Francisco José García García

Efficient Query Processing in Distributed Spatial Data Management Systems

Doctoral Thesis

Supervisors
Dr. Antonio Corral
Dr. Luis Iribarne

Departament of Informatics
Applied Computing Group

UNIVERSIDAD DE ALMERÍA

Departament of Informatics
University of Almería

Efficient Query Processing in Distributed Spatial Data Management Systems

DOCTORAL THESIS

by

Francisco José García García

Supervisors

Dr. Antonio Corral
Associate Professor
Department of Informatics
University of Almería, Spain

Dr. Luis Iribarne
Associate Professor
Department of Informatics
University of Almería, Spain

Almería, June, 2021

Written by: Francisco José García García
Printed by: Murex (Almería, Spain)

June 2021

Departamento de Informática
Universidad de Almería

Procesamiento Eficiente de Consultas en Sistemas de Gestión de Datos Espaciales Distribuidos

TESIS DOCTORAL

by

Francisco José García García

Dirigida por

Dr. Antonio Corral
Profesor Titular de Universidad
Departamento de Informática
Universidad de Almería, España

Dr. Luis Iribarne
Profesor Titular de Universidad
Departamento de Informática
Universidad de Almería, España

Almería, Junio, 2021

Escrito por: Francisco José García García
Impreso por: Murex (Almería, Spain)

Junio 2021

This file has been generated using LaTeX.

All Figures and Tables in this file are originals

Efficient Query Processing in
Distributed Spatial Data Management Systems

Francisco José García García
Departament of Informatics
Applied Computing Group (TIC-211)
University of Almería
Almería, June, 2021

http://acg.ual.es

To Susana,
to my parents, José and Brígida,
to all my family and the people I love

Acknowledgements

This thesis is the end of a journey that would not have been possible without the contribution and help of many people. A path that by walking has allowed me to meet and learn from great people, visit incredible places, and that has also had its awful moments due to the COVID pandemic.

In the first place, this journey in the world of research could not have started if it was not for Associate Professor Antonio Corral. At a family meal, my brother-in-law agreed to introduce me to the thrilling world of spatial data and their exciting queries. His help has been essential since he has always been there to give me that push that I needed, and among all that he has taught me, I have to highlight two things: that one should never give up, and that in the research field, you have to write a lot, although not I like it.

I would also thank Associate Professor Luis Iribarne for his supervision and advice throughout this thesis. Not only has he provided me with the necessary means to attend conferences and other research needs, but I have also been able to share experiences and knowledge with the different members of the Applied Computing Group (TIC-211) to enrich my research capabilities.

And speaking of families, I cannot forget the great Greek family. First of all, Professor Yannis Manolopoulos, in addition to being an excellent researcher in so many fields, taught me to appreciate good food and *chiringuitos*. As for Associate Professor Michael Vassilakopoulos, he has always found and suggested new ideas that have considerably improved the research work quality. Also, together with his wife, Eli, he showed me a small and beautiful part of Greece, which I would like to visit again. Finally, it has been a pleasure to meet Dr. Panagiotis Moutafis and Dr. George Mavrommatis that have been available whenever needed and with whom we have combined our knowledge and experience to obtain even better results.

I cannot forget Dr. Manolo Torres and Dr. José Antonio Martínez for providing me with an OpenStack infrastructure to carry out the experiments of this thesis. I swear that one day I will free my resource quota.

Moreover, I am grateful for the funding given by the EU ERDF and the Spanish Government under AEI Projects TIN2013-41576-R ("*Evolving dynamic systems in the cloud: A framework toward the smart user interfaces*") and TIN2017-83964-R ("*Co-Smart: Study of a holistic approach for the interoperability and coexistence of dynamic systems: Implication in Smart Cities models*") in which I have had the honor to participate.

And all this would not be possible without my parents, José and Brígida, who are the ones who started me on the journey of life. I am the person that I am thanks to them, and I will always follow their main advice: "My son, be good to people so that they can always speak well of you."

Nor could I have traveled this long road without the best travel companion you can have in life, my wife, Susana. I knew it from the first moment I met her, and I hope that destiny continues to provide us with good moments to enjoy her goodness and love. I know that I have had to borrow a long time, which I will gladly return with interest.

To finish, I would like to thank all who have directly or indirectly done their bits, such as my sister-in-law Rosa, Antonio Becerra, the people at breakfast, my coworkers, Javi, Miguel, Sera, Cristo and many others whom I may leave unmentioned.

Francisco José García García
Departament of Informatics
Applied Computing Group
University of Almería
Almería, 2021

Table of Contents

© 2021 García-García, F.J.

List of Figures

© 2021 García-García, F.J.

List of Tables

Abstract

Spatial Computing covers ideas, solutions, tools, technologies, and systems that transform our lives and society by creating a new understanding of spaces, locations, places, and properties. Since the term *Big Data* was coined for the first time in 2005, it has unleashed a worldwide revolution in scientific research and business. *Big Spatial Data* (BSD), the Big Data associated with spatial information, is now one of the most active research fields in spatial computing, mainly motivated by the rapid development of smart, sensor, and mobile technologies. Current usage of the term *Big Spatial Data* tends to refer to the process of capturing, storing, managing, analyzing, and visualizing huge amounts of spatial data, not using traditional tools and systems. Recent big spatial data developments have motivated the emergence of novel technologies for distributed processing of large-scale spatial data in shared-nothing clusters of computers, leading to *Distributed Spatial Data Management Systems* (DSDMSs). Distributed cluster-based computing systems can be classified as Hadoop-based or Spark-based systems. Based on this classification, two of the most leading DSDMSs are *SpatialHadoop* (disk-based DSDMS) and *LocationSpark* (in-memory-based DSDMS). These distributed systems support several characteristics like spatial data partitioning, indexing methods, and spatial query processing. An important aspect of these DSDMSs is to adopt a layered architecture for distributed computing and inject spatial data awareness into each layer. For example, the layers in SpatialHadoop are *Language*, *Storage*, *MapReduce* and *Operations*. Considering that SpatialHadoop is a comprehensive extension to the Hadoop ecosystem, it is a scalable and efficient cloud computing framework that allows distributed processing of large-scale spatial datasets using the MapReduce programming model.

In this thesis, we study and enrich *SpatialHadoop* by implementing new Distance-Based Query (DBQ) MapReduce algorithms in the *Operations* layer: εDistance Range Query (εDRQ), kNearest Neighbor Query (kNNQ), kClosest Pairs Query (kCPQ), kNearest Neighbor Join Query (kNNJQ), εDistance Join Query (εDJQ), εDistance Range Join Query (εDRJQ), Reverse kNearest Neighbor Query (RkNNQ), etc. Moreover, we improve the *Storage* layer with a new spatial partitioning technique (Voronoi-Diagram based partitioning), and a new local indexing structure (Quadtree) to optimize the distributed spatial query processing in shared-nothing clusters. This study and the knowledge of SpatialHadoop helps us identify new opportunities to enrich *LocationSpark* (a spatial data processing system built on top of Spark ecosystem) too, with the design and implementation of new distributed Distance-based Join Query (DJQ) algorithms (kCPQ, εDJQ and εDRJQ), extensions, and improvements over them. Additionally, we

propose other enhancements and optimizations for distributed spatial query processing that leverage both data and algorithmic properties. Furthermore, we compare these DSDMSs by evaluating the performance of several distributed DJQ algorithms under different settings with large spatial real-world datasets from *OpenStreetMap*.

To develop this thesis, we start by reviewing the most relevant DSDMSs (research prototypes), the state-of-the-art spatial partitioning techniques in DSDMSs, and the most representative and common DBQs. Then, we focus our study on the structure and operations of spatial data partitioning methods and indexing structures in SpatialHadoop, by proposing a spatial partitioning technique based on Voronoi-Diagrams and including the Quadtree as a local index in such a DSDMS. Driven by an exhaustive analysis on the spatial query processing in SpatialHadoop, we identify and implement new spatial queries (εDRQ, kCPQ, εDJQ, εDRJQ, kNNJQ, RkNNQ, etc.) with different extensions (e.g., for non-points spatial data types) and improvements (e.g., *repartitioning* methods, *less data* technique, new pruning rules, etc.) in this DSDMS. Next, we analyze the general spatial query processing scheme of LocationSpark to extend it with new distributed DJQ algorithms and improvements. Afterward, we achieve an extensive performance evaluation of such enhancements (distributed spatial query algorithms, extensions, and improvements) in SpatialHadoop and LocationSpark. Finally, we carry out a comparative study between SpatialHadoop and LocationSpark by executing an exhaustive set of experiments of several DJQs to identify which DSDMS is the most appropriate for the distributed query processing on large volumes of spatial data.

Keywords: *Big Spatial Data*, *Distributed Spatial Data Management Systems*, *Distance-Based Queries*, *Distance-Based Join Queries*, *kClosest Pairs Query*, *kNearest Neighbor Join*, *Spatial Indexes*, *Quadtree*, *Spatial partitioning*, *Voronoi-Diagrams*, *Spatial data processing*, *SpatialHadoop*, *MapReduce*, *LocationSpark*, *Resilient Distributed Dataset*, *Spatial query evaluation*, *OpenStreetMap*.

© 2021 García-García, F.J.

Resumen

La *Computación Espacial* (Spatial Computing) engloba ideas, soluciones, herramientas, tecnologías y sistemas que transforman nuestras vidas y la sociedad al crear una nueva comprensión de los espacios, ubicaciones, lugares y propiedades. Desde que se acuñó el término *Big Data* por primera vez en 2005, éste ha desencadenado una revolución mundial en la investigación científica y en los negocios. *Big Spatial Data* (BSD), el Big Data asociado con información espacial, es ahora uno de los campos de investigación más activos en computación espacial, motivado principalmente por el rápido desarrollo de tecnologías inteligentes, de sensores y móviles. El uso actual del término *Big Spatial Data* tiende a referirse al proceso de capturar, almacenar, gestionar, analizar y visualizar grandes cantidades de datos espaciales, sin utilizar herramientas y sistemas tradicionales. El reciente desarrollo de sistemas que manipulen grandes volúmenes de datos espaciales han motivado la aparición de nuevas tecnologías para el procesamiento distribuido de datos espaciales a gran escala en *clústeres shared-nothing* (clústeres en los que cada nodo es independiente y autosuficiente) de computadoras, surgiendo así los *sistemas de gestión de datos espaciales distribuidos* (Distributed Spatial Data Management Systems — DSDMSs). Los sistemas de procesamiento distribuido basados en clústeres se pueden clasificar como sistemas basados en Hadoop o en Spark. Según esta clasificación, dos de los DSDMS más importantes son *SpatialHadoop* (DSDMS basado en disco) y *LocationSpark* (DSDMS basado en memoria). Estos sistemas distribuidos proporcionan varias características, como el particionado de datos espaciales, métodos de indexación y el procesamiento de consultas espaciales. Un aspecto importante de estos DSDMS es que adoptan una arquitectura en capas para la computación distribuida e inyectan la capacidad de manipular datos espaciales en cada una de ellas. Por ejemplo, las capas en SpatialHadoop son *Language* (Lenguaje), *Storage* (Almacenamiento), *MapReduce* y *Operations* (Operaciones). Dado que SpatialHadoop es una extensión integral del ecosistema Hadoop, se trata de un marco de computación en la nube escalable y eficiente que permite el procesamiento distribuido de conjuntos de datos espaciales a gran escala utilizando el modelo de programación MapReduce.

En esta tesis, estudiamos y enriquecemos *SpatialHadoop* mediante la implementación de algoritmos MapReduce para consultas basadas en distancia en la capa *Operations*: εDistance Range Query (εDRQ), kNearest Neighbor Query (kNNQ), kClosest Pairs Query (kCPQ), kNearest Neighbor Join Query (kNNJQ), εDistance Join Query (εDJQ), εDistance Range Join Query (εDRJQ), Reverse kNearest Neighbor Query (RkNNQ), etc. Además, mejoramos la capa *Storage* con una nueva técnica de particionado espacial (particionado basado en diagramas de Voronoi) y una nueva estructura de indexación

local (Quadtree) para optimizar el procesamiento de consultas espaciales distribuidas en clústeres shared-nothing. Este estudio, y el conocimiento adquirido sobre SpatialHadoop, nos ayuda a identificar nuevas oportunidades para, también, enriquecer *LocationSpark* (un sistema de procesamiento de datos espaciales construido sobre el ecosistema Spark), con consultas de *join basados en distancias* (DJQ) mediante el diseño e implementación de nuevos algoritmos distribuidos (kCPQ, εDJQ y εDRJQ), extensiones y mejoras sobre ellos. Además, proponemos otras mejoras y optimizaciones para el procesamiento de consultas espaciales distribuidas que aprovechan tanto los datos como las propiedades algorítmicas. Por último, comparamos estos dos DSDMS evaluando el rendimiento de varios algoritmos DJQ distribuidos según diferentes configuraciones con grandes conjuntos de datos espaciales reales procedentes de *OpenStreetMap*.

Para desarrollar esta tesis, se revisan los DSDMS (prototipos de investigación) más relevantes, el estado del arte de las técnicas de particionado espacial en DSDMS, y las DBQ más representativas y comunes. Luego, se estudian la estructura y operaciones de los métodos de particionado de datos espaciales y estructuras de indexación en SpatialHadoop, proponiendo una técnica de particionado espacial basada en diagramas de Voronoi y la incorporación del Quadtree como índice local en dicho DSDMS. Dirigidos por un estudio exhaustivo sobre el procesamiento de consultas espaciales en SpatialHadoop, identificamos e implementamos nuevas consultas espaciales (εDRQ, kCPQ, εDJQ, εDRJQ, kNNJQ, R kNNQ, etc.) con diferentes extensiones (por ejemplo, para tipos de datos espaciales que no son puntos) y mejoras (por ejemplo, métodos de *reparticionamiento*, la técnica *less data*, nuevas reglas de poda, etc.) en este DSDMS. A continuación, se analiza el esquema general de procesamiento de consultas espaciales de LocationSpark para extenderlo con nuevos algoritmos DJQ distribuidos y diversas mejoras. Posteriormente, se realiza una extensa evaluación del rendimiento de las diferentes propuestas (algoritmos de consulta espacial, extensiones y mejoras) en SpatialHadoop y LocationSpark. Finalmente, también se lleva a cabo un estudio comparativo entre SpatialHadoop y LocationSpark mediante la ejecución de un conjunto exhaustivo de experimentos de varias DBQ para identificar qué DSDMS es el más adecuado para el procesamiento de consultas distribuidas sobre grandes volúmenes de datos espaciales.

Palabras clave: *Big Spatial Data*, *Sistemas de gestión de datos espaciales distribuidos*, *Consultas basadas en distancia*, *Consultas de join basados en distancia*, *Consulta de los k pares más cercanos*, *Join de los k vecinos más próximos*, *Índices espaciales*, *Quadtree*, *Particionado espacial*, *Diagramas de Voronoi*, *Procesamiento de datos espaciales*, *SpatialHadoop*, *MapReduce*, *LocationSpark*, *Conjunto de datos distribuidos y flexibles*, *Evaluación de consultas espaciales*, *OpenStreetMap*.

© 2021 García-García, F.J.

Chapter 1

Introduction

© 2021 García-García, F.J.

Chapter 1

Introduction

Contents

Spatial Computing covers the ideas, solutions, tools, technologies, and systems that transform our lives and society by creating a new understanding of spaces, locations, places, and properties. Moreover, this computing paradigm helps us know, communicate, and visualize our relation to places in a space of interest; and how to navigate through those places [Evans et al., 2014].

Spatial data are discrete representations of continuous phenomena. Discretization of continuous space is required by the nature of digital representation. There are three basic models to represent spatial data: raster (images), vector (points, lines, regions), and network (spatial networks). *Spatial data types* provide a fundamental abstraction for modeling the geometric (or spatial) structure of objects in space as well as their relationships, properties, and operations [Schneider, 2009]. Examples of 2d spatial data types are points, lines, regions, spatial networks, etc., and examples of 3d spatial data types are surfaces, volumes, etc. *Spatial Big Data* (SBD) are defined as simply instances of these *spatial data types* that exhibit at least one of the 3 V's: *volume*, *velocity*, and *variety* [Evans et al., 2014]. Spatial data frequently demonstrate at least one of these core features, given the variety of data types in spatial computing, such as points, lines, regions, etc. Moreover, spatial analytics have shown to be more computationally expensive than the non-spatial ones as they need to account for spatial autocorrelation and non-stationarity, among other properties. Examples of SBD can be: (1) temporally tagged road maps that provide traffic speed values every minute for all roads in a city, (2) global positioning system (GPS) trajectory data from smartphones, (3) engine measurements of fuel consumption and gas emissions, (4) geotagged tweets issued from Twitter, etc. [Evans et al., 2014]. Other authors refer to the term spatial big data as *Big Spatial Data* (BSD) [Eldawy and Mokbel, 2017]. For instance, according to [Alam et al., 2021], a huge volume of geo-referenced data (from sensor devices, GPS-enabled devices, location-based services, spatial applications, etc.), generated every day, are often called big spatial data (see Figure 1.1). In this thesis, we will use the term *Big Spatial Data* to describe the process of capturing, storing, managing, analyzing, and visualizing huge amounts of spatial data, not using traditional tools and systems [Alam et al., 2021].

Distributed Computing is a reference to computation on a platform with multiple nodes, each with its own hardware (computers) and software (operating systems). The nodes in a distributed computing platform could be in close proximity connected via a local area network (LAN) or dispersed over a large geographic area connected via a wide area network (WAN) [Sharker and Karimi, 2014]. One of the main features of distributed computing is *scalability*, which means that the platform allows participation of a different number of computing nodes as the demand changes (i.e., it can scale down or up). Increasing the number of nodes in a distributed computing platform is one possible approach for handling *large-scale problems*. *Cluster Computing* refers to commonly distributed computing platforms, where nodes are connected through dedicated network systems and protocols, all of them running under one centralized operating system.

In the age of smart cities and mobile environments, the increase of the volume of available spatial data (e.g., location, routing, etc.) is huge all over the world. Recent

© 2021 García-García, F.J.

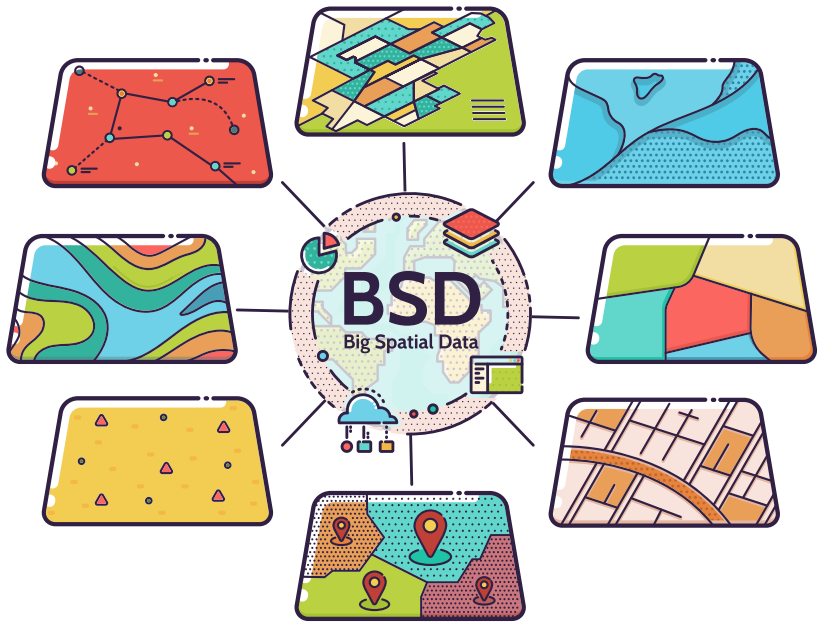


Figure 1.1: Multiple sources and layers that form part of the *Big Spatial Data*.

developments of big spatial data systems have motivated the emergence of novel technologies for processing large-scale spatial data on shared-nothing clusters in a distributed environment. A *shared-nothing architecture* of machines has proved to be a popular design choice for the implementation and deployment of big data platforms. Clusters of machines are often favored over expensive infrastructure because of their low operational costs. In such clusters, data are partitioned and distributed over several machines, usually leveraging the functionality of a distributed file system [Doulkeridis and Nørvåg, 2014]. The processing component is often deployed on the same cluster of machines in order to leverage data locality; the incentive is to process data on the machine where it is already stored and avoid expensive network transfers. For this reason, shared-nothing clusters are generally preferable to other forms of clustering. Furthermore, the scalability of shared-nothing clusters makes it optimal for intensive analytical and query processing.

Recent big spatial data developments have motivated the emergence of novel technologies for distributed processing of large-scale spatial data in shared-nothing clusters of computers, leading to *Distributed Spatial Data Management Systems* (DSDMSs) or Big Spatial Data Analytics Systems (BSDASs) [Pandey et al., 2018]. These DSDMSs (research prototypes) can be classified in disk-based [Li et al., 2014], which are characterized by being Hadoop-based systems, and in-memory-based [Zhang et al., 2015], generally based on Spark. Apache Hadoop[1] is a reliable, scalable, and efficient cloud

[1]Available at `https://hadoop.apache.org/`

computing framework allowing distributed processing of large datasets using the MapReduce programming model. However, it is a kind of disk-based computing framework, which writes all intermediate data to disk between *map* and *reduce* tasks. MapReduce [Dean and Ghemawat, 2004] is a framework for processing and managing large-scale datasets in a distributed cluster. It was introduced with the goal of providing a simple yet powerful parallel and distributed computing paradigm, offering good scalability and fault tolerance mechanisms. Apache Spark[2] is a fast, reliable and distributed in-memory large-scale data processing framework. It takes advantage of the Resilient Distributed Dataset (RDD), which allows us to transparently store data in memory and persisting it to disk only if it is needed [Zaharia et al., 2012]. Hence, it can reduce a huge number of disk writes and reads to outperform the Hadoop platform. Since Spark maintains the status of assigned resources until a job is completed, it reduces time consumption in resource preparation and collection.

Both Hadoop and Spark have weaknesses related to efficiency when applied to spatial data. One main shortcoming is the lack of any indexing mechanism that would allow selective access to specific regions of spatial data, which would in turn yield more efficient query processing algorithms. A solution to this problem is an extension of Hadoop, called *SpatialHadoop* [Eldawy and Mokbel, 2015], which is a framework that supports spatial indexing on top of Hadoop, i.e., it adopts a two-level index structure (global and local) to organize the stored spatial data. In this distributed framework, the spatial data are partitioned and scattered to the nodes of the cluster so that objects with spatial proximity are in the same partition. Besides, these generated partitions are indexed, allowing efficient query algorithms that access only a part of the data while still returning the correct result. In Spark, there are similar solutions like *LocationSpark* [Tang et al., 2016, Tang et al., 2020], which is a spatial data processing system built on top of Spark that employs various spatial indexes for in-memory data. It provides a wide range of spatial features and supports a rich set of spatial queries. Moreover, it samples the input dataset and partitions data accordingly by using several spatial partitioning schemes. It also provides flexibility for local indices, where the data are locally indexed within a concrete partition.

DSDMSs are cluster-based systems that support spatial data management, query processing, and analytics over distributed data using a cluster of commodity machines. Several characteristics are supported in these systems, like spatial data partitioning, indexing schemes, and spatial queries. An important aspect of these DSDMSs is to adopt a layered architecture for distributed computing and inject spatial data awareness into each layer. For example, *SpatialHadoop* [Eldawy and Mokbel, 2015] is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the *language*, *storage*, *MapReduce*, and *operations* 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 as Grid, R-tree, Quadtree, etc., 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. At the *Operations* layer,

[2]Available at `https://spark.apache.org/`

© 2021 García-García, F.J.

SpatialHadoop is also equipped with several spatial operations, including range query, nearest neighbor query, and spatial join.

In this thesis, we study and enrich two of the most leading DSDMSs, *SpatialHadoop* [Eldawy and Mokbel, 2015] (disk-based DSDMS) and *LocationSpark* [Tang et al., 2016] (in-memory-based DSDMS), by implementing new distance-based queries (εDistance Range Query — εDRQ, kNearest Neighbor Query — kNNQ, kClosest Pairs Query — kCPQ, kNearest Neighbor Join Query — kNNJQ, εDistance Join Query — εDJQ, εDistance Range Join Query — εDRJQ and Reverse kNearest Neighbor Query — RkNNQ), new spatial partitioning techniques (Voronoi-Diagram based partitioning) and new local indexing structures (Quadtree) for distributed spatial query processing in shared-nothing clusters. Furthermore, we propose and implement additional improvements and optimizations for distributed spatial query processing that leverage both data and algorithmic properties. Besides, we compare both DSDMSs by evaluating the performance of several new distributed distance-based query algorithms under various settings with large spatial real-world datasets. We start our study by reviewing the most relevant DSDMSs, the state-of-the-art spatial data partitioning techniques in DSDMSs, and the most representative distance-based queries (DBQ). Then, we focus our study on the structure and operations of spatial partitioning techniques and indexing methods in SpatialHadoop, by proposing a spatial data partitioning technique based on Voronoi-Diagrams and including the Quadtree as a local index in SpatialHadoop. Driven by an exhaustive study on the spatial query processing in SpatialHadoop, we identify and implement new spatial queries (kCPQ, εDJQ, kNNJQ, RkNNQ, etc.) in SpatialHadoop and, different extensions and improvements of these spatial query algorithms are also incorporated. Next, the general spatial query processing scheme of LocationSpark is studied and, new DBQs, extensions, and improvements are also implemented in LocationSpark. Finally, an extensive performance evaluation of the different enhancements (spatial query algorithms, extensions, and improvements) in SpatialHadoop is achieved, and a comparative study between these two DSDMSs (SpatialHadoop and LocationSpark) is also carried out.

1.1 Research Objectives

We thoroughly research two of the most leading DSDMSs: *SpatialHadoop* and *LocationSpark*, and we enhance them by including new spatial data partitioning techniques, new local spatial indexing methods, and new DBQ algorithms for processing large real-world spatial datasets. In particular, we aim at making existing DSDMSs more valuable and complete by implementing Voronoi-Diagram based partitioning technique in SpatialHadoop, Quadtree as a local index in SpatialHadoop and new distance-based queries (εDRQ, kNNQ, kCPQ, kNNJQ, RkNNQ, εDJQ, εDRJQ, etc.) in SpatialHadoop and LocationSpark. Our main goal is to enrich current DSDMSs, concerning the distributed storage for spatial query processing and the number of supported spatial queries (see Figure 1.2). To this end, we analyze existing DSDMSs, identify lacks and limitations, and implement specific data partitioning techniques and local indexes (Spatial Storage layer) and spatial queries (Spatial Operations layer), which can be easily integrated into

these popular distributed platforms.

