, an Euclidean distance call followed. The calculation times were almost equal. Because in Phase 2 the distances are relatively small (inside and closely around the Query MBR) the sum of x-distances of a Training point rarely becomes bigger than the K-th best sum of Euclidean distances found so far. This would work in Phase 3, that checks distant cells, but the pruning heuristics in Mappers hardly let any points through.

- The algorithm generally favors quite large numbers of cells. The curves start descending from fewer cells areas (left on the Grid charts and right on the Quadtree ones) and then become almost horizontal as cells number increases. This can be observed both in Grid and Quadtree graphs. The explanation comes from Hadoop's architecture, when there are few, big cells there are large numbers of Query and Training points inside them and the Reducers have to perform a huge number of computations. Distributed computing is better exploited using many small cells, since the Reducers will finish faster using more computing nodes at the same time. But the performance increase stagnates after a point, because more Reducers and computing nodes have to be added to the system.

- Grid and Quadtree perform similarly, with the exception of Query 2, where Grid wins by about 20%. The best performing Grid in this dataset ($N = 400$) has $400 \times 400 =$

$160,000$ cells, while the best performing Quadtree with capacity 15 has 22,594 cells only (Table 5.3). Quadtree data space partitioning is generally more refined than Grid, because it chooses where to split instead of applying blind cuts. It also creates fewer and more equibalanced cells, reducing network congestion and number of Reducers. But Quadtree has a recursive search function for locating points and creating cells, which is inefficient when called often and perhaps costed its victory. In this particular dataset we can also observe Grid's signifficant performance variations in neighboring N's (from 5000 seconds for $N = 500$, up to 6000 seconds for $N = 600$ and then down to 5000 seconds again for $N = 700$).

Table 5.3: Number of cells for each N and capacity

Grid		**Quadtree**	
N	**cells**	**capacity**	**cells**
100	10,000	15	22,954
200	40,000	25	13,351
300	90,000	50	6,922
400	160,000	100	3,472
500	250,000	200	1,762
600	360,000	300	1,111
700	490,000	400	811
800	640,000	500	658

- We did not notice any MBR shape (squares vs horizontal vs vertical rectangles) specific behavior. Because we kept the Query points cardinality constant, the performance differences seem to stem from the number of contained Training points and their relative position and concentration.

- The completion time of each phase as a percentage of the total time is shown in Table 5.4, for a specific dataset, space partitioning and calculation technique. Other combinations exhibit similar performance. We can see that Phase 2 is the most demanding by far, followed by the other two parallel phases, while the local ones cost less than 2% of total time each. Phase 3 is much faster than Phase 2 because its Mapper prunes

almost every distant cell, so the Reducer has little work to do, while being similar to Phase 2 Reducer. This particular conclusion motivated us to introduce a more efficient cell selection algorithm, shown in the next version of the algorithm (see Figure 5.11).

Table 5.4: Phases relative performance, GD+PS, Query 1, $N = 400$

Phase	% of total time
Preliminary	2
Phase 1	5.5
Phase 1.5	0.5
Phase 2	80.5
Phase 2.5	0.5
Phase 3	10.5
Phase 3.5	0.5
Total	100

5.7 Experimental evaluation of the centroid algorithm

These experiments were presented in [53]. The most notable differences compared to the first version are:

1. A new data filtering method for Phase 2 is introduced (centroid circle overlapping, see Section 5.3.6) which boosts the algorithm's performance by almost 90%.

2. A new method ("Fast Sums") that saves calculations from distant sums, when used for pruning purposes (see Section 5.3.3).

3. Minimization of MapReduce phases output size by using *null* instead of *cell_id* in Phases 2 and 3 output and crosschecking each cell's neighbor list from Phase 3 with Phase 2.5 best neighbors, such that only better candidates will proceed to the output.

4. Implementation of the algorithm to SpatialHadoop which also introduces a new two-level partitioning method.

5. Investigation of the algorithm's backstage behavior, using comparative cluster-wide metrics.

6. Extensive experimentation using a variety of real world and synthetic datasets.

In order to evaluate the behaviour of the proposed algorithms, we have used three real world datasets as Training and one real and two clustered synthetic ones as Query. The Training datasets contain the coordinates of parks, buildings and road networks around the world from OpenStreetMap [17] and have 11.5M, 114.7M and 717M points, respectively. The real Query is based on one that contains linear hydrography coordinates inside the USA, but we have trimmed the farthest points and halved it, so it now contains 2.8 million points. The synthetic Querys originally contained 10M and 50M points, scattered in concentrated regions all around the globe. We have cropped them to a rectangle that roughly covers the African continent and they now contain fewer points. This way we can measure the algorithm's behaviour with varying cardinality and distribution of the Query. All datasets' coordinates are normalized in $[0, 1]$ range. We also performed several experiments with the Querys moved to other locations, for performance comparisons. Figures 5.27 and 5.28 show the parks and hydrography datasets on the plane. Figure 5.29 shows the smaller clustered synthetic and the road networks datasets. Figure 5.30 shows the larger clustered synthetic and the buildings datasets. Figure 5.31 shows the default and new hydrography dataset locations over the road networks dataset.

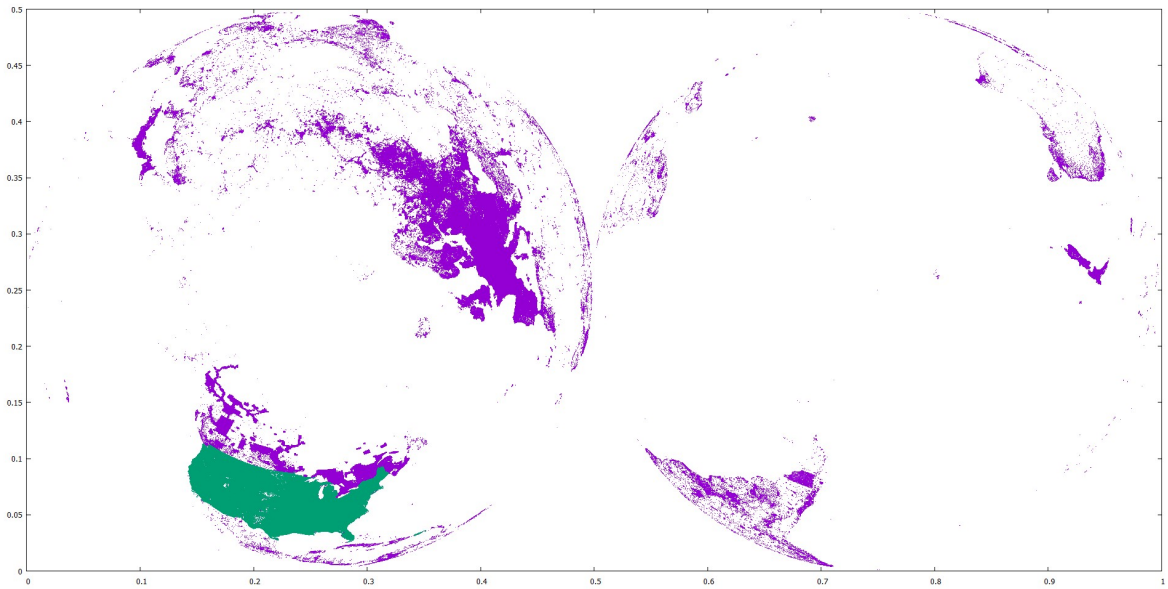

Figure 5.27: Parks (purple) and hydrography (green) datasets in default location.

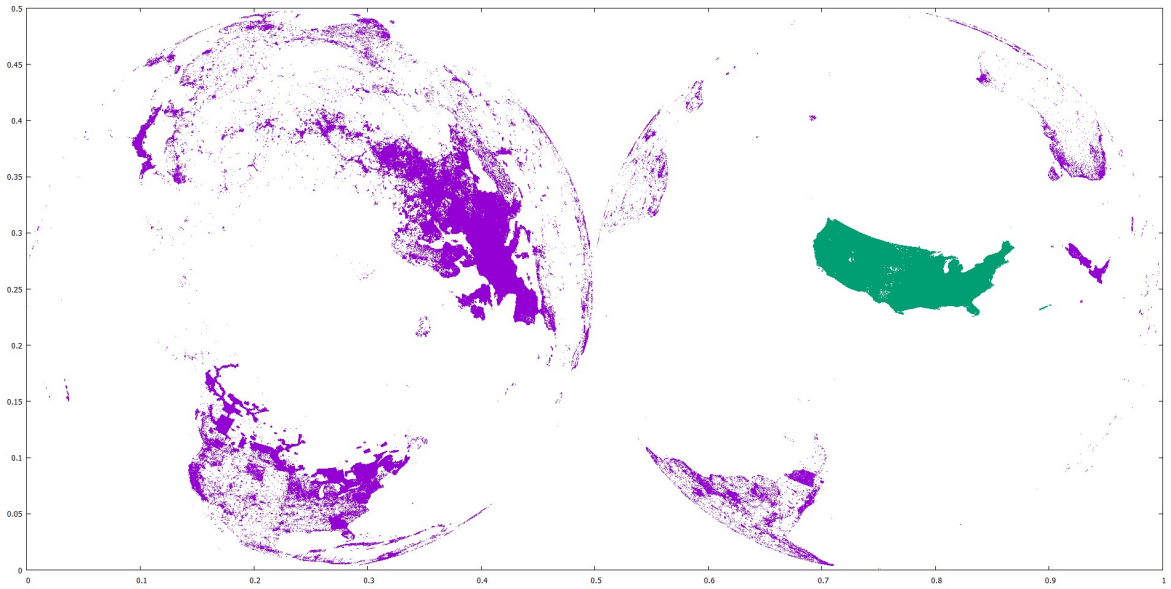

Figure 5.28: Parks (purple) and hydrography (green) datasets in new location.

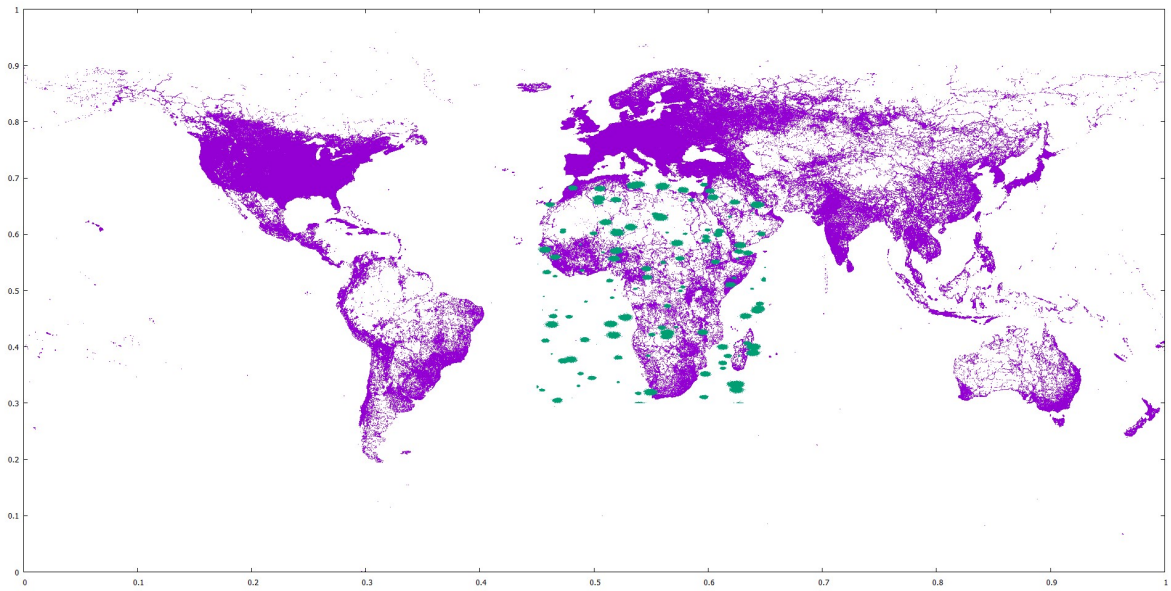

Figure 5.29: Road networks (purple) and synthetic 706K (green) datasets.

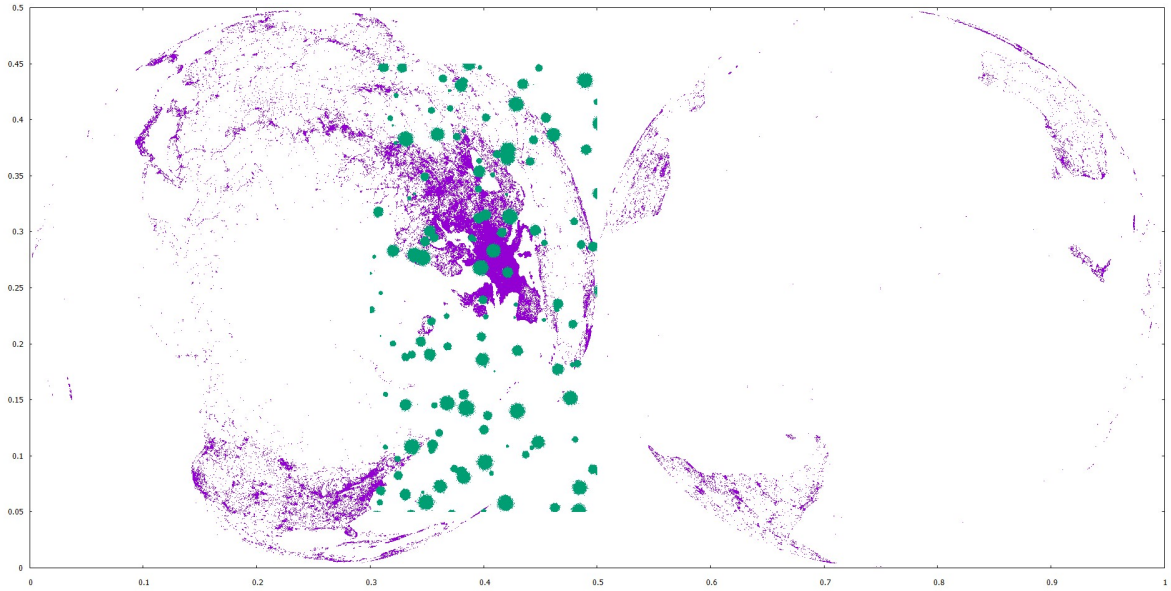

Figure 5.30: Buildings (purple) and synthetic 3.9M (green) datasets.

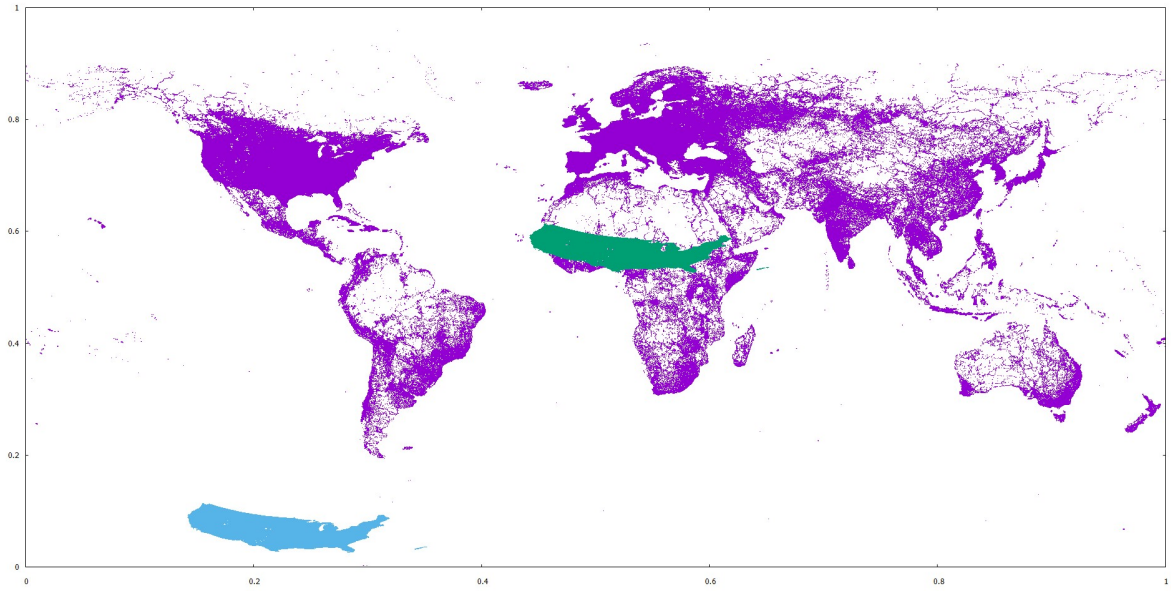

Figure 5.31: Road networks (purple) and hydrography datasets in default (cyan, below South America) and new (green, over Sahara desert) location.