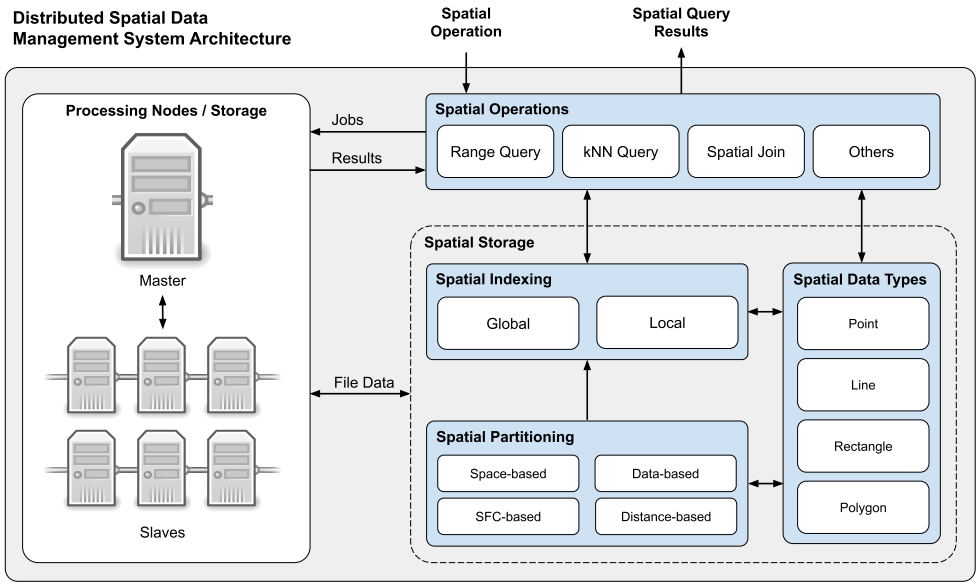


Figure 1.2: Overview of a Distributed Spatial Data Management System architecture (Spatial Operations and Spatial Storage layers).

To achieve this main goal, several specific objectives must be addressed. These are described below:

- **Analysis of existing DSDMSs**. In the first stage of this thesis, a detailed analysis of the state-of-the-art of different DSDMSs (Hadoop-based and Spark-based) is carried out. Next, we choose the options that can be considered more mature and robust for further study in terms of physical and logical architectures, spatial data types, partitioning techniques, indexing methods, and spatial queries supported.

- **Generation of spatial datasets for the experimental evaluation**. We study and identify the best spatial data sources to test the proposed enhancements. The information identified as relevant must be converted into spatial data (according to the available spatial data types, e.g., Points, Lines, Rectangles, Polygons, etc.) that can be stored to be processed by the chosen DSDMS. These datasets can be of two types: (1) synthetic data (following distributions like uniform or clustered) that allow us to generate baseline scenarios with configurable parameters of interest, and (2) real-world data obtained from open-data sources such as

© 2021 García-García, F.J.

OpenStreetMap[3] to check the performance of new DBQs in real-world contexts.

- **Implementation of new spatial partitioning techniques**. Spatial data partitioning is a powerful mechanism for improving the efficiency of DSDMSs since it improves the overall manageability of large datasets, and it also speeds up spatial query performance. By partitioning large datasets into smaller units, it enables the processing of a spatial query in parallel and reduces the I/O activity by only scanning a few partitions that contain data relevant to the query constraints. The use of the most appropriate spatial partitioning technique will improve the efficiency of the proposed spatial query algorithms in a particular DSDMS. In this thesis, we have implemented in SpatialHadoop a new data partitioning technique based on Voronoi-Diagrams.

- **Implementation of new spatial indexing methods**. Spatial indexing is a robust mechanism for enabling fast access to spatial data and accelerate spatial query processing. In a particular DSDMS, the spatial storage level is adapted to include spatial indexes and use them to support spatial queries efficiently. To tackle the building of spatial indexes, a two-layers (global and local) indexing approach is commonly used. The implementation of a two-level index structure in a DSDMS could lead to efficient distributed algorithms for processing spatial queries over large-scale real-world spatial datasets. In this thesis, the Quadtree is included, as a local index, in SpatialHadoop to speed up the spatial query processing.

- **Implementation of new spatial queries**. In this objective, popular spatial queries, which are not present in the selected DSDMSs, will be implemented. The traditional spatial query algorithms will be optimally adapted to a distributed programming model (MapReduce or Resilient Distributed Datasets — RDD), taking into account the advantages and characteristics that the distributed environment provides us. DBQs have received considerable attention from the database community due to their importance in numerous applications, such as spatial databases and geographic information systems (GIS), data mining, multimedia databases, etc. In this thesis, we have implemented the most representative DBQs like εDRQ, kNNQ, kCPQ, kNNJQ, RkNNQ, εDJQ, εDRJQ, among others in the selected DSDMSs. An example of these DBQs could be the kCPQ in a transportation monitoring and moving objects scenario, considering two spatial datasets: locations of users of a taxi app and positions of free taxis. kCPQ could *find the 10 pairs of app users and taxis with the shortest distances between them*, to be able to offer these users fast service at a reduced price (as a promotion strategy), or for analysis by the taxi service.

- **Comparison of DSDMSs**. The evaluation of the experimental results obtained after executing spatial queries is key to identify the DSDMSs that are the most suitable for the distributed processing of large volumes of spatial data. Therefore, the creation of a reference framework that allows us to compare DSDMSs, which can be so heterogeneous, is crucial to choose the DSDMS that best adapts to the

[3]Available at `https://www.openstreetmap.org/`

characteristics of the spatial data and the spatial query that we are examining. Furthermore, this objective will try to define a series of performance metrics and experiments that, through different dimensions, allow us to define what are the advantages and disadvantages of using one DSDMS or another. In this thesis, we compare SpatialHadoop and LocationSpark using several performance measures with respect to the most significant distance-based join queries (DJQs).

1.2 Research Methodology

In this section, we provide an overview of the methodology used in this thesis. We give a brief description of the general principles we have followed for the design, implementation, and optimization of the main contributions of this thesis. We also discuss several challenges we faced and how we decided to overcome each of them.

1.2.1 General Approach

Among the wide variety of data-intensive applications and platforms, we focus mainly on spatial query algorithms and DSDMSs for two reasons. First, spatial query algorithms are crucial in modern DSDMSs since the analysis of spatial data is a core issue for companies that use geographic location to support strategic decisions and to enhance the user experience. These companies have a massive advantage over their competitors and are able to react quickly to business conditions changes. Second, DSDMSs present interesting research challenges and open issues. They provide specialized functionalities (complex and hard to implement) to manage and process huge volumes of spatial data using parallel and distributed data processing frameworks (e.g., Hadoop and Spark).

As in [Hassani, 2017], we consider a *research methodology* as a scientific approach that investigates, compares, contrasts, and explains the different ways that research could be conducted alongside several methods that could be used in these processes. That is, a methodology discusses the alternative approaches and methods to tackle the research problem. It discusses the advantages/disadvantages, properness/improperness, feasibility, practicality, ethical issues, and such parameters for the approaches to do the research. A *research method* can be considered as an approach, procedure, and guidelines that are used in conducting research. A method might require different tools, instruments, equipment, etc. As a result, research in computing might be of *theoretical* or *experimental* nature or a combination of them; it appreciates different paradigmatic views and utilizes best-suited tools and approaches from both quantitative and qualitative methods.

Experimental methodologies are broadly used in Computer Science to evaluate new solutions to problems. Experimental evaluation is often divided into two phases. First, an *exploratory phase* where the researcher takes measurements that will help identify what are the questions that should be asked about the system under evaluation. Second, an *evaluation phase* will attempt to answer these questions. A well-designed experiment will start with a list of the questions that the experiment is expected to answer.

© 2021 García-García, F.J.

In this thesis, we adopt an *experimental approach*, common to computer systems research, instead of using a theoretical methodology. A high-level overview of our research methodology is shown in Figure 1.3 (the numbers represent chronological order).

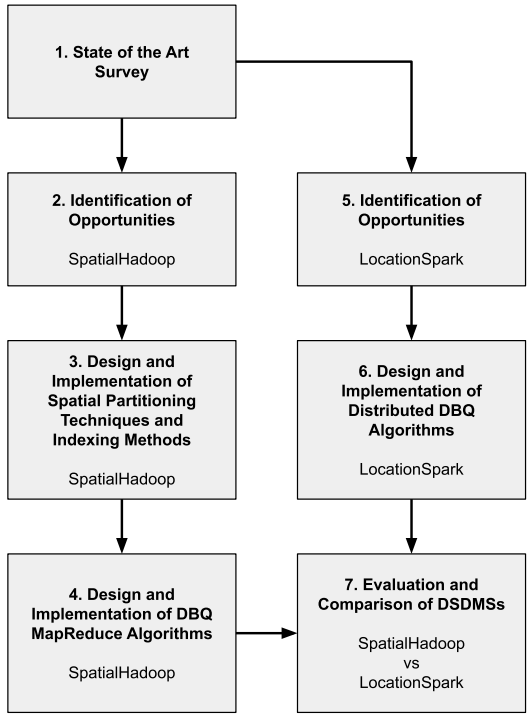


Figure 1.3: High-level overview of the followed research methodology.

First, we identify functionality gaps and limitations in DSDMSs, and then we design and implement new algorithms to overcome these shortcomings. We start by conducting a literature study of recent research results on existing DSDMSs, spatial data partitioning techniques, and the most representative DBQs (Chapter 2). The result of this thorough study provides us an overview of the state-of-the-art in the research field and reveals open issues. For instance, which are the best Hadoop-based and Spark-based DSDMSs to choose for spatial query processing, and which are the most popular DBQs to be included in these distributed platforms.

Next, we study the SpatialHadoop system architecture and the implementations in each layer. In particular, the spatial partitioning techniques and the indexing methods included in the Storage layer are examined. This study helps us identify the new opportunities for enhancing this layer of SpatialHadoop, with the design and implementation of a spatial partitioning technique based on Voronoi-Diagrams and the use of the

Quadtree as a local index. To prove the performance and efficiency of both proposals, a set of experiments has been carefully designed, implemented, and executed using real-world datasets (Chapter 3).

In order to continue enriching SpatialHadoop in the Operations layer, we center our research around the design and implementation of new distributed algorithms of distance-based queries (εDRQ, kNNQ, kCPQ, kNNJQ, RkNNQ, εDJQ, etc.). Furthermore, we describe and discuss the implementation of different extensions (for non-points spatial data types) and improvements. To show the performance and efficiency of all DBQ MapReduce algorithms (extensions and improvements), an exhaustive experimental evaluation has been run in SpatialHadoop (Chapter 4).

For the next experimental target, we study the LocationSpark system architecture (it is built as a library on top of Spark) and the implementation of the layers (Memory Management, Spatial Index, and Spatial Operators) that most affect spatial query processing. This study and the knowledge of SpatialHadoop helps us identify the new opportunities for enriching LocationSpark, with the design and implementation of several new distributed DBQ (εDRQ, kCPQ, εDJQ, and εDRJQ) algorithms, extensions, and improvements over them. To report the performance and efficiency of all distributed distance-based query algorithms (extensions and improvements), a comprehensive experimental evaluation has been executed in LocationSpark, and a comparison with SpatialHadoop has been also carried out (Chapter 5). The main performance measures that we have taken into account in our experiments are: *total execution time* (i.e., total running time or total response time), *shuffled data* (for read and write operations) and *peak execution memory*.

1.2.2 Implementations

We have used open-source, widely-used, and mature systems and libraries to implement and evaluate the distributed spatial query algorithms and techniques proposed in this thesis. Essentially, we have used SpatialHadoop[4] for the implementation of spatial partitioning techniques and indexing methods discussed in Chapter 3 and the new DBQ MapReduce algorithms described in Chapter 4. For the development of our Voronoi-Diagram based partitioning method, the ELKI library [Schubert and Zimek, 2019] has been used, which has provided various clustering algorithms (Sort-Means, k-means++, OPTICSxi, etc.) for the pivot selection. On the other hand, LocationSpark[5] has been used to develop distributed DBQ algorithms in Spark-based environments, whose performance and comparison with SpatialHadoop appear in Chapter 5. The implementations of our algorithms and techniques are free to use, open-source, and documented. They are available at the next `github` repository: `https://github.com/acgtic211`

1.2.3 Experimental Evaluation

For our experiments, we always choose the latest stable released version of the considered DSDMSs (SpatialHadoop and LocationSpark). We have used real-world datasets from

[4] Available at `https://github.com/aseldawy/spatialhadoop2`
[5] Available at `https://github.com/merlintang/SpatialSpark`

© 2021 García-García, F.J.

OpenStreetMap[6], and when using synthetic datasets, we have described in detail each data distribution (uniform or clustered) and parameters used in the creation process. Moreover, we have shared our detailed setup configuration in each of our works in order to facilitate reproducibility. All experiments were conducted on a cluster of virtual machines in an OpenStack environment, where each computing node has 4 vCPUs with 8 GBs of main memory running Linux operating systems. Therefore, we are able to create a clean and isolated environment with only the necessary tools installed. We choose representative performance metrics for our evaluation (total execution time or total response time, shuffled read/write cost, and peak execution memory) by using the ones that appear often in related research to show the efficiency and scalability of our distributed DBQ algorithms and techniques.

1.2.4 Improvements Overview

In this section, we present a brief overview of the most relevant improvements that have been applied to the first original solution of the problems. Most of the available improvements have been inspired by our previous knowledge in the context of distributed computing and spatial query processing. First of all, to improve the Voronoi-Diagram based partitioning technique in SpatialHadoop, we have used three sampling methods (random, k-means++, and DENDIS) to sample large spatial datasets in the sampling phase, and three clustering algorithms (random, k-means++ and OPTICS) for the pivot selection in the space subdivision phase. The experimental results showed that the use of k-means++ in both phases (sampling and space subdivision) is the best choice.

The main improvement for the computation of the k closest pairs, both in SpatialHadoop and LocationSpark, has been related to the computation of β (i.e., the upper bound of the distance value of the k-th closest pair of the joined datasets). The computation of β can be carried out (1) by sampling globally both large datasets and executing a kCPQ plane-sweep algorithm over the two samples, or (2) by appropriately selecting a specific pair of partitions to which the two large datasets are partitioned and either (2.a) by sampling locally the partitions of this pair and executing a kCPQ plane-sweep algorithm over the two samples, or (2.b) by applying an approximate variation (α-allowance approximate technique) of a kCPQ plane-sweep algorithm over the selected pair of partitions. After an exhaustive experimental study, the fastest and the most accurate method to compute β is by *local sampling* (2.a). In Figure 1.4, we can see the general scheme for kCPQ processing in SpatialHadoop consists of four steps: Preprocessing, Pruning, Local Spatial Query Processing, and Global Processing. The aforementioned improvement in the β computation is carried out in the *Pruning* step, as we can observe in Figure 1.4.

For the computation of all k nearest neighbors (kNearest Neighbor Join query) both in SpatialHadoop and LocationSpark, several improvements can be applied. The first improvement is related to solve the problem of skewed data. For this purpose, we can apply the *repartitioning* technique by splitting again the densest partitions using once more any spatial partitioning method. From our experimental results, the best method to make the algorithms faster is to use the Quadtree-based repartitioning technique,

[6]Available at `http://spatialhadoop.cs.umn.edu/datasets.html`

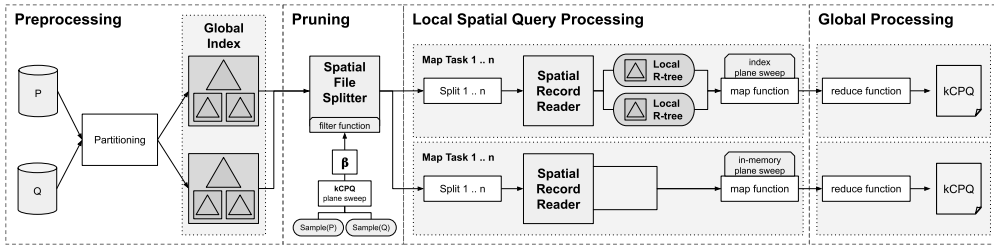


Figure 1.4: General scheme for kCPQ processing in SpatialHadoop.

and it also reduces the shuffled data. Another improvement for this DJQ is to use *less data technique* in order to reduce the size of the shuffled data between computing nodes (shuffled read/write costs) and the size of the output data of the *kNNJ on Overlapping Partitions* phase in the distributed kNNJQ algorithm.

One of the most important improvements in DBQ processing is the use of effective *pruning rules* to avoid unnecessary distance computations. Several efficient pruning rules have been presented for both kCPQ and kNNJQ in this thesis. For kCPQ, we have proposed a new pruning rule called *Pair of Partitions Pruning* to apply in the *filter function* for pruning combinations of partitions from the two datasets in order to reduce the number of *map* tasks that the distributed kCPQ algorithm needs to perform to get the final query result. For kNNJQ, several pruning rules have been used to reduce the number of distance computations and, they are based on concepts like *core-distance* and *support-distance* of a Voronoi-cell.

1.2.5 Challenges

In this section, we highlight the challenges found in the development of this thesis. These are described in more detail in their corresponding chapters.

When implementing DBQs in SpatialHadoop, a series of challenges appeared inherent to the distributed platform in question. On the one hand, the size of *shuffled data* must be minimized to obtain efficient and scalable algorithms. This performance metric refers to the amount of information that travels between computing nodes in the cluster, especially between the *map* and *reduce* phase tasks. Therefore, if the shuffled data increase, the execution time of the algorithms will also grow by increasing both transmitted and processed data. To tackle this challenge, we must wisely use the filtering capabilities provided by SpatialHadoop along with heuristics and indexes that allow us to quickly discard data that are not part of the final solution. On the other hand, since SpatialHadoop is a disk-based DSDMS, both the size of the *intermediate files* and the size of the *result files* must be taken into account, especially in join queries. For example, when performing a kNNJ query, the output size is equal to the size of the first set times k. Therefore, sophisticated techniques must be applied to reduce intermediate file sizes, and also the disk resources of the distributed cluster could be increased if needed.

© 2021 García-García, F.J.

The design of efficient distributed DBQ algorithms that perform optimally on Spark-based DSDMSs, like LocationSpark, is a crucial target in the context of this thesis. For this reason, the reduction of the size of the *shuffled data* is also essential for the reduction of the total execution time of the distributed algorithms. In this case, the *shuffled data* represents the amount of information that is redistributed across partitions that may or may not cause moving data across processes, executors, or nodes. To this end, we must reduce the use of Spark transformation operations that produce *wide dependencies*, such as *groupByKey*, which also increases the *stages* of the algorithm. Therefore, we should encourage the use of operations that generate *narrow dependencies*, such as *zipPartitions*, *aggregate*, or *union*, which do not require data redistribution. Furthermore, the configuration and tuning of nodes in Spark-based systems are quite complex. Access to guides, provided by the community, such as *Tuning Spark*[7], *Set up your Apache Spark cluster*[8] and research works like [Gounaris and Torres, 2018], can help to generate a correct configuration of the distributed environment for conducting experiments in the best possible way.

Finally, there are some challenges and difficulties that are common to both DSDMSs. First, it was necessary to set up the hardware and software infrastructure on which to run the experiments. For this aim, *OpenStack* has been used for the creation of virtual machines, and *Apache Ambari* for the creation and administration of the Hadoop and Spark clusters. These tools have allowed us to set up the experimentation environments in a simple way, with efficient administration and flexibility of being able to modify the characteristics of the cluster, depending on the type of experiment to be executed. Another problem was the great diversity in the characteristics of the execution environments that appear in the studied research works. Indeed, some of these configurations are difficult to access, as they have been made in high-capacity and costly payment clusters. In this way, it is not possible to directly use the results found in these studies, and therefore, we had to repeat the experiments in our own (local) cluster. To conclude, and perhaps the most important challenge in this thesis is the difficulty in *debugging* the distributed algorithms due to a large number of nodes (machines) and the use of large spatial datasets. Depending on the algorithm to be debugged, the execution can take a long time, and finding why an error has occurred is very laborious. To solve this problem, it has been necessary to rely on local execution with reduced datasets and the use of writing in logs that have allowed us to correct and advance in the development of the distributed DBQ algorithms.

1.3 Thesis Contributions

This section summarizes the most important contributions of this thesis from different points of view. First, we synthesize the main contributions of this research work by a brief description of each of them. Next, we list the `github` repositories where the source code of this dissertation is open-access available. Afterward, all publications (conferences and journals) that support the research of this thesis are detailed, highlighting the qual-

[7]Available at `https://spark.apache.org/docs/latest/tuning.html`
[8]Available at `http://sedona.apache.org/download/cluster/`

ity parameters of such publications. Finally, other contributions (publications closely related) that are not included in this thesis but the author has actively participated during the course of the author's Ph.D. studies are also outlined.

1.3.1 Summary of Contributions

The main contributions of this thesis are the following:

- A survey of the state-of-the-art in Hadoop-based and Spark-based DSDMSs. We explore the most representative DSDMSs (research prototypes) that appear in the literature and compare them based on features like spatial index and spatial queries they support. Moreover, we also review the most common spatial partitioning techniques and classify them by how they use data or space properties. Finally, a thorough overview of the most studied and known DBQs in the context of spatial databases. In particular, we emphasize the review on distance-based join queries (kCPQ, kNNJQ, εDJQ, εDRJQ and other distance-based joins).

- We have proposed a new spatial data partitioning technique based on Voronoi-Diagrams in SpatialHadoop. This data partitioning scheme is especially suitable for DBQs as kNNQ and kNNJQ. An extensive experimental evaluation of the spatial partitioning methods that are implemented in SpatialHadoop and a comparison with the Voronoi-Diagram based technique for kCPQ and kNNJQ is also accomplished [García-García et al., 2018a, García-García et al., 2020b].

- We have studied and included the Quadtree as a local index in SpatialHadoop since this spatial access method is widely used in commercial spatial database systems. A comparative study between R-tree and Quadtree as local indexes for kNNQ and kCPQ has been carried out and has demonstrated the excellent performance of the Quadtree for these top-k queries in SpatialHadoop [García-García et al., 2020a].

- We have proposed new MapReduce algorithms in SpatialHadoop to perform efficient distance range queries (εDistance Range query and εDistance Range Join query) on large-scale spatial datasets. We have also evaluated the performance of the proposed algorithms in distinct scenarios with large synthetic and real-world datasets [García-García et al., 2016a].

- One of the main and original contributions of this thesis has been the design and implementation of new MapReduce algorithms to perform efficiently kCPQ and εDJQ in SpatialHadoop. For this aim, we have utilized plane-sweep-based kCPQ algorithms and improved them to compute an upper bound of the distance of the k-th closest pair and make the original version of kCPQ MapReduce algorithm much more efficient and faster. We have evaluated the performance of the proposed algorithms in several situations with large real-world as well as synthetic datasets. The experimental results have demonstrated the efficiency and scalability of our proposed MapReduce algorithms [García-García et al., 2016b, García-García et al., 2018b].

© 2021 García-García, F.J.

– Another distinguished contribution of this thesis has been to compare two of the most leading DSDMSs, SpatialHadoop (Hadoop-based) and LocationSpark (Spark-based), by evaluating the performance of several existing and newly proposed distributed distance-based join query (kCPQ, kNNJQ, εDJQ, εDRJQ, etc.) algorithms under various settings with large real-world datasets. We have also extended the distributed DBQ algorithms for managing spatial objects more complex than points, like polygons or line-segments. Moreover, improved kNNJQ and εDRJQ MapReduce algorithms have been also implemented by using *repartitioning techniques* in dense areas (skewed data handling). Several interesting conclusions have been obtained after an exhaustive experimental study. For instance, while SpatialHadoop is a robust and efficient system when large spatial datasets are joined (since it is built on top of the mature Hadoop platform), LocationSpark is the clear winner in total execution time when small-medium spatial datasets are combined (due to in-memory processing provided by Spark) [García-García et al., 2017a, García-García et al., 2020c].

– The Reverse kNearest Neighbor (RkNN) query has been recently studied very thoroughly since it is of particular interest in a wide range of applications, such as decision support systems, resource allocation, profile-based marketing, location-based services, etc. In this thesis, we have proposed the design and implementation of new RkNNQ MapReduce algorithms, MRSFT and MRSLICE, in SpatialHadoop. We have also evaluated and compared their performances with large real-world datasets, showing interesting conclusions and demonstrating the efficiency and scalability of MRSLICE in comparison with the other proposal [García-García et al., 2017b, García-García et al., 2019].

DSDMS	**Spatial Partitioning**	**Spatial Indexing**	**Spatial Query**
SpatialHadoop	Voronoi-Diagram based partitioning	Quadtree	εDRQ, kNNQ, **RkNNQ**, **kCPQ**, **kNNJQ**, εDJQ, εDRJQ - points and non-points (line-segments and polygons) - computing of β, repartitioning, less data, new pruning rules
LocationSpark	—	—	**kCPQ**, εDJQ, εDRJQ
SpatialHadoop vs. **LocationSpark**	*Quadtree*	—	kCPQ, kNNJQ, εDJQ, εDRJQ

Table 1.1: The most important contributions of this thesis.

1.3.2 Software

The following software was developed in the course of this thesis:

– A distributed spatial partitioning algorithm based on Voronoi-Diagrams has been implemented in SpatialHadoop. We describe the spatial partitioning method in detail in Chapter 3. The technique uses several clustering algorithms (Sort-Means,

OPTICSxi, etc.) from the ELKI library [Schubert and Zimek, 2019] for the pivot selection step. The code is available at `https://github.com/acgtic211/spatialhadoop2/tree/voronoi` under the Apache 2.0 license.

– A new local index based on Quadtree has been implemented in SpatialHadoop. It allows us to accelerate the local computation phase of several spatial query algorithms, and it is fully described in Chapter 3. The code is available at `https://github.com/acgtic211/spatialhadoop2/tree/quadtree` under the Apache 2.0 license.

– Several DBQs have been implemented in SpatialHadoop: εDRQ, kCPQ, εDJQ, kNNJQ, εDRJQ and RkNNQ. We describe their implementations in detail in Chapter 4, together with various extensions and improvements of the initial versions of the algorithms. The code is available at `https://github.com/acgtic211/spatialhadoop2/` under the Apache 2.0 license.

– Several DBQs have been also implemented in LocationSpark: kCPQ, εDJQ, and RkNNQ. The implementation details of each of these distributed DBQ algorithms are found throughout Chapter 5. The code is available at `https://github.com/acgtic211/LocationSpark/tree/DJQ`.

1.3.3 Publications

Results presented in this thesis have been published as papers in international and national conference proceedings (2 *ADBIS*, 3 *MEDI*, 1 *JISBD*, 1 *PCI*) and in prestigious international journals (*GeoInformatica*, *Information Sciences* and *Future Generation Computer Systems*) as follows.

– [García-García et al., 2016b] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2016). **Enhancing spatialhadoop with closest pair queries**. In *ADBIS Conference*, pages 212–225. *ADBIS* (Advances in Databases and Information Systems) is a prestigious European conference on the fields of databases and information systems. In 2016, the conference attracted 85 paper submissions and, after a rigorous review process, only 21 papers were accepted (25%) to be included in LNCS proceedings. **CORE B**.

– [García-García et al., 2016a] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2016). **Distance range queries in spatialhadoop**. In *JISBD Conference*, pages 1–14. *JISBD* (Jornadas de Ingeniería del Software y Bases de Datos) is the most important Spanish conference in databases and software engineering. In 2016, this paper was published in the *Data Management* track.

– [García-García et al., 2017b] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2017). **Rknn query processing in distributed spatial infrastructures: A performance study**. In *MEDI Conference*, pages 200–207. *MEDI* (Model and Data Engineering) is an emerging international conference on models and data engineering, the development of advanced technologies related to

© 2021 García-García, F.J.

models and data, as well as their advanced applications. *MEDI* 2017 received 69 paper submissions, and only 27 (20 full papers and 7 short papers) were accepted (39%) to be included in LNCS proceedings.