Tables 5.5 and 5.6 show some properties of the datasets used.

We ran our algorithm in many different combinations for each Query and Training couple: Hadoop and SpatialHadoop, Grid and Quadtree partitioning, with Brute-Force or Plane-Sweep Reducers and MBR overlapping vs centroid circle overlapping. We present results of the average total time of five measurements in seconds, as a function of the space partitioning parameter, which is N for Grid ($N \times N$ cells) and capacity for Quadtree (maximum capac-

Table 5.5: Training datasets

Dataset	Num. of pts	MBR (x_{min}, x_{max}) (y_{min}, y_{max})	disk size
parks	11.5M	(0.0013, 0.9969) (0.0009, 0.4973)	373 MB
buildings	114.7M	(0.0023, 0.9969) (0.0086, 0.4916)	3.7 GB
roads	717M	(0.0000, 1.0000) (0.0002, 0.9584)	30 GB

Table 5.6: Query datasets

Dataset	Num. of pts	MBR (x_{min}, x_{max}) (y_{min}, y_{max})	centroid x, y	disk size
hydro (def. loc.)	2.8M	(0.1423, 0.3518) (0.0256, 0.1137)	0.2342, 0.0555	91.4 MB
hydro (new. loc.)	2.8M	(0.6923, 0.9018) (0.2256, 0.3137)	0.7842, 0.2555	91.4 MB
small synth. (def. loc.)	706K	(0.4500, 0.6500) (0.3000, 0.6957)	0.5518, 0.5184	29.5 MB
large synth. (def. loc.)	3.9M	(0.4500, 0.6500) (0.3000, 0.7000)	0.5417, 0.4942	165.7 MB
small synth (new loc.)	706K	(0.3000, 0.5000) (0.0500, 0.4457)	0.4018, 0.2684	29.5 MB
large synth. (new loc.)	3.9M	(0.3000, 0.5000) (0.0500, 0.4500)	0.3917, 0.2442	165.7 MB

ity of Training points in the sample). In Grid it is obvious that the number of cells increases (quadratically) with N, while in Quadtree it decreases with capacity (bigger capacities mean more points per cell, thus fewer and bigger cells). For SpatialHadoop experiments, Grid or Quadtree refers only to top level partitioning, since the bottom level is always Grid. Unless

stated otherwise, all experiments regard the discovery of $K = 10$ neighbors.

We will also present some comparative metrics for selected experiments, which show what what happens internally (under the hood) during the operation of the algorithm, like how many points or cells were pruned or how many times a certain function was called during the execution. These metrics will help us investigate and explain the behavior of the algorithm in cases where it may look incomprehensible.

Finally, we will show what happens when we turn off the cell-pruning heuristics in Mapper 3.1, so that Reducer 3 will process *all* the cells and Training points left over from Phase 2.

A synopsis of the abbreviations we used in presenting our algorithm follows:

- *GD*, or *Grid*: Grid space partitioning used by Mappers,
- *BF*, or *Brute-Force*: Brute-Force processing in Reducers,
- *PS*, or *Plane-Sweep*: Plane-Sweep processing in Reducers,
- *QT*, or *Quadtree*: Quadtree space partitioning used by Mappers.
- *MBR*: MBR overlapping refinement technique for Phase 1.5.
- *Centroid*: Centroid circle overlapping refinement technique for Phase 1.5.

These abbreviations will also be used in combinations expressing the combined application of techniques, like *QT+PS* (application of Quadtree Mapper and Plane-Sweep Reducer) or *QT-GD* (SpatialHadoop's Quadtree top layer combined with Grid bottom layer).

First, we will study how all presented methods and partitioning techniques perform against each other, using the smallest Training, parks, and the hydrography Query in default and new locations. Afterwards, we will take the best method and partitioning combination and test it on a variety of Query and Training datasets.

5.7.1 Hydrography and Parks datasets

We will first present experiments for the hydrography dataset as Query and the parks dataset as Training, for both Hadoop and SpatialHadoop.

Hadoop experiments

Preliminary Phase times are:

- Calculation of the Query MBR, its centroid and sum of distances from centroid to all Query points: about 12 seconds.
- Query points sorting by ascending x-distance into a list: about 35 seconds (43 msec for sorting and the rest is HDFS I/O for a 70MB file).
- Quadtree sampling and creation: from 7 to 20 seconds (the lower the capacity, the higher the creation time).

Total Preliminary Phase time is about 60 seconds and will not be included in the following graphs, because it was only run once.

Figures 5.32 and 5.33 depict the comparative performance of BF vs PS Reducers and MBR vs Centroid circle overlapping refinement technique, using GD and QT Mappers, for the default location of the Query (Figure 5.27).

Five different Grid N's (400 to 1200) and five different Quadtree capacities (2 to 25) were tested, creating vastly divergent cell numbers and points per cell distributions. Table 5.7 gives the number of cells for each N and capacity. In our experiments we used a 1% sample from the Training to create the Quadtree. A maximum capacity e.g. 5 means that there must be five points per cell at most, *in the sample*. So, in the complete dataset, this translates to maximum five hundred points per cell.

Table 5.7: Number of cells for each N and capacity

Grid		**Quadtree**	
N	**cells**	**capacity**	**cells**
400	160,000	2	149,812
600	360,000	3	100,774
800	640,000	5	60,991
1000	1,000,000	10	30,694
1200	1,440,000	25	12,259

We can draw some useful conclusions from these charts.

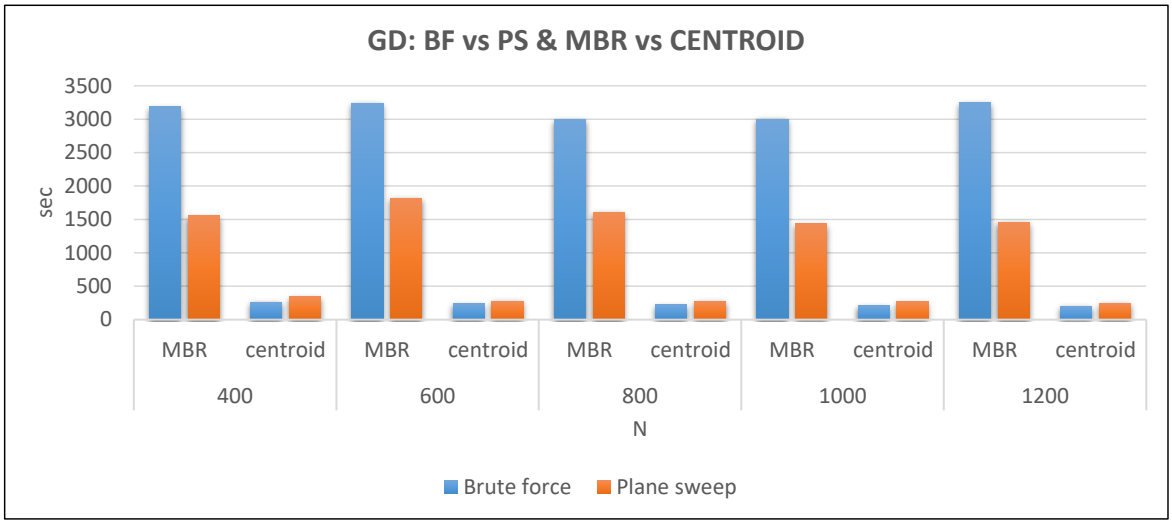

Figure 5.32: Hadoop, default location hydro & parks, GD, BF vs PS, MBR vs Centroid.

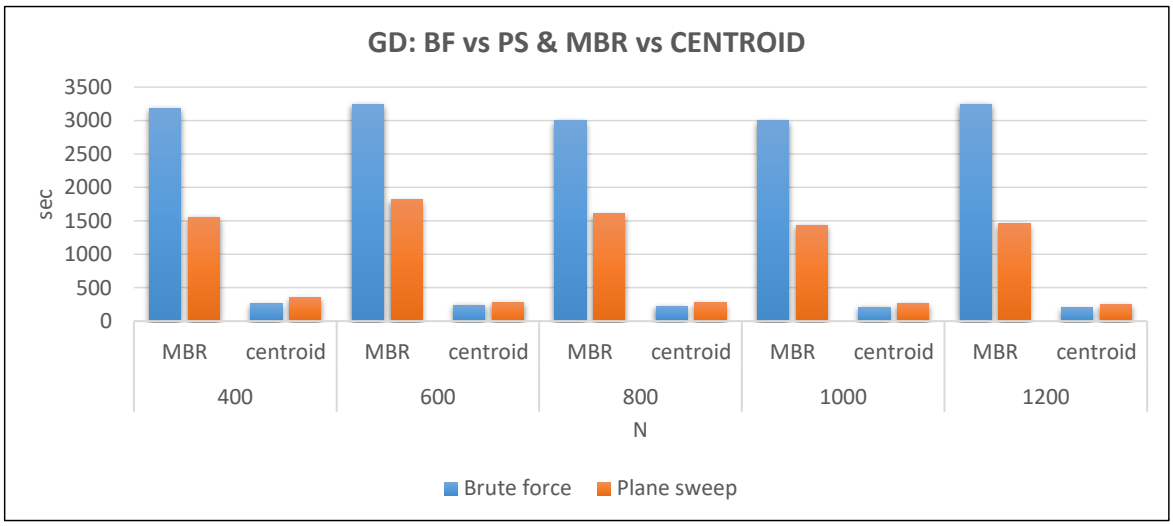

Figure 5.33: Hadoop, default location hydro & parks, QT, BF vs PS, MBR vs Centroid.

- Centroid circle overlapping refinement method in Phase 1.5 is the clear performance winner, finishing almost 90% faster than MBR overlapping method. The explanation is the much smaller number of cells it feeds to Phase 2 for processing, more than 90% less. As noted in Section 5.3.6, Phase 2 was the hardest one to complete, when using MBR overlapping. Heuristic 4 check failed most of the time, due to small distances, so full Query distance sums were calculated for most Training points. Centroid circle method did not make that heuristic succeed, but it greatly reduced the number of cells (and Training points) processed.

- Regarding the MBR overlapping method, Plane-Sweep is the clear winner for both

Grid and Quadtree partitioning. The greater number of pruned Training points due to its x-distance check is the reason, as will be shown later when we present what happens under the hood. Brute-Force's only pruning method, heuristic 4, was not very successful.

- The algorithm generally favors large numbers of cells. Completion times are descending from areas of fewer cells (left on the Grid charts and right on the Quadtree ones). This can be observed both in Grid and Quadtree graphs. The explanation comes from Hadoop's architecture, when there are few, big cells there are large numbers of Query and Training points inside them and the Reducers have to perform a huge number of computations. Distributed computing is better exploited using many small cells, since the Reducers will finish faster using more computing nodes at the same time. But the performance increase stagnates after a point, because more Reducers and computing nodes have to be added to the system.

- Grid seems to perform a little better than Quadtree in most cases. The best performing Grid in this dataset ($N = 1000$) has 1,000,000 cells, while the best performing Quadtree with capacity 2 has 149,812 cells only (Table 5.7). Their Brute-Force performance is almost equal (3,002 seconds for Grid and 3,055 seconds for Quadtree), but Plane-Sweep time is 1,436 seconds for Grid and 1,858 seconds for Quadtree, which is about 30% worse. Point distribution is far more balanced inside Quadtree cells and all Reducers have almost the same number of points to work with, but still the percentage of pruned points is greater in Grid. This cannot be easily analyzed, because it involves thousands of points and their relative distances that result in heuristic 4's and Plane-Sweep's x-distance check's success or failure. Our hypothesis is that it is heavily dataset dependent, involving both datasets. It is also a result of Quadtree's more complicated procedure of top to bottom pointer based query for point and cell location calculation.

- Centroid circle method's performance is fluctuating between 200 and 300 seconds, in most cases. Having in mind that Hadoop's MapReduce phases need about 30 seconds each just to start and finish (without performing any calculations), it is obvious that there are about 100 - 200 seconds left where the algorithm actually works. So, there is not much to compare to between different partitioning and computation tech-

niques, since performance variation is almost flat. Plane-Sweep seems a little slower than Brute-Force because the x-distance check is becoming obsolete in such small point populations.

Figure 5.34 shows per phase comparison of the two refining methods as percentages of total time, in the best performing Quadtree, using Plane-Sweep. It is obvious that Phase 2 is the dominant one in MBR overlapping method, while all distributed phases are more balanced when using centroid overlapping. Local phases are definitely overshadowed, especially in the MBR case. Their combined time is less than 1% and about 4% of total, in MBR and centroid methods, respectively.

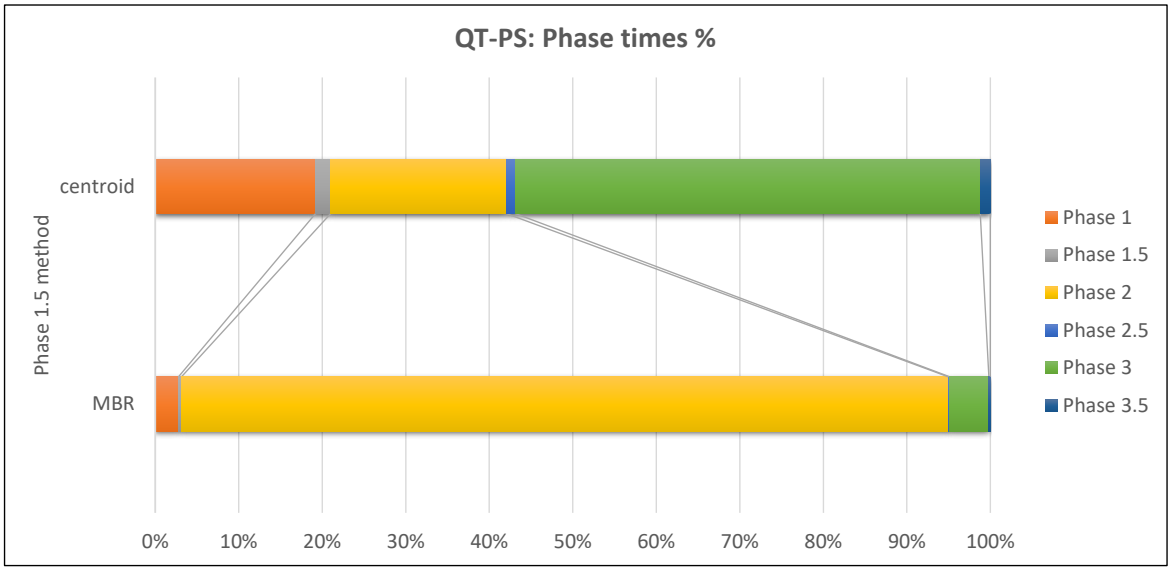

Figure 5.34: QT-PS, capacity = 2, phase times %.

Figures 5.35 and 5.36 depict the comparative performance of BF vs PS Reducers and MBR vs Centroid circle overlapping refinement technique, using GD and QT Mappers, for the new location of the Query (Figure 5.28).

The effect of low density of Training points underneath the Query is more than obvious. The MBR and Centroid refinement methods are directly competing with each other. The reason is the very small number of cells that Phase 1.5 feeds to Phase 2, in both methods. Plane-Sweep is once again coming second, as explained before, except for some cases in Quadtree partitioning, where the number of pruned points was unpredictably bigger than Brute-Force's. Grid partitioning wins this round too, by a close head.

Now we will present some metrics to explain what's going on under the hood, where in some cases the performance was better or worse than expected. Table 5.8 shows performance

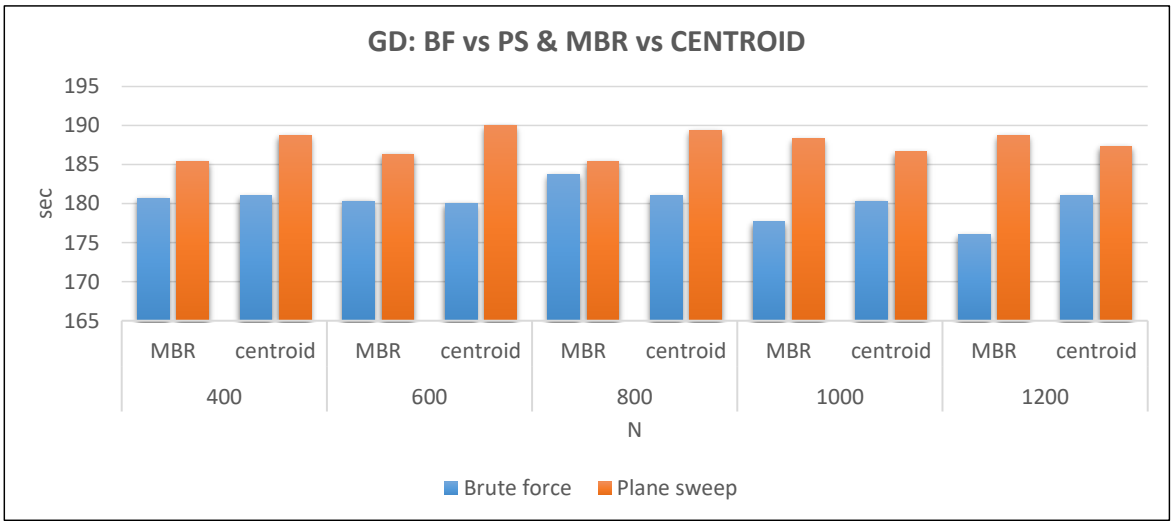

Figure 5.35: Hadoop, new loc. hydro & parks, GD, BF vs PS, MBR vs Centroid.

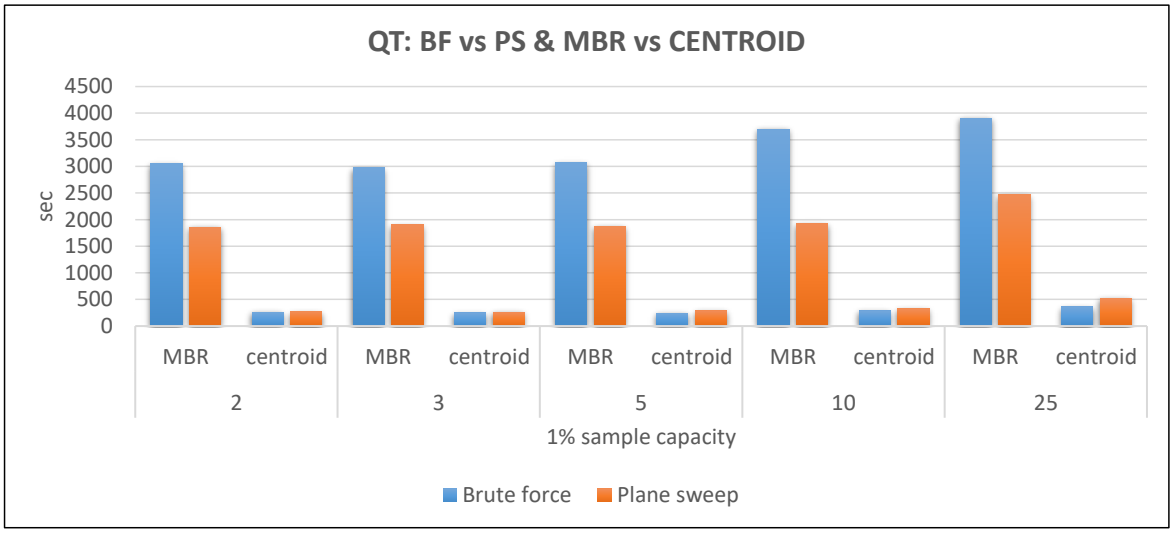

Figure 5.36: Hadoop, new loc. hydro & parks, QT, BF vs PS, MBR vs Centroid.

metrics for the best performing Grid on the default location Query.

The first column on the left of Table 5.8 contains the performance metrics, the second column contains the metrics for Phase 2, the third column contains the metrics for Phase 3 when the heuristics are turned on (default mode, as shown in previous performance charts) and the last column contains the metrics for Phase 3 when the heuristics are turned off (the cell-pruning heuristics 1, 2, 3 in Mapper 3_1). The phases columns have two sub-columns each, for BF and PS.

The performance metrics are:

- *Total time*: the algorithm's total time in seconds. It is entered under Phase 3 entries. It

Table 5.8: Best GD performance metrics, default location hydro & parks

Grid N = 1000						
	Phase 2		**Phase 3 (heuristics ON)**		**Phase 3 (heuristics OFF)**	
	BF	**PS**	**BF**	**PS**	**BF**	**PS**
MBR method						
Total time (sec)			3,002	1,436	3,755	1,753
cells processed	7,620		0		32,876	
tpoints processed	1,629,158		0		9,874,877	
SumDx calls	*NA*	368,357	*NA*		*NA*	59,826
SumDist calls	*NA*	364,372	*NA*		*NA*	59,249
SumDx successes	*NA*	4,289	*NA*		*NA*	22,638
% of tpoints pruned	16.33%	77.54%			98.86%	99.38%
Centroid method						
Total time (sec)			211	265	2,938	1,578
cells processed	5		117		40,491	
tpoints processed	1,150		3,628		11,502,885	
SumDx calls	*NA*	1,125	*NA*	3,628	*NA*	399,290
SumDist calls	*NA*	1,150	*NA*	3,628	*NA*	394,105
SumDx successes	*NA*	0	*NA*	0	*NA*	27,250
% of tpoints pruned	0.0%	0.0%	0.0%	0.0%	87.22%	96.54%

is *not* Phase 3's time, it's *total* time.

- *cells processed*: the number of cells processed by each phase. They are the same for BF and PS inside each phase, since their number depends on partitioning only.
- *tpoints processed*: The number of Training points contained inside the above cells.
- *SumDx calls*: how many times the function calculating the sum of x-distances was called (PS only). *NA* = Not Available, for BF.
- *SumDist calls*: how many times the function calculating the sum of Euclidean distances was called (PS only). Actually, this is also called in BF, but we are interested in PS's

case to compare it with SumDx calls. In BF it's the only distance function available, anyway.

- *SumDx successes*: how many SumDx function calls (PS only) were successful in pruning some points, i.e. when the sum of x-distances was bigger than the maximum distance in the neighbors heap. The number of successes itself represents pruned points which are added to the total. This metric shows the efficiency or inefficiency of PS.

- *% of tpoints pruned*: The percentage of Training points pruned in the Reducers by heuristic 4 or PS's SumDx successes.

We will now point out some interesting conclusions drawn from Table 5.8.

- The first thing someone could notice is the incredible difference in Phase 2's processed cells number between the MBR and the Centroid methods: 7,620 vs 5. This alone is explaining the huge performance gain of the second method, as we have analyzed in Section 5.3.6. A second look tells us that Centroid's method Phase 3 (with heuristics on) is processing some cells (117), while MBR is not. This shows that the list of Phase 2.5 was the final one in MBR method and not the final one in Centroid, that's why the pruning heuristics let some cells pass through and got their points checked. The "easy trip" through Phase 2 had its toll, which was some sub-optimal candidate neighbors. But this error was successfully and painlessly corrected in Phase 3. We have also analyzed that in Section 5.3.9. It must be noted that this does not happen every time, only in certain Grid or Quadtree configurations.

- Looking at Phase 2 metrics for the MBR method, we can see that PS prunes 77.54% of the Training points that it gets, compared to BF's only 16.33%. BF prunes points by heuristic 4 only, while PS adds its x-distance check successes to it (and compensates the overhead of sorting Training points, once in each cell), which was 4,289 times successful inside the Query MBR. Judging only from Phase 2's pruned points percentages, one would expect that PS would be about 5 times faster than BF. But we should also account the x-distance check failures (not shown), which resulted in calling the SumDist function almost as many times as the SumDx function, which precedes it in the algorithm.

- Phase 3 with heuristics on has pruned every cell that Phase 2 left behind, so there is not much to discuss here (MBR method). When we turn the cell-pruning heuristics to off, thousands of cells containing millions of Training points are swarming into Phase 3. Because of their long distances from the centroid, heuristic 4 in the Reducers (not deactivated) and PS's x-distance check prune almost every point checked. However, BF's total time is now 3,755 seconds (25% raise), while PS's time is 1,753 seconds (22% raise). This means that PS' x-distance check pruning (22,638 times successful) is "cheaper" than BF's heuristic 4 only, because it prunes points at large numbers at once, not just one at a time.

- Looking at the Centroid method's metrics, we can see that the pruned points percentage is 0 for both Phases 2 and 3 (with heuristics on), and for both BF and PS. This explains why PS finished second. Every calculation of SumDx resulted in an unsuccessful x-distance check, therefore a subsequent calculation of SumDist followed, for every point. This means that PS called two distance functions for every point, in contrast to BF which called only one. Things change, however, when Phase 3's heuristics are turned off. BF's total time is launched to 2,938 seconds (1,292% increase from 211), while PS's time reaches 1,578 seconds (495% increase from 265). This difference is easily explained by looking at their pruned points percentages and mainly the number of PS's x-distance check successes ("cheap" pruning) in Phase 3.

We have gathered similar performance metrics for both Query datasets (default and new location hydro), for both partitioning methods (GD and QT) and also some other metrics, like the absolute values and percentages of points and cells pruned by each heuristic separately and the failures of SumDx. But we decided to show only one representative table with clean and simple figures, since the other results are very much alike the ones presented.

SpatialHadoop experiments

Preliminary Phase times are:

- Calculation of the Query MBR, its centroid and sum of distances from centroid to all Query points: about 15 seconds.

- Query points sorting by ascending x-distance into a list: about 50 seconds.

- Grid or Quadtree sampling and creation: about 12 minutes with default parameters and 4 minutes with an optimization on the number of splits.

Like in Hadoop's experiments, these procedures were only run once and will not be included in the following performance graphs.

Figures 5.37 and 5.38 depict the comparative performance of BF vs PS Reducers and MBR vs Centroid circle overlapping refinement technique, using GD-GD and QT-GD Mappers, for the default location of the Query (Figure 5.27).

Five different Grid N's (400 to 1200) for two different SpatialHadoop partitioning techniques (Grid and Quadtree) were tested, creating similar cell numbers and points per cell distributions to Hadoop. However, SpatialHadoop creates a different number of partitions for each partitioning technique. For instance, giving a default HDFS block size of 128 MBytes, we have 2 partitions for Grid and 16 for Quadtree. Each partition is divided by a Grid with the same cell size as for Hadoop, as depicted in Section 5.4.3.

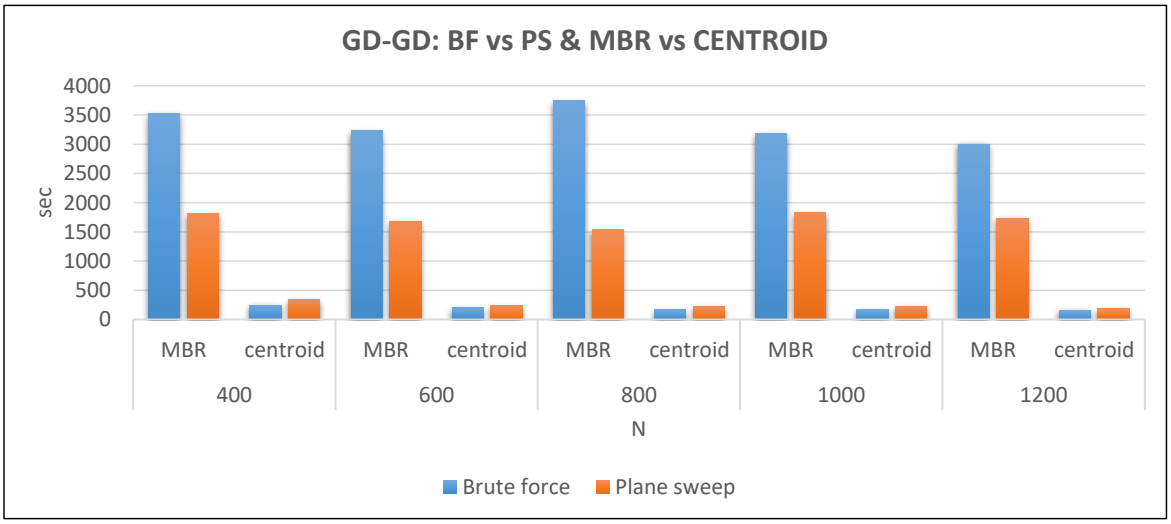

Figure 5.37: SpatialHadoop, default location hydro & parks, GD-GD, BF vs PS, MBR vs Centroid.

Next we show several conclusions that can be obtained from these charts.

- Similar to Hadoop, the Centroid circle overlapping refinement method in Phase 1.5 is also the clear performance winner, showing the same relative performance of almost 90% gain over the MBR overlapping method. The reduction in the number of cells processed in Phase 2 explains this behavior since the number of distance sums increases considerably for both partitioning methods when MBR overlapping is used.

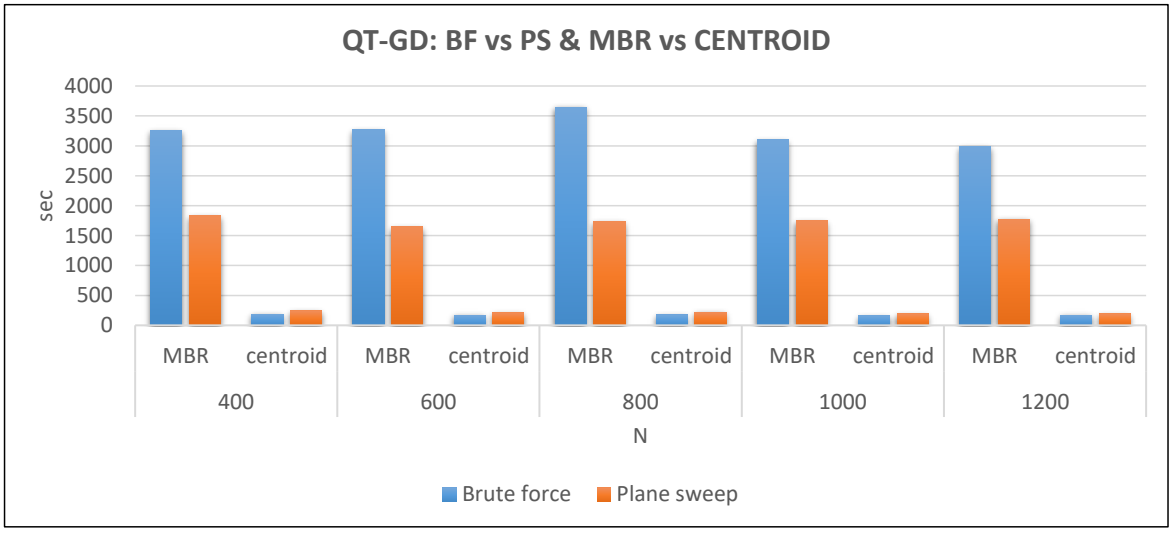

Figure 5.38: SpatialHadoop, default location hydro & parks, QT-GD, BF vs PS, MBR vs Centroid.

- MBR overlapping method also shows similar behavior to Hadoop in which Plane-Sweep gets better performance for both Grid and Quadtree partitioning. The main reason is also the use of the x-distance check that prunes a higher number of Training points compared to the Brute-Force approach.