– [García-García et al., 2017a] García-García, F., Corral, A., Iribarne, L., Mavrommatis, G., and Vassilakopoulos, M. (2017). **A comparison of distributed spatial data management systems for processing distance join queries**. In *ADBIS Conference*, pages 214–228. *ADBIS* 2017 conference attracted 107 paper submissions and only 26 papers were accepted (24%) to be included in LNCS proceedings. **CORE B**.

– [García-García et al., 2018a] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2018). **Voronoi-diagram based partitioning for distance join query processing in spatialhadoop**. In *MEDI Conference*, pages 251–267. *MEDI* 2018 conference received 86 paper submissions and only 27 (23 full papers and 4 short papers) were accepted (31%) to be included in LNCS proceedings.

– [García-García et al., 2018b] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2018). **Efficient large-scale distance-based join queries in spatialhadoop**. *GeoInformatica*, 22(2): 171–209. *GeoInformatica* (ISSN: 1384:6175, Springer) is a prestigious journal on advances of computer science for geographic information systems (GISs), covering research fields like spatial modeling and databases; parallelism, distribution and communication through GIS; spatio-temporal reasoning; etc. In 2018, *GeoInformatica* journal had a JCR Impact Factor of **1.317**; Computer Science, Information Systems 118/155 **Q4**.

– [García-García et al., 2019] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2019). **MRSLICE: efficient rknn query processing in spatialhadoop**. In *MEDI Conference*, pages 235–250. *MEDI* 2019 conference received 41 paper submissions, and only 21 (11 full papers, 7 short papers, 2 application papers, and 1 vision paper) were accepted (51%) to be included in LNCS proceedings.

– [García-García et al., 2020a] García-García, F., Corral, A., and Iribarne, L. (2020). **Including the quadtree index in spatialhadoop**. In *Pan-hellenic Conference on Informatics*, pages 376–379. PCI (Pan-Hellenic Conference on Informatics) is the most important Greek conference on computer science and emerging fields of informatics. *PCI* 2020 conference received 171 paper submissions, and only 93 (80 full papers and 13 short papers) were accepted (54%) to be included in PCI 2020 proceedings volume. This paper was accepted in the special session on Parallel and/or Distributed Databases (PDDB).

– [García-García et al., 2020c] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2020). **Efficient distance join query processing in distributed spatial data management systems**. *Information Sciences*, 512:985–1008. *Information Sciences* (ISSN: 0020:0255, Elsevier)

is a prestigious multidisciplinary journal that publishes high-quality and refereed articles and, it emphasizes a balanced coverage of both theory and practice. In 2019, *Information Sciences* journal had a JCR Impact Factor of **5.910**; Computer Science, Information Systems 9/156 **Q1**.

– [García-García et al., 2020b] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2020). **Improving distance-join query processing with voronoi-diagram based partitioning in spatialhadoop**. *Future Generation Computer Systems*, 111:723–740. *Future Generation Computer Systems* (ISSN: 0167:739X, Elsevier) is a prestigious journal that aims to lead the way in advances in the fields of distributed systems, Big Data, clouds, among others. In 2019, *Future Generation Computer Systems* journal had a JCR Impact Factor of **6.125**; Computer Science, Theory & Methods, 8/108 **Q1**.

1.3.4 Other Contributions

Other related contributions that are not included in this thesis, but have been developed during the course of the author's PhD studies, are the following:

– [Mavrommatis et al., 2017] Mavrommatis, G., Moutafis, P., Vassilakopoulos, M., García-García, F., and Corral, A. (2017). **SliceNBound: Solving closest pairs and distance join queries in apache spark**. In *ADBIS Conference*, pages 199–213. This paper addresses the problem of answering the kCPQ in Apache Spark, by presenting a specialized and fast algorithm (SliceNBound) that can easily be imported in any, spatial-oriented or general, Spark-based system. Furthermore, it presents a variant of this algorithm that solves the εDJQ. Experiments and comparison to other solutions indicate that this new method is fast and efficient. *ADBIS* 2017 conference attracted 107 paper submissions and only 26 papers were accepted (24%) to be included in LNCS proceedings. **CORE B**.

– [Moutafis et al., 2019a] Moutafis, P., García-García, F., Mavrommatis, G., Vassilakopoulos, M., Corral, A., and Iribarne, L. (2019). **Mapreduce algorithms for the K group nearest-neighbor query**. In *SAC Conference*, pages 448–455. This paper presents a multi-phased algorithm, consisting of alternating local and parallel phases, which can be used to effectively process the Group Nearest Neighbor (GNN) query when the query dataset fits in memory, but the training one belongs to the Big Data category. Moreover, some pruning heuristics and effective calculation techniques are used, as well as different indexing methods. Finally, some comparative benchmarks with several synthetic and real-world datasets are performed. *SAC* (Symposium on Applied Computing) is a prestigious multidisciplinary conference to present the results of strategic research and experimentation (innovative application fields, technology transfer, experimental computing, strategic research, management of computing, etc.). In 2019, the *SAC* conference attracted 1067 paper submissions and, after a rigorous review process, only 258 papers were accepted (24.2%) for inclusion in the conference proceedings and presented during the symposium. **CORE B**.

© 2021 García-García, F.J.

– [Moutafis et al., 2021] Moutafis, P., García-García, F., Mavrommatis, G., Vassilakopoulos, M., Corral, A., and Iribarne, L. (2021). **Algorithms for processing the group k nearest-neighbor query on distributed frameworks**. *Distributed and Parallel Databases*, *In Press*. This paper presents a significantly improved version of the GkNNQ MapReduce algorithm presented in *SAC* 2019 conference that incorporates a new high-performance refining method, a fast way to calculate distance sums for pruning purposes, and several other coding and algorithmic improvements. Moreover, this algorithm is transformed to SpatialHadoop, using a novel two-level partitioning method. A thorough experimental study of the Hadoop and SpatialHadoop versions of the algorithm is also presented for synthetic and real-world datasets, including a backstage analysis of the algorithm's performance, using metrics that highlight its internal functioning. Finally, an experimental comparison of the Hadoop, the SpatialHadoop versions, and the version of our previous work (SAC 2019 conference) is presented, showing that the improved versions are the big winners, being the SpatialHadoop version faster than its Hadoop counterpart. *Distributed and Parallel Databases* (ISSN: 0926:8782, Springer) is a prestigious journal in the fields of distributed and parallel database technology. In 2019, Distributed and Parallel Databases journal had a JCR Impact Factor of **0.757**; Computer Science, Theory & Methods, 84/108 **Q4**.

– [García-García et al., 2021] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2021). **Enhancing sedona (formerly geospark) with efficient k nearest neighbor join processing**. In *MEDI Conference*, *Accepted*. This paper investigates how to design and implement an efficient distributed kNNJQ algorithm in Sedona (formerly GeoSpark), using the most appropriate spatial partitioning technique and other improvements. Finally, the results of an extensive set of experiments with real-world datasets are presented, demonstrating that the proposed kNNJQ algorithm in Sedona is efficient, scalable, and robust. This paper has been accepted in the *MEDI* 2021 conference. *MEDI* 2021 received 47 paper submissions and only 16 full papers have been accepted (34%) to be included in LNCS proceedings.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 details the state-of-the-art and the needed background on Distributed Spatial Data Management Systems (DSDMSs), spatial data partitioning techniques, and the most representative DBQs, paying special attention to DJQs.

Chapter 3 describes the structure and operations of spatial partitioning and indexing in SpatialHadoop. Then, this chapter proposes a data partitioning technique based on Voronoi-Diagrams to split the spatial dataset into smaller units, enabling the processing of a spatial query in parallel and reducing the I/O activity by only scanning a few partitions that contain the relevant data to the query constraint. Moreover, the Quadtree is included, as a local index, in SpatialHadoop. Finally, a set of experiments evaluates the

performance and efficiency of both proposals with respect to other spatial partitioning techniques implemented in SpatialHadoop and R-tree local index.

Chapter 4 focuses on a detailed description of the spatial query processing in SpatialHadoop. It presents the general scheme for distributed spatial query processing, built-in queries, and tools in SpatialHadoop. Next, it describes and discusses the implementation of new DBQs and several extensions and improvements. Finally, it shows an experimental evaluation of the proposed DBQ MapReduce algorithms and a comparison with their extensions and improvements.

Chapter 5 details the different spatial capabilities and spatial queries that LocationSpark provides to Spark. Next, this chapter proposes several new DJQs, extensions, and improvements over it. Finally, it presents the most representative results of an extensive set of experiments and a comparison with SpatialHadoop.

Chapter 6 provides the conclusions arising from the research of this thesis and discusses related future work on open research lines.

Finally, a list of Acronyms to describe those initials used in the document, and a *Bibliography* section, that contains the references used in this research work, is included at the end of this thesis.

© 2021 García-García, F.J.

Chapter 2

State of the Art

© 2021 García-García, F.J.

Chapter 2

State of the Art

Contents

This chapter introduces a detailed description of the state-of-the-art and the needed background. First, the most outstanding Distributed Spatial Data Management Systems (DSDMSs) are exposed in Section 2.1. Next, the state-of-the-art of spatial partitioning techniques used in DSDMSs is discussed in Section 2.2. Finally, the most representative Distance-based Queries (DBQs) are described and formally defined in Section 2.3.

2.1 Distributed Spatial Data Management Systems

Nowadays, researchers, developers and practitioners worldwide have started to take advantage of parallel and distributed computing using shared nothing clusters. The most relevant processing frameworks for these Big Data environments are Apache Hadoop and Apache Spark.

However, both Hadoop and Spark are less efficient when are applied to spatial data [You et al., 2015, Yu et al., 2015]. The main shortcoming is that there is no indexing mechanism for selective access to specific regions of spatial data, which would make query processing algorithms more efficient. This problem could be solved by the use of novel technologies for the distributed processing of large-scale spatial data on clusters of computers [Chen and Zhang, 2014], leading to *Distributed Spatial Data Management Systems* (DSDMSs), also called Big Spatial Data Analytics Systems (BSDASs) [Pandey et al., 2018]. These DSDMSs can be classified as disk-based [Li et al., 2014] or in-memory-based [Zhang et al., 2015], as we can see in [Pandey et al., 2018, de Carvalho Castro et al., 2020, Velentzas et al., 2021]. For instance, in [de Carvalho Castro et al., 2020], a comparative study of both DSDMSs (Hadoop and Spark) based on the *user-centric* view is presented. These comparisons help users to understand how the characteristics of DSDMSs are useful to meet the specific requirements of their spatial applications.

2.1.1 Disk-based DSDMSs

The disk-based DSDMSs are characterized as Hadoop-based systems. Apache Hadoop[1] is a reliable, scalable, and efficient cloud computing framework enabling distributed processing of large datasets using the MapReduce programming model [Bechini et al., 2016]. MapReduce [Dean and Ghemawat, 2004] is a framework for processing and managing large-scale datasets in a distributed cluster. In fact, the MapReduce programming paradigm has become a *de-facto standard* for processing large amounts of data (Big Data). It was introduced to provide a simple yet powerful parallel and distributed computing paradigm, offering good scalability and fault tolerance mechanisms. However, Hadoop is a type of disk-based computing framework, which writes to disk all intermediate data between *map* and *reduce* tasks.

[1] Available at `https://hadoop.apache.org/`

© 2021 García-García, F.J.

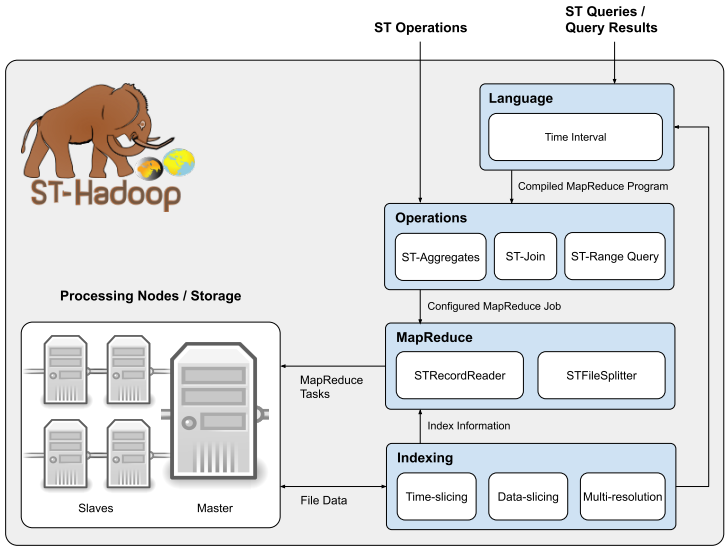


Figure 2.1: ST-Hadoop system architecture [Alarabi et al., 2018].

The most representative disk-based DSDMSs are the following research prototypes:

– *Parallel-Secondo* [Lu and Güting, 2012] is a parallel spatial DBMS that uses Hadoop as a distributed task scheduler. It integrates Hadoop with SECONDO [Güting et al., 2010], a database that can handle non-standard data types, like spatial data, usually not supported by standard systems. It only supports *uniform* spatial data partitioning techniques, which cannot efficiently handle the spatial data skewness problem.

– *Hadoop-GIS*[2] [Aji et al., 2013] extends Hive [Thusoo et al., 2009], a data warehouse infrastructure built on top of Hadoop with a uniform grid index for range queries, spatial joins and other spatial operations. It adopts the Hadoop Streaming framework and integrates several open-source software packages for spatial indexing and geometry computation. It utilizes SATO spatial partitioning [Vo et al., 2014] (similar to kd-tree [Bentley, 1975]) and local spatial indexing to achieve efficient spatial query processing.

– *SpatialHadoop*[3] [Eldawy and Mokbel, 2015] is a full-fledged MapReduce framework with native support for spatial data. It tightly integrates well-known spatial operations (including range queries, kNN query, spatial join and CG_Hadoop [Eldawy et al., 2013]) into Hadoop. It supports various spatial data types (point,

[2]Available at `https://github.com/bunnyg/Hadoop-GIS`
[3]Available at `https://github.com/aseldawy/spatialhadoop2`

line string, polygon, multi-point, etc.), several spatial partitioning techniques [Eldawy et al., 2015] (uniform Grids, STR (Sort-Tile-Recursive algorithm), Quadtree [Finkel and Bentley, 1974], kd-tree [Bentley, 1975], Hilbert-curve [Faloutsos and Roseman, 1989] and Z-curve [Orenstein and Merrett, 1984]) and local spatial indexes (Grid file [Nievergelt et al., 1984], R-tree [Guttman, 1984] and R$^+$-tree [Sellis et al., 1987]). In addition, SpatialHadoop has an excellent performance and it is one of the best maintained Hadoop-based DSDMS [Pandey et al., 2018]. For all these reasons, we have focused on SpatialHadoop and no in others Hadoop-based DSDMSs.

– *ST-Hadoop*[4] [Alarabi et al., 2018] extends Hadoop and SpatialHadoop by adding spatio-temporal data awareness to their language, indexing and operation layers as shown in Figure 2.1. It supports several spatio-temporal operations (including ST-range query, ST-join, ST-aggregates, and kNNQ). It adds various spatio-temporal data types (STPoint, TIME, INTERVAL, etc.) and uses a spatio-temporal index that consists of two layers of temporal slices and then spatial partitions. Finally, ST-Hadoop replicates its index into a temporal hierarchy index structure where the same data is replicated on distinct levels but with different spatio-temporal granularities.

According to [Yao and Li, 2018, Alam et al., 2021], Table 2.1 lists the most representative existing DSDMSs based on Hadoop, which are compared from three aspects, namely spatial partitioning, spatial indexing, and spatial query.

DSDMS	Spatial Partitioning	Spatial Indexing	Spatial Query
Parallel-Secondo	3D Grid	B-tree, R-tree	range query, spatial join
Hadoop-GIS	SATO framework	Two-Level (Global, Local) R*-tree	range query, spatial join
ST-Hadoop	Time-slice, Data-slice	Two-Level (Temporal, Spatial) ST-index	ST-range query, ST-join, ST-aggregates, kNNQ
SpatialHadoop	Grid, STR, STR+, Quadtree, kd-tree, H-curve, Z-curve	Two-Level (Global, Local) Grid File, R-tree, R$^+$-tree	range query, kNNQ, spatial join

Table 2.1: The most representative existing DSDMSs based on Hadoop.

Finally, there are other DSDMSs that show some interesting features. For instance, **CloST** [Tan et al., 2012] is a MapReduce-based storage system for big spatio-temporal data analytics on Hadoop. It uses a simple data model composed of three main attributes (*id*, *location* and *time*) and a hierarchically partitioning method that enables efficient parallel processing of spatio-temporal range scans and both single-object and all-object queries.

[4]Available at `http://st-hadoop.cs.umn.edu/`

© 2021 García-García, F.J.

2.1.2 In-memory-based DSDMSs

Considering in-memory-based DSDMSs, they are characterized as Spark-based systems. Apache Spark[5] is a fast, reliable and distributed in-memory large-scale data processing framework. It takes advantage of Resilient Distributed Datasets (RDDs) that allow data to be stored transparently in memory and persisted to disk only if necessary [Zaharia et al., 2012]. Hence, it can avoid a huge number of disk writes and reads, and outperform the Hadoop platform. Since Spark maintains the status of assigned resources until a job is completed, it reduces the consumed time in resource preparation and collection [Karim et al., 2018].

The most remarkable in-memory research prototypes are the following:

- *SpatialSpark*[6] [You et al., 2015] is a lightweight implementation of several spatial operations on top of Spark in-memory big data system. It targets at in-memory processing for higher performance. SpatialSpark adopts data partition strategies, like fixed grid or kd-tree on data files in HDFS and builds an index to accelerate spatial operations. It supports range queries and spatial joins over geometric objects using spatial conditions, like *intersect* and *within*.

- *GeoSpark*[7] [Yu et al., 2015, Yu et al., 2019], currently *Sedona*, extends Spark for processing spatial data. It provides a new abstraction, called Spatial Resilient Distributed Datasets (SRDDs), and a few spatial operations. It allows an index (e.g., Quadtree and R-tree) to be the object inside each local RDD partition. From the query processing point of view, GeoSpark supports range query, kNNQ, spatial joins, and distance join over SRDDs. Figure 2.2 summarizes its architecture and the relations between layers.

- *Simba*[8] (Spatial In-Memory Big data Analytics) [Xie et al., 2016] offers scalable and efficient in-memory spatial query processing and analytics for big spatial data. Simba is based on Spark and runs over a cluster of commodity machines. In particular, Simba extends the Spark SQL engine to support rich spatial queries and analytics through both SQL and the DataFrame API. It introduces spatial partitioning techniques (e.g., Sort-Tile-Recursive (STR) algorithm [Leutenegger et al., 1997]), spatial indexes (global and local) based on R-trees over RDDs to work with big spatial data, and complex spatial operations (e.g., range query, kNNQ, distance join and kNNJQ).

- *STARK*[9] [Hagedorn and Räth, 2017] is a framework that adds spatio-temporal support to Spark, includes spatial partitioners, several modes for indexing, as well as filter, join, and clustering operators. More precisely, STARK includes spatial partitioning (grid and binary space) and indexing techniques (R-tree) for fast and efficient execution of the data analysis tasks. STARK also supports spatial

[5] Available at `https://spark.apache.org/`
[6] Available at `https://github.com/syoummer/SpatialSpark`
[7] Available at `http://sedona.apache.org/`
[8] Available at `http://www.cs.utah.edu/~dongx/simba/`
[9] Available at `https://github.com/dbis-ilm/stark`

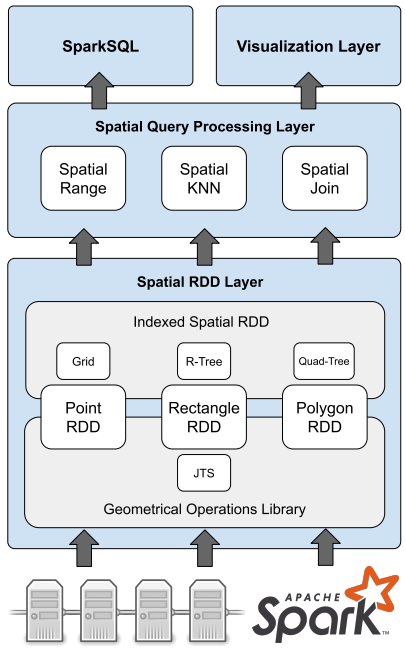


Figure 2.2: GeoSpark system architecture [Yu et al., 2015].

operations like intersect, contains, containedBy, spatial join, skyline, kNNQ, and a density-based clustering operator that allows us to find groups of similar events.

– *LocationSpark*[10] [Tang et al., 2016, Tang et al., 2020] is an efficient in-memory distributed spatial query processing system that is characterized as a Spark-based system. It provides promising features for the efficiency of query processing, like data and query skew components to improve load balancing while executing spatial operators (e.g., spatial range, kNN search, spatial range join, and kNN join), by generating cost-optimized query execution plans over in-memory distributed spatial data. Moreover, LocationSpark builds two layers of spatial indexes: global and local. The global index partitions the entire dataset equally between the available processing nodes, and it uses Grid, R-tree, or Quadtree. Each data partition has a local spatial index (e.g., a Grid local index, an R-tree, a variant of the Quadtree, or an IR-tree [Li et al., 2011]).

Note that [Pandey et al., 2018] explores the availability of spatial analytics systems (based on Spark) and compares their features and queries by running experiments that evaluate their performance and other metrics using real-world datasets. For kNNJQ, only *LocationSpark* and *Simba* support it, and *LocationSpark* obtains the best results in terms of performance, scalability and shuffle cost. As a conclusion, the authors highlight

[10] Available at `https://github.com/purduedb/LocationSpark`

© 2021 García-García, F.J.

that *LocationSpark* is a very interesting option since it has a very good query scheduler and optimizer. Also, it has a *spatial bloom filter* (sFilter) which brings the query costs down. Moreover, they also suggest that these features could be incorporated in the other studied Spark-based systems.

According to [Yao and Li, 2018, Alam et al., 2021], Table 2.2 lists the most representative existing DSDMSs based on Spark, which are compared with the same aspects as in Table 2.1.

DSDMS	Spatial Partitioning	Spatial Indexing	Spatial Query
SpatialSpark	Grid, Binary-split, STR	R-tree	range query, spatial join
GeoSpark	Grid, Voronoi, R-tree, Quadtree, kDB-tree	R-tree, Quadtree	range query, kNNQ, spatial join, distance join
Simba	STR	R-tree (multi-level)	range query, kNNQ, distance join, kNNJQ
STARK	Grid, Binary-split	R-tree (live and persistent)	range query, kNNQ, spatial join
LocationSpark	Grid, R-tree, Quadtree	R-tree, Quadtree, IR-tree (multi-level)	range query, kNNQ, spatial range join, distance join, kNNJQ

Table 2.2: The most representative existing DSDMSs based on Spark.

In addition to the previous in-memory-based DSDMSs, there are others that present some quite promising characteristics:

- **Magellan** [Sriharsha, 2021] is a distributed execution engine on top of Apache Spark that optimizes spatial queries over big data. It extends SparkSQL with spatial datatypes, geometric predicates and, range and join queries on top of a Z-curve index.
- **SparkGIS** [Baig et al., 2017] is a distributed, in-memory spatial data processing framework that combines the in-memory distributed processing capabilities of Apache Spark and the efficient spatial query processing of Hadoop-GIS. Experiments with medical images and geographical data from OpenStreetMap have proved its performance in real scenarios.
- **GeoTrellis** [Kini and Emanuele, 2014] is an open-source library focused on the management of large spatial raster datasets over Apache Spark. It relies on files written using the GeoTIFF[11] format and the use of multi-dimensional space-filling curves for indexing.
- **GeoMatch** [Zeidan et al., 2018] is a scalable and efficient big-data pipeline for large-scale map matching on Apache Spark. It leverages an effective indexing technique based on the Hilbert space-filling curve and a load balancing algorithm that evenly distributes the dataset across compute nodes.
- **SciSpark** [Wilson et al., 2016] is a big data framework focused on scientific computations built on top of Apache Spark. In its current state, it provides time and

[11] Available at `https://www.ogc.org/standards/geotiff`

space partitioning, n-dimensional array operations, parallel computation of time-series statistical metrics, and a frontend interface that uses Apache Zeppelin[12] notebook software.

Finally, there are other in-memory-based DSDMSs whose main contribution is the support for spatio-temporal data handling:

- **BinJoin** [Whitman et al., 2017, Whitman et al., 2019] is a spatio-temporal attribute join implementation in Apache Spark that uses a local index and a query optimizer to increase its performance. Two interesting conclusions can be extracted from its experimental study: the efficiency of a distributed spatial join algorithm depends on the characteristics of the two input datasets, and the temporal conditions decrease the effects of spatial skew.
- **GeoMesa** [Hughes et al., 2015] provides spatio-temporal indexing on top of different distributed data storage systems. It supports range and spatial join queries optimized by the use of an R-tree spatial partitioning technique and a Grid-based local index.

2.2 Spatial Data Partitioning

Data partitioning is a powerful mechanism to improve the efficiency of data management systems, and it is a standard feature in modern database systems. Aside from the fact that data partitioning improves the overall manageability of large datasets, it also speeds up query performance. Partitioning such datasets into smaller units enables the processing of a query in parallel and reduces the I/O activity by only scanning a few partitions that contain relevant data to the query constraints. When we partition spatial data in a distributed framework, we are talking about *spatial data partitioning*. DSDMSs have to take into account several factors of their execution environment and the characteristics of real-world spatial objects. Therefore, the following **factors** [Yao et al., 2017] must be analyzed for the *spatial partitioning techniques* to get better algorithms with optimal performance:

F.1 *Spatial Objects.* They are the smallest unit of non-divisible / non-splittable spatial information (e.g., points, line-segments, polygons, regions, etc.).

F.2 *Spatial Location.* The chosen representation to store the *Spatial Object*. Normally, instead of using a complex geometry, exactly describing the spatial object, an approximation is used (e.g., center, centroid, MBR, etc.).

F.3 *Spatial Distribution.* By the nature of spatial objects, they usually show localization patterns that tend to show skew. In addition, adjacent spatial objects must be partitioned in the same blocks as much as possible while seeking a balance that reduces skew problems. For complex spatial objects, we also have to decide how to handle spatial objects that are within the boundaries between partitions.

[12] Available at `https://zeppelin.apache.org/`

© 2021 García-García, F.J.

F.4 *Object Volume.* Size of the spatial object (bytes) in the physical storage layer (disk or memory).

F.5 *Block Size.* It determines when a block of data in HDFS is subdivided or merged (e.g., the default value for Hadoop 2 is 128 MB). Optimally, the size of the partitions should approximate this value.

Moreover, *spatial data partitioning* is challenging due to several important properties that are particular to spatial data and query processing, especially spatial data skew and boundary object handling [Aji et al., 2015].