- As SpatialHadoop is a distributed system based on Hadoop, its architecture is also benefited by a large number of cells, although each partitioning technique shows different behaviors. On the one hand, the Centroid overlapping method shows better results for Quadtree partitioning when the value of N is small and as this value increases the results are equalized ($N = 800$). This is due to the number of partitions created by each technique, that is, 2 for Grid and 16 for Quadtree. The filtering function is more effective for Quadtree because space is divided into more partitions, obtaining a smaller number of cells and therefore Training points. By increasing the number of cells, the effectiveness of both methods is matched because the cell size is smaller and the heuristics select a similar number of cells / Training points. On the other hand, MBR overlapping shows similar values for Grid and Quadtree due to the high number of cells selected in Phase 2 that make the pruning methods that SpatialHadoop partitioning provides ineffective.

- The Centroid circle method's performance is even under 200 seconds for SpatialHadoop in most cases. This further accentuates the effectiveness of Brute-Force against

Plane-Sweep, that is, it is preferable to use a greedy approach when the number of Training points/cells is very low than having to order each cell to use heuristics that may not be successful.

- If we compare the results of SpatialHadoop with those of Hadoop, the improvements in performance can be seen as a product of using the features provided by the former. The use of the two-level indexing technique and heuristic-based filters reduce both the physical reading of Training points and the search space. This can be seen especially in the combination of Quadtree and Grid against plain Quadtree and in all centroid circle based configurations. However, when we compare Grid partitioning with plain Grid using the MBR method, there is no clear winner and sometimes SpatialHadoop shows lower performance. This is because of the low number of partitions created by SpatialHadoop using the Grid partitioning technique for this Training, for instance, only 2. Therefore, the performance depends a lot on the morphology of the Query and the different values of N, and then the two-level index can cause an overhead.

In the same way as for Hadoop, Figure 5.39 shows per phase comparison of the two refining methods as percentages of total time, in the best performing Quadtree, using Plane-Sweep. Note a very similar behavior, in which for the centroid method the distribution of the times of each phase is very balanced, while Phase 2 take almost all the execution time for the MBR overlapping method. Local phases present again an insignificant time with respect to the distributed ones, although the times are lower (both less than 0.5%) than those of Hadoop thanks to the use of the SpatialHadoop indices.

Figures 5.40 and 5.41 depict the comparative performance of BF vs PS Reducers and MBR vs Centroid circle overlapping refinement technique, using GD Mappers, for the new location of the Query (Figure 5.28) for both Grid and Quadtree partitioning techniques in SpatialHadoop.

When the number of points/cells in the Training, that is overlapping with the Query, is very low, we have similar behavior to Hadoop. Therefore, the number of cells evaluated in Phase 2 is very low and alike between the MBR and Centroid refinement methods and the performance does not show differences. However, the relative performance against Hadoop has an average improvement of 30%, that is, we can see how the two-level indexing technique and heuristic-based filters avoid processing points/cells more effectively. Quadtree partitioning wins by a small gap thanks to the higher number of partitions.

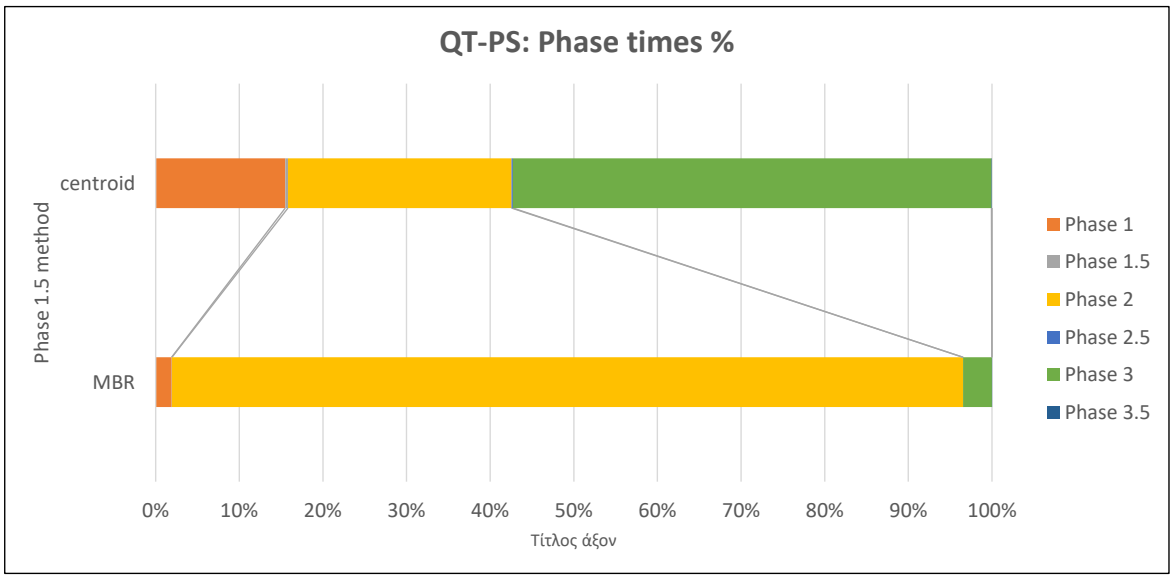

Figure 5.39: QT-PS, capacity = 2, phase times %.

Now we will show some metrics to elaborate more on the behaviour of the algorithm in SpatialHadoop. Table 5.9 shows performance metrics for the best performing Grid on the default location Query for both Grid and Quadtree partitioning techniques having a value of N of 800 and 600 respectively.

The first column on the left of Table 5.9 contains the performance metrics, the second and third columns contains the metrics for Phase 2 and Phase 3 for Grid partitioning, and the fourth and fifth columns contains the metrics for Phase 2 and Phase 3 for Quadtree partitioning. All heuristics are turned on when available. The phases columns have two sub-columns each, for

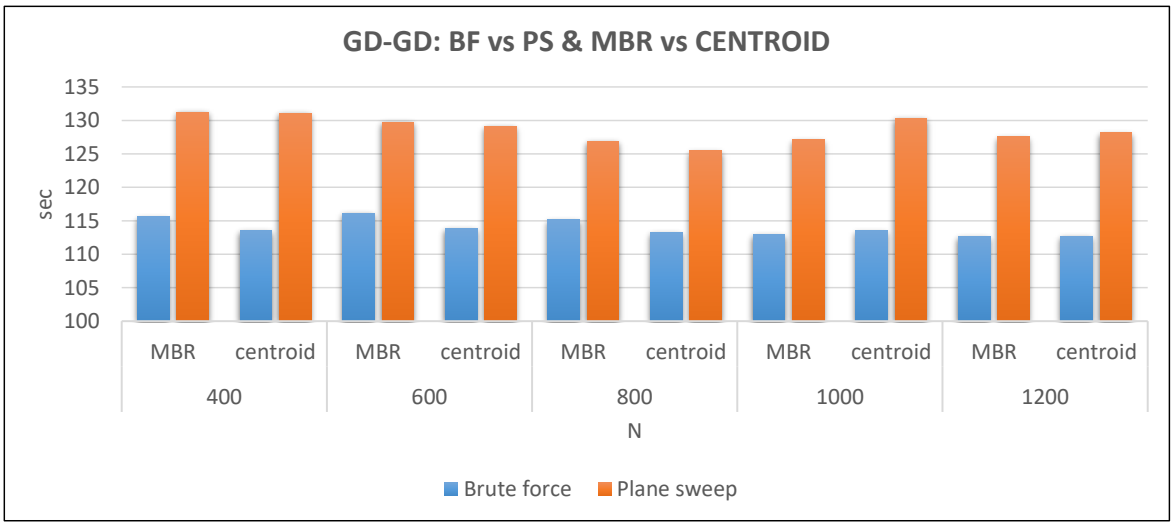

Figure 5.40: SpatialHadoop, new loc. hydro & parks, GD, BF vs PS, MBR vs Centroid.

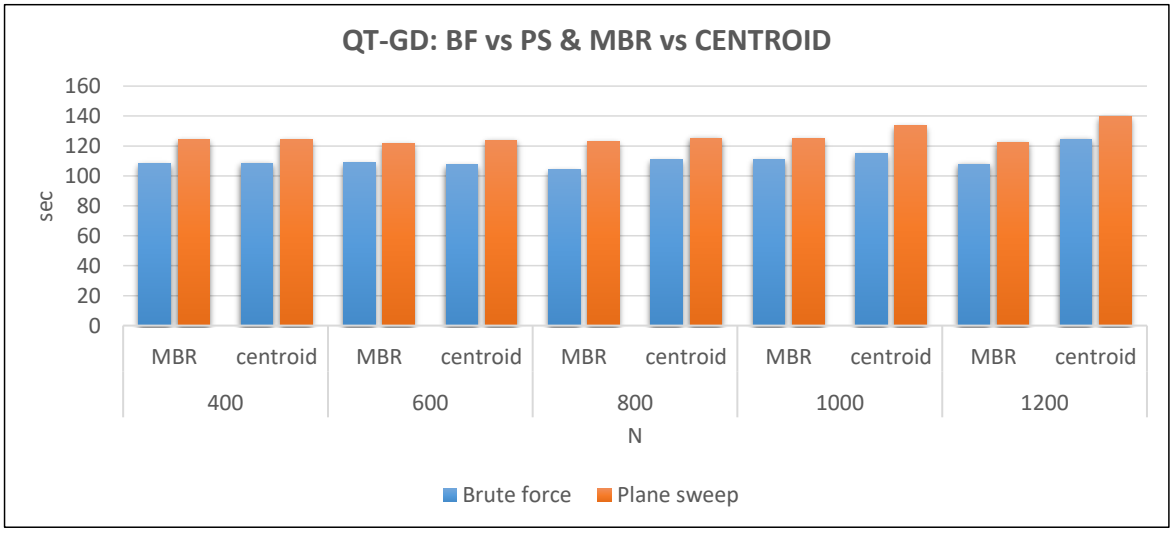

Figure 5.41: SpatialHadoop, new loc. hydro & parks, QT, BF vs PS, MBR vs Centroid.

BF and PS. The performance metrics are the same as for Hadoop in Table 5.8.

We can compare both partitioning techniques and draw some useful conclusions from Table 5.9.

- The first thing to notice is that the best performing Grid is obtained for Quadtree partitioning with a lower value of N than for Grid partitioning (800 vs 600). This is due to the number of partitions created by SpatialHadoop for each partitioning technique, that is, 2 partitions for Grid and 16 for Quadtree. Despite having more and larger cells, Quadtree gets better results because it can avoid reading more Training points/cells that are not part of the result. However, Grid partitions are selected in almost all cases cause of their small number.

- Similar to Hadoop, the Centroid's method Phase 3 is processing some cells, while MBR is not. This shows that the list of Phase 2.5 does not need more refinement in the MBR method, and again this does not happen every time, only in certain configurations.

- Looking at the Phase 2 metrics for the MBR method, we see that the results are consistent with those of Hadoop and that PS prunes around 78% of the Training points that it gets, compared to BF's only 16% for both partitioning techniques.

- On one hand, observing the number of cells processed in Phase 2 for the MBR method, BF gets better performance (3273 sec) when using Quadtree with a larger number of cells (6121) even if they are larger ($N = 600$) and approximately 10,000 more

Table 5.9: Best GD-GD and QT-GD performance metrics, default location hydro & parks

	Grid-Grid N = 800				**Quadtree-Grid N = 600**			
	Phase 2		**Phase 3**		**Phase 2**		**Phase 3**	
	BF	**PS**	**BF**	**PS**	**BF**	**PS**	**BF**	**PS**
				MBR method				
Total time (sec)			3,743	1,536			3,273	1,653
cells processed	5,567		0		6,121		0	
tpoints processed	1,626,040		0		1,636,229		0	
SumDx calls	*NA*	355,311	*NA*		*NA*	361,717	*NA*	
SumDist calls	*NA*	352,288	*NA*		*NA*	358,357	*NA*	
SumDx successes	*NA*	3,187	*NA*		*NA*	3,525	*NA*	
% of tpoints pruned	16.36%	78.26%			16.42%	78.01%		
				Centroid method				
Total time (sec)			169	220			170	211
cells processed	2		117		2		143	
tpoints processed	833		4,579		417		6,618	
SumDx calls	*NA*	823	*NA*	4,579	*NA*	405	*NA*	6,618
SumDist calls	*NA*	833	*NA*	4,579	*NA*	417	*NA*	6,618
SumDx successes	*NA*	0	*NA*	0	*NA*	0	*NA*	0
% of tpoints pruned	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

points are processed. This is because it favors itself more to parallelization than PS. On the other hand, PS shows better behavior when using Grid with a smaller number of cells (5567), since successful x-distance checks compensate the overhead of sorting of Training Points (once per cell) and of the calculation of heuristics that do not succeed.

- As for the Centroid method's metrics, the number of pruned points inside the cells is again 0 for both Phases 2 and 3, both BF and PS and both Grid and Quadtree partitioning. Therefore, this shows the overhead, caused by the sorting and the calling of SumDx and SumDist functions for every point without success, which makes BF again the winner. Comparing the partitioning techniques, Quadtree is faster than Grid despite processing more cells and points in Phase 3. This is because Quadtree expends

less time in Phase 3 due to the fact that it processes approximately half of points than Grid. However, this has raised more cells and points to be processed in Phase 3 than for Grid but in a smaller time. Therefore, this shows that the quality of Phase 2 candidates is not as important as their existence to accelerate the subsequent Phase 3.

- SpatialHadoop is again the winner against Hadoop concerning execution time and it is easy to see the proper functioning of the two-level indexing technique that allows processing fewer points in both Phase 2 and Phase 3.

Fast Sums experiments

We will now show the efficiency of the "Fast Sums" technique. The performance gain is far more obvious when using MBR overlapping than centroid, so this is what we will present. We used the default location Query. Figures 5.42 and 5.43 show how the best performers from Hadoop and SpatialHadoop behave when switching the "Fast Sums" to "on".

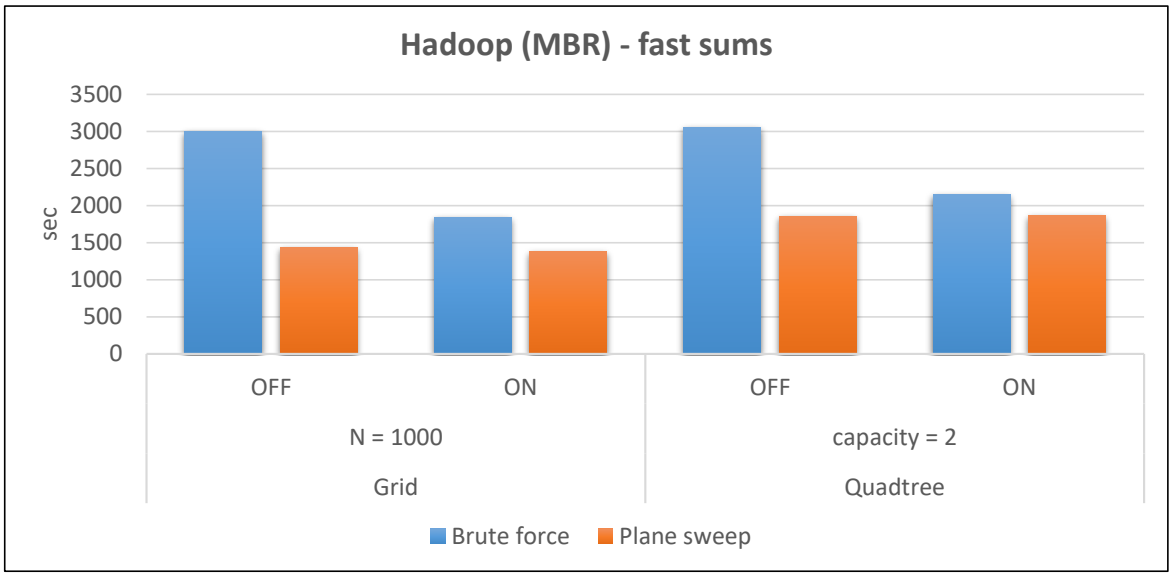

Figure 5.42: Hadoop, "Fast Sums", hydro & parks, GD vs QT, BF vs PS, MBR.