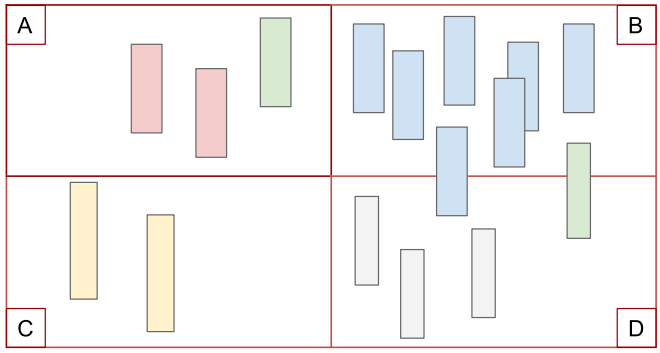


Figure 2.3: Partitions of a spatial dataset that exhibit spatial data skew and boundary objects.

In general, the problem of data skew is that some partitions contain more data elements, and their processing creates a delay in obtaining the final result of the query. As shown in Figure 2.3, in spatial applications, we can find regions or countries, similar to cell B, that have more density because they contain a greater number of spatial objects than others. Furthermore, these most populated areas will normally have a high number of other space entities, such as buildings, that will increase the magnitude of the problem when they are part of a join-type query. Therefore, to increase the performance of the spatial queries, the different partitioning techniques must take into account this characteristic presented by the spatial data when dividing the space into partitions and/or using a preprocessing prior to the spatial query that performs some re-distribution to adjust the data in units of work as uniform as possible.

As for boundary objects, spatial partitioning techniques should treat them in a special way. Complex spatial objects, unlike points, fill a certain area that can involve multiple partitions. For example, Figure 2.3 shows two rectangles that are in the boundary between cells B and C. Some partitioning techniques use replication methods that copy each geometry in all partitions with which it interacts. Therefore, it is necessary to eliminate possible duplicate results using techniques such as the *reference-point duplicate avoidance* technique [Eldawy and Mokbel, 2015], which consists of fixing a single point of the geometry and working only with the partition in which it is located. However, this causes both an increase in the physical size of the spatial dataset and a rise in the

processing time. Other partitioning methods, such as partitioning based on Z-curve or Voronoi-Diagrams, associate each spatial object to exactly one partition avoiding this problem.

The subsequent sections describe the most important spatial data partitioning techniques [Eldawy and Mokbel, 2015] classified by how they use spatial data or space properties. More specifically, different data structures are presented, along with the procedures used to obtain the final partitions and auxiliary indexes.

2.2.1 Space-based Partitioning Techniques

Space-based partitioning techniques perform the division of space through an algorithm that uses some of the geometric characteristics of the space of the dataset to be partitioned. Among the most important partitioning techniques are:

– *Grid-based partitioning.* It consists of the division of space into a matrix of $m \times n$ cells of the same size. The main advantage of this partitioning method is that no previous preprocessing is necessary to divide a dataset into a certain number of columns and rows. Given the Minimum Bounding Rectangle (MBR) $(x_{min}, y_{min}, x_{max}, y_{max})$ of a dataset $\mathbb{P}$ and the number of columns m and rows n, the corresponding cell c of a point $p \in \mathbb{P}$ is given by the following equations:

$$
\begin{aligned}
dx &= (x_{max} - x_{min})/m \\
dy &= (y_{max} - y_{min})/n \\
i &= (p_x - x_{min})/dx \\
j &= (p_y - y_{min})/dy \\
c &= i + j * n
\end{aligned}
$$

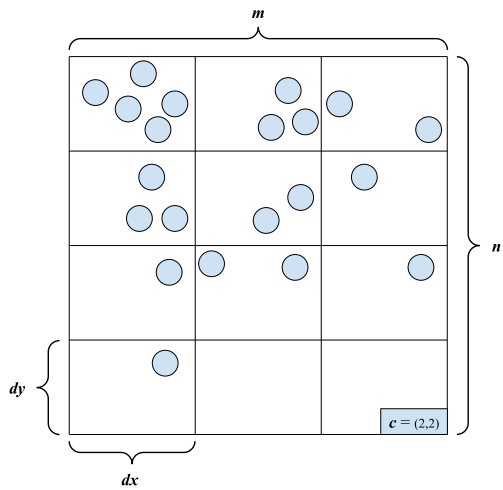


Figure 2.4: Spatial dataset partitioned by a 3×4 grid.

© 2021 García-García, F.J.

For instance, SpatialHadoop has an index based on an uniform Grid [Eldawy et al., 2015], where $m = n$, which performs the partitioning of the spatial dataset based on their MBR without using any sampling or other preprocessing method. Furthermore, given a particular cell, neighboring cells are also easily located. Figure 2.4 shows a dataset partitioned by a grid of $m = 3$ and $n = 4$ where the bottom right cell c is identified by $i = 2$ and $j = 2$.

– *Quadtree-based partitioning.* It uses a Quadtree [Finkel and Bentley, 1974] for the partitioning of the data through the recursive decomposition of a two-dimensional space in 4 quadrants or regions. According to [Samet, 1984], a *Quadtree* is a set of hierarchical data structures that present a series of common properties such as recursive subdivision and the use of 2 types of nodes: *internal* nodes and *leaf* nodes. Each internal node of the tree has 4 children, representing the 4 resulting regions: NW(North West), NE(North East), SW(South West), and SE(South East). Moreover, internal nodes are often accompanied by information about the region they represent, such as the MBR. Normally, Quadtree-based partitioning techniques recursively divide the nodes/regions that contain a larger number of elements until a certain condition or restriction is satisfied. For example, until the number of elements contained in the leaf nodes occupy a certain size of main memory or disk (capacity). In the case of SpatialHadoop, a sampling of s elements of the input spatial dataset is performed previously, which are inserted one by one in a Quadtree with a node capacity of s/n, where n is the desired number of partitions [Eldawy et al., 2015].

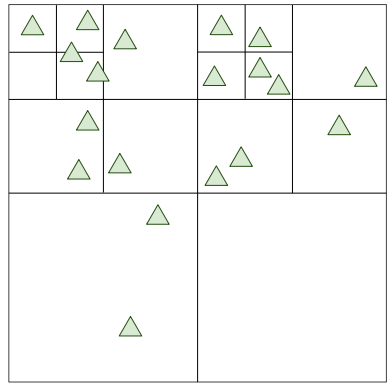


Figure 2.5: Spatial dataset partitioned by a Quadtree with a maximum of two elements per leaf.

Finally, these space-based partitioning techniques can store the complete structure of the tree as an index or only use the boundaries of the leaf nodes or some level, which represent the final partitions. Figure 2.5 shows the final partitions of a dataset by means of a Quadtree based on a maximum of 2 elements per leaf.

2.2.2 Data-based Partitioning Techniques

Data-based partitioning techniques are defined as those partitioning methods that use characteristics of the data, such as their number or position, to divide the space in which they are located. The most outstanding techniques are:

– *R-tree-based partitioning.* This technique partitions the data using the R-tree resulting from the insertion of the elements of a dataset. An R-tree [Guttman, 1984] is a height-balanced tree similar to a B-tree [Comer, 1979] whose structure is designed to perform spatial searches by visiting as few nodes as possible. To this end, each node contains entries with a pointer to a children node and its MBR that is tested with a geometric operation (intersection, contains, etc.) to traverse the tree during the spatial query. This data structure divides the space hierarchically into sets, possibly overlapping, with the data elements on the leaf nodes. In contrast to a Quadtree, nodes are divided by balancing the number of elements using different element/node heuristics without explicitly dividing the space. Moreover, the degree of an R-tree is the maximum number of entries per node. There are several methods for its creation, such as one-by-one insertion or bulk loading. The techniques based on bulk loading consider all data to be inserted as a whole to take advantage of their joint properties. Therefore, they usually have higher creation, storage, and query performance than one-by-one insertion methods. When adapting the partitioning to the various factors discussed in the

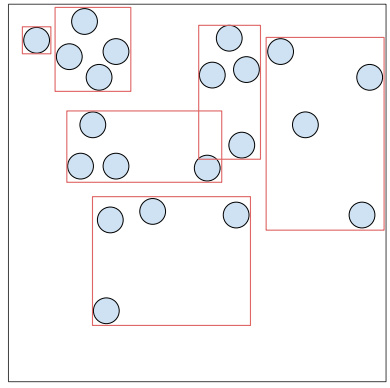


Figure 2.6: Spatial dataset partitioned by an R-tree.

previous section, we can use the R-tree degree to limit the number of children per node. Furthermore, we can select a tree-level or elected nodes as we go through it to obtain the partitions based on some restrictions. For instance, the STR partitioning method in SpatialHadoop [Eldawy et al., 2015] bulk loads a sample of s elements from the spatial dataset into an R-tree using the Sort-Tile-Recursive (STR) algorithm [Leutenegger et al., 1997]. The degree of the R-tree used is s/n, where n is the number of partitions, to ensure that there are at least n partitions

© 2021 García-García, F.J.

on the second level. Then the MBRs of these nodes are used as boundaries of the final partitions.

As with Quadtree, the STR partitioning technique can preserve the complete structure of the R-tree as an index, as SpatialHadoop does. Figure 2.6 shows the result of applying R-tree-based partitioning to a dataset and also limiting the number of elements per leaf to 4. Notice that there are overlaps between partitions, and their size is less regular compared to Quadtree-based partitioning.

– *kd-tree-based partitioning.* It uses a kd-tree [Bentley, 1975] to partition a spatial dataset by the result of its construction. A kd-tree is a multidimensional binary tree where each node contains a point with k dimensions and two pointers to each child node. Therefore, for $k = 2$ it splits a two-dimensional space into two half-spaces by using planes defined by the inserted points. Moreover, for each level of the binary tree the reference dimension/axis for inserting/searching is swapped, that is, for two-dimensional spaces, it switches between horizontal and vertical planes. To obtain a well-balanced tree there are different techniques mainly based on median-finding algorithms. In these methods, the median from the dataset is chosen as the root and the algorithm is recursively applied to the two new generated splits. For spatial partitioning, the recursive algorithm stops when there are no more elements to split or some restriction is reached, such as the number of partitions. For instance, kd-tree-based partitioning in SpatialHadoop starts with the MBR of the full dataset and inserts $n - 1$ median elements from a sample of the dataset into the kd-tree in order to get n partitions [Eldawy et al., 2015].

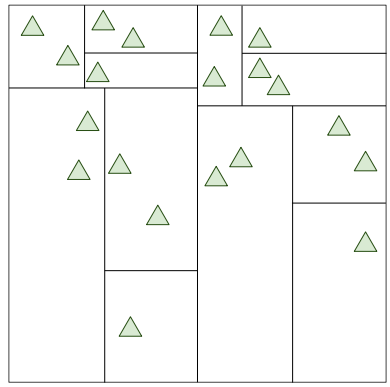


Figure 2.7: Spatial dataset partitioned by a kd-tree where $k = 2$.

Finally, the partitions correspond to the divisions of the spatial space generated by the planes present in the tree. Figure 2.7 shows the final partitions of a dataset partitioned by a kd-tree with $k = 2$ and with a limit on the number of elements per leaf of 2. Notice that the size of the partitions is still irregular, but in this case, there are no overlaps between them.

There are other *Data-based partitioning techniques* that use variations of these data structures. For instance, in [Elashry et al., 2018] the 2DPR-tree partitioning method for SpatialHadoop is presented, and it combines properties of both R-tree and kd-tree. This method uses a two-dimensional Priority R-tree (PR-tree) [Arge et al., 2008] which is an R-tree that presents optimal performance when answering window queries. Basically, it is just a kd-tree that stores the partitions as four-dimensional points, except that four extra leaves are added below each internal node with the actual points arranged by maximum / minimum values in each axis, e.g., one leaf has points with the lowest X-axis values. Given a node capacity and starting with the root, the partitioning algorithm generates two more child nodes with the exceeding elements once the node is filled, and then it starts sorting the elements in the four leaves. This bulk-loading process is done recursively.

2.2.3 Space-Filling Curve-based Partitioning Techniques

The *space-filling curve-based partitioning techniques* [Sagan, 2012, Mokbel et al., 2003] map each multi-dimensional element of the input dataset using a mathematical function to a one-dimensional space that will be used to obtain the final partitions. In the case of two-dimensional space, a space-filling curve is a continuous curve that contains the entire 2-dimensional unit square in the unit interval $[0, 1]$, that is, a curve that passes through every point of the 2-dimensional region. An important feature about space-filling curves is that it allows one-dimensional techniques to be applied to data of multidimensional origin. Therefore, we can use much less complex processing methods and algorithms that are independent of the dimensionality of the data. The main difference between the types of space-filling curves is how the mapping is done towards one-dimensional space, that is, its way of going through the multi-dimensional space. They can also be classified into *recursive* and *non-recursive*, according to their construction. If we call I the one-dimensional interval that maps to square Q, if we partition I into 4 intervals and Q into 4 subsquares, each subinterval $(I')_i$ can be mapped into its corresponding subsquare $(Q')_i$. Furthermore, the squares can be arranged in such a way that the adjacent subintervals correspond to subsquares that have a common edge. This process allows us to preserve locality by transforming data from multidimensional to one-dimensional space, a very important characteristic when measuring the quality of a space-filling curve. Finally, partitioning techniques, which use this type of curve, usually order the elements of the input dataset based on their mapping and then divide them into sets that meet some restrictions to create the final partitions based on their centroids or MBRs.

Some of the most common Space-Filling Curves are:

– *Z-curve-based partitioning.* Z-curve (Z-order curve) [Peano, 1890] maps the unit square using an interval that goes through it in the form of a Z. They are characterized by the fact that algorithms for performing the mapping between parameter space and curve indexes should be time-efficient. For instance, in order to partition a spatial dataset with this spatial partitioning technique, SpatialHadoop sorts the elements of a sample by their order on the Z-curve and takes n equally sized splits to get n partitions defined by the MBR of the points inside each split

© 2021 García-García, F.J.

[Eldawy et al., 2015]. Figure 2.8 shows how the interval maps with two numbers of partitions (4 vs. 16) for the Z-curve.

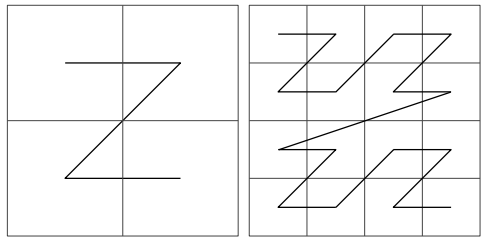


Figure 2.8: Z-curve-based partitioning with two number of partitions (4 vs. 16).

– *H-curve-based partitioning.* H-curve (Hilbert-curve) [Hilbert, 1891] maps the unit square using an interval that runs through it in a U-shape and reorder the subintervals to present good locality. Therefore, if two elements are together on the H-curve, the corresponding elements of the multidimensional dataset are also close. For example, SpatialHadoop uses the same process as with Z-curve partitioning but considering the order of the sampled elements in the H-curve [Eldawy et al., 2015]. Figure 2.8 shows how the interval maps with two numbers of partitions (4 vs. 16) for the H-curve.

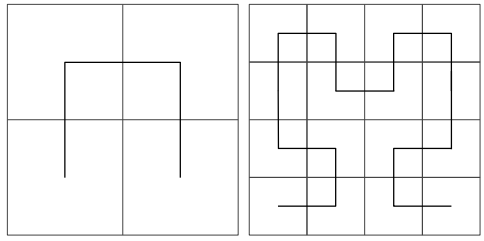


Figure 2.9: H-curve-based partitioning with two number of partitions (4 vs. 16).

2.2.4 Distance-based Partitioning Techniques

Distance-based partitioning techniques use several distance metrics and geometric relationships between the spatial objects to partition the input dataset. They can be considered as a constrained version of the space-based partitioning techniques.

The most important distance-based partitioning technique is the *Voronoi-Diagram*-based partitioning. A *Voronoi-Diagram* divides space into disjoint partitions where the nearest neighbor of any point inside a partition is the generator or *pivot* of the partition.

Each partition of the Voronoi-Diagram, called *Voronoi-Cell*, is associated with a point p (pivot), such that any point inside *p's* cell has p as its nearest neighbor [Okabe et al., 2000, Aurenhammer, 1991]. The resulting data structure from the Voronoi-Diagram is very efficient in exploring a local neighborhood in a geometric space. Voronoi-Diagrams are used in many algorithmic applications, like closest-site problems (nearest neighbor queries and closest pairs), clustering point sites (partitioning and hierarchical clustering methods), placement and motion planning, triangulating sites, connectivity graphs for sites, etc. [Aurenhammer, 1991].

In order to *partition a spatial dataset*, it is divided into partitions based on a Voronoi-Diagram with a careful method for selecting a set of suitable *pivots*. Then, these data partitions (Voronoi-Cells) are clustered into groups only if the distances between them are restricted by a specific distance bound. The *pivot selection strategy* is crucial for the creation of Voronoi-Diagrams [Lu et al., 2012, García-García et al., 2020b] and therefore for the query processing. Hence, the use of a *clustering algorithm* improves the quality of the selected pivots, which separate the whole dataset more evenly and also improves the performance of queries. This is because clustering algorithms aim at grouping objects in such a way that similar ones belong to the same cluster and are different from the ones belonging to other clusters [Jain et al., 1999, Xu and Tian, 2015]. Figure 2.10 shows a dataset example where each element is selected as a pivot and the resulting Voronoi-Cells or partitions. Finally, to optimize and speed up existing clustering algorithms, *sampling*

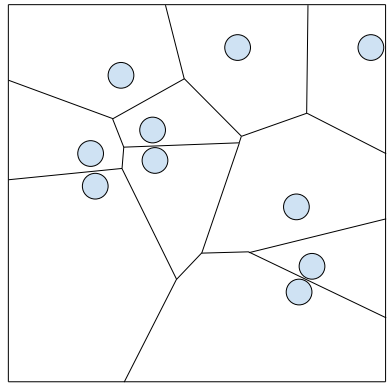


Figure 2.10: Spatial dataset partitioned by a Voronoi-Diagram.

is a very interesting technique when large datasets are managed [Ros and Guillaume, 2017]. It can be considered as a preprocessing step for clustering algorithms, and it should provide a representative and relevant set of samples from the input dataset.

Voronoi-Diagrams can partition spatial datasets into set spaces and are effective in the study of local neighborhoods for each partition. For this reason, Voronoi-Diagrams are used as a spatial partitioning technique in distributed environments. At the same time, Voronoi-Diagrams can help improve the performance of MapReduce distance-based join queries [Akdogan et al., 2010, Lu et al., 2012, Kim et al., 2016, Kuhlman et al., 2017, García-García et al., 2018a, Hu et al., 2020, García-García et al., 2020b].

© 2021 García-García, F.J.

2.3 Distance-based Query Processing

Distance-based Queries (DBQs) in spatial databases have received considerable attention from the database community due to their importance in numerous applications, such as geographical information systems (GIS), location-based systems, continuous monitoring in streaming data settings, and processing of road network constrained data, among others. DBQs are especially costly queries when they combine two or more datasets, considering a certain distance metric as the main query constraint. These DBQs could require the use of special query processing techniques, for example, the *plane-sweep* technique [Jacox and Samet, 2007] is used when the datasets are not indexed. Besides, several research works have been devoted to improving the performance of these DBQs by proposing efficient algorithms or designing new complex spatial index structures. However, all these approaches focus on methods that are to be executed in a centralized environment. Furthermore, with the fast increase in the generation of large datasets from spatial applications, processing large-scale data in a parallel and distributed way is becoming a popular practice. For this reason, a considerable number of parallel and distributed DBQ algorithms in MapReduce have been recently designed and implemented. A special case of DBQs is the Distance-based Join Query (*DJQ*) [Li and Taniar, 2017] where two datasets are combined, taking into account a distance metric. These DJQs could be very costly when the size of the datasets is huge, and for this reason, they have lately been thoroughly investigated.

The next sections describe the semantic details of the most representative DBQs, along with the corresponding notation and processing paradigms. Moreover, their most important characteristics are reviewed, assuming that the Euclidean distance, *dist*, is the distance used in these DBQs.

2.3.1 kNearest Neighbor Query

Given a set of points, the kNearest Neighbor Query (kNNQ) [Roussopoulos et al., 1995] discovers the k closest points to a given query point (i.e., it reports only the top k points). It is one of the most important and studied spatial operations, where one spatial dataset and a distance function are involved. The formal definition of the kNNQ for points is the following:

Definition 2.1. **kNearest Neighbor query, *kNN* query**
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ a set of points in E^d (d-dimensional Euclidean space), a query point q in E^d, and a number $k \in \mathbb{N}^+$ $(k > 0)$. Then, the result of the kNearest Neighbor Query with respect to the query point q is an ordered collection, $kNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains the k $(1 \leq k \leq |\mathbb{P}|)$ different points of $\mathbb{P}$, with the k smallest distances from q: $kNN(\mathbb{P}, q, k) = (p_1, p_2, \cdots, p_k) \in \mathbb{P}$, such that for any $p_i \in \mathbb{P} \setminus kNN(\mathbb{P}, q, k)$ we have $dist(p_1, q) \leq dist(p_2, q) \leq \cdots \leq dist(p_k, q) \leq dist(p_i, q)$.

One **application case** (Accommodation Services), with a spatial dataset of locations of hotels and the position of a conference center, kNNQ could *find the 3 nearest possible hotels to the conference center* in order to select the hotel ($k = 1$) closest to the conference where the user is attending. Figure 2.11 illustrates this example with the

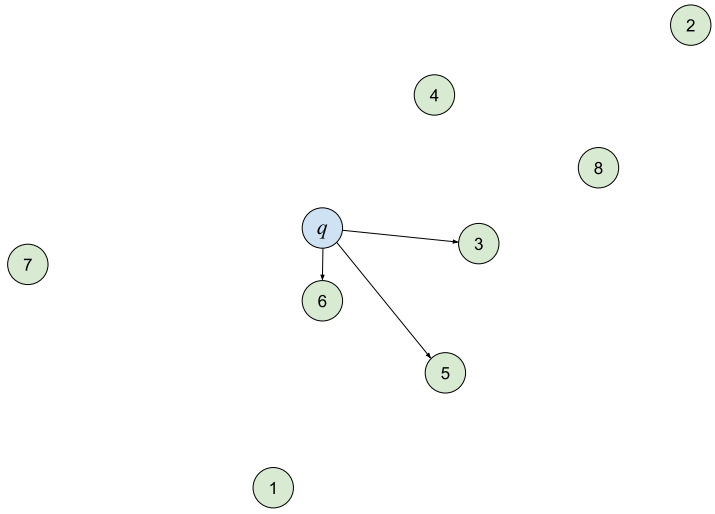


Figure 2.11: kNearest Neighbor query with $k = 3$.

numbered circles being the hotels ($\mathbb{P}$) and the q circle is the conference center and the result of the query is $kNN(\mathbb{P}, q, 3) = (3, 6, 5)$.

There are several works on kNNQ in MapReduce. For instance, in [Zhang et al., 2009a] a brute force approach calculates the distance to each point and selects the nearest k points, while another approach [Akdogan et al., 2010] partitions points using a Voronoi-Diagram and finds the answer in partitions close to the query point.

2.3.2 εDistance Range Query

The εDistance Range Query (εDRQ), given a set of points, finds all points in the dataset that fall on the circular shape, centered in a query point (q) with radius a distance threshold ε. Note that this query is also called *circle range query* or *circular query*. It is a special case of *Regional Query* [Shekhar and Chawla, 2003], which allows us to search regions that have arbitrary orientations and shapes. The formal definition of the εDRQ for points is as follows.

Definition 2.2. εDistance Range query, εDR query
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ a set of points in E^d, a query point q in E^d, and a distance threshold $\varepsilon \in \mathbb{R}^+$ ($\varepsilon > 0$). Then, the result of the εDistance Range query with respect to the query point q is a set, $\varepsilon DR(\mathbb{P}, q, \varepsilon) \subseteq \mathbb{P}$, which contains all points $p_i \in \mathbb{P}$ that fall on the circular shape, centered in q with radius ε:
$\varepsilon DR(\mathbb{P}, q, \varepsilon) = \{p_i \in \mathbb{P} \ : \ dist(p_i, q) \ \leq \ \varepsilon\}$

One **application case** (Fitness Finding Service), with a spatial dataset of fitness centers and a position given by the user, εDRQ could *find all fitness centers at 2 kilome-*

© 2021 García-García, F.J.

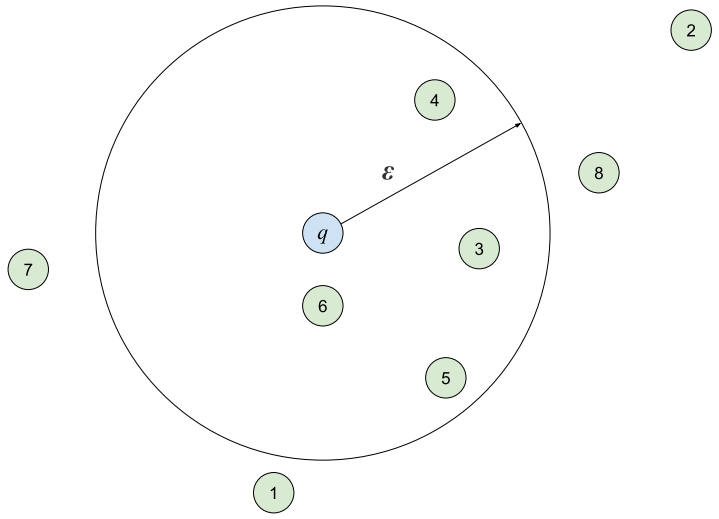


Figure 2.12: εDistance Range query with distance ε.

ters from my home. Figure 2.12 shows this situation by making circle q the home and using an ε value of 2 kilometers, and the result of the query is $\varepsilon DR(\mathbb{P}, q, 2) = \{3, 4, 5, 6\}$.

This spatial query has been extensively studied in centralized environments, however, when the dataset resides in a parallel and distributed framework, it has not received the same attention. Examples of such distributed works are [Zhang et al., 2009a, Ma et al., 2009], where the input file is scanned, and each record is compared against the query range.

2.3.3 Reverse kNearest Neighbor Query

For Reverse kNearest Neighbor Query (RkNNQ), given a set of points $\mathbb{P}$ and a query point q, a point p is called the *Reverse kNearest Neighbor* of q, if q is one of the k nearest points of p. That is, a RkNNQ issued from point q returns all points of $\mathbb{P}$ whose k nearest neighbors include q. Note that, this query is also called *Monochromatic* RkNNQ [Korn and Muthukrishnan, 2000].