In Hadoop, the gain for BF is amazing: almost 40% improvement in total time for GD and 30% for QT. Unfortunately, this achievement is not repeated in PS, the performance is almost unchanged. This can be explained from Table 5.8. Most of the work is done inside Phase 2, where BF prunes only 16.33% of the Training points, so the "Fast Sums" are used by the rest 83.67% of the non-pruned points (about 1.3M), where full distance sums had to be calculated for each. PS prunes 77.54% of the Training points, so the "Fast Sums" are mostly

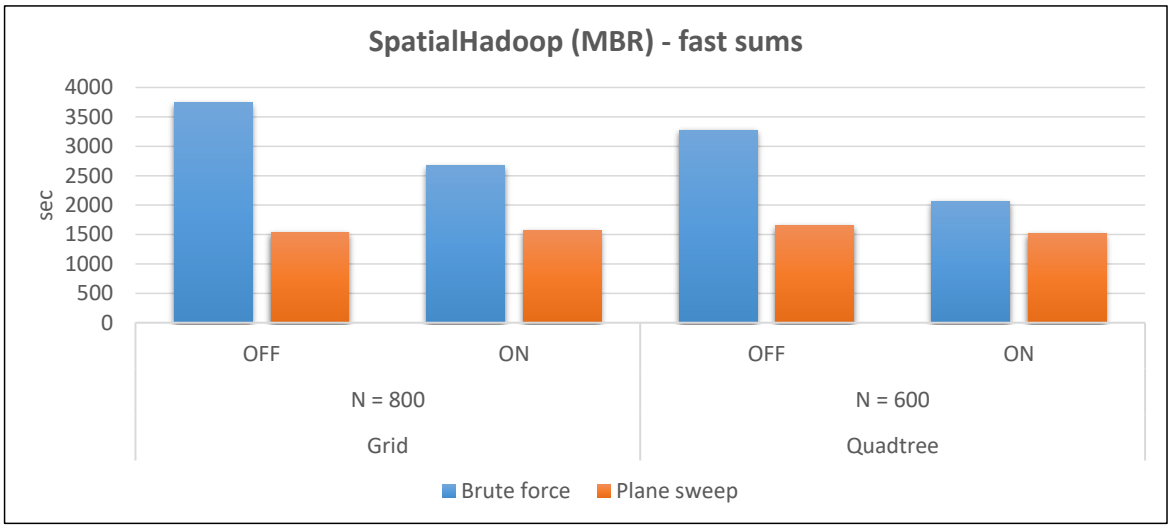

Figure 5.43: SpatialHadoop, "Fast Sums", hydro & parks, GD vs QT, BF vs PS, MBR.

used only by the rest 22.46% of the non-pruned points (about 365K). Also, both SumDx and SumDist functions are called in PS (BF only calls SumDist), where SumDx's x-distances sum, if successful, will probably get bigger than the max heap distance near the end of the loop, so the jump-off will not have much gain.

The results for SpatialHadoop are similar, 30% improvement for GD-BF and 40% for QT-BF. PS performance is the same in GD, but it is about 10% better in QT. Table 5.9 does not completely justify this, so we can assume that the sum calculations happened to favor the early loop breaks in many cases.

5.7.2 Scaling experiments

In this section we will use various combinations of Query and Training datasets, including big ones, and a variation of the number of computing nodes. In both cases we are searching for $K = 10$ neighbors. We have also tried scaling K to 100 and 1000, but the performance remained practically the same, so we will not present these experiments.

Figure 5.44 shows how the algorithm on both frameworks behaves when deactivating some computing nodes. The Query and Training datasets are hydrography and parks (Figure 5.27). We used QT-PS with centroid overlapping and "Fast Sums" turned on in both frameworks, selecting the best performers from each.

There is a 30% increase in total time when going from 8 to 4 nodes and a 63% increase from 4 to 2 nodes, in Hadoop. The corresponding percentages for SpatialHadoop are 23%

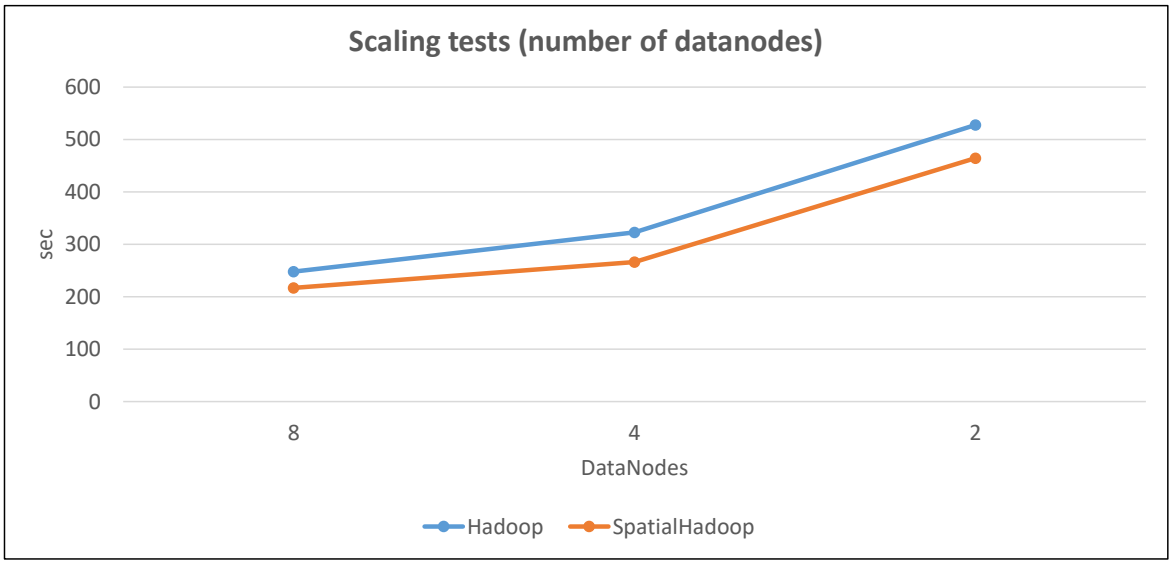

Figure 5.44: Scaling tests, datanodes.

and 74% respectively.

Figures 5.45, 5.46 and 5.47 show a direct comparison of Hadoop vs SpatialHadoop for three different Query datasets (two clustered synthetics and one real) vs three different Training datasets with vastly different cardinality. GD-BF with centroid circle overlapping is used in all cases, with varying N to achieve optimal performance and "Fast Sums" turned on. The dataset combinations refer to Figures 5.27, 5.29, 5.31 and 5.30. Last two bar pairs in Figure 5.47 regard the default and the new location of hydrography dataset in Figure 5.31.

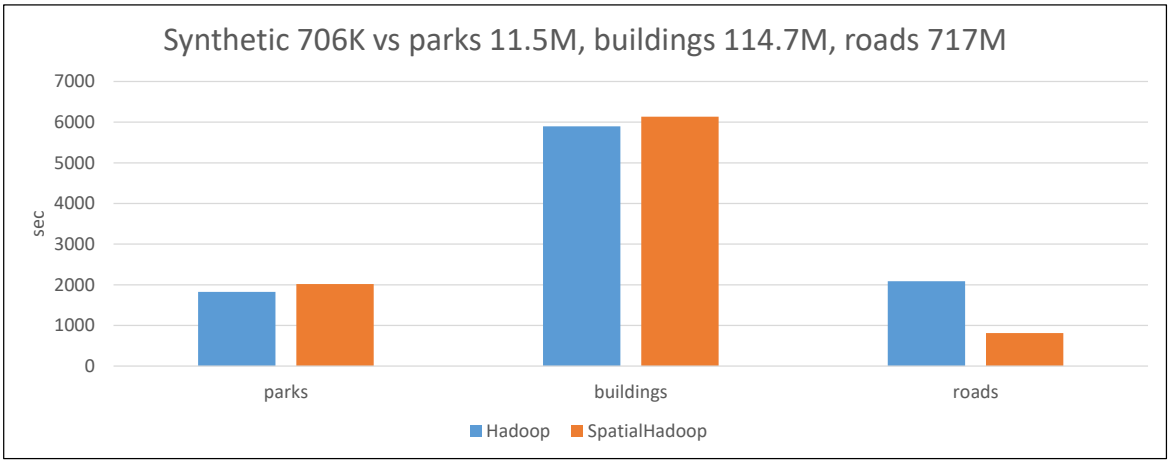

Figure 5.45: Scaling tests, small synthetic.

The synthetic Query datasets that were originally cropped over the African continent of the road networks file, are incompatible with the parks and buildings files and are moved to an arbitrary position to match them, as in Figure 5.30.

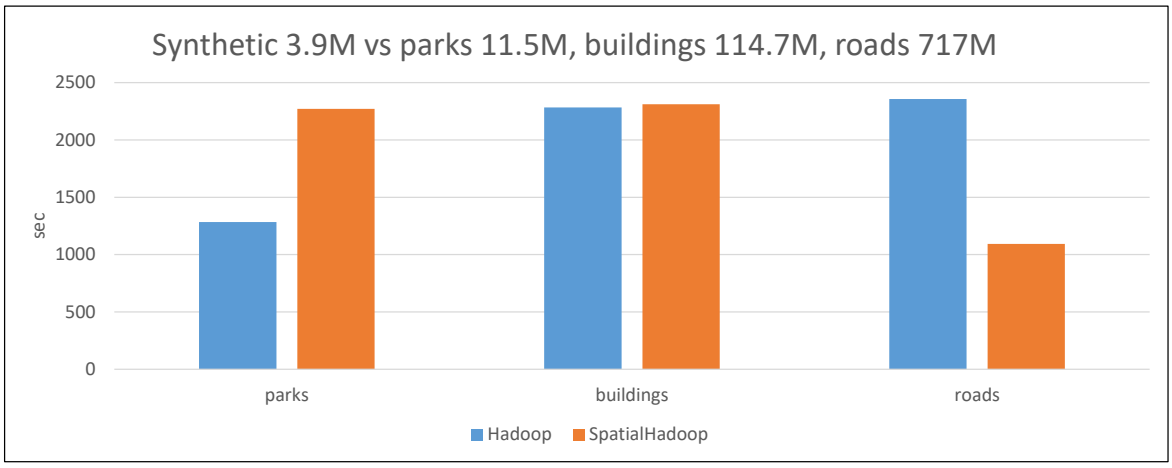

Figure 5.46: Scaling tests, large synthetic.

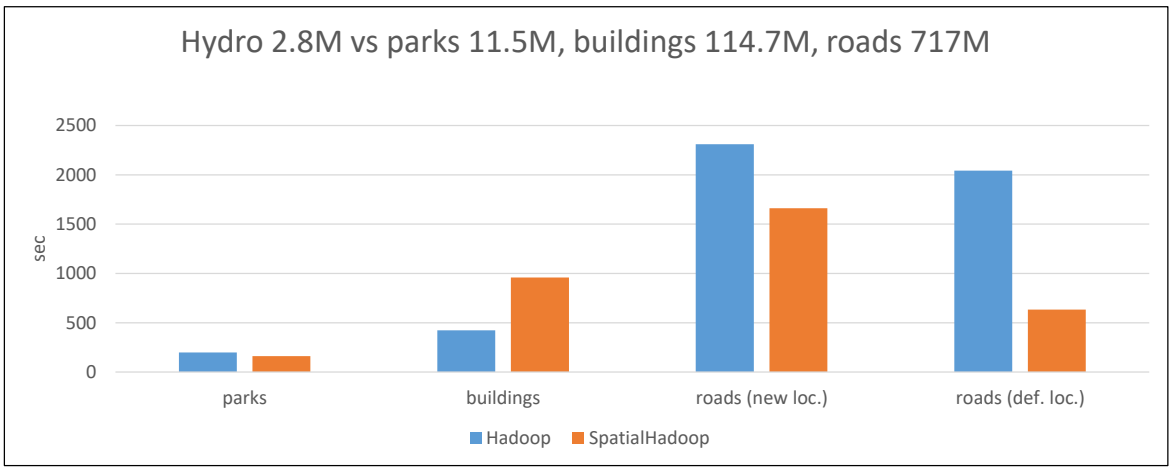

Figure 5.47: Scaling tests, hydro.

In Figure 5.45, we can observe that the two systems perform similarly in small and medium size Training datasets, with Hadoop being a little faster. However, SpatialHadoop clearly dominates in the largest Training, being more than twice as fast as plain Hadoop. Another thing to notice is that the buildings dataset is far more slower than the other two, even than the much larger roads. This proves that the number of points alone is not the decisive factor in this algorithm's performance. The relative position of the datasets is clearly more important because it controls the success or failure of the heuristics and the size of data transferred across the network.

In Figure 5.46, the performance differences between Hadoop and SpatialHadoop were maximized in the small and large Training datasets. The middle one remains neutral, performance wise. Also notice the total time scale in these graphs; the second and larger Query performs faster than the smaller one. This is a demonstration of the importance of the Query

datasets' distribution, even though the two synthetic datasets are located inside the same MBR.

In Figure 5.47, we can observe a smoother scaling of performance, as a function of the Training cardinality. SpatialHadoop loses in the middle Training, but wins again in the largest one. The hydrography dataset is compatible with parks and buildings, but not with roads and is scaled differently, hence the odd "default" location (below South America) and shape when paired with roads. Because in its default position it has almost no Training points below it, we decided to move it to an arbitrary new position (over Sahara desert). In the default location, while the metrics show that there are only about ten points examined in the second phase and less than ten in the third one, Hadoop shows little performance gain, compared to the new location, where many more points are processes in both second and third phases. We conclude that the performance in the default location is completely I/O bound; the only thing that matters here is the data transfer between nodes through the HDFS. The sheer size of the roads dataset (30 GB) which has to be read and transferred in every phase, dominates total performance. SpatialHadoop clearly benefits more and broadens the performance gap. This is the result of its effective pre-partitioning phase, which is not counted in total time.

5.7.3 Comparison to the older version

The final section of the experiments will show what we have achieved with all these aggregated major and minor contributions to the original algorithm of [52].

We have tested the original algorithm's QT-PS version ("old") from [52] against the current best performers ("new") from Hadoop and SpatialHadoop, as presented in previous sections. The Query and Training datasets are hydrography and parks and we are searching for $K = 10$ neighbors. The new methods have the "Fast Sums" turned on. The results are depicted in Figure 5.48.

Even when using the same Phase 1.5 cell refinement method (MBR), the improvement is an amazing 70% in Hadoop and 76% in SpatialHadoop. This is the added effect of the "Fast Sums" method and of several minor code optimizations in calculations and output data management.

The improvement reaches a sky high percentage of 96% and 97% in Hadoop and SpatialHadoop, respectively, when using the new Phase 1.5 refinement algorithm (centroid circle) and transforms a slow algorithm to a blazing fast one.

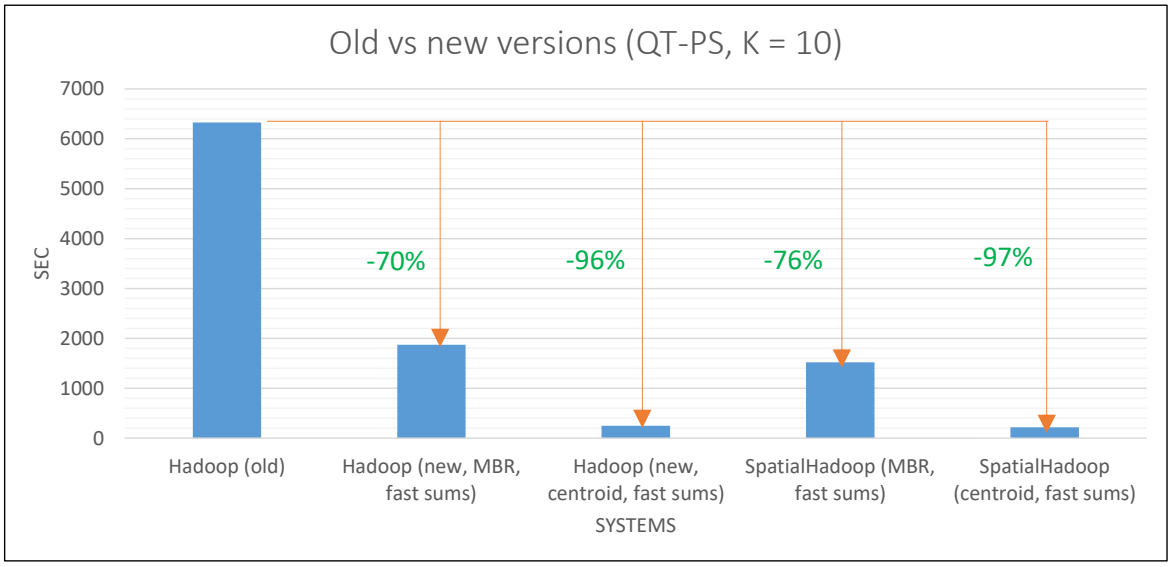

Figure 5.48: Old vs new configurations.