Definition 2.3. Reverse kNearest Neighbor query, RkNN query [Wu et al., 2008]
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ a set of points in E^d, a query point q in E^d, and a number $k \in \mathbb{N}^+$ $(k > 0)$. Then, the result of the Reverse kNearest Neighbor query with respect to the query point q is a set, $RkNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains all points of $\mathbb{P}$ whose k nearest neighbors include q:
$RkNN(\mathbb{P}, q, k) = \{p_i \in \mathbb{P} : q \in kNN(\mathbb{P} \cup q, p_i, k)\}$

One **application case** (Wi-Fi Signal Coverage), with a spatial dataset of locations of Wi-Fi access points in a university campus and the location of a new Wi-Fi access point,

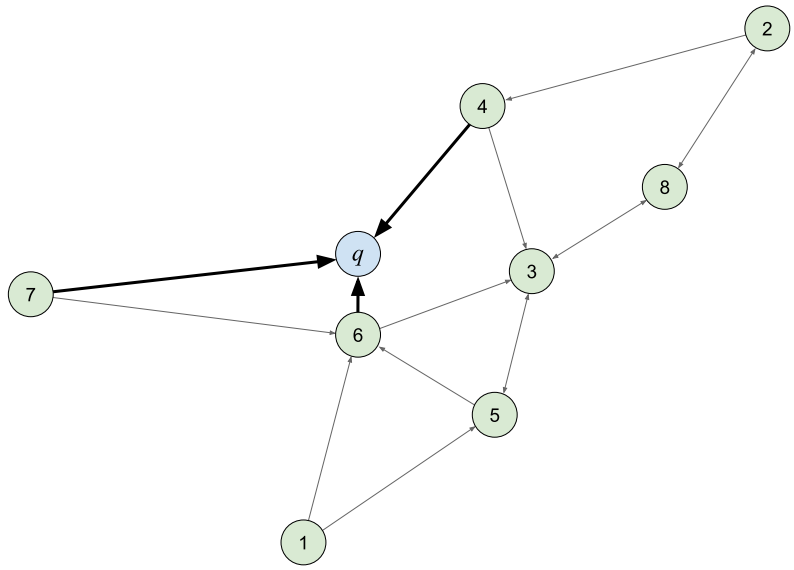


Figure 2.13: Reverse kNearest Neighbor query with $k = 2$.

RkNNQ could *find which access points would connect with the new one if each access point can connect only with 2 other access points* in order to select the best location for the new access point. Figure 2.11 represents this example with the numbered circles being the actual Wi-Fi access points ($\mathbb{P}$) and the q circle the new one and the result of the query is $RkNN(\mathbb{P}, q, 2) = \{4, 6, 7\}$.

The RkNN problem has been studied extensively in the past few years. As shown in a recent experimental study [Yang et al., 2015], which is the state-of-the-art RkNN algorithms for two-dimensional location data in centralized environments, and the most relevant contributions are the following. In [Korn and Muthukrishnan, 2000], RkNNQ was first introduced and its processing is based on a pre-computation step (for each data point $p \in \mathbb{P}$ the kNearest Neighbor, *kNN(p)*, is pre-computed and its distance is denoted by *kNNdist(p)*) and has three phases: *pruning*, *containment* and *verification*. In the *pruning* phase, for each $p \in \mathbb{P}$ a circle centered at p with radius *kNNdist(p)* is drawn, and the space that cannot contain any RkNN is pruned by using the query point q. In the *containment* phase, the objects that lie within the unpruned space are the RkNN candidates. Finally, in the *verification* phase, a *range query* is issued for each candidate to check if the query point is one of its *kNN* or not. That is, for the query point q, it determines all circles (*p, kNNdist(p)*) that contain q and return their centers p. In [Stanoi et al., 2000], the *Six-Regions* algorithm is presented, and the need for any pre-computation is eliminated by utilizing some interesting properties of RkNN retrieval. The authors solve RkNNQ by dividing the space around the query point into six equal partitions of 60° each (R_1 to R_6). In each partition R_i, the *k-th* nearest neighbor of the query point defines the pruned area. In [Singh et al., 2003]

© 2021 García-García, F.J.

the multistep *SFT* algorithm is proposed. It: (1) finds (using an R-tree) the *kNNs* of the query point q, which constitute the initial candidates; (2) eliminates the points that are closer to some other candidate than q; and (3) applies *boolean range queries* on the remaining candidates to determine the actual *RNNs*. In [Tao et al., 2004], the *TPL* algorithm, which uses the property of perpendicular bisectors located between the query point for facilitating pruning the search space, is presented. In the *containment* phase, TPL retrieves the objects that lie in the area not pruned by any combination of k bisectors. Therefore, TPL has to consider each combination of k bisectors. To overcome the shortcomings of this algorithm, a new method named *FINCH* is proposed in [Wu et al., 2008]. Instead of using bisectors to prune the objects, the authors use a convex polygon that approximates the unpruned area. *Influence Zone* [Cheema et al., 2011] is a half-space based technique proposed for RkNNQ, which uses the concept of *influence zone* to significantly improve the *verification* phase. Influence zone is the area, such that a point p is an RkNN of q if and only if p lies inside it. Once the influence zone is computed, RkNNQ can be answered by locating the points lying inside it. In [Yang et al., 2014], the *SLICE* algorithm is proposed, which improves the filtering power of *Six-Regions* approach while utilizing its strength of being a cheaper filtering strategy. In [Yang et al., 2015] a comprehensive set of experiments to compare some of the most representative and efficient RkNNQ algorithms under various settings is presented, and the authors propose an optimized version of TPL (called TPL++) for arbitrary dimensionality RkNNQs. One of the main conclusions of this comparative research study is that *SLICE* is the state-of-the-art RkNNQ algorithm since it is the best for all considered performance parameters in terms of CPU cost.

With the fast increase in the scale of large input datasets, processing such datasets in parallel and distributed frameworks is becoming a popular practice. However, there is not much work in developing efficient RkNNQ algorithms in DSDMSs. The only contributions that have been implemented in MapReduce frameworks are [Akdogan et al., 2010, Ji et al., 2013, Ji et al., 2015]. In [Akdogan et al., 2010], the MRVoronoi algorithm is presented, which adopts the Voronoi-Diagram partitioning-based approach and applies MapReduce to answer *RNNQ* and other spatial queries. In [Ji et al., 2013], the Basic MapReduce RkNNQ method based on the *inverted grid index* over large-scale datasets is investigated. An optimization method, Lazy-MapReduce RkNNQ algorithm, that prunes the search space when all data points are discovered, is also proposed. In [Ji et al., 2015] several improvements of [Ji et al., 2013] have been presented. For instance, a novel decouple method is proposed to decomposes pruning-verification into independent steps, and it can increase opportunities for parallelism. Moreover, new optimizations to minimize the network and disk input/output cost of distributed processing systems have been also investigated.

2.3.4 kNearest Neighbor Join Query

The kNearest Neighbor Join Query (kNNJQ) [Böhm and Krebs, 2004] is a type of DJQ where, given two datasets of points ($\mathbb{P}$ and $\mathbb{Q}$) and a positive number k, it finds for each point of $\mathbb{P}$, its k nearest neighbors in $\mathbb{Q}$. The formal definition of this kind of DJQ is given below.

Definition 2.4. k**Nearest Neighbor Join query,** ***kNNJ*** **query**
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ *and* $\mathbb{Q} = \{q_1, q_2, \cdots, q_m\}$ *be two sets of points in* E^d*, and a natural number* $k \in \mathbb{N}^+$ $(k > 0)$*. Then, the result of the kNearest Neighbor Join query is a set* $kNNJ(\mathbb{P}, \mathbb{Q}, k) \subseteq \mathbb{P} \times \mathbb{Q}$*, which contains for each point of* $\mathbb{P}$ $(p_i \in \mathbb{P})$ *its k nearest neighbors in* $\mathbb{Q}$*:*
$kNNJ(\mathbb{P}, \mathbb{Q}, k) = \{(p_i, q_j) : \ \forall\ p_i \in \mathbb{P},\ q_j \in kNN(\mathbb{Q}, p_i, k)\}$

The most important properties of kNNJQ are the following:

1. kNNJQ is **asymmetric**, i.e., $kNNJ(\mathbb{P}, \mathbb{Q}, k) \neq kNNJ(\mathbb{Q}, \mathbb{P}, k)$, since kNNQ is asymmetric (i.e., if $kNN(\mathbb{P}, q, 1) = \{p\}$, there may exist another $q' \in \mathbb{Q}$ $(q' \neq q)$ such that $dist(p, q') < dist(p, q)$ and $kNN(\mathbb{Q}, p, 1) = \{q'\}$). Moreover, given $\mathbb{P} \neq \mathbb{Q}$ $(|\mathbb{P}| \neq |\mathbb{Q}|)$, the cardinality of $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ is $k \times |\mathbb{P}|$ (similarly $|kNNJ(\mathbb{Q}, \mathbb{P}, k)|$ $= k \times |\mathbb{Q}|$), and therefore the results are different. In the case of $|\mathbb{P}| = |\mathbb{Q}|$ $(\mathbb{P} \neq \mathbb{Q})$, although the cardinalities of the results are the same, the content is different, $kNNJ(\mathbb{P}, \mathbb{Q}, k) \neq kNNJ(\mathbb{Q}, \mathbb{P}, k)$, due to kNNQ is asymmetric.

2. Given $k \ll |\mathbb{Q}|$, the cardinality of $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ is predictable $(|\mathbb{P}| \times k)$, since it returns k nearest neighbors in $\mathbb{Q}$ for each point of $\mathbb{P}$.

3. The distance from each point of $\mathbb{P}$ to its k nearest neighbors is unknown a priori.

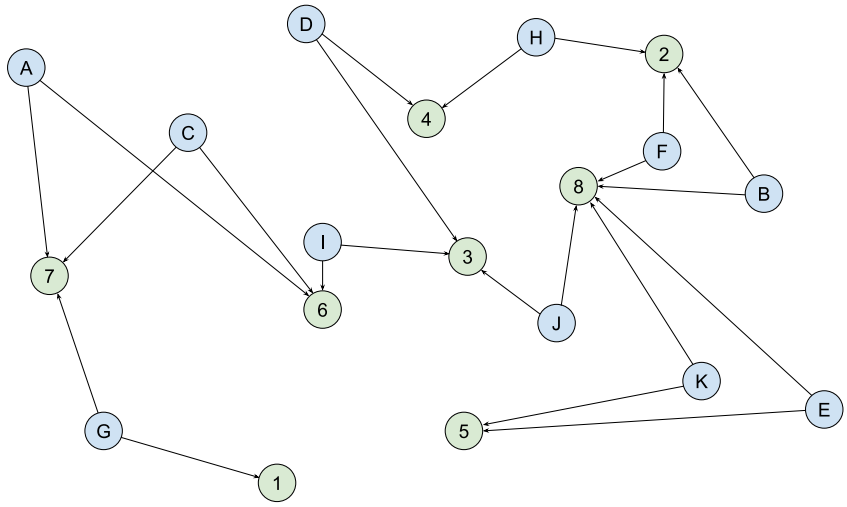


Figure 2.14: kNearest Neighbor Join query with $k = 2$.

One **application case** (Mobile Location Services), with two spatial datasets, locations of shopping centers and positions of possible customers using a smart phone with mobile data and GPS enabled. kNNJQ could *find the 100 nearest possible customers to each shopping center* for sending an advertising SMS about a fashion brand available there. Figure 2.14 represents this example with the numbered circles being the

© 2021 García-García, F.J.

customers ($\mathbb{P}$) and the lettered circles being the shopping centers ($\mathbb{Q}$). To reduce complexity, k value is 2 and the result of the query is $kNNJ(\mathbb{P}, \mathbb{Q}, 2) = \{(7, A), (6, A), (8, B), (2, B), (7, C), (6, C), (4, D), (3, D), (8, E), (5, E), (8, F), (2, F), (7, G), (1, G), (2, H), (4, H), (6, I), (3, I), (3, J), (8, J), (8, K), (5, K)\}$.

Since, the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ returns for each point in $\mathbb{P}$, its kNNs in $\mathbb{Q}$; it is equivalent to the query called *All-k-Nearest Neighbor* (AkNN) query [Zhang et al., 2004, Chen and Patel, 2007] in the context of DJQs. Several research works have been devoted to improve the performance of this DJQ by proposing efficient algorithms and specialized index structures in centralized environments [Xia et al., 2004, Chen and Patel, 2007, Emrich et al., 2010].

The kNNJQ MapReduce algorithm has been extensively studied in the literature [Nodarakis et al., 2016b], and the most representative contributions are the following. In [Lu et al., 2012], the problem of answering the kNNJ using MapReduce is extensively analyzed and solved. This is achieved by exploiting the Voronoi-Diagram based partitioning method, which divides the input datasets into groups, such that kNNJ can answer by only checking object pairs within each group. Moreover, several pruning rules to reduce the shuffling cost as well as the computation cost are developed in the PGBJ (Partitioning and Grouping Block Join) algorithm, which works with two MapReduce phases. In [Zhang et al., 2012], the authors propose novel (exact and approximate) algorithms in MapReduce to perform efficient parallel kNNJQ on large datasets, and they use the R-tree, and Z-value-based partition joins to implement them. In [Song et al., 2016], the existing solutions that perform the kNNJ operation in the context of MapReduce are reviewed and studied from the theoretical and experimental point of view. In [Yokoyama et al., 2012], a kNNJQ MapReduce algorithm for 2d spatial data is presented. It decomposes the data space into small equal cells (Grid), and afterward, it merges some neighboring cells, always in 2×2 sets, if they do not contain k points or more in total. In this way, the algorithm creates bigger cells so that the kNN list of a query point will always be complete. In [Nodarakis et al., 2016a], the algorithm of [Yokoyama et al., 2012] is improved by replacing the merging step with a circle of increasing radius around the query point so that it checks for candidate neighbors in nearby cells. In this way, the merging step is not needed, and the number of distance calculations may be significantly reduced. In [Moutafis et al., 2019b], the work presented in [Nodarakis et al., 2016a] has been extended. The *information distribution* phase has been implemented by Quadtrees, utilizing dataset sampling to capture the skewness of data distribution, to balance the load, and to free the end-user from having to refine parameters of data partitioning. The *primitive computation* phase has employed plane-sweep to reduce distance calculations. The *update lists* and the *unify lists* phases have been restructured to reduce network traffic.

2.3.5 εDistance Range Join Query

The εDistance Range Join Query (εDRJQ), given two datasets of points ($\mathbb{P}$ and $\mathbb{Q}$) and a distance threshold ε, finds, for each point $p_i \in \mathbb{P}$, all points in $\mathbb{Q}$ that fall on the circular shape, centered in p_i with radius ε. This query is also called *spatial range join query*. The formal definition is as follows.

Definition 2.5. εDistance Range Join query, εDRJ query
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ and $\mathbb{Q} = \{q_1, q_2, \cdots, q_m\}$ be two sets of points in E^d, and a distance threshold $\varepsilon \in \mathbb{R}^+$ ($\varepsilon > 0$). Then, the result of the εDistance Range Join query is a set, $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon) \subseteq \mathbb{P} \times \mathbb{Q}$, which contains for each point of $\mathbb{P}$ ($p_i \in \mathbb{P}$) all points from $\mathbb{Q}$ ($q_j \in \mathbb{Q}$) that fall on the circular shape, centered in p_i with radius ε:
$\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon) = \{(p_i, q_j) : \ \forall\ p_i \in \mathbb{P},\ \forall q_j \in \varepsilon DR(\mathbb{Q}, p_i, \varepsilon)\}$

This query is also related to the *similarity join* in multidimensional databases, where the problem of deciding if two objects are similar is reduced to the problem of determining if two multidimensional points are within a certain distance of each other. In the MapReduce framework, the most representative work is [Silva and Reed, 2012], where a partition-based similarity join for MapReduce is proposed (called $MRSimJoin$). This approach is based on the $QuickJoin$ algorithm [Jacox and Samet, 2008], and iteratively partitions the data until each partition can be processed on a single reducer (i.e., it partitions and distributes the data until the subsets are small enough to be processed in a single node).

One **application case** (Resource Management in Agriculture), authorities planning the sustainable exploitation of water resources are considering two spatial datasets, locations of water wells and areas of cultivable lands, εDRJQ could *find all land areas within 3 Km from every water well* (the borders or the centroid of each land area could be used for processing this query). Figure 2.15 represents this example with the numbered circles ($\mathbb{Q}$) being the cultivable lands, the lettered circles ($\mathbb{P}$) being the water wells, and ε value is 3 Km. Therefore, the result of the query is $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, 3) = \{(I, 6), (F, 8)\}$.

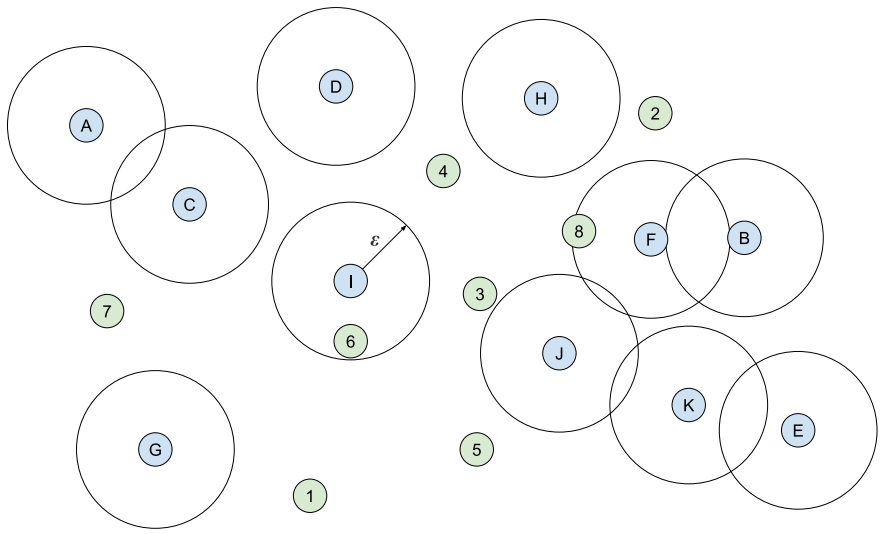


Figure 2.15: εDistance Range Join query with distance ε.

© 2021 García-García, F.J.

2.3.6 kClosest Pairs Query

The kClosest Pairs Query (kCPQ) discovers the k pairs of points formed from two datasets ($\mathbb{P}$ and $\mathbb{Q}$) having the k smallest distances between them (i.e., it reports only the top k pairs). This query is also called *kDistance Join Query*. The formal definition of this DJQ is as follows.

Definition 2.6. kClosest Pairs query, kCP query
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ and $\mathbb{Q} = \{q_1, q_2, \cdots, q_m\}$ be two sets of points in E^d, and a natural number $k \in \mathbb{N}^+$ $(k > 0)$. Then, the result of the kClosest Pairs query is an ordered collection, $kCP(\mathbb{P}, \mathbb{Q}, k)$, containing k different pairs of points from $\mathbb{P} \times \mathbb{Q}$, ordered by distance, with the k smallest distances between all possible pairs:
$kCP(\mathbb{P}, \mathbb{Q}, k) = ((p_1, q_1), (p_2, q_2), \cdots, (p_k, q_k))$, $(p_i, q_i) \in \mathbb{P} \times \mathbb{Q}$, $1 \leq i \leq k$, *such that for any* $(p, q) \in \mathbb{P} \times \mathbb{Q} \setminus kCP(\mathbb{P}, \mathbb{Q}, k)$ *we have* $dist(p_1, q_1) \leq dist(p_2, q_2) \leq \cdots \leq dist(p_k, q_k) \leq dist(p, q)$.

kCPQ has the following properties:

1. kCPQ is symmetric, i.e., $kCP(\mathbb{P}, \mathbb{Q}, k) = kCP(\mathbb{Q}, \mathbb{P}, k)$, since it discovers the k pairs of points with the k smallest distances from all possible pairs that can be formed from the join of two datasets, and the Euclidean distance is symmetric $dist(p_i, q_j) = dist(q_j, p_i)$.

2. The cardinality of the result is known beforehand $|kCP(\mathbb{P}, \mathbb{Q}, k)| = k$.

3. The distance of the k closest pairs of points is unknown a priori.

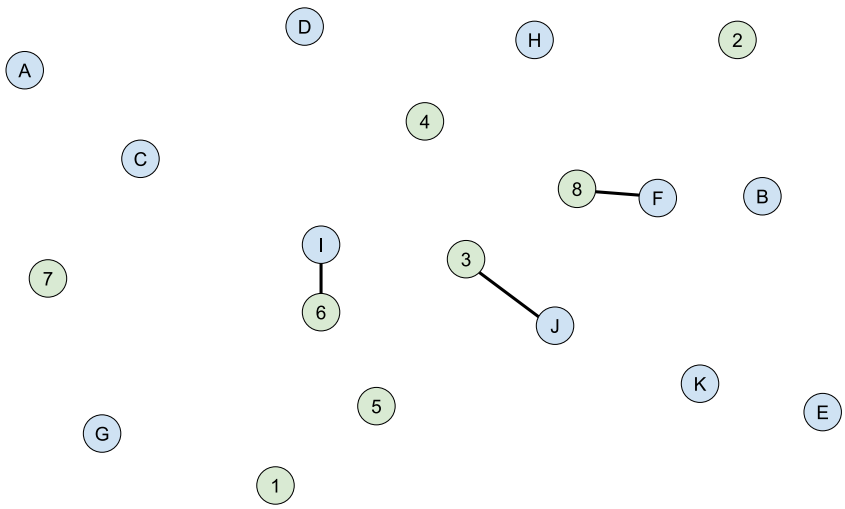


Figure 2.16: kClosest Pairs query with $k = 3$.

One **application case** (Transportation Monitoring and Moving Objects), considering two spatial datasets, locations of users of a taxi app and positions of free taxis, kCPQ could *find the 3 pairs of app users and taxis with the shortest distances between them*, to be able to offer these users fast service at a reduced price (as a promotion strategy), or for analysis by the taxi service. Figure 2.16 represents this example with the numbered circles ($\mathbb{Q}$) being the taxis and the lettered circles ($\mathbb{P}$) being the users. For $k = 3$ the result of the query is $kCP(\mathbb{P}, \mathbb{Q}, 3) = ((I, 6), (J, 3), (F, 8))$.

This spatial query has been actively studied in centralized environments, regardless whether both spatial datasets are indexed or not [Corral et al., 2004a, Roumelis et al., 2014, Roumelis et al., 2016, Corral et al., 2000, Corral, 2002, Corral et al., 2006, Hjaltason and Samet, 1998, Shin et al., 2003, Yang and Lin, 2002, Kim and Patel, 2010, Gutierrez and Sáez, 2013]. If both $\mathbb{P}$ and $\mathbb{Q}$ are indexed by R-trees, the concept of synchronous tree traversal and Depth-First (DF) or Best-First (BF) traversal order can be combined for the query processing [Hjaltason and Samet, 1998, Corral et al., 2000, Corral, 2002, Corral et al., 2004a]. For a more detailed explanation of the processing of kCPQ-DF and kCPQ-BF algorithms on two R*-trees from the non-incremental point of view, see [Corral et al., 2004a]. In [Hjaltason and Samet, 1998], incremental and non-recursive algorithms based on Best-First traversal using R-trees and additional priority queues for DJQs were presented. In [Shin et al., 2003], sophisticated techniques, such as sorting and application of plane-sweep during the expansion of node pairs and the use of the estimation of the distance of the k-th closest pair to avoid the computations of unnecessary MBR distances, are included to improve the algorithms proposed in [Hjaltason and Samet, 1998]. In [Yang and Lin, 2002], a new index structure, the bichromatic Rddn-tree (bRddn-tree), is proposed for improving kCPQ and related DJQs by keeping track of the nearest neighbor distance for each data point. Consequently, this index prunes the search path more efficiently and allows the implementation of several algorithms in a non-incremental (DF) way. This new index, similar to Rddn-tree [Yang and Lin, 2001] for the RkNNQ, outperformed R*-tree for kCPQ with respect to the number of disk accesses. In [Corral and Almendros-Jimenez, 2007], a Recursive Best-First Search (RBF) algorithm for DBQs (kNNQ, εDRQ, kCPQ, εDJQ) between spatial objects indexed in R-trees is presented with an exhaustive experimental study that compares DF, BF, and RBF for several DBQs. In [Kim and Patel, 2010], an extensive experimental study comparing the R*-tree and Quadtree-like index structures for kNNJQ and kCPQ (also called kDistance Join query) together with index construction methods (dynamic insertion and bulk-loading algorithm) is presented. It was shown that when spatial data are static, the R*-tree shows the best performance. However, when spatial data are dynamic, a bucket-Quadtree begins to outperform the R*-tree. This is due to, once the dynamic R*-tree algorithm is used, the overlap among MBRs grows with the increment of the dataset sizes, and the R*-tree performance decreases.

In the case where just only one dataset is indexed ($\mathbb{P}$ or $\mathbb{Q}$), in [Gutierrez and Sáez, 2013] an algorithm has been proposed for kCPQ. The main idea is to partition the space occupied by the dataset without an index into several cells or subspaces (according to the VA-File index structure [Weber et al., 1998]) and to make use of the properties of a set of distance functions defined between two MBRs [Corral et al., 2004a].

When the two datasets are not indexed and stored in main-memory or disk, a new

© 2021 García-García, F.J.

plane-sweep algorithm for kCPQ, called *Reverse Run*, was proposed in [Roumelis et al., 2014, Roumelis et al., 2016]. Two improvements on the *Classic plane-sweep* algorithm for this spatial query were presented as well. Experimentally, the *Reverse Run plane-sweep* algorithm proved to be faster since it minimized the number of Euclidean distance computations. However, when the datasets reside in a parallel and distributed framework like SpatialHadoop, the *Classic* and *Reverse Run* plane-sweep algorithms had similar results in terms of execution times [García-García et al., 2016b, García-García et al., 2018b].

2.3.7 εDistance Join Query

The εDistance Join Query (εDJQ) finds all possible pairs of points from two datasets that are within a distance threshold ε of each other. The formal definition of this query is given below.

Definition 2.7. εDistance Join query, εDJ query
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ and $\mathbb{Q} = \{q_1, q_2, \cdots, q_m\}$ be two sets of points in E^d, and a distance threshold $\varepsilon \in \mathbb{R}^+$ ($\varepsilon > 0$). Then, the result of the εDistance Join query is the set, $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon) \subseteq \mathbb{P} \times \mathbb{Q}$, containing all possible different pairs of points from $\mathbb{P} \times \mathbb{Q}$ that have a distance of each other smaller than, or equal to ε:
$\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon) = \{(p_i, q_j) \in P \times Q \ : \ dist(p_i, q_j) \ \leq \ \varepsilon\}$

Note that εDJQ can be considered as an extension of the kCPQ, where the distance threshold of the pairs (ε) is known beforehand. Therefore, studying the results of εDRJQ and εDJQ, we note that they are equivalent [García-García et al., 2020c], i.e., both DJQs report the same result set ($\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon) \equiv \varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$). The main difference resides in the order of the pairs returned in the final result. While $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports pairs in a clustered way around every point of $\mathbb{P}$ (i.e., for each point $p_i \in \mathbb{P}$, it returns all points in $\mathbb{Q}$ overlapping with a circular shape, centered in p_i with radius ε), $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports pairs of points without any relationship among them (i.e., it returns a sequence of pairs of points within a distance threshold (ε) of each other). Another difference between both DJQs is the algorithmic technique used to solve them. While the processing method of εDRJQ is based on multiple executions of εDRQ on $\mathbb{Q}$ for every point in $\mathbb{P}$, the algorithm to solve εDJQ is based on a sort-merge join approach (i.e., it is a plane-sweep algorithm between $\mathbb{P}$ and $\mathbb{Q}$).

2.3.8 Other related Distance Join Queries

Other related DJQs can be deduced from the previous ones. For instance,

- *εDistance Range k Query*, that returns the k points from $\mathbb{Q}$ with the smallest distances within the specified distance threshold ε around each query point $p_i \in \mathbb{P}$.
- *εDistance Join k Query*, that returns only the k pairs with the smallest distances from all possible different pairs of points, having a distance less than or equal to ε of each other.

- *Iceberg Distance Join Query* [Shou et al., 2003], that returns object pairs (p_i, q_j) such $p_i \in \mathbb{P}$ and $q_j \in \mathbb{Q}$, within distance ε of each other, provided that p_i appears at least k times in the join result.