5.8 Experiments using prepartitioning

These experiments were presented in [55]. The major difference compared to [53] is the prepartitioning phase which preceeds the others and the prepartitioned dataset that effectively replaces the Training as input to all phases.

Only the GD-BF version for Hadoop was prepared and presented, because this was the winning combination in [53]. The Quadtree version is still a work in progress, but it looks promising so far.

In order to evaluate the behaviour of the proposed algorithm, we have used one real world dataset as Training and one real and two clustered synthetic ones as Query. The Training dataset contains the coordinates of road networks around the world from OpenStreetMap [17] and has 717M points. The real Query is based on one that contains linear hydrography coordinates inside the USA, but we have trimmed the farthest points and halved it, so it now contains 2.8 million points. To test different configurations, we moved it on the ocean, somewhere below South America ("default location"), and we also moved it over Sahara desert ("new location") for another set of experiments. The synthetic Query datasets originally contained 10M and 50M points, scattered in clustered regions all around the globe. We have cropped them to a rectangle that roughly covers the African continent and they now contain fewer points (705K and 3.9M respectively). All datasets' coordinates are normalized in $[0, 1]$ range.

We have set up a cluster of 9 virtual machines (1 NameNode and 8 DataNodes) running

Ubuntu Linux 18.04 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 2.8.5, 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 ran the new algorithm using the best parameters from [53] in order to perform a direct comparison with it and emphasize the optimizations. We also ran these tests with smaller Training datasets (the other ones from [53], with 114M and 11.5M points), but the improvements were negligible, so we won't present these tests. After all, real Big Data performance is what really matters. All experiments regard the discovery of $K = 10$ neighbors, partitioning method is Grid, Reducers use Brute-Force calculations and Phase 1.5 filtering method is centroid circle.

Figures 5.49, 5.50, 5.51 and 5.52 show the comparative performance of base algorithm (orange) vs the new one (blue) in six column pairs: the first column from the left is the prepartitioning time only, the next three are comparisons of the three distributed phases times and the last two columns on the right are the total time, with and without including prepartitioning time.

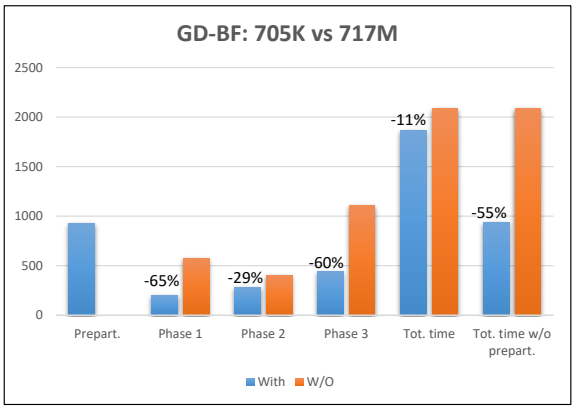

Figure 5.49: Synthetic 705K vs 717M.

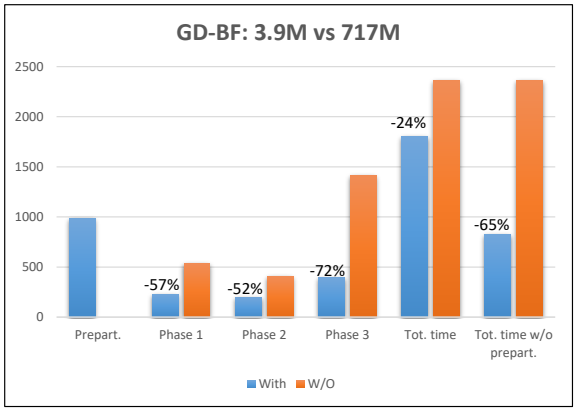

Figure 5.50: Synthetic 3.9M vs 717M.

In all graphs, we can see that the prepartitioning time takes about 1000 seconds for this Training dataset. This is the time to calculate the cell for each point, group the points by cell and write it to HDFS. Total time itself is about 2000 seconds for all Query datasets, so it takes half that time for prepartitioning only.

As seen in the graphs, every phase is benefited from prepartitioning of the Training dataset. Phase 1 time is reduced by 57% to 65%, Phase 2 time is reduced by 29% to 52% and Phase 3 time is reduced by 56% to 73%. Phase 3 profits the most, justifying our original

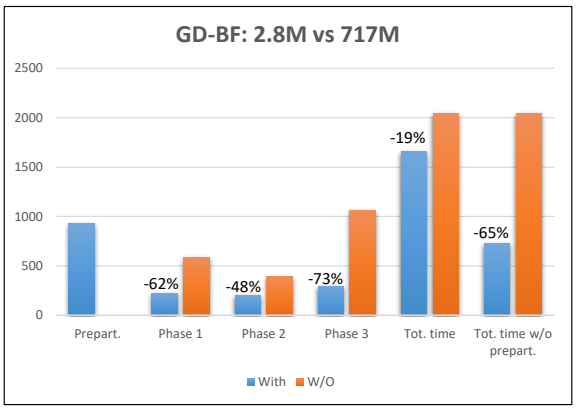

Figure 5.51: Real 2.8M (default location) vs 717M.

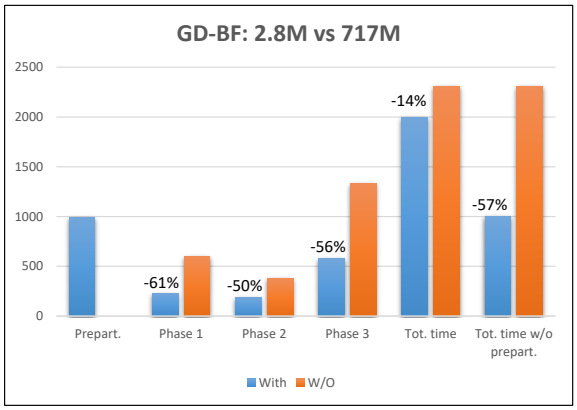

Figure 5.52: Real 2.8M (new location) vs 717M.

estimation, because it has to deal with most of the Training dataset (non-overlapped cells) and a lot of data is transferred between nodes.

Total time is reduced by 11% to 24% and by 55% to 65%, when including or not prepartitioning time. So, we observe that even taking into account the 1000 seconds of prepartitioning, which is half the algorithm's total time, it still has a positive effect in performance. But if the prepartitioning is done once for each Training dataset and we plan to use it with many Query's, the prepartitioning time is negligible and the performance is almost doubled.

5.9 Conclusions

The experiments on the first version of the algorithm [52] have shown that:

- The best performing computation approach was Brute-Force. The x-distance check of Plane-Sweep failed, mainly because it was applied in Phase 2, were the relative distances are small. Quad tree partitioning generally performed well, while keeping the number of cells extremely low, except for Query dataset 2, where it came behind Grid partitioning by 20%.

- The heuristics were applied in parallel Phases 2 and 3, in both Mappers and Reducers. They resulted in efficient pruning of many intermediate points and cells, saving lots of unnecessary computations. This becomes more obvious in Phase 3, by looking at its relative to Phase 2 performance. Phase 3 Mapper prunes almost every distant cell, leaving little work for the Reducer.

- Performance favors fine grained space partitioning. More Reducers with less work for each are available at the same time, exploiting distributed computing more efficiently.

- Finally, Query MBR shape seems to be of little importance, compared to the cardinality and relative position of both datasets. Space partitioning plays a major role for the pairing of points in every cell. Its effect will generally be coincidental and difficult to foresee.

The experiments on the improved version of the algorithm [53] have shown that:

- The decisive factor for the performance boost proved to be the new refinement method, which drastically decreased the number of processed cells and points. The processed cells number dropped by a factor of 1000, saving thousands of time consuming calculations.

- The "Fast Sums" technique allowed us to early jump off the iterative sum computation loops when we had to compare the final sum to a specific value, for pruning purposes. This proved very effective in certain configurations (MBR refinement and Brute-Force reducers) and less effective in others (centroid circle refinement, Plane-Sweep reducers).

- Plane-sweep reducers were very effective combined with MBR refinement. They kept a clear performance distance from their Brute-Force counterparts. However, when using centroid circle method, these differences were flattened and Brute-Force was marginally better.

- When using Hadoop, between Grid and Quadtree partitioning methods, Grid was crowned winner, even by short head. This comes in contrast to the much more sophisticated partitioning approach of the latter. We assume that Quadtree gets slowed down by the point and cell location queries, which are much more costly than Grid's.

- The first pair of datasets put the query points MBR on top of a dense population of training points. When we moved the same query points over a sparse area, the experiments showed a dramatic increase in performance, which confirmed what was logically expected. Fewer points, fewer calculations, the farther the points, the more efficient the heuristics.

- SpatialHadoop's two level partitioning technique is more effective than Hadoop's simpler one. This was shown in the combination of Quadtree and Grid against plain Quadtree and in all centroid circle based configurations. The pre-partitioning phase that effectively distributes data to the nodes, as well as pre-processing of points before the mapper with its dedicated filters played the most important role.

- Analysis of what happens internally (under the hood) during the operation of the algorithm revealed the secret work of the heuristics and the major role they play in the algorithm's performance. When we disabled them, we saw how much the total time was increased in each case. The performance metrics also gave us some insight to the effectiveness of Plane-Sweep's mass pruning x-distance check, against Brute-Force's single point pruning only heuristic.

- We saw that the algorithm behaves normally when the datanodes number varies. However, some further tuning might be needed to achieve optimal performance. While testing synthetic Query's and large Training datasets, we observed a variation in both systems' performance which showed the importance of both datasets' relative location over cardinality and the major role of data locality, the advantage that crowned SpatialHadoop the clear winner for the couple of large Training datasets.

- Finally, we have reached a performance increase that approaches 100%, compared to our previous (and the only in literature) algorithm. This is the result of the aggregated improvements we have presented in this paper.

The experiments on the prepartitioning version of the algorithm [55] have shown that:

- The prepartitioning phase is taking almost half of the algorithm's time. It is an I/O costly process, but it proved very beneficial for each phase and the total time, especially when we plan to use the same Training dataset with many Query ones.

- Total time was reduced by 11% - 24%, when prepartitioning time is included and by 55% - 65% when it is not included.

- Per phase times are also reduced by a significant percentage, with Phase 3 gaining up to 73%.

Chapter 6

Conclusions and Future Directions

In this thesis we presented new and improved algorithms for processing several spatial queries in parallel and distributed environments. Some of these algorithms were the first ones in the literature to solve the queries in multiple machines shared-nothing setups. All the algorithms were presented in detail, using flowcharts and pseudo-code and furthermore they were extensively tested against a variety of real world and synthetic datasets in 2 and 3 dimensions, using popular frameworks such as Apache Hadoop, Apache Spark and SpatialHadoop. Implementations in different frameworks were also tested against each other. Several tuning parameters were tested for each algorithm, including space partitioning, refining methods, distance calculation, output data savings and variability of K. Whenever these algorithms were tested against other popular ones from the literature, they always performed better.

6.1 Conclusions

The numerous experiments conducted for each algorithm using combinations of different frameworks, datasets and tuning parameters have led to many interesting conclusions which will be presented separately for each query case.

6.1.1 KCPQ and DJQ

Although processing of the KCPQ has been studied extensively for centralized environments, few solutions have appeared for parallel and distributed frameworks. In [49], we presented an algorithm for KCPQ processing in Spark, separating data in strips and processing by plane sweep within each eligible pair of strips. To the best of our knowledge, this is

the first KCPQ algorithm in Spark.

In [50], we extended [49] by developing three alternative algorithms. Two variations of the Binary Space Partitioning (BSP) technique are used to partition the data, based on two different criteria: equal size (of contained points) and equal width of the child strips. We have also removed the second one of the two sampling stages used in [49]. These schemes were compared to the splitting strategy of [49]). We have performed an extensive set of experiments to evaluate the efficiency and scalability of the algorithm and the performance of the different partitioning schemes by using large real-world datasets. Results show that splitting into strips by means of BSP achieves better performance. This is mainly due to the fact that selecting the number of points within each strip as the preset criterion, instead of the number of strips, provides more flexibility in fine tuning the system. The performance was improved by 15-20% for larger datasets, although for smaller datasets' performance was barely improved. Single sampling also had a small positive impact. Lastly, equal size partitions exhibited great uniformity in points distributions inside stripes, as was expected, in contrast to equal width partitions, where the points cardinality varied significantly from stripe to stripe, leading to imbalanced computations. Hence the superiority of the former partitioning method.

Motivated by these observations, we utilized, elaborated and adapted ideas presented in [49,50,64] and proposed a new algorithm for the KCPQ in Spark, called SliceNBound (SnB) [51]. This is a four-phased, iterative algorithm in "plain" Apache Spark to perform efficient parallel KCPQ processing on big spatial datasets. It is based on a simple and, therefore, not very computationally demanding partitioning scheme that enables the two datasets to share a common partitioning. Additionally, it only exploits built-in functions of Spark, thus making it easy to be imported in any spatially-oriented, or general, Spark-based parallel system. To compute good upper bounds of the distance value of the K-th closest pair, we proposed a couple of fast heuristic methods that use a two-stage sampling technique. These methods can be used as a preprocessing phase for any technique that uses preprocessing. We also presented a version of the SnB algorithm for DJQs. More specifically, we introduced a couple of pair-partitioning techniques, called "Parent-child" and "Common-merged" partition. This way the points from both datasets are divided into the same stripes and a first, approximate solution is computed, using only the points in the same stripe. Later, the exact solution is found by using cross-border computations and distant stripes are pruned, using the upper bound we found in the previous step. Finally, we performed an extensive set of experiments using big real-

world points datasets, to study the performance of our algorithms and compared the DJQ one against the DJQ algorithm embedded in Simba, showing that our algorithm won by almost 75% in a wide range of the experiments. SnB also outperformed the previous method of [49] by more than 30%, when using larger datasets.

6.1.2 AKNNQ

The work published in [54] presents efficient algorithms for processing AKNN queries in the MapReduce programming framework. It is based on [58] which uses a five-phase MapReduce algorithm to classify spatial data using AKNNQ. We introduced three methods for improved partitioning, faster computations and intermediate phase output data reduction. The improvements work in two and three dimensional datasets. We put these methods to excessive experimental testing and found them to be faster than the base algorithm and even than other popular methods in the literature.

The first improvement is about performing faster calculations in the second and third phases reducers. Instead of Brute-Force distance calculations, which involve all points inside a cell, we introduce a Plane-Sweep method, which can prune thousands of points per cell and save a lot of computing time.

The second improvement consists in intermediate phase output data reduction ("Less-Data") which had a huge impact in the network traffic of the cluster. More specifically, Phase 3 produced a very large amount of data and was by far the slowest to complete. We managed to significantly reduce this amount by cutting out some data and significantly accelerate Phase 3.

The third and final improvement has to do with partitioning in the second and third phases mappers, uses Quadtrees in 2D and Octrees in 3D instead of plain grid. This partitioning method recursively divides the space in four equal quadrants in 2D and eight in 3D, until a given capacity requirement is met. This way the cells created are unequal in size but they contain almost the same number of points, leading to more balanced calculations across the cluster. Grid blindly cuts the space in equal cells, which may contain vastly different numbers of points, leading to imbalanced calculations and even timeouts.

These improvements were tested using two real world datasets as Input and Training consisting of about 11.5M points each and another real world dataset consisting of 110M points as Training only. The latter was used to test the scaling of the algorithm.

The first group of experiments showed that Plane-Sweep is more than 20% faster than the best Brute-Force when using Grid partitioning ($N \times N$ cells) with the optimal N for Brute-Force. Pairing Plane-Sweep with LessData gave our algorithm another performance boost, which grows almost linearly to K and puts the combination 40% ahead of Brute-Force. LessData method was found to save 40% - 50% of Phase 3's output data, which explains its performance gain due to network traffic savings.

The next set of experiments tested Quadtree against Grid, using Plane-Sweep + LessData. The best Quadtree (by tuning its capacity) won the best Grid by 30% in total time and produced about 77% less output data in Phase 3.

The algorithm was also tested using 2, 4 and 6 Datanodes (instead of 8) and it proved to scale well. Furthermore, we tested the algorithm using the 110M dataset as Training, which required to re-calibrate the new best performing N (Grid) and capacity (Quadtree). Grid's previous best performing N was 50% slower than the newest, while Quadtree proved more resilient, since its new best capacity beat the older one by only 17%.

Quadtree was also found to divide the cells in a way that non-empty cells (cells that contain at least one point) percentage was about 93%, while Grid's one was 4% - 8% only. This means that Grid unnecessarily cuts empty space to many small cells.

The above experiments regarded two dimensions. We artificially expanded the datasets adding a third dimension by randomizing z. 3D Grid proved to be even more less efficient than Octree, losing by more than 65%. Amplified by the z coordinate, Phase 3's output was even larger than 2D and Octree's more refined partitioning played a major role in its superior performance, by drastically reducing this phase's output size.

Scalability experiments in 3D were also conducted, similarly to 2D, and the conclusions were also similar. They showed that the naive Grid partitioning is much more affected by the "dimensionality curse" than its Octree counterpart.

Finally, we compared our algorithm to two others from the literature, HBRJ and PGBJ, the best of which was the latter, using Voronoi partitioning. However, our algorithm still proved to be faster in both 2D and 3D, when using Quad/Octree, Plane-Sweep and LessData, combined.

6.1.3 GKNNQ

The GKNNQ was introduced in [60] and [61] where the query was studied in a single machine environment. We extended this research for use in multiple machines using the MapReduce framework and exploited several methods and pruning heuristics from the literature. Our algorithm was the first to solve this query in parallel and distributed environments and uses a combination of totally seven (four local and three distributed) phases. It makes use of several geometric heuristics that easily prune distant cells and applies some Plane-Sweep formulas to easily compute sums of distances, were applicable. The space partitioning techniques used (in mappers) were Grid and Quadtree and the computational methods (in reducers) were Brute-Force and Plane-Sweep. This work was published in [52].

We later presented a SpatialHadoop version, which introduces a novel two-level partitioning method, along with several other enhancements for both Hadoop and SpatialHadoop versions, which greatly improved its performance. The most important of these enhancements was the improved selection method of eligible cells in an early local phase which led to a significant reduction of calculations in the subsequent distributed phase and to a general boost of the algorithm's performance. Furthermore, we incorporated special cluster-wide metrics to the code, which we switched on and off to measure the impact of the enhancements and the pruning heuristics and it led to very interesting conclusions. In the experiments we used combinations of real world and synthetic clustered datasets, measuring up to hundreds of millions points. This work was published in [53].

Finally, in [55] we presented a modified version of the algorithm in which there was a preliminary phase that pre-partitioned the Training dataset. This proved to be very effective because there were several repeating point location functions scattered around the code and performed the same actions multiple times.

The general conclusions derived from the experiments were as follows:

1. Grid partitioning almost always wins Quadtree partitioning, despite the latter's more sophisticated and balanced cell-points grouping. We concluded that this was the result of Quadtree's recursive point location functions, which are often called inside the code.

2. The algorithm generally favors quite large numbers of cells. This is true for both Grid and Quadtree partitioning. More but smaller cells contain fewer points and the reducers have less computations to perform in each one. But the performance stagnates after a

point, because more reducers and computing nodes have to be added to the system.

3. Plane-Sweep is generally less efficient than Brute-Force, despite its x-distance pruning. The reason behind this is the geometric pruning heuristic that is always computed, without paying-off by pruning enough distance calculations.

4. The new cell refinement method proved to be almost 90% faster than the older one, thanks to the much smaller number of cells and points it feeds to the next distributed phase, thus saving lots of calculations. The new method favors Brute-Force, while the older one favors Plane-Sweep.

5. Both refinement methods perform almost the same when the Query MBR encloses very few Training points. After all, this is exactly the reason behind the new method's success.

6. The method we used to save distance sums computations gave up to 40% performance improvement when using Brute-Force combined with the old refinement method. However, it had little effect on the new refinement method and/or Plane-Sweep.

7. Scaling K had negligible impact on performance. Scaling computing nodes had an expected performance impact. Scaling Training dataset (using a much larger one) had a great impact and SpatialHadoop performed much better than plain Hadoop.

8. SpatialHadoop generally performed better than plain Hadoop, mainly as a result of its two-level partitioning and its spatial filtering functions.

9. Comparison between the first and the improved version of the algorithm gave the latter a clear victory with up to 97% performance improvement.

10. Using prepartitioning on the Training dataset gave it another 11% - 24% improvement, with prepartitioning time included, and up to 65% with prepartitioning time excluded. Prepartitioning time took almost half of the whole time and each distributed phase gained from 30% to 75% improvement.

6.2 Future directions

Although, there are many future research directions that could build on the work presented in the thesis, the most mature ones are summarized in the following:

- To extend the KCPQ algorithms and the GKNNQ algorithms to 3-dimensional spatial data and to extend the algorithms of all the queries studied to multidimensional data of more than three dimensions. Extending to more dimensions is always a challenge, because of the "dimensionality curse", as the authors of [58] note. We already saw a massive drop in performance when moving from two to three dimensions in AKNN, where distance functions and data structures became more complex (e.g. Euclidean distance formula for two points to three and Quadtree to Octree). Two prevalent phenomena that arise when the number of dimensions grows are Data Sparsity and Distance Concentration, which lead to the problem of all the pairwise distances between different points in the space converging to similar values, as the dimensionality increases. The algorithms we developed utilize distance-based pruning and will probably be affected by the dimensionality curse. Redesigning of the algorithms may be required, as well as the use of alternative techniques.

- To extend and apply the algorithms to spatiotemporal data and implement them in Beast [2] and ST-Hadoop [77]. Spatiotemporal data are data which are collected across both space and time and thus they include at least one spatial and one temporal property. For example, in the queries we studied, the Query datasets could be taxi trajectory, geo-tagged tweets or astronomical data. Current big spatial systems have difficulties processing queries with a temporal property because they produce stationary spatial indexes which may associate outdated data and so they would need to be rebuilt perpetually, thus greatly increasing the processing time. ST-Hadoop is built upon Spatial-Hadoop and injects it with spatiotemporal awareness, mainly because it creates a two-level, first temporal and then spatial, index. Beast on the other hand is based on Spark and stands for "Big Exploratory Analytics on Spatio-Temporal data" and it claims to support both vector and raster data with multidimensional data types and index structures. Both systems, especially Beast, are quite recent and remain largely unexplored so far.

- To implement the AKNNQ and GKNNQ algorithms in Apache Spark [75] (regard-

ing GKNNQ, we are already working in this direction and the implementation approaches completion) and all the algorithms in Apache Sedona [71] (formerly known as GeoSpark). Porting the GKNNQ algorithm to Spark has shown several benefits; the code can be written as a main class which calls auxiliary external classes and functions, instead of writing several distinct MapReduce phases, the datasets can be read only once and then placed into RDDs, and lengthy MapReduce functions can be implemented as one-line series of transformations and actions on a single RDD, thanks to Spark's functional programming inherent structure. However, the algorithm largely remains the same. Our early experiments have shown a significant performance increase compared to Hadoop and SpatialHadoop. We expect similar behavior of the AKNNQ algorithm's transfer to Spark. Sedona provides SpatialRDDs for storing spatial data as points, polygons and lines objects, and claims to outperform other spatial-aware frameworks, such as Simba and SpatialHadoop.

- To use alternative partitioning schemes that produce a space hierarchy, such as members of the R-tree family and Voronoi diagrams [24]. A partitioning based on R-trees has already been implemented in KCPQ [50]. We also competed against a Voronoi partitioning from the literature in our AKNNQ, but our algorithm proved to be faster. The R-tree is a data structure often used in spatial applications and it consists of minimum bounding rectangles containing spatial objects. We haven't used it so far in our AKNNQ and GKNNQ algorithms because its nodes may be overlapping and our algorithms are tailored to non-overlapping data structures. A Voronoi diagram consists of several disjoint polygonal regions constructed around specific points in space called pivots. Those polygons' characteristic is that an arbitrary point inside each one is closer to its pivot than to any other pivot of any other polygon. Voronoi diagrams have a wide range of applications in many scientific fields, such as Computer Science, Biology, Astronomy, etc.

- To transform the algorithms for utilizing multiple CPU or GPU cores in data nodes of the shared-nothing distributed system, exploiting techniques presented in [80]. Modern commercial GPUs are equipped with several gigabytes of very fast memory, thousands of computing cores and specialized programming interfaces for general purpose use. They can host a very large number of threads accessing the same memory and in many applications they have proven to be faster than CPUs. However, for datasets that exceed

device memory capacity, the communication overhead between host (CPU) and GPU may degrade the overall performance. So, the biggest challenge will be to create an efficient partitioning and indexing that will assign partitioned data to cluster nodes and in-node CPU and GPU cores.

- Using the techniques of this thesis, to create algorithms for other spatial queries, like skyline [36], reverse nearest neighbor [23] and spatio-textual queries [93]. Skyline query takes a dataset with a given dominance relationship and returns those points that cannot be dominated by others. Given a dataset P and a point q, a reverse (K) nearest neighbor query retrieves all the points of P that have q as their nearest neighbor or as one of their K nearest neighbors. Spatio-textual queries retrieve the most similar objects with respect to a given location and a keyword set. All these queries could benefit from the methods we have developed and used in this thesis, such as Plane-Sweep and Quadtree partitioning and are related to the nearest neighbor query.

- To develop approximate versions of the algorithms [12] that balance between speed and accuracy. So far, in most of our algorithms we computed an intermediate approximate solution first and we found the exact one afterwards. In many cases the approximate and the exact solutions were identical, while in others they differed considerably. We could further study the parameters that led to an erroneous approximate solution for each query and try to tune them to achieve a desirable approximation, while avoiding complex calculations in order to improve performance. For example, while drawing the circle to discover eligible cells in AKNNQ and GKNNQ we could experiment with different starting radii and increment steps or decrease of the distance accuracy.

- To develop versions of the algorithms for uncertain data [42, 44, 46]. Uncertain data is data which contains noise that makes it deviate from the correct, intended or original values. Such data are collected e.g. from sensor networks of limited accuracy, like GPS sensors of commercial smartphones. Because of their uncertainty, very large amounts of data need processing to extract credible results. There are numerous papers in the literature that deal with the queries studied here in a single machine when the data is uncertain, but very few attempt to solve them in a parallel and distributed environment. Adding uncertainty to the big volume of data, the algorithms will probably need redesigning to discover nearest classes rather than nearest objects.

Appendix A

Publications

This thesis has led to the following publications:

- George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos. Closest pairs query processing in Apache Spark. *In Proc. of the 8th Int. Conf. on Cloud Computing, GRIDs and Virtualization, CLOUD COMPUTING '17*, Athens, Greece, February 19-23, 2017, pages 26–31, 2017.
  `https://www.thinkmind.org/articles/cloud_computing_2017_2_10_28010.pdf`
  BEST PAPER AWARD: `http://www.iaria.org/conferences2017/awardsCLOUDCOMPUTING17/cloudcomp2017_a2.pdf`

- George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos. Binary space partitioning for parallel and distributed closest pairs query processing. *International Journal on Advances in Software*, 10(3-4):275–285, 2017.
  `https://www.thinkmind.org/articles/soft_v10_n34_2017_10.pdf`

- George Mavrommatis, Panagiotis Moutafis, Michael Vassilakopoulos, Francisco García, and Antonio Corral. Slicenbound: Solving closest pairs and distance join queries in Apache Spark. *In Proc. of the 21st European Conf. on Advances in Databases and Information Systems, ADBIS '17, Nicosia, Cyprus, September 24-27, 2017,* vol. 10509 of LNCS, pages 199–213, Springer, 2017.
  `https://doi.org/10.1007/978-3-319-66917-5_14`

- Panagiotis Moutafis, Francisco García, George Mavrommatis, Michael Vassilakopoulos, Antonio Corral, and Luis Iribarne. Mapreduce algorithms for the K group nearest

neighbor query. *In Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, Limassol, Cyprus, April 8-12, 2019,* pages 448–455, ACM, 2019.
https://doi.org/10.1145/3297280.3299733

- Panagiotis Moutafis, Francisco García, George Mavrommatis, Michael Vassilakopoulos, Antonio Corral, and Luis Iribarne. Algorithms for processing the group k nearest neighbor query on distributed frameworks. *Distributed and Parallel Databases*, Published online: Nov. 9, 2020.
https://doi.org/10.1007/s10619-020-07317-8

- Panagiotis Moutafis, George Mavrommatis, Michael Vassilakopoulos, and Spyros Sioutas. Efficient processing of all k nearest neighbor queries in the mapreduce programming framework. *Data & Knowledge Engineering*, 121:42–70, 2019.
https://doi.org/10.1016/j.datak.2019.04.003

- Panagiotis Moutafis, George Mavrommatis, and Polychronis Velentzas. Prepartitioning in mapreduce processing of group nearest neighbor query. *In Proc. of the 24th Panhellenic Conf. on Informatics, PCI '20, Athens, Greece, November 20-22, 2020,* pages 380-385, ACM, 2020.
https://doi.org/10.1145/3437120.3437345

References

[1] Ablimit Aji, Hoang Vo, and Fusheng Wang. Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910, 2015.

[2] Beast homepage. `https://bitbucket.org/eldawy/beast/src/master/`.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, pages 322–331. ACM, 1990.

[4] Christian Böhm and Florian Krebs. The *k*-nearest neighbour join: Turbo charging the KDD process. *Knowledge and Information Systems*, 6:728–749, 2004.

[5] Francisco Borges, Albert Gutierrez-Milla, Remo Suppi, and Emilio Luque. Strip partitioning for ant colony parallel and distributed discrete-event simulation. In *Proc. of the Int. Conf. on Computational Science, ICCS '15, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015*, volume 51 of *Procedia Computer Science*, pages 483–492. Elsevier, 2015.

[6] Huu Vu Lam Cao, Trong Nhan Phan, Minh Quang Tran, Thanh Luan Hong, and Minh Nhat Quang Truong. Processing all k-nearest neighbor query on large multidimensional data. In *Proc. of the 2016 Int. Conf. on Advanced Computing and Applications, ACOMP '16, Can Tho City, Vietnam, November 23-25, 2016*, pages 11–17. IEEE, 2016.

[7] Georgios Chatzimilioudis, Constantinos Costa, Demetrios Zeinalipour-Yazti, Wang-Chien Lee, and Evaggelia Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *IEEE Trans. on Knowledge & Data Engineering*, 28(4):925–938, 2016.

[8] Dehua Chen, Changgan Shen, Jieying Feng, and Jiajin Le. An efficient parallel top-k similarity join for massive multidimensional data using spark. *Int. Journal of Database Theory and Application*, 8(3):57–68, 2015.

[9] Yun Chen and Jignesh M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *Proc. of the 23rd Int. Conf. on Data Engineering, ICDE '07, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1056–1065. IEEE, 2007.

[10] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data, Dallas, Texas, USA, May 16-18, 2000*, pages 189–200. ACM, 2000.

[11] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data & Knowledge Engineering*, 49(1):67–104, 2004.

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

[13] Antonio Corral and Michael Vassilakopoulos. Query processing in spatial databases. In Laura C. Rivero, Jorge Horacio Doorn, and Viviana E. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 511–516. Idea Group, 2005.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Daniel J. Eisenstein and Piet Hut. Hop: A new group-finding algorithm for n-body simulations. *The Astrophysical Journal*, 498(1):137–142, 1998.

[16] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial partitioning techniques in spatial hadoop. *Proc. of the VLDB Endowment*, 8(12):1602–1605, 2015.

[17] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Proc. of the 1st IEEE Int. Conf. on Data Engineering, ICDE '15, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363. IEEE, 2015.