- *Closest Pair Queries with Spatial Constraints* [Papadopoulos et al., 2006], that studies constrained closest-pair queries, between two distinct datasets $\mathbb{P}$ and $\mathbb{Q}$, where objects from $\mathbb{P}$ must be enclosed by a spatial region $\mathbb{R}$.

- *kMulti-Way Distance Join Query* (*kMWDJQ*) [Corral et al., 2004b] is a multi-way spatial join that finds the k n-tuples of points among n spatial datasets that have the k smallest $D_{distance}$ values. An n-tuple is a set of n points from n different datasets that obeys a query graph. The $D_{distance}$ value of an n-tuple is the result of a linear function of distances of the n points that make up this n-tuple, according to the edges of the query graph. Finally, the query graph, in this case, is a weighted directed graph where the nodes represent datasets of points, and edges correspond to distance functions.

- An interesting extension of RkNNQ is the *Reverse kNearest Neighbor Join Query* (RkNNJQ), where the query does not consist of a single query point but a whole set of query points, and for each of which an RkNNQ has to be performed [Emrich et al., 2013b]. Despite the potential applications of this kind of join operation, in the context of databases and decision-making applications, it has only received little attention in the literature [Emrich et al., 2013b, Emrich et al., 2013a, Emrich et al., 2015].

- In a similar situation that RkNNJQ, we find the *Spatial Reverse Top-k Query* (SRTkQ) [Yang et al., 2017], which is also an interesting extension of RkNNQ that, given a linear scoring function W that computes the score of a facility f for a given point p. A *SRTkQ* returns every point $p \in \mathbb{P}$ for which q is one of the top-k facilities according to the scoring function W.

All these DJQs are considered extensions of the main DBQs studied in the previous sections, and they could be regarded as a target of further research in the context of DSDMSs.

2.4 Conclusions

This section summarizes the main conclusions of this chapter. It introduces a detailed description of the state-of-the-art and the needed background. First, the most relevant Distributed Spatial Data Management Systems (DSDMSs) are exposed in Section 2.1. These DSDMSs are classified as disk-based (DSDMSs based on Hadoop) or in-memory-based (DSDMSs based on Spark). Next, we review the most relevant spatial partitioning techniques used in DSDMSs. These partitioning techniques are organized into four categories: space-based, data-based, space-filling curve-based, and distance-based. Finally, we describe and formally define the most representative distance-based queries (DBQs). These DBQs can be classified if only one dataset is involved in the query or two datasets

© 2021 García-García, F.J.

are combined, taking into account a distance metric. For instance, εDRQ, kNNQ, and RkNNQ are DBQs where only one dataset is taken in, while εDRJQ, kCPQ, kNNJQ, and εDJQ are clear examples of Distance-based Join Queries (DJQs) where two datasets are combined. Furthermore, several application cases have been shown to help in the understanding of each query.

Chapter 3

Spatial Partitioning and Indexing in SpatialHadoop

© 2021 García-García, F.J.

Chapter 3

Spatial Partitioning and Indexing in SpatialHadoop

Contents

In this chapter, the structure and operations of spatial partitioning and indexing in SpatialHadoop are detailed. First, Section 3.1 presents the general spatial partitioning scheme in SpatialHadoop together with its spatial indexing mechanism for enabling fast access to spatial data in Hadoop. Next, Section 3.2 exposes the spatial partitioning techniques already implemented in SpatialHadoop. Moreover, the spatial indexes available in SpatialHadoop are described in Section 3.3. Then, Section 3.4 proposes a data partitioning technique based on Voronoi-Diagrams in SpatialHadoop. Furthermore, the motivation and the process to include the Quadtree as a local index in SpatialHadoop is discussed in Section 3.5. Finally, Section 3.6 offers an experimental evaluation of the Quadtree-based partitioning technique and a comparison with the new Voronoi-Diagram based technique for kNNJQ and kCPQ, and Quadtree-based local index for top-k query algorithms (kNNQ and kCPQ).

3.1 Spatial Partitioning and Indexing in SpatialHadoop

Spatial Partitioning is a powerful and crucial mechanism for enabling fast access to spatial data in a distributed system like Hadoop. SpatialHadoop implements several spatial partitioning techniques adapted to HDFS (Hadoop Distributed File System), so they can be used to support spatial queries in MapReduce. Moreover, the use of *Spatial Indexing* is one of the most common techniques employed to accelerate spatial query processing. Therefore, to tackle the building of spatial indexes in Hadoop, SpatialHadoop uses a two-layers indexing approach of *global* and *local* indexes. Each index contains one *global* index, stored in the master node, that partitions the spatial dataset across a set of partitions. Each partition is stored in slave nodes with a *local* index, organizing the data of such partition. The main advantage of this structure is that each partition can be treated in parallel by a slave node. To make this *two-level index structure* accessible to MapReduce programs, SpatialHadoop introduces two new components in the MapReduce layer: *SpatialFileSplitter* and *SpatialRecordReader*. The *SpatialFileSplitter* allows us to access to the *global* index to select only the partitions needed for the current query, and the *SpatialRecordReader* provides the *local* index of each selected partition as input to the *map* task to enable quick access to spatial data. Therefore, in order to effectively process a spatial dataset in SpatialHadoop, the following preliminary phases must be carried out: *Partitioning*, *Local Indexing* and *Global Indexing*.

Firstly, the *Partitioning* phase splits the input dataset into several partitions by a particular partitioning method. Moreover, each partition must meet a series of restrictions, based on the **factors** presented in Section 2.2, to obtain the best possible performance:

1. Spatial objects that are close to each other (**F.3**) must be assigned to the same partition, and partitions that are close to each other will contain objects close to each other.

© 2021 García-García, F.J.

2. Partitions should be approximately the same size (**F.3**) to avoid skew problems and balance the workload as best as possible.

3. The size of each partition must be less than the HDFS block size (**F.5**) to prevent it from being split by Hadoop in its block duplication process.

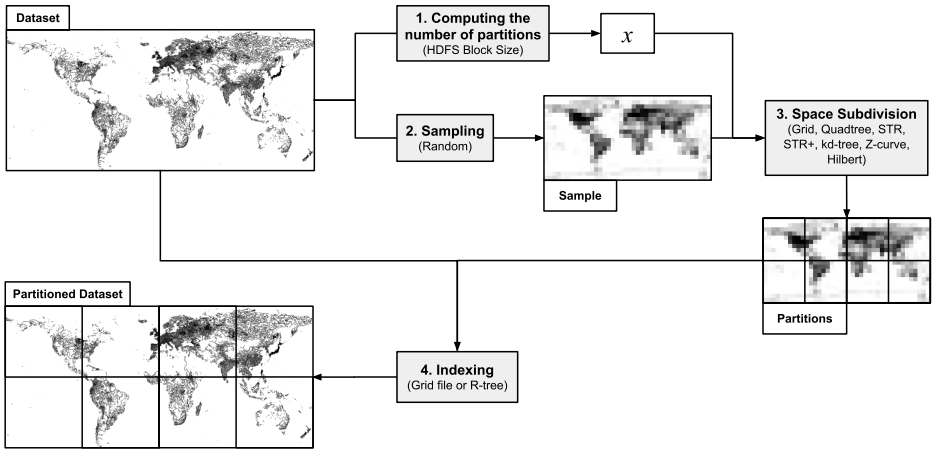


Figure 3.1: Spatial Partitioning phase in SpatialHadoop.

Considering these restrictions, Figure 3.1 shows the four steps of the *Spatial Partitioning* phase in SpatialHadoop [Eldawy and Mokbel, 2015, Eldawy et al., 2015]:

1. *Computing the number of partitions.* The first step computes the number of desired partitions x based on file size and HDFS block capacity, which are both fixed for all spatial partitioning techniques.

2. *Sampling.* The second step reads a random sample, with a sampling ratio ρ, from the input file.

3. *Space subdivision.* The third step uses the sample to partition the space into x cells or partitions, such that the number of sample points in each partition is at most $\lfloor s/x \rfloor$, where s is the sample size.

4. *Indexing.* The fourth step partitions the input file by assigning each point to one or more partitions, i.e., every partition becomes a file that is duplicated to the number of nodes defined by the Hadoop cluster replication factor.

Next, the *Local Indexing* phase builds the requested spatial index as a local index (e.g., Grid file or R-tree) on the spatial data contents of each physical partition. Finally, the *Global Indexing* phase merges the information of all local indexes to generate the required global index.

3.2 Spatial Partitioning Techniques in SpatialHadoop

The *Partitioning* phase in SpatialHadoop divides the input dataset into serveral partitions (*Space subdivision* in Figure 3.1) by a particular spatial partitioning technique. In [Eldawy et al., 2015], seven different *spatial partitioning techniques* in SpatialHadoop are presented, and an extensive experimental study on the quality of the generated partitions and the performance of range and spatial join queries is reported. They are classified as *space-based* (Grid and Quadtree), *data-based* (STR, STR+ and kd-tree) and *space-filling curve-based* (Z-curve and Hilbert-curve) partitioning strategies. Furthermore, these seven partitioning techniques are also classified in two categories according to boundary object handling: *replication-based techniques* (Grid, Quadtree, STR+ and kd-tree) and *distribution-based techniques* (STR, Z-curve and Hilbert-curve) [Eldawy et al., 2015]. The *distribution-based techniques* assign an object to exactly one overlapping partition and the partition has to be expanded to enclose all contained points. The *replication-based techniques* avoid expanding partitions by replicating each point to all overlapping partitions, but the query processor has to employ a duplicate avoidance technique to account for replicated elements.

SpatialHadoop supports seven spatial partitioning strategies to handle large-scale spatial data:

- *Grid-based partitioning.* It consists of the division of the spatial dataset into a uniform grid of x cells of the same size. The main advantage of this method is that the *Partitioning* phase does not need any *Sampling* step or other preprocessing methods. Therefore, in order to obtain the partition boundaries, the *Space subdivision* step splits the MBR of the spatial dataset in $\sqrt{x}$ rows $\times$ $\sqrt{x}$ columns. Finally, the *Indexing* step replicates each spatial object to the partitions it overlaps using a *Grid file* for each partition.

- *Quadtree-based partitioning.* This method employs a Quadtree to recursively partition a two-dimensional dataset in 4 quadrants or regions. First, the *Sampling* step takes a sample of spatial objects from the input dataset. Next, in the *Space subdivision* step, the centroid of each sampled spatial object is bulk loaded into an in-memory Quadtree, with a leaf node capacity of $\lfloor s/x \rfloor$, using the PR-Quadtree bulk loading algorithm [Hjaltason and Samet, 1999], which works in a bottom-up fashion. This technique sorts the centroids using a Z-curve so that spatial objects found on a leaf node appear consecutively. Then, the algorithm considers that all spatial objects belong to the root node and checks whether there are nodes to divide. A node is divided into its four children quadrants if the current number of spatial objects exceeds the established node capacity. Once the Quadtree is built using the sample, the spatial dataset is partitioned using the boundaries of the leaf nodes. Therefore, the *Indexing* step uses the Quadtree to find the leaf nodes or partitions a spatial object overlaps with and replicates it using a *Grid file* for each partition. Figure 3.2 shows the partitions of a real-world dataset of 115M records (points) of buildings with Quadtree-based partitioning. Note that the partitions

© 2021 García-García, F.J.

are not exactly quadrants due to the MBRs being adjusted to the existing records.

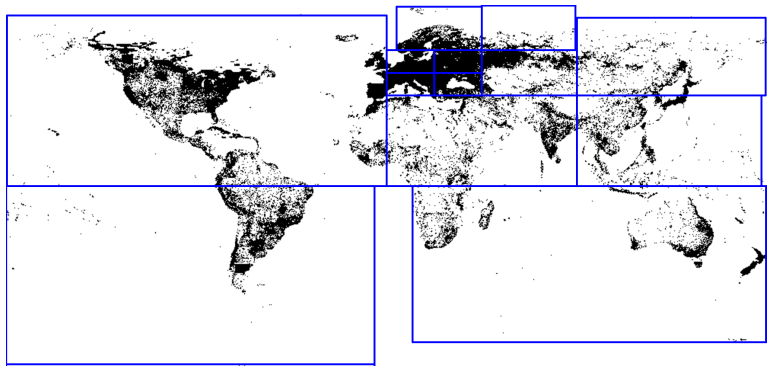


Figure 3.2: Real-world dataset of 115M records of buildings with Quadtree-based partitioning.

– *STR-based partitioning.* It gets the partitions from the spatial dataset by using an in-memory R-tree. To do this, it uses the Sort-Tile-Recursive (STR) bulk loading algorithm [Leutenegger et al., 1997] with the centroids of the sample obtained in the *Sampling* step. The degree of the R-tree to build in the *Space subdivision* step is set to $\sqrt{x}$ in order to get at least x nodes in the second level of the tree. Next, using the MBRs of the latter as the actual partitions, the *Indexing* step assigns each spatial object to the partition that has a larger intersection area to avoid replication. Moreover, the boundaries of each of the partitions are enlarged to contain all spatial objects they have assigned. Finally, this partitioning technique receives two different names depending on the *local index* used: *STR* when using a *Grid file* for each partition, or *R-tree* when an individual R-tree is populated per partition. Figure 3.3 shows the partitions of a real-world dataset of 115M records (points) of buildings with STR-based partitioning. Note that the partitions are less regular than those of Quadtree-based partitioning.

– *STR+-based partitioning.* This method performs a similar process to the STR-based partitioning but using an R^+-tree [Sellis et al., 1987]. The main difference of this type of tree is that it allows us to have disjoint nodes at each level of the tree. Also, spatial objects that overlap with multiple nodes or partitions are replicated to them because it is a replication-based technique. Therefore, the *Indexing* step does not make any modification of the partition boundaries. In the same way, as for STR-based partitioning, this technique presents two different variants depending on the *local index* used: *STR+* when using a *Grid file* for each partition, or R^+-tree when an individual R^+-tree is populated per partition.

– *kd-tree-based partitioning.* This technique partitions a spatial dataset by means of a *k*d-tree [Bentley, 1975]. Given a sample obtained in the *Sampling* step and

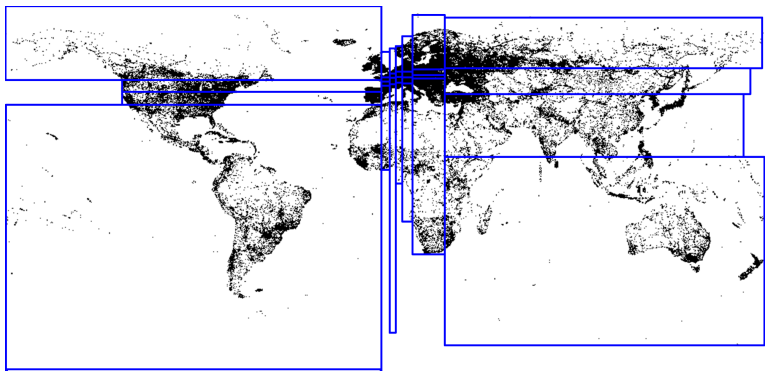


Figure 3.3: Real-world dataset of 115M records of buildings with STR-based partitioning.

the MBR of the dataset as the initial node, the *Space subdivision* step recursively partitions $x-1$ times a kd-tree in order to get at least x leaf nodes. Moreover, in each step of the subdivision, it alternates between horizontal and vertical planes that seek to keep a balanced number of spatial objects in the leaf nodes. Besides, the technique uses the median spatial object of the respective axis for the node to split to achieve this balance. This partitioning technique is also known as Binary Space Partitioning (BSP) [Fuchs et al., 1980]. Finally, the *Indexing* step uses the boundaries of the leaf nodes as the partitions and replicates each spatial object to the ones it overlaps using a *Grid file*.

– *Z-curve-based partitioning.* This method uses several properties of the Z-curve to partition a spatial dataset [Mokbel et al., 2003]. First, the spatial objects of the sample obtained in the *Sampling* step are sorted based on the order of their centroid on the Z-curve. Then, the *Space subdivision* step divides this set of centroids into x subsets of equal size (containing roughly $\lfloor s/x \rfloor$ spatial objects) for each one of the partitions to generate. Next, the *Indexing* step uses the centroid of each subset as a pivot and assigns each spatial object to the partition of the closest one. Finally, the boundaries of each partition are calculated based on the contained spatial objects, and the *Grid files* are written to HDFS. Note that this partitioning technique presents several overlaps between the generated partitions due to the partial loss of spatial locality that presents the Z-curve.

– *Hilbert-curve-based partitioning.* This partitioning technique is similar to the Z-curve-based partitioning, but employing a Hilbert-curve (H-curve) [Hilbert, 1891] instead. Moreover, the overlaps between partitions are reduced thanks to the fact that the Hilbert-curve preserves the locality properties of spatial objects to a greater extent, and it completely avoids the long jumps on the Z-curve.

The most important conclusions of [Eldawy et al., 2015] about the previous partitioning techniques for distributed spatial join processing, using the *overlap* spatial predicate,

© 2021 García-García, F.J.

are the following: (1) the lowest running time is obtained when the same partitioning technique (for both input datasets) is used for the spatial join processing; (2) Quadtree outperforms all other techniques with respect to running time since it minimizes the number of overlapping partitions by employing a regular space partitioning; (3) Z-curve reports the worst running times; and (4) kd-tree gets very similar results to STR.

There are other partitioning methods and techniques implemented in SpatialHadoop but not included in the official release:

- *Spatial coding-based* [Yao et al., 2017]. This partitioning technique does not need the *Sampling* step and processes the full spatial dataset instead. First, the *Space subdivision* step uses a spatial coding matrix (SCM), based on a spatial code (e.g., Hilbert, Grid, etc.), to compress the spatial dataset into a sensing information set (SIS). Moreover, the SIS storages different properties, like the size or spatial object count, for each spatial coded cell. Then, this information combined with other HDFS properties, like the block size, is used to compute a spatial partitioning matrix (SPM) that assigns each spatial code to a block or partition. Finally, in the *Indexing* step, each spatial object is partitioned by calculating its spatial code and looking at the assigned partition in the SPM.

- *2DPR-Tree* [Elashry et al., 2018]. This method uses a two-dimensional Priority R-tree (PR-tree) [Arge et al., 2008] which is an R-tree that presents optimal performance to answer window queries. It uses again the full dataset instead of the *Sampling* step. Basically, the *Space subdivision* step employs the 2DPR-tree to store the partitions as four-dimensional points and adds four extra leaves below each internal node with the actual spatial objects arranged by maximum or minimum values in each axis, e.g., one leaf has points with the lowest X-axis values. Given a spatial dataset $\mathbb{P}$, the node capacity as $\lfloor |\mathbb{P}|/x \rfloor$ and starting with the full dataset MBR as root, the bulk-loading process recursively generates two more child nodes with the exceeding elements once the node is filled and then it starts sorting the elements in the four leaves. The process stops when there are x leaf nodes or partitions. Finally, the *Indexing* step stores the spatial objects of each one as the partitions using a *Grid file*.

- *R*-Grove* [Vu and Eldawy, 2020]. This partitioning technique uses an R*-tree [Beckmann et al., 1990] to obtain high-quality square-like partitions with high load balance and block utilization. To achieve this, it expands the *Sampling* step by optionally building a histogram of storage size that assists in the partitioning algorithm at the *Space subdivision* step. Therefore, this histogram is used to assign a weight to each sample based on the total size of all its neighbors. Next, the *Space subdivision* adapts R*-tree-based algorithms to produce partition boundaries with higher load balance. Basically, it considers all split points and chooses the one that minimizes some cost function which is typically the total area of the two resulting partitions. Finally, in the *Indexing* step, a kd-tree-based structure improves the performance of this step and allows us to replicate each point to all overlapping partitions using disjoint partitions.

To end this section, choosing the best partitioning method is essential to obtain the best results when performing spatial queries. This selection must take into account the different properties and distribution of the spatial dataset to be partitioned. Below, there are two selection methods applied to several partitioning techniques implemented in SpatialHadoop:

- *Skewness-based selection* [Belussi et al., 2020b]. This paper proposes a partitioning technique selection method based on the skewness degree of the input spatial dataset. Furthermore, the detection is based on a box-counting function and a heuristic, as well as several properties and experimental observations to get the best performance while executing some spatial queries over the partitioned spatial dataset. Besides, for the calculation of the box-counting function, a MapReduce algorithm is presented that allows us to obtain two exponents, namely E^0 and E^2, that refer respectively to the existence of empty areas (dataset diffusion) and the concentration of objects in some areas (dataset distribution). Therefore, with these properties, a heuristic is proposed that enables the choice of the most appropriate partitioning technique. In summary, this selection method uses Grid-based partitioning when the distribution is uniform, and otherwise, it uses Quadtree-based partitioning when there is some clustering of the data and R-tree-based partitioning when the data shows some connection.

- *Deep Learning selection* [Vu et al., 2020]. The authors propose a method based on deep learning techniques for selecting the best partitioning technique on a set of these, for instance, kd-tree, R*-Grove, STR, Z-curve, Grid, and RR*-tree. This method consists of a *training phase* and an *application phase*. During the *training phase*, several synthetic datasets are generated to choose which two types of summarization techniques, fractal-based, and histogram-based techniques, are applied. The former is based on the box-counting plots presented in the previous method [Belussi et al., 2020b] along with the Morans index, which is a measure of spatial autocorrelation (e.g., concentration, dispersion), and the number of empty cells. Regarding the latter, a histogram is obtained using a uniform grid that counts the spatial objects within each cell. Together with the summary vector, quality metrics are used to calculate the best partitioning technique for a given dataset and thus be able to train the model. Finally, the *application phase* calculates the summary vector of the dataset to be partitioned, and with the trained model, selects the partitioning technique to apply. The results of different experiments show up to 87% accuracy of the proposed model in recommending the best partitioning technique.

3.3 Spatial Indexing in SpatialHadoop

For spatial indexing, SpatialHadoop uses a *two-level index structure*, composed of *global* and *local* indexes, to accelerate spatial query processing in MapReduce. Figure 3.4 shows how this structure is distributed among the master and slave nodes and the different types of available indexes. On the one hand, each slave node has the data

© 2021 García-García, F.J.

of a particular partition and its associated *local* index, which allows us to speed up the local spatial queries. On the other hand, the master node stores the *global* index, which has the general information of all partitions generated by a selected partitioning method. Moreover, its combined use with the *SpatialFileSplitter* allows us to discard the partitions that are not needed for the current spatial query.

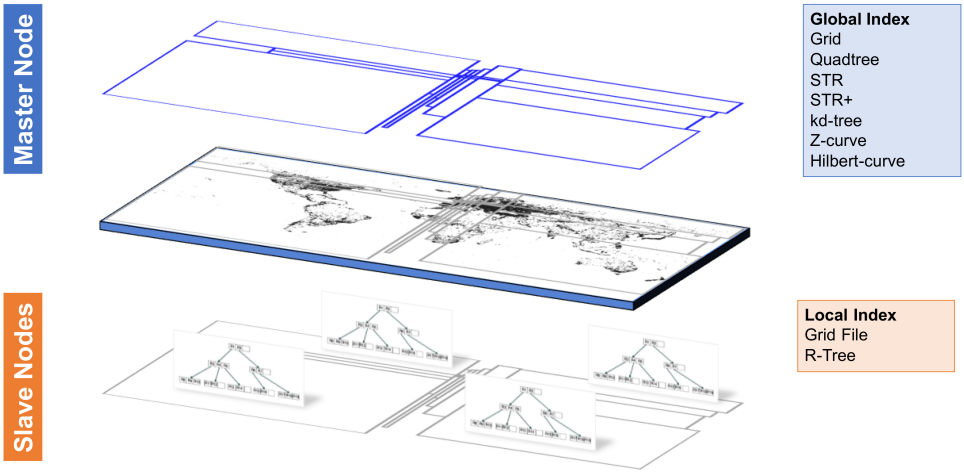


Figure 3.4: A two-level index structure in SpatialHadoop for spatial indexing.

As described in Section 3.1, after the *Partitioning* phase, the *Local Indexing* phase builds the requested spatial index as a local index (e.g., Grid file or R-tree) on the spatial data contents of each physical partition. This process is carried out as a *reduce* function that takes the records assigned to each partition and stores them in a spatial index, written in a local index file. Moreover, the combined size of the records and index structure has to fit in one HDFS block. The block in a local index file is generated with its Minimum Bounding Rectangle (MBR) of its contents, which is calculated while building the local index. The latest released implementation of SpatialHadoop[1] supports Grid files and R-trees as local indexes:

- For the *Grid file*, the records of each grid cell are just written to a heap file without building any local indexes because the grid index is a one-level flat index where contents of each grid cell are stored in no particular order.
- For the *R-tree*, the records of each partition are bulk loaded into an R-tree using the STR algorithm [Leutenegger et al., 1997], which is then dumped into a file.

Finally, the *Global Indexing* phase builds the requested spatial index (e.g., Grid file or R-tree) as a global index structure that indexes all partitions by concatenating the general information of all local index files into one file.

[1]Available at `https://github.com/aseldawy/spatialhadoop2/tree/shadoop-2.4.2`

3.4 Voronoi-Diagram based Partitioning

The *Voronoi-Diagram* is a partitioning method of a geometric space that contains points data. Each data partition of the Voronoi-Diagram (*Voronoi-Cells*) is associated with a generator point or *pivot p*, such that any point inside the partition has p as its nearest neighbor. The resulting data structure from the Voronoi-Diagram is very efficient in exploring a local neighborhood in a geometric space. Voronoi-Diagrams are used in many algorithmic applications, like closest-site problems (nearest neighbor queries and closest pairs), clustering point sites (partitioning and hierarchical clustering methods), placement and motion planning, etc. In our case, the large dataset of points is divided into data partitions based on a Voronoi-Diagram with a careful method for selecting a set of suitable pivots. Then, these data partitions (Voronoi-Cells) are clustered into groups only if the distances between them are restricted by a specific distance bound. This new data partitioning technique in SpatialHadoop is an approach based on *Voronoi-Diagrams* [García-García et al., 2018a, García-García et al., 2020b], and according to [Song et al., 2016] it could be considered as a *distance-based partitioning strategy*. A *Voronoi-Diagram* divides a geometric space into disjoint partitions where the nearest neighbor of any point inside a partition is the *pivot* of the partition. Several related definitions are shown below.

Definition 3.1. Voronoi-Diagram, VD
Let $\mathcal{R} = \{r_1, r_2, \cdots, r_t\}$ *be a set of* t *distinct points in the plane (2D); these points can be called generators or pivots. Then, the Voronoi-Diagram of* $\mathcal{R}$ *is defined as the subdivision of the plane into* r *cells, one for each pivot* r_i *in* $\mathcal{R}$*, with the property that a point* p *lies in the cell corresponding to a pivot* r_i *if and only if* $dist(p, r_i) < dist(p, r_j)$ *for each* $r_j \in \mathcal{R}$ *with* $j \neq i$*. We can denote the Voronoi-Diagram generated by* $\mathcal{R}$ *as* $VD(\mathcal{R})$*.*

Definition 3.2. Voronoi-Cell, V_i
The cell of $VD(\mathcal{R})$ *that corresponds to a pivot* r_i *is called Voronoi-Cell of* r_i *and is denoted by* $VC(r_i)$ *or* V_i *for short. The Voronoi-Diagram of a set of point* $\mathcal{R}$*,* $VD(\mathcal{R})$*, is unique and it also satisfies the following property:* $VD(\mathcal{R}) = \bigcup_{i=1}^{t} V_i$ *and* $\bigcap_{i=1}^{t} V_i = \emptyset$*, where* $V_i = \{p : \ dist(p, r_i) < dist(p, r_j)$ *for* $j \neq i\}$*.*

According to [Lu et al., 2012], given a set of points $\mathbb{P}$, the main idea of *Voronoi-Diagram based partitioning* technique is to select a set $\mathcal{R}$ of points (which may not necessarily belong to $\mathbb{P}$) as *pivots*, and then split the points of $\mathbb{P}$ into $|\mathcal{R}|$ disjoint partitions, where each point is assigned to the partition of its closest pivot r_i in $\mathcal{R}$. In the case of multiple pivots that are closest to a particular point, then that point is assigned to the partition with the smallest number of points. In this way, the whole data space is split into $|\mathcal{R}|$ disjoint Voronoi-Cells. In summary, the set of points are divided into partitions based on a Voronoi-Diagram with carefully selected *pivots*. Then, data partitions (i.e., Voronoi-Cells) are clustered into groups only if the distances between them are restricted by a specific bound.

Moreover, two distance metrics are defined, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$, to be used in DBQ MapReduce algorithms.

© 2021 García-García, F.J.

Definition 3.3. Maximum and Minimum distance $U(\mathcal{P}_i^{\mathbb{P}})$, $L(\mathcal{P}_i^{\mathbb{P}})$
Let $\mathcal{R}$ be the set of selected pivots, $\forall r_i \in \mathcal{R}$, $\mathcal{P}_i^{\mathbb{P}}$ denotes the set of points from $\mathbb{P}$ that has r_i as its closest pivot. We denote $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ as the maximum and minimum distance from the pivot r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$, respectively. That is:
$U(\mathcal{P}_i^{\mathbb{P}}) = \max\{dist(p, r_i) : \forall p \in \mathcal{P}_i^{\mathbb{P}}\}$
$L(\mathcal{P}_i^{\mathbb{P}}) = \min\{dist(p, r_i) : \forall p \in \mathcal{P}_i^{\mathbb{P}}\}$

Table 3.1 shows the symbols and their meanings used throughout this section.

Symbol	Definition
k	number of the NNs or the CPs, $k \geq 1$
$\mathbb{P}$	set of points $\mathbb{P}$
$dist(p_i, q_j)$	distance from p_i to q_j
$\mathcal{S}^{\mathbb{P}}$	sample set from $\mathbb{P}$
ρ	sampling ratio, $0 \leq \rho \leq 1$
$\mathcal{R}^{\mathbb{P}}$	set of selected pivots from $\mathbb{P}$
r_i	a pivot in $\mathcal{R}^{\mathbb{P}}$
$VD(\mathcal{R}^{\mathbb{P}})$	a Voronoi-Diagram of $\mathcal{R}^{\mathbb{P}}$
V_i	a Voronoi-Cell of r_i
$\mathcal{P}^{\mathbb{P}}$	set of partitions from $\mathbb{P}$
$\mathcal{P}_i^{\mathbb{P}}$	subset from $\mathbb{P}$, having r_i as its closest pivot
$U(\mathcal{P}_i^{\mathbb{P}})$	maximum dist. from r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$
$L(\mathcal{P}_i^{\mathbb{P}})$	minimum dist. from r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$
$MBR(\mathcal{P}_i^{\mathbb{P}})$	MBR covering the points of $\mathcal{P}_i^{\mathbb{P}}$

Table 3.1: Symbols and their meanings.

In order to include the new data partitioning technique based on Voronoi-Diagram into SpatialHadoop, we have followed the steps for the *Spatial Partitioning* phase in SpatialHadoop (see Figure 3.1):

1. *Computing the number of partitions.* As usual, the number of desired partitions x is computed based on file size and HDFS block capacity.
2. *Sampling.* A set $\mathcal{S}^{\mathbb{P}}$ of samples from an input dataset $\mathbb{P}$ is provided.
3. *Space subdivision.* A set $\mathcal{R}^{\mathbb{P}}$ of x *pivots* is obtained from the sample set $\mathcal{S}^{\mathbb{P}}$, using some *pivot selection technique*.
4. *Indexing.* The points from the input dataset $\mathbb{P}$ are assigned to their closest *pivot* $r_i \in \mathcal{R}^{\mathbb{P}}$ and some properties of the pivot are calculated and stored in the global index.

3.4.1 Sampling large datasets

Sampling is an effective way to deal with large datasets, which attempts to find a small but representative profile of the dataset. The sample-set is required to be small enough

to satisfy the dataset size constraints and, and at the same time, the result of the sampling should be reliable and close enough to approximately represent the whole dataset. However, sampling methods cannot take into account the correlation among the data hence it is hard to obtain the perfect sample. For example, if we have an input dataset that contains k clusters, ideally, the sample set should also contain k clusters. For this reason, ideal clustering result is difficult to obtain since the clusters cannot be determined easily.

For implementing this new partitioning technique, three sampling methods have been studied: (1) *uniform random sampling* [Eldawy and Mokbel, 2015], it is the simplest and the most common; (2) *partition-based sampling* [Ros and Guillaume, 2017], where the sampling is carried out according to a split of the dataset into a number of disjoint partitions that optimize a criterion function; and (3) *density-based sampling* [Ros and Guillaume, 2016], where distance concepts are managed for sampling to ensure space coverage and fit cluster shapes.

- For *uniform random sampling* on large datasets, the size of the sample is usually set to a ratio between the sample dataset size and its original dataset size [Eldawy and Mokbel, 2015], that is $|\mathcal{S}^{\mathbb{P}}| = \rho \times |\mathbb{P}|$, where $0 \leq \rho \leq 1$ is the ratio of the sampled dataset. In [Zhao et al., 2018], when $0.01 \leq \rho \leq 0.02$, the execution times are minimized for kNNJQ since both small and large sample sizes tend to deteriorate the performance (i.e., small ratios are unable to accurately estimate dataset distribution and large ratios lead to high sampling overhead). In our experiments, we have chosen by default $\rho = 0.01$ (1%), since it was the best ratio value for real datasets when kNNJQ is executed [Zhao et al., 2018], also for the kCPQ performance [García-García et al., 2018b], and it produces high quality partitions in SpatialHadoop [Eldawy et al., 2015]. To generate the random sample efficiently when the input file is very large, SpatialHadoop provides a MapReduce job that scans all records and outputs each one with a probability of 1% ($\rho = 0.01$).

- For *partition-based sampling*, k-means clustering algorithm [MacQueen, 1967] has been successfully used as a preprocessing sampling step for sophisticated and expensive clustering techniques. It is executed with $k = |\mathcal{S}^{\mathbb{P}}|$, where $|\mathcal{S}^{\mathbb{P}}|$ is the desired sample size of $\mathbb{P}$, such as $|\mathcal{S}^{\mathbb{P}}| \ll |\mathbb{P}|$ [Ros and Guillaume, 2017]. For this reason, we can use k-means++ [Arthur and Vassilvitskii, 2007] for sampling purposes. To generate this kind of sample efficiently from a large dataset, we have implemented a MapReduce job, where the input dataset is split into a number of necessary parts to fit in the main memory of the *mappers*. Therefore, in the *map* phase, each $mapper_i$ performs the k-means++ algorithm from ELKI library [Schubert and Zimek, 2019] on its part with $k_i = s_i$, where s_i is the number of points resulting from applying the ratio ρ on the number of points that such $mapper_i$ receives. The final result of this MapReduce job is the combination of the partial results of applying k-means++ in each mapper. The study of the theoretical analysis of error bounds of sampling to select the pivots for partitions in metric similarity join in MapReduce can be found in [Wu et al., 2019]. In addition, in [Blömer et al., 2016], the study of the seeding methods for the k-means algorithm is presented, providing also the lower bound on the expected error of picking k

© 2021 García-García, F.J.

initial centers for the k-means algorithm. According to [Arthur and Vassilvitskii, 2007], the k-means method does not perform well since the random seeding will inevitably merge clusters together, and the algorithm will never be able to split them apart. The careful seeding method of k-means++ avoids this problem altogether, and it almost always attains the optimal results.

– For *density-based sampling*, we will use the DENDIS clustering algorithm [Ros and Guillaume, 2016] since it combines both DENsity and DIStance concepts to ensure space coverage and fit cluster shapes. In general, at each step of the algorithm, a new point is added to the sample, choosing the furthest from the representative in the most important group. Like the previous sampling cases, we have implemented a MapReduce job, where the input dataset is also split into a number of necessary parts whose size can be processed by each *mapper*. Then, each $mapper_i$ executes the DENDIS algorithm on each part, using granularity $gr = 0.001$, as recommended in [Ros and Guillaume, 2016] to get a good accuracy. Finally, the individual results of each *mapper* are combined to obtain the final result.

3.4.2 Pivot selection techniques for space subdivision

The Voronoi-Diagram based partitioning technique is well-known for maintaining data proximity, and it is especially appropriate for distance-based queries. For the creation of Voronoi-Diagrams, the method to select of suitable pivots is very important and therefore, in the *Space subdivision* step of the *Partitioning* phase in SpatialHadoop (see Figure 3.1), a module for *selecting a set of pivots* should be executed. In [Lu et al., 2012] three pivot selection strategies are proposed: random selection, furthest selection and k-means selection. Random selection was faster than k-means, but during the kNN join phase, the performance of k-means selection was better. For this reason, we have adapted *random selection* and *clustering selection* strategies to be included in SpatialHadoop. For the *random* selection technique, $\lfloor |\mathcal{S}^{\mathbb{P}}|/k \rfloor$ random sets of points are generated, then for each set, the total sum of the distances between every two points are computed, and the points from the set with the largest total sum of distances are chosen as pivots.

Taking into account the results of [Lu et al., 2012] and [García-García et al., 2018a], the use of a clustering algorithm improves the quality of the selected pivots for splitting the whole dataset more evenly, and the partition-based and density-based clustering algorithms are the most appropriate for spatial big data [Schoier and Gregorio, 2017]. *Partition-based clustering* attempts to directly decompose the dataset into a set of disjoint clusters. More specifically, this type of clustering algorithm attempts to determine an integer number of partitions that optimize a certain criterion function. On the other hand, the key idea of *density-based clustering* is to group neighboring objects of a dataset into clusters based on density conditions.

– For the *partition-based clustering* category we have chosen the k-means clustering algorithm [MacQueen, 1967], leading to the *k-means* selection technique. We have used the best recommendation for the k-means family in ELKI library [Schubert and Zimek, 2019], this is *Sort-Means* [Phillips, 2002], which accelerates k-means,

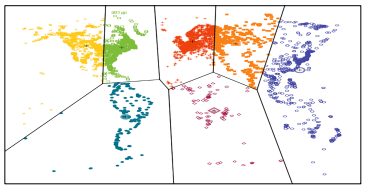
exploiting the triangle inequality and pairwise distances of means to prune candidate means (with sorting). Moreover, it uses k-means++ [Arthur and Vassilvitskii, 2007] to initialize means. When the k clusters have been generated, the center point of each cluster is chosen as a *pivot* for the Voronoi-Diagram based partitioning.

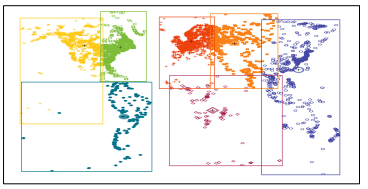
– For the *density-based clustering* class, we have chosen the OPTICS algorithm [Ankerst et al., 1999], resulting the *OPTICS* selection technique. OPTICS is a density-based clustering algorithm that attempts to overcome some of the drawbacks of its most famous counterpart DBSCAN [Ester et al., 1996]. The major weaknesses of DBSCAN are the inability to detect clusters in zones of varying density and the choice of parameter values, for which it is very sensitive. The main difference between them is the ϵ value; in OPTICS, it is an upper bound instead of a specific distance value. We have used the best recommendation for the density-based clustering family in ELKI library [Schubert and Zimek, 2019], this is OPTICSxi [Schubert and Gertz, 2018] with the implementation of FASTOptics [Schneider and Vlachos, 2013]. In general terms, OPTICSxi generates a hierarchical classification of the clusters obtained when OPTICS is applied. The main parameters of OPTICSxi are ϵ (an upper bound of the distance to be considered), *minpts* (the minimum number of points required to form a cluster) and *xi* (contrast parameter that establishes the relative decrease in density). For our experiments, we have used $\epsilon = 2$, $minpts = 100$ and $xi = 0.025$. Since the output of the algorithm is a hierarchical structure, we have to find a level where at most k clusters are stored. When the k clusters have been selected, the center point of each cluster is chosen as a *pivot* for the Voronoi-Diagram based partitioning.

3.4.3 Indexing data

It should be recalled that the *Indexing* step of the *Partitioning* phase in SpatialHadoop splits the data file by assigning each point to one or more partitions. The main idea of this step in Voronoi-Diagram based partitioning technique is to allocate each point of $\mathbb{P}$ to the partition with its closest pivot in $\mathcal{R}^{\mathbb{P}}$. That is, the points from the input dataset $\mathbb{P}$ are assigned to their closest pivot $r_i \in \mathcal{R}^{\mathbb{P}}$, leading to $|\mathcal{R}^{\mathbb{P}}|$ possible partitions. Moreover, some properties of the pivot r_i are calculated and stored for each partition, such as the number of points $|\mathcal{P}_i^{\mathbb{P}}|$, the $MBR(\mathcal{P}_i^{\mathbb{P}})$ which is the Minimum Bounding Rectangle (MBR) covering the points of $\mathcal{P}_i^{\mathbb{P}}$, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$. Figure 3.5 illustrates the result of applying the Voronoi-Diagram based partitioning technique in SpatialHadoop. For more details, in the left chart, the data partitions, using the Voronoi-Diagram based partitioning technique from the selected pivots, are shown. The chart in the center shows the same data partitions, represented as pivots with their MBRs, in the same way, that other spatial partitioning techniques are represented in SpatialHadoop. Finally, on the right, there is a table that summarizes the values of some properties of the pivots available for each partition.

© 2021 García-García, F.J.





Partition	$MBR(\mathcal{P}_i^D)$	r_i	$\lvert\mathcal{P}_i^D\rvert$	$L(\mathcal{P}_i^D)$	$U(\mathcal{P}_i^D)$
1	$MBR(\mathcal{P}_1^D)$	r_1	50000	0	30
2	$MBR(\mathcal{P}_2^D)$	r_2	60000	0	20
3	$MBR(\mathcal{P}_3^D)$	r_3	45000	0	60
4	$MBR(\mathcal{P}_4^D)$	r_4	35000	5	40
5	$MBR(\mathcal{P}_5^D)$	r_5	55000	5	35
6	$MBR(\mathcal{P}_6^D)$	r_6	50000	0	25
7	$MBR(\mathcal{P}_7^D)$	r_7	40000	5	60

Figure 3.5: Overview of the Voronoi-Diagram based partitioning technique in SpatialHadoop.

3.5 Quadtree-based Local Index

The use of a spatial index is one of the most common techniques employed to accelerate spatial query processing. Many different spatial indices have been proposed in the literature [Gaede and Günther, 1998], but the most influential ones have been the R-trees and the Quadtrees. One of the main characteristics of DSDMSs is to include spatial indexes that would allow selective access to specific regions of spatial data, which would in turn yield more efficient distributed query processing algorithms. In general, DSDMSs employ spatial indices for two main purposes: (1) to distribute data among slave nodes and possibly reduce the number of partitions visited during a spatial query (spatial partitioning); and (2) to process spatial queries in slave nodes (spatial indexing). Because of these advantages, spatial indices are supported by all proposed DSDMSs, and the R-tree is used for both purposes [Pandey et al., 2018]. The Quadtree is only used by GeoSpark and LocationSpark, both Spark-based DSDMSs. As we have seen above, in SpatialHadoop, the Quadtree is used as a spatial partitioning technique to split the large datasets into smaller units, but it is not used to index the data of each partition. In this section, we will study how to include the Quadtree as a local index in SpatialHadoop (Hadoop-based DSDMS).

3.5.1 Implementing a Quadtree-based local index in SpatialHadoop

In the *local indexing* phase, each partition is bulk loaded into a Quadtree using the PR algorithm [Hjaltason and Samet, 1999], similar to the Quadtree partitioning. First, the records of each partition are sorted using a Z-curve so that those found on a leaf node appear consecutively. Then, the algorithm considers that all records belong to the root node and checks whether there are nodes to divide. A node is split into its four children if the current number of records exceeds the established node capacity. Figure 3.6 shows a partition indexed by a Quadtree-based local index, with the resulting regions and tree structure. Finally, the Quadtree of each partition is dumped to a file along with the partition records. The index header contains information about its length, MBR of its contents and, the record offsets and sizes of each leaf node. As usual in SpatialHadoop, the *global indexing* phase concatenates all local index files and creates the global index using their MBRs as the index key.

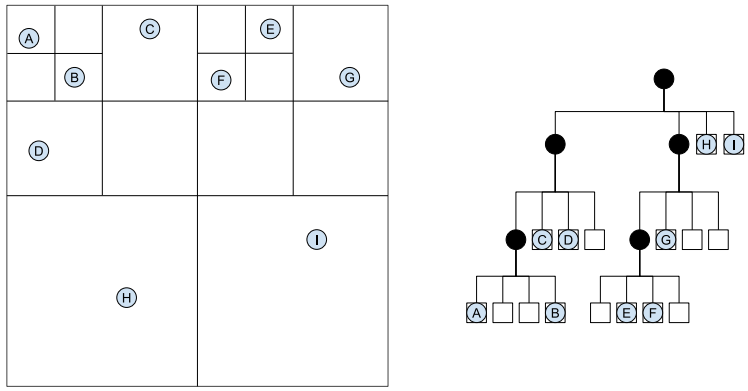


Figure 3.6: Overview of a partition indexed by a Quadtree-based local index.

3.5.2 kNNQ and kCPQ MapReduce algorithms with Quadtrees in SpatialHadoop

The general scheme of the kNNQ algorithm in SpatialHadoop is as follows [Eldawy and Mokbel, 2015]: (1) A *filtering* function selects the partition in which the query point q is found. (2) Then, the *map* task is responsible for obtaining the *initial answer* by using a local kNNQ algorithm on the selected partition and in the *reduce* task, the global k nearest neighbors from q are returned. (3) The *correctness check* phase evaluates whether the result obtained is less than k, or there are partitions within the circular range query, centered in q with a radius equal to the k-th largest distance obtained so far. (4) In this case, the *answer refinement* starts by rerunning the previous MapReduce job from the first step, but at this time, more partitions are within the range query, and therefore, they are selected by the *filtering* function. Otherwise, the final result has already been obtained. If the input dataset contains a local Quadtree-based index, the local kNNQ algorithm traverses it by Breadth-First search of the tree, using a queue. Each step checks whether the current node intersects with the circular range and if so, its four children are added to the end of the queue for further testing. In the case of leaf nodes, each one of the spatial objects they contain is tested to see if they are part of the range query.

In general, the kCPQ MapReduce algorithm [García-García et al., 2018b] in SpatialHadoop consists of the following steps: (1) The *upper bound calculation* step finds an upper bound of the distance value of the k-th closest pair of the joined datasets, called β, (2) that the *filtering* step uses to prune combinations of pairs of partitions. (3) The *local kCPQ* step consists of a *map* function that uses a plane-sweep kCPQ algorithm between each local pair of partitions. (4) Finally, the *global kCPQ* is a *reduce* function that merges the local sets into the final set of the k closest pairs. When both datasets are indexed through a local Quadtree-based index, instead of using a plane-sweep on all its records, both trees are traversed using Depth-First search. For this, a stack stores

© 2021 García-García, F.J.

the children of the pairs of nodes whose distance is less than the current k-th closest pair distance. When dealing with a pair of leaf nodes, the general kCPQ plane-sweep algorithm is applied to the spatial objects stored in them.

3.6 Performance Evaluation

This section provides the results of an extensive experimental study aiming at measuring and evaluating the efficiency of the spatial partitioning and indexing techniques proposed in Sections 3.4 and 3.5. In particular, Subsection 3.6.1 describes the experimental settings for this performance study in SpatialHadoop. Next, Subsection 3.6.2 studies the effects of applying Voronoi-Diagram based partitioning technique in two DJQs (kNNJQ and kCPQ). Finally, Subsection 3.6.3 shows a comparison of the Quadtree-based local index against the R-tree local index in SpatialHadoop over two top-k queries (kNNQ and kCPQ).

3.6.1 Experimental Setup

For the experimental evaluation, we have used real-world 2d point datasets to test both our Voronoi-Diagram based partitioning technique and Quadtree-based local index in SpatialHadoop. We have used datasets from OpenStreetMap[2]:

- *LAKES* (L), which contains 8.4M records (8.6 GB) of boundaries of water areas (polygons).
- *PARKS* (P), which contains 10M records (9.3 GB) of boundaries of parks or green areas (polygons).
- *ROADS* (R), which contains 72M records (24 GB) of roads and streets around the world (line-strings).
- *BUILDINGS* (B), which contains 115M records (26 GB) of boundaries of all buildings (polygons).
- *ROAD_NETWORKS* (RN), which contains 717M records (137 GB) of road networks represented as individual road segments (line-strings).

To create sets of points from these five spatial datasets, we have transformed the MBRs of line-strings into points by taking the center of each MBR. In addition, we have considered the *centroid* of each polygon to generate individual points for this type of spatial object.

The main performance measures that we have used in our experiments have been the *total execution time* (i.e., total response time) and the *total indexing time* (i.e., total creation time). For the performance evaluation, we have employed distance-based queries (i.e., kCPQ and kNNJQ for Voronoi-Diagram based partitioning, and kNNQ and kCPQ for Quadtree-based local index) although they are described in more detail in Chapter 4.

[2] `http://spatialhadoop.cs.umn.edu/datasets.html`

Table 3.2 summarizes the configuration parameters used in our experiments in this section.

Parameter	Values (default)
Sampling	Random, k-means++, DENDIS
Pivot selection	Random, k-means, OPTICS

Table 3.2: Configuration parameters used in our experiments.

All experiments were conducted on a cluster of 12 nodes on an OpenStack environment. Each node has 4 vCPU with 8GB of main memory running Linux operating systems and Hadoop 2.7.1.2.3. Each node has a capacity of 3 vCores for MapReduce2 / YARN use. Finally, we used the latest code available in the repositories of SpatialHadoop[3].

3.6.2 Voronoi-Diagram based Partitioning experiments

Subsections 3.6.2.1 and 3.6.2.2 experimentally show the advantages of the use of sampling and space subdivision in the building of the Voronoi partitioned dataset. Subsection 3.6.2.3 presents all experiments for kNNJQ using the Voronoi-Diagram based partitioning technique, paying special attention to the execution time needed to perform this DJQ and the increment of the k value. Subsection 3.6.2.4 exposes all experiments related to kCPQ, comparing *Quadtree* spatial partitioning technique, which is the best spatial partitioning method in SpatialHadoop for distributed spatial join according to [Eldawy et al., 2015], with the proposed data partitioning technique, and analyzing the increment of k value. Finally, in Subsection 3.6.2.5 a summary of the most important conclusions from the experimental results is reported.

3.6.2.1 Effect of sampling methods

During the *Partitioning* phase, in the *Sampling* step, we collect a set of samples (e.g., $|\mathcal{S}^{\mathbb{P}}| = 0.01 \times |\mathbb{P}|$) from the input dataset to capture its distribution as best as possible, since this sample set will affect query performance. In this experiment, we evaluate three sampling techniques for the building of the Voronoi partitioned dataset (**R**andom, **k**-means++ and **D**ENDIS) for kNNJQ (Fig. 3.7) and kCPQ (Fig. 3.8) by considering the three pivot selection techniques: **R**andom (V_R), **k**-means (V_k) and **O**PTICS (V_O). Figure 3.7 shows that, on average, *k-means++* sampling exhibits the best global performance (execution time) for kNNJQ, although *Random* and *DENDIS* report good results with V_k. *Random* sampling is the fastest, but it has a great component of randomness that exists between two different executions of the same query. *DENDIS* needs more time than *k-means++* to be run, since it requires many distance computations and consumes many resources in its execution. For kCPQ, Figure 3.8 reveals again that *k-means++* sampling shows the best global performance, mainly for V_k. *Random* and *DENDIS* with V_k get good results as well, but they have the previous drawbacks.

[3] `https://github.com/aseldawy/spatialhadoop2`

© 2021 García-García, F.J.

The main conclusion of these results indicates that *k-means++* is the best sampling technique (partition-based sampling) for the creation of Voronoi partitioned datasets in SpatialHadoop for DJQs.

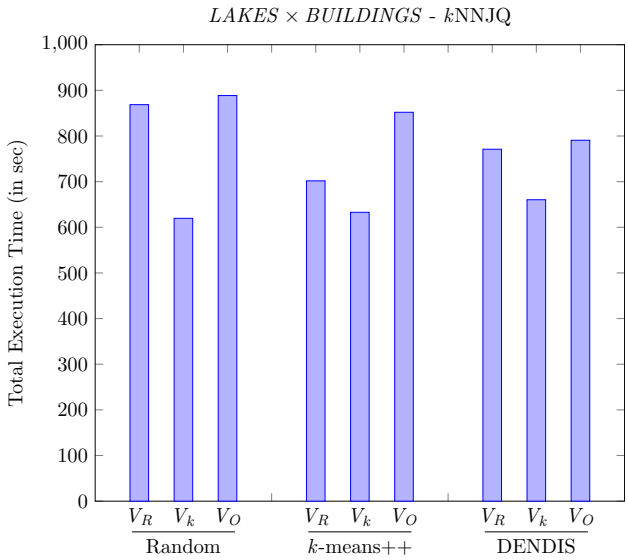


Figure 3.7: kNNJQ cost, the total execution time for the combination of the datasets, $LAKES \times BUILDINGS$, considering different sampling methods and pivot selection techniques for $k = 10$.

3.6.2.2 Effect of space subdivision and indexing

In this experiment, we will compare our new proposed Voronoi-Diagram based partitioning algorithms with the *Quadtree* (Q) built-in partitioning technique which has shown to obtain the best performance results with the different spatial queries present in SpatialHadoop [Eldawy et al., 2015, Eldawy and Mokbel, 2015, García-García et al., 2016b, García-García et al., 2018b]. We will consider the k-means++ sampling (the best one of the previous experiment), and the three pivot selection techniques: *random selection* ($Voronoi_{kR}$, V_{kR}), *k-means selection* ($Voronoi_{kk}$, V_{kk}) and *OPTICS selection* ($Voronoi_{kO}$, V_{kO}) for the *Space subdivision* step, and the *Indexing data* step.