[18] Ahmed Eldawy and Mohamed F. Mokbel. The era of big spatial data: A survey. *Foundations & Trends in Databases*, 6(3-4):163–273, 2016.

[19] Ahmed Eldawy and Mohamed F. Mokbel. The era of big spatial data. *Proc. of the VLDB Endowment*, 10(12):1992–1995, 2017.

[20] Hicham G. Elmongui, Mohamed F. Mokbel, and Walid G. Aref. Continuous aggregate nearest neighbor queries. *GeoInformatica*, 17(1):63–95, 2013.

[21] Tobias Emrich, Franz Graf, Hans-Peter Kriegel, Matthias Schubert, and Marisa Thoma. Optimizing all-nearest-neighbor queries with trigonometric pruning. In *Proc. of the 22nd Int. Conf. on the Scientific and Statistical Database Management, SSDBM '10, Heidelberg, Germany, June 30 - July 2, 2010*, volume 6187 of *LNCS*, pages 501–518. Springer, 2010.

[22] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[23] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. MRSLICE: efficient rknn query processing in spatialhadoop. In *Proc. of the 9th Int. Conf. on Model and Data Engineering, MEDI '19, Toulouse, France, October 28-31, 2019*, volume 11815 of *LNCS*, pages 235–250. Springer, 2019.

[24] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. Improving distance-join query processing with voronoi-diagram based partitioning in spatialhadoop. *Future Generation Computer Systems*, 111:723–740, 2020.

[25] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Enhancing spatialhadoop with closest pair queries. In *Proc. of the 20th East European Conf. on Advances in Databases and Information Systems, ADBIS '16, Prague, Czech Republic, August 28-31, 2016*, volume 9809 of *LNCS*, pages 212–225. Springer, 2016.

[26] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Efficient large-scale distance-based join queries in spatialhadoop. *GeoInformatica*, 22(2):171–209, 2018.

[27] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Efficient distance join query processing in distributed spatial data management systems. *Information Sciences*, 512:985–1008, 2020.

[28] Fangda Guo, Ye Yuan, Guoren Wang, Lei Chen, Xiang Lian, and Zimeng Wang. Cohesive group nearest neighbor queries over road-social networks. In *Proc. of the 35th IEEE Int. Conf. on Data Engineering, ICDE '19, Macao, China, April 8-11, 2019*, pages 434–445. IEEE, 2019.

[29] Gilberto Gutierrez and Pablo Sáez. The k closest pairs in spatial databases - when only one set is indexed. *GeoInformatica*, 17(4):543–565, 2013.

[30] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD Int. Conf. on Management of Data, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57. ACM, 1984.

[31] Apache software foundation: Hadoop homepage. `https://hadoop.apache.org/`.

[32] Tanzima Hashem, Lars Kulik, and Rui Zhang. Privacy preserving group nearest neighbor queries. In *Proc. of the 13th Int. Conf. on Extending Database Technology, EDBT '10, Lausanne, Switzerland, March 22-26, 2010*, volume 426 of *ACM ICPS*, pages 489–500. ACM, 2010.

[33] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Washington, USA, June 2-4, 1998*, pages 237–248. ACM, 1998.

[34] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[35] Tao Jiang, Yunjun Gao, Bin Zhang, Qing Liu, and Lu Chen. Reverse top-k group nearest neighbor search. In *Proc. of the 14th Int. Conf. on Web-Age Information Management, WAIM '13, Beidaihe, China, June 14-16, 2013*, volume 7923 of *LNCS*, pages 429–439. Springer, 2013.

[36] Christos Kalyvas and Manolis Maragoudakis. Skyline and reverse skyline query processing in spatialhadoop. *Data & Knowledge Engineering*, 122:55–80, 2019.

[37] Sven Koenig and Yury V. Smirnov. Graph learning with a nearest neighbor approach. In *Proc. of the 9th Annual Conf. on Computational Learning Theory, COLT '96, Desenzano del Garda, Italy, June 28-July 1, 1996*, pages 19–28. ACM, 1996.

[38] Jae-Gil Lee and Minseo Kang. Geospatial big data: Challenges and opportunities. *Big Data Research*, 2(2):74–81, 2015.

[39] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. STR: A simple and efficient algorithm for r-tree packing. In *Proc. of the 13th Int. Conf. on Data Engineering, ICDE '97, April 7-11, 1997, Birmingham, UK*, pages 497–506. IEEE, 1997.

[40] Feifei Li, Bin Yao, and Piyush Kumar. Group enclosing queries. *IEEE Trans. on Knowledge & Data Engineering*, 23(10):1526–1540, 2011.

[41] Hongga Li, Hua Lu, Bo Huang, and Zhiyong Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *Proc. of the 13th ACM Int. Workshop on Geographic Information Systems, ACM-GIS '05, Bremen, Germany, November 4-5, 2005*, pages 192–199. ACM, 2005.

[42] Jiajia Li, Botao Wang, Guoren Wang, and Xin Bi. Efficient processing of probabilistic group nearest neighbor query on uncertain data. In *Proc. of the 19th Int. Conf. on Database Systems for Advanced Applications, DASFAA '14, Part I, Bali, Indonesia, April 21-24, 2014*, volume 8421 of *LNCS*, pages 436–450. Springer, 2014.

[43] Jing Li, Jeppe Rishede Thomsen, Man Lung Yiu, and Nikos Mamoulis. Efficient notification of meeting points for moving groups via independent safe regions. *IEEE Trans. on Knowledge & Data Engineering*, 27(7):1767–1781, 2015.

[44] Xiang Lian and Lei Chen. Probabilistic group nearest neighbor queries in uncertain databases. *IEEE Trans. on Knowledge & Data Engineering*, 20(6):809–824, 2008.

[45] Xutong Liu, Feng Chen, and Chang-Tien Lu. Robust prediction and outlier detection for spatial datasets. In *Proc. of the 12th IEEE Int. Conf. on Data Mining, ICDM '12, Brussels, Belgium, December 10-13, 2012*, pages 469–478. IEEE, 2012.

[46] Zhang Liu, Chaokun Wang, and Jianmin Wang. Aggregate nearest neighbor queries in uncertain graphs. *World Wide Web*, 17(1):161–188, 2014.

[47] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. of the VLDB Endowment*, 5(10):1016–1027, 2012.

[48] Yanmin Luo, Hanxiong Chen, Kazutaka Furuse, and Nobuo Ohbo. Efficient methods in finding aggregate nearest neighbor by projection-based filtering. In *Proc. of the Int. Conf. on Computational Science and Its Applications, ICCSA '07, Part III, Kuala Lumpur, Malaysia, August 26-29, 2007*, volume 4707 of *LNCS*, pages 821–833. Springer, 2007.

[49] George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos. Closest-pairs query processing in apache spark. In *Proc. of the 8th Int. Conf. on Cloud Computing, GRIDs and Virtualization, CLOUD COMPUTING '17, Athens, Greece, February 19-23, 2017*, pages 26–31, 2017.

[50] George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos. Binary space partitioning for parallel and distributed closest-pairs query processing. *Int. Journal on Advances in Software*, 10(3-4):275–285, 2017.

[51] George Mavrommatis, Panagiotis Moutafis, Michael Vassilakopoulos, Francisco García-García, and Antonio Corral. Slicenbound: Solving closest pairs and distance join queries in apache spark. In *Proc. of the 21st European Conf. on Advances in Databases and Information Systems,*

ADBIS '17, Nicosia, Cyprus, September 24-27, 2017, volume 10509 of *LNCS*, pages 199–213. Springer, 2017.

[52] Panagiotis Moutafis, Francisco García-García, George Mavrommatis, Michael Vassilakopoulos, Antonio Corral, and Luis Iribarne. Mapreduce algorithms for the K group nearest-neighbor query. In *Proc. of the 34th ACM/SIGAPP Symp. on Applied Computing, SAC '19, Limassol, Cyprus, April 8-12, 2019*, pages 448–455. ACM, 2019.

[53] Panagiotis Moutafis, Francisco García-García, George Mavrommatis, Michael Vassilakopoulos, Antonio Corral, and Luis Iribarne. Algorithms for processing the group k nearest-neighbor query on distributed frameworks. *Distributed and Parallel Databases*, 2020. Published online, `https://doi.org/10.1007/s10619-020-07317-8`, Nov. 9, 2020.

[54] Panagiotis Moutafis, George Mavrommatis, Michael Vassilakopoulos, and Spyros Sioutas. Efficient processing of all-k-nearest-neighbor queries in the mapreduce programming framework. *Data & Knowledge Engineering*, 121:42–70, 2019.

[55] Panagiotis Moutafis, George Mavrommatis, and Polychronis Velentzas. Prepartitioning in mapreduce processing of group nearest-neighbor query. In *Proc. of the 24th Panhellenic Conf. on Informatics, PCI '20, Athens, Greece, November 20-22, 2020*, pages 380–385. ACM, 2020.

[56] Sansarkhuu Namnandorj, Hanxiong Chen, Kazutaka Furuse, and Nobuo Ohbo. Efficient bounds in finding aggregate nearest neighbors. In *Proc. of the 19th Int. Conf. on Database and Expert Systems Applications, DEXA '08, Turin, Italy, September 1-5, 2008*, volume 5181 of *LNCS*, pages 693–700. Springer, 2008.

[57] Thao P. Nghiem, David Green, and David Taniar. Peer-to-peer group k-nearest neighbours in mobile ad-hoc networks. In *Proc. of the 19th IEEE Int. Conf. on Parallel and Distributed Systems, ICPADS '13, Seoul, Korea, December 15-18, 2013*, pages 166–173. IEEE, 2013.

[58] Nikolaos Nodarakis, Evaggelia Pitoura, Spyros Sioutas, Athanasios K. Tsakalidis, Dimitrios Tsoumakos, and Giannis Tzimas. kdann+: A rapid aknn classifier for big data. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 24:139–168, 2016.

[59] United Nations Initiative on Global Geospatial Information Management. Future trends in geospatial information management: the five to ten year vision. `https://ggim.un.org/documents/Future-trends.pdf`, 2013.

[60] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group nearest neighbor queries. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proc. of the 20th Int. Conf.*

on Data Engineering, ICDE '04, 30 March - 2 April 2004, Boston, MA, USA, pages 301–312. IEEE, 2004.

[61] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. on Database Systems*, 30(2):529–576, 2005.

[62] Dustakar Nagarjuna Rao and Dustakar Surendra Rao. Computational geometry leveraged by apache spark. *Journal of Innovation in Electronics and Communication Engineering*, 5(2):15–31, 2015.

[63] Philippe Rigaux, Michel Scholl, and Agnès Voisard. *Spatial databases - with applications to GIS*. Elsevier, 2002.

[64] George Roumelis, Antonio Corral, Michael Vassilakopoulos, and Yannis Manolopoulos. New plane-sweep algorithms for distance-based join queries in spatial databases. *GeoInformatica*, 20(4):571–628, 2016.

[65] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. The k group nearest-neighbor query on non-indexed ram-resident data. In Cédric Grueau and Jorge Gustavo Rocha, editors, *Geographical Information Systems Theory, Applications and Management*, pages 69–89. Springer Int. Publishing, 2016.

[66] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. A new plane-sweep algorithm for the k-closest-pairs query. In *Proc. of the 40th Int. Conf. on Current Trends in Theory and Practice of Computer Science, SOFSEM '14: Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014*, volume 8327 of *LNCS*, pages 478–490. Springer, 2014.

[67] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. Plane-sweep algorithms for the K group nearest-neighbor query. In *Proc. of the 1st Int. Conf. on Geographical Information Systems Theory, Applications and Management, GISTAM '15, Barcelona, Spain, 28-30 April, 2015*, pages 83–93. SciTePress, 2015.

[68] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 71–79. ACM, 1995.

[69] Maytham Safar. Group K -nearest neighbors queries in spatial network databases. *Journal of Geographical Systems*, 10(4):407–416, 2008.

[70] Hanan Samet, C. A. Shatter, Randal C. Nelson, Y.-G. Huang, Kikuo Fujimura, and A. Rosenteld. Recent developments in linear quadtree-based geographic information systems. *Image & Vision Computing*, 5(3):187–197, 1987.

[71] Apache software foundation: Sedona homepage. `http://sedona.apache.org/`.

[72] Shashi Shekhar, Hui Xiong, and Xun Zhou, editors. *Encyclopedia of GIS*. Springer, 2017.

[73] Hyoseop Shin, Bongki Moon, and Sukho Lee. Adaptive and incremental processing for distance join queries. *IEEE Trans. on Knowledge & Data Engineering*, 15(6):1561–1578, 2003.

[74] SIMBA homepage. `http://www.cs.utah.edu/~dongx/simba/`.

[75] Apache software foundation: Spark homepage. `https://spark.apache.org/`.

[76] SpatialHadoop homepage. `https://github.com/aseldawy/spatialhadoop2/`.

[77] ST-Hadoop homepage. `http://st-hadoop.cs.umn.edu/`.

[78] Nusrat Sultana, Tanzima Hashem, and Lars Kulik. Group nearest neighbor queries in the presence of obstacles. In *Proc. of the 22nd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, pages 481–484. ACM, 2014.

[79] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proc. of the VLDB Endowment*, 9(13):1565–1568, 2016.

[80] Polychronis Velentzas, Panagiotis Moutafis, and George Mavrommatis. An improved gpu-based algorithmfor processing the k nearest neighbor query. In *Proc. of the 24th Panhellenic Conf. on Informatics, PCI '20, Athens, Greece, November 20-22, 2020*, pages 372–375. ACM, 2020.

[81] Kai Wang, Jizhong Han, Bibo Tu, Jiao Dai, Wei Zhou, and Xuan Song. Accelerating spatial data processing with mapreduce. In *Proc. of the 16th IEEE Int. Conf. on Parallel and Distributed Systems, ICPADS '10, Shanghai, China, December 8-10, 2010*, pages 229–236. IEEE, 2010.

[82] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An efficient method for KNN join processing. In *Proc. of the Thirtieth Int. Conf. on Very Large Data Bases, VLDB '04, Toronto, Canada, August 31 - September 3 2004*, pages 756–767. Morgan Kaufmann, 2004.

[83] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proc. of the 2016 SIGMOD Int. Conf. on Management of Data, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1071–1085. ACM, 2016.

[84] Congjun Yang and King-Ip Lin. An index structure for improving closest pairs and related join queries in spatial databases. In *Proc. of teh Int. Database Engineering & Applications Symp., IDEAS '02, Edmonton, Canada, July 17-19, 2002*, pages 140–149. IEEE, 2002.

[85] Takuya Yokoyama, Yoshiharu Ishikawa, and Yu Suzuki. Processing all k-nearest neighbor queries in hadoop. In *Proc. of the 13th Int. Conf. on Web-Age Information Management, WAIM '12, Harbin, China, August 18-20, 2012*, volume 7418 of *LNCS*, pages 346–351. Springer, 2012.

[86] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *Proc. of the 31st IEEE Int. Conf. on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pages 34–41. IEEE, 2015.

[87] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based KNN join processing for high-dimensional data. *Information and Software Technology*, 49(4):332–344, 2007.

[88] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *Proc. of the 23rd SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 70:1–70:4. ACM, 2015.

[89] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation, NSDI '12, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[90] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[91] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proc. of the 15th Int. Conf. on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012*, pages 38–49. ACM, 2012.

[92] Dongxiang Zhang, Chee-Yong Chan, and Kian-Lee Tan. Nearest group queries. In *Proc. of the Conf. on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 7:1–7:12. ACM, 2013.

[93] Yu Zhang, Youzhong Ma, and Xiaofeng Meng. Efficient spatio-textual similarity join using mapreduce. In *Proc. of the Int. Joint Conf. on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), IEEE/WIC/ACM '14, - Volume II, Warsaw, Poland, August 11-14, 2014*, pages 52–59. IEEE, 2014.

[94] Liang Zhu, Yinan Jing, Weiwei Sun, Dingding Mao, and Peng Liu. Voronoi-based aggregate nearest neighbor query processing in road networks. In *Proc. of the 18th ACM SIGSPATIAL Int. Symp. on Advances in Geographic Information Systems, ACM-GIS '10, November 3-5, 2010, San Jose, CA, USA*, pages 518–521. ACM, 2010.