In Figure 3.9, the partitioning cost of different datasets is shown with respect to the execution time, for both the *Space subdivision* and *Indexing* phases. The first conclusion we can draw is that the total execution times for $Voronoi_{kR}$ and *Quadtree* grow similarly as the size of the datasets is increased. For $Voronoi_{kk}$ the increase in execution time is larger, since a *k-means* algorithm is used in the *Space subdivision* phase. This *k-means* algorithm takes longer times to converge towards a solution as the size of the datasets increases. The costliest pivot selection technique is $Voronoi_{kO}$, because the execution of *OPTICSxi* clustering algorithm is more expensive than *k-means*, being the number of partitions smaller. Finally, $Voronoi_{kR}$ presents the fastest execution times,

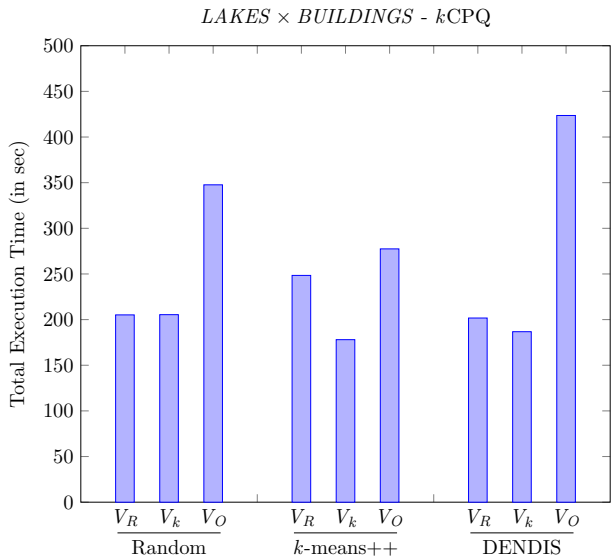


Figure 3.8: kCPQ cost, total execution time for the combination of the datasets, $LAKES \times BUILDINGS$, considering different sampling methods and pivot selection techniques for $k = 100$.

mainly because it consumes the smallest time in the *Indexing* phase of the data since in the *Space subdivision* phase, the total execution times are very similar to those of *Quadtree*. In Table 3.3, we can observe information of data distribution (points per partition) about the partitioning of *ROAD_NETWORKS* dataset for each partitioning technique. On the one hand, $Voronoi_{kO}$ presents a higher mean value due to having a lower number of partitions than the other techniques. On the other hand, $Voronoi_{kk}$ has a much lower standard deviation that allows better handling of data skew problems by having a more proportional distribution of the points in all partitions. This metric provides information about the gap between the different partitioning techniques and how it affects the performance of the DJQs since the skewed data is one of the main factors for the increase of the execution time. In addition, this result is aligned with the behavior obtained in Figures 3.7 and 3.8, where the best performance (the lowest execution time) is obtained by applying k-means++ algorithm, either in sampling or partitioning phases, confirming that the results are close to the optimal values.

	NUM	MEAN	MIN	MAX	STDEV
$Voronoi_{kk}$	512	1400486	19914	3684694	623909
$Voronoi_{kR}$	512	1400486	18347	6228082	985297
$Voronoi_{kO}$	72	9959011	1149113	40703435	8512796
$Quadtree$	430	1667555	218	4275451	1130277

Table 3.3: Information of data distribution (points per partition) of *ROAD_NETWORKS* dataset per partitioning technique.

© 2021 García-García, F.J.

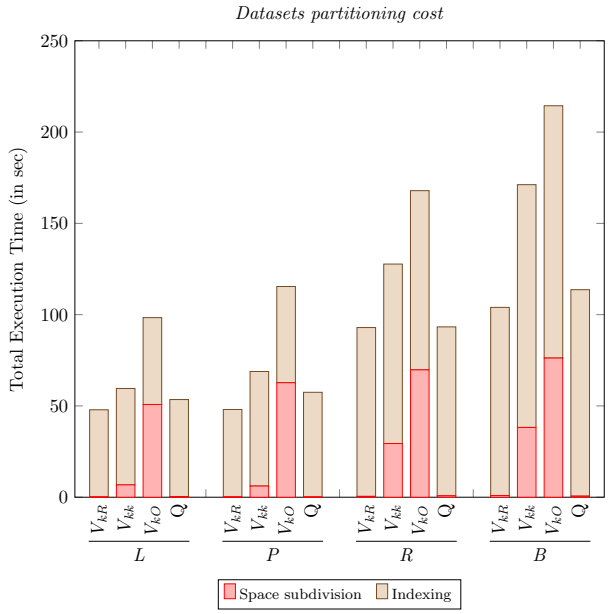


Figure 3.9: Partitioning cost, total execution time per phase, considering different partitioning techniques and datasets.

3.6.2.3 Effect of pivot selection techniques - kNNJQ

This experiment compares the three *pivot selection* techniques (**R**andom, **k**-means and **O**PTICS) with *k-means++* as the sampling method and *Quadtree* (Q) for the kNNJQ in SpatialHadoop, based on the total execution time. They are denoted as $Voronoi_{kk}$ (V_{kk}), $Voronoi_{kR}$ (V_{kR}) and $Voronoi_{kO}$ (V_{kO}).

In Figure 3.10, left chart, the kNNJQ for the combination of different datasets ($L \times P$, $L \times R$, $L \times B$ and $L \times RN$) is shown for each *pivot selection* technique and for a fixed $k = 10$. We can observe that $Voronoi_{kk}$ exhibits the best performance in all cases. Moreover, *Quadtree* is much slower than any of the other variants of Voronoi-Diagram based partitioning technique. This behavior is due to the fact that with the three *Voronoi* variants, every point of $\mathbb{P}$ is assigned to a $\mathbb{Q}$ partition that contains at least k elements, and therefore the processing time of a big part of the points is reduced. However, for *Quadtree* there is a large growth of the number of partitions to search for kNN candidates. Notice the high execution time needed for $L \times RN$ using V_{kO}, this is because the OPTICS algorithm does not generate a fixed number of clusters, but it depends strongly on the data distribution (and the number of clusters is less than k). In this figure, we can also highlight that the differences in execution time between the four partitioning techniques are reduced with the combination with the largest dataset, $L \times RN$, mainly because the *Quadtree* technique finds more final results faster. As the volume and size of $\mathbb{Q}$ are much greater, the volume of points of $\mathbb{P}$ that fall into partitions of $\mathbb{Q}$ is also greater, obtaining final results earlier, reducing the execution time of the kNNJQ MapReduce algorithm.

Moreover, similar behavior can be observed in Figure 3.10, right chart, where, as the k value is increased for the combination of the datasets, $LAKES \times ROADS$. The execution time of the kNNJQ algorithm is also higher. We have also to emphasize the high execution time needed for $k = 75$ using V_{kR}, which is mainly due to the random nature of the random selection technique.

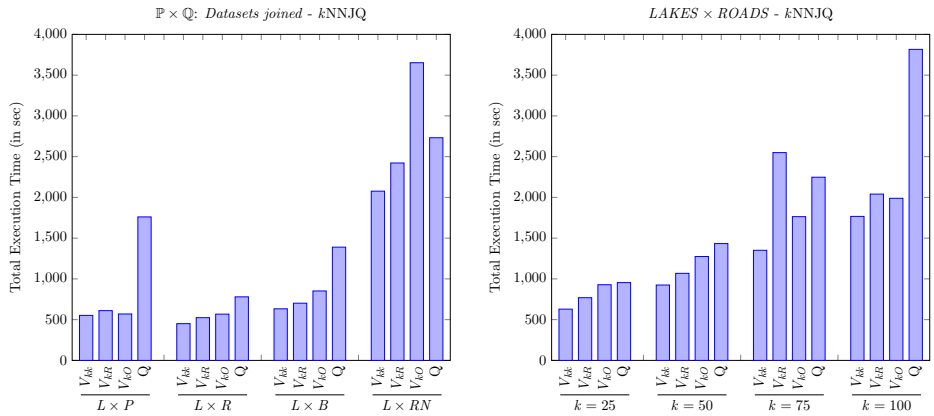


Figure 3.10: kNNJQ cost, total execution time of different dataset combinations (left) and varying the k values (right) for $L \times R$.

3.6.2.4 Effect of pivot selection techniques - kCPQ

These experiments aim to measure the behavior of the kCPQ MapReduce algorithm in SpatialHadoop, varying different parameters as the dataset sizes to be joined, the partitioning techniques, and the values of k. In Figure 3.11, left chart, the kCPQ for a fixed $k = 100$ and for real spatial datasets ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) is shown with respect to the execution time for the different partitioning techniques (*Voronoi*$_{kk}$, *Voronoi*$_{kR}$, *Voronoi*$_{kO}$ and *Quadtree*). We can observe that the total execution times in all partitioning techniques grow almost linearly as the size of the datasets is increased, except *Voronoi*$_{kO}$ that for $P \times R$ the time is very high, due to mainly the high preprocessing cost. For kCPQ, the best partitioning technique is *Quadtree*, which is approximately 18% faster than *Voronoi*$_{kk}$. Moreover, for the combinations of $L \times P$ and $P \times R$, *Voronoi*$_{kk}$ is slightly faster than *Quadtree* (e.g., for $L \times P$ *Voronoi*$_{kk}$ is 14 sec faster than Quadtree), but for the combinations of the largest datasets ($R \times B$ and $B \times RN$) *Quadtree* is the fastest, e.g., for $B \times RN$ *Quadtree* is 18% (254 sec) faster than *Voronoi*$_{kk}$. That is, *Voronoi*$_{kk}$ exhibits smaller runtime values for smaller dataset sizes since it produces a slightly larger number of partition combinations (e.g., 24 vs. 23 partition pairs for $L \times P$) that are better distributed in tasks for this cluster of nodes. But for larger dataset sizes, *Quadtree* is the fastest for kCPQ since it minimizes the number of partitions for each dataset and the number of the ones that overlap between each other. For instance, for the combination of $B \times RN$, *Quadtree* obtains $78 \times 430 = 33540$ possible pairs of partitions, with only 711 pairs of partitions (2%) considered, with a total execution time of 1220 sec. In the case of *Voronoi*$_{kk}$, it generates $81 \times 512 = 41472$

© 2021 García-García, F.J.

pairs of partitions, with only 1191 pairs of partitions (2.8%) considered, with a total execution time of 1474 sec, which is slightly higher than for Quadtree due to the increase on the number of *map* tasks. Finally, $Voronoi_{kO}$ shows the worst results, noting that the indexing time of $Voronoi_{kO}$ is much higher and the number of partitions is smaller. Figure 3.11, right chart, shows the effect of increasing the k value for the combination of the largest datasets ($BUILDINGS \times ROAD\_NETWORKS$) for kCPQ. This experiment shows that the total execution time grows slowly as the number of results to be obtained (k) increases. All partitioning techniques report very stable execution times, even for large k values (e.g., $k = 10^5$), although, we can see that *Quadtree* still exhibits the lowest execution times.

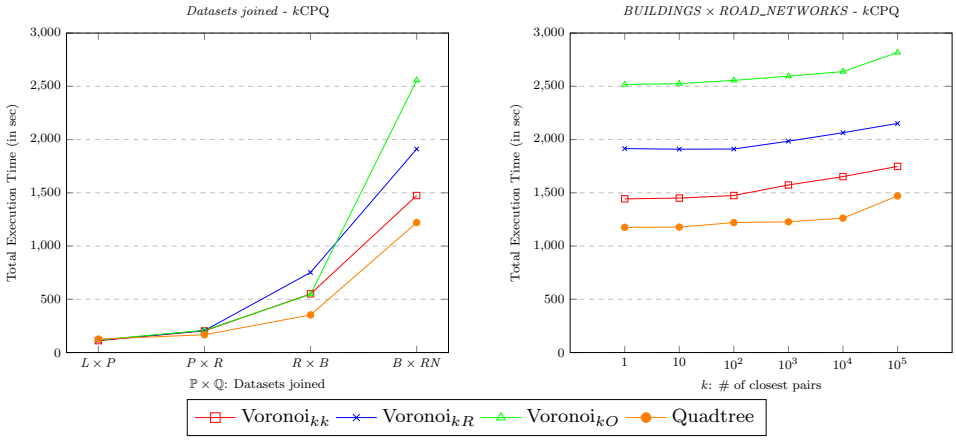


Figure 3.11: kCPQ cost, total execution time of different partitioning techniques (left) and varying the k values (right) for $B \times RN$.

3.6.2.5 Conclusions from the experimental results

The main conclusions extracted for this set of experiments on the proposed Voronoi-Diagram based partitioning techniques in SpatialHadoop for DJQ MapReduce algorithms are the following:

1. The best sampling technique to find a small but representative profile from big spatial datasets for DJQ processing in SpatialHadoop is k-means++, which is a partition-based sampling method.

2. Using the k-means++ sampling, we have compared three clustering algorithms (Random, k-means, and OPTICS) for the pivot selection. The partitioning execution times for V_{kR} are the smallest and grow almost linearly as the size of the datasets, while, for V_{kk}, this increment is larger due to the use of k-means++ clustering algorithm. The use of OPTICS, V_{kO}, is the slowest. But V_{kk} exhibits the best global performance in all cases for kNNJQ because this combination of k-means algorithms partitions the dataset appropriately for the kNNJQ MapReduce algorithm in SpatialHadoop.

3. For kNNJQ (it follows a multiple nearest neighbor query processing schema), V_{kk} is faster than *Quadtree* partitioning because it deals better with skewed data and it gets more final results earlier.

4. *Quadtree* partitioning outperforms all other variants of partitioning techniques based on Voronoi-Diagrams, with respect to the total execution time for the kCPQ (it follows a global query processing schema), although $Voronoi_{kk}$ or V_{kk} techniques present slightly better performance, for the combinations of the smallest datasets.

3.6.3 Quadtree-based local index experiments

In this section, we present the most representative results of our experimental evaluation, comparing the Quadtree-based local index against the built-in R-tree local index in SpatialHadoop [García-García et al., 2020a]. To this end, we have used real-world 2d point datasets to test the local indexes and the top-k query MapReduce algorithms (kNNQ and kCPQ) in SpatialHadoop.

The conducted experiments results are shown in Figure 3.12 and correspond to the following: (1) Local index creation time; (2) kNNQ execution time varying k using $BUILDINGS$; (3) kNNQ with $k = 100$ for different spatial datasets; (4) kNNQ varying cluster node count (η); kCPQ execution time varying k joining (5) $LAKES \times PARKS$ and (6) $PARKS \times BUILDINGS$; (7) kCPQ with $k = 100$ for different spatial datasets combinations; and (8) kCPQ varying cluster node count (η).

3.6.3.1 Conclusions from the experimental results

By analyzing the experimental results showed in Figure 3.12, we can extract the following conclusions:

1. The indexing times (creation times) are similar with a small advantage for *Quadtree* since it has a higher number of partitions and the workload is shared among the nodes, e.g., $BUILDINGS$ (78 partitions with *Quadtree* vs. 28 partitions with STR).

2. For both DBQs, when the value of k or the size of the datasets varies, *Quadtree* is the clear winner. Furthermore, its execution times are more stable than those of R-tree when dealing with higher values of k or larger datasets.

3. For kCPQ, the large differences in execution time between R-tree and *Quadtree*, are due to the morphology of the nodes and the number of partitions. The nodes of the *Quadtree* show a more regular shape that causes a smaller number of overlaps between nodes, which implies a reduction in the number of pairs to compare. Moreover, the number of partitions generated by *Quadtree* is larger, which means they are smaller in size and allows us to reduce the impact of skew data problems.

4. For kNNQ, the use of computing nodes (η) by both *Quadtree* and R-tree is small thanks to the efficient utilization of indices, allowing the execution of several queries in parallel.

© 2021 García-García, F.J.

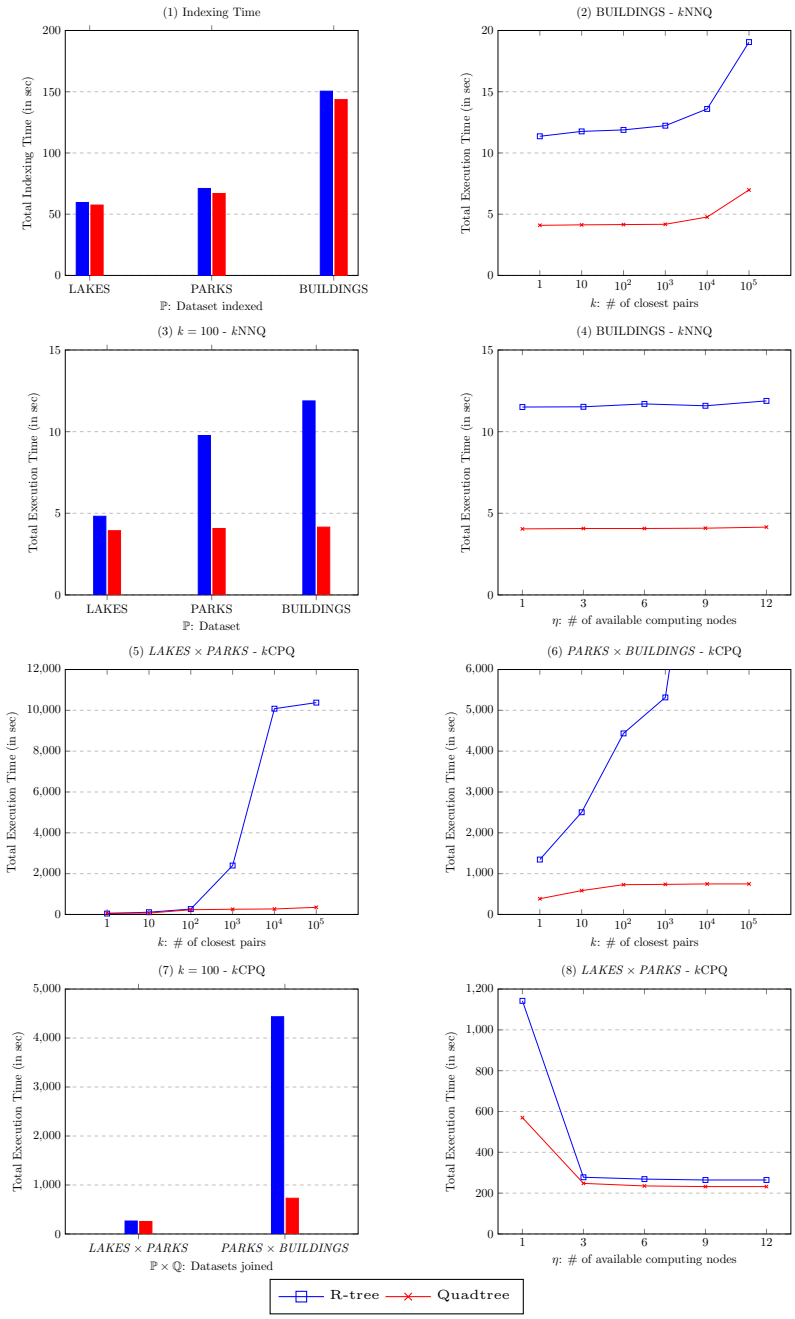


Figure 3.12: Experimental results comparing Quadtree and R-tree performance with the top-k queries (kNNQ and kCPQ).

5. For kCPQ, both R-tree and *Quadtree* show better performance when the number of computing nodes (η) is increased, but if there are not enough tasks available for a specific number of nodes, no performance improvements are obtained. On the one hand, *Quadtree* takes much less time than R-tree when there is only one node available. On the other hand, the existence of skew data problems, which generate *map* tasks with a larger execution time compared to the rest, reduces the benefits of adding more nodes.

3.7 Conclusions

This section highlights the main conclusions of this chapter. First, we have detailed the general spatial partitioning scheme and spatial indexing mechanism of SpatialHadoop to enable fast access to spatial data in Hadoop. Next, we have described the available spatial partitioning techniques in SpatialHadoop. Furthermore, we have presented a detailed description of the built-in spatial indexes in SpatialHadoop. We have also proposed a data partitioning technique based on Voronoi-Diagrams in SpatialHadoop. We have explained and discussed the reasons and the process to include the Quadtree as a local index in SpatialHadoop since this spatial access method is widely used in commercial spatial database systems. Finally, an experimental evaluation of different spatial partitioning methods and a comparison with the new Voronoi-Diagram based technique and Quadtree-based local index in SpatialHadoop have demonstrated their importance and great performance for DBQs (kNNQ, kNNJQ and kCPQ).

© 2021 García-García, F.J.

Chapter 4

Spatial Query Processing in SpatialHadoop

© 2021 García-García, F.J.

Chapter 4

Spatial Query Processing in SpatialHadoop

Contents

© 2021 García-García, F.J.

In this chapter, we focus on a detailed description of the spatial query processing in SpatialHadoop. First, in Section 4.1, the general spatial query processing scheme in SpatialHadoop is presented together with the different features and tools that it provides to obtain better performance over Hadoop. Next, the spatial queries already supported by SpatialHadoop are exposed in Section 4.2. Moreover, new Spatial Queries implemented in SpatialHadoop are described in Section 4.3. Then, useful extensions and improvements of the spatial query algorithms are discussed in Section 4.4. Finally, a performance evaluation of several spatial query algorithms, and a comparison with their extensions and improvements, are presented in Section 4.5.

4.1 SpatialHadoop for Spatial Query Processing

The main goal of SpatialHadoop is to provide spatial query processing capabilities to Hadoop by injecting spatial data awareness in each of its layers. In the previous chapter, it has been described how the *Storage* layer seeks to distribute, organize and index the big spatial datasets in an optimal way for its processing. However, the layers that support the different spatial queries are the *MapReduce* and *Operations* layers.

4.1.1 MapReduce layer

The *MapReduce* layer is the query processing layer that runs MapReduce programs, and SpatialHadoop enriches it to support the use of spatially indexed input files. Therefore, SpatialHadoop provides to the MapReduce layer two new components, namely, *SpatialFileSplitter* and *SpatialRecordReader*, to implement efficient and scalable spatial data processing [Eldawy and Mokbel, 2015].

– *SpatialFileSplitter*. This new file splitter, which has knowledge about the spatial nature of the datasets, obtains blocks from the input files based on the partitions defined by global indexes. Besides, it avoids the processing of those partitions that do not contain elements that are part of the result of a certain spatial query. To do this, a *filtering* function can be defined to select the partitions to be processed later by implementing some heuristic that exploits their spatial properties. For instance, for the εDistance Range query, the *filtering* function can prune the partitions whose MBR has a distance value from the query point q larger than ε. Another feature that it provides is the ability to combine two input files by using *CombineFileSplits*. This kind of split contains a pair of partitions or blocks, one from each of the input files, and is especially suitable for queries that perform some spatial join. For instance, for the εDistance Join query, the *filtering* function receives a list of partitions from each of the input files and returns, through *CombineFileSplits*, the combinations of pairs of partitions whose MBRs are separated at most a distance of ε. Finally, the partitions outputted by the *SpatialFileSplitter*, whether they have one or two input files, are further processed by the *SpatialRecordReader*.

© 2021 García-García, F.J.

– *SpatialRecordReader*. It is a *record reader* that obtains the input data of the *map* function from the splits generated by the *SpatialFileSplitter*. This *map* function takes the MBR of the partition being processed as the *key* parameter, and the local index (or the iterator if it does not exist) as the *value* parameter, which enables access to all elements in the partition. The *SpatialRecordReader* allows us to handle all elements of the partition in the same *map* function in a more optimal way and without the need to regroup by forwarding them to a *reducer* [Eldawy and Mokbel, 2015]. Moreover, it also allows us to exploit the characteristics of the local index so that we can apply different heuristics that prevent us from having to scan all elements in the partition but only to those that are necessary for the spatial query. Finally, in the case of *CombineFileSplits*, the behavior is similar but the *map* functions receive a pair of MBRs, as the *key* parameter, and a pair of each of the indices (or iterators) of the split partitions, as the *value* parameter.

4.1.2 Operations layer

The *Operations* 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. The general spatial query processing scheme in SpatialHadoop consists of five steps [Eldawy and Mokbel, 2015, García-García et al., 2016b, García-García et al., 2018b, Li et al., 2019], as we can observe in Figure 4.1:

1. *Preprocessing*, also called *Partitioning*, is the step where the dataset is distributed according to a specific spatial partitioning technique (e.g., Grid, Quadtree, STR, Hilbert-curve, etc.) [Eldawy et al., 2015], generating a set of partitions or cells. In this partitioning process, spatial data locality is fulfilled since spatially nearby objects are assigned to the same partition [Eldawy and Mokbel, 2015]. Each partition corresponds to an HDFS block, and the HDFS blocks in each file are globally indexed, generating a *spatially indexed file* (indexing).

2. *Filtering*, when the query is issued, this is the optional step where the master node examines all partitions and prunes (by a *filtering* function) those that are guaranteed not to include in any possible result of the spatial query. Furthermore, in this step, the *SpatialFileSplitter* component exploits the global index(es) on input file(s) and the partition boundaries to prune easily file partitions not contributing to the answer of the spatial query [Eldawy and Mokbel, 2015].

3. *Local Spatial Query Processing*, is the step where a local spatial query is performed on each non-pruned partition in parallel on different machines. In this step, the *SpatialRecordReader* allows us to read a split originating from the spatially indexed input file(s) and exploiting local index(es) to efficiently process the spatial query [Eldawy and Mokbel, 2015]. However, it is possible to employ the *SpatialRecordReader* component without using the local index(es) (i.e., it would not exploit the advantages of them) and access the whole set of elements of each partition in the input of the *map* function to perform, for example, a plane-sweep-based algorithm over them.

4. *Pruning*, which is an optional step, is executed after the local spatial query has ended in each machine. Its main goal is to detect which elements do not need further processing or are not part of the query result. The former ones are written directly to the output files, while the latter are directly pruned. This action reduces the amount of data sent to the next step, reducing both memory requirements and processing time.

5. *Global Processing*, also called *Merging*, is the step where the results are collected from all nodes (machines) in the previous step, and the final result of the concerned spatial query is computed. A *combine* function can be applied in order to decrease the volume of data that is sent from the *map* task. The *reduce* function can be omitted when the results from the *map* phase are final. Furthermore, when the total size of the elements collected from all nodes is very large, they cannot be optimally processed in a single node. Therefore, in [Li et al., 2019] an additional processing scheme is presented consisting of two steps: (1) a *reduce* function, which runs in parallel on various nodes, that processes and reduces the size of the data that remains to be processed, so that it fits into a single node, and (2) a *post-processing* step that lastly merges the data on a single node. Finally, this additional scheme can be executed in several rounds.

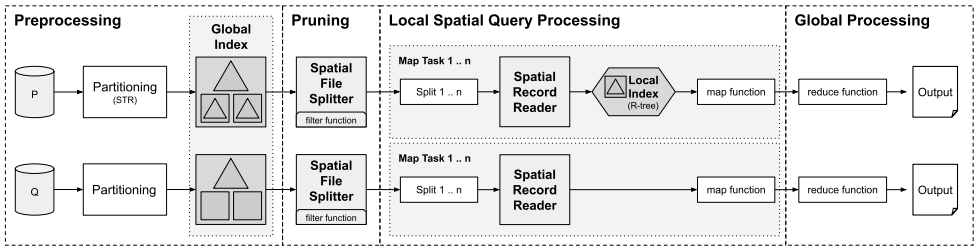


Figure 4.1: General spatial query processing scheme in SpatialHadoop.

4.2 Spatial Queries supported by SpatialHadoop

SpatialHadoop is equipped in the *Operations* layer with several spatial operations, including range, $k$nearest neighbor, and spatial join queries. Other computational geometry algorithms (e.g., polygon union, skyline, convex hull, farthest pair, closest pair, and Voronoi-Diagram) are also implemented following the similar general approach described in Section 4.1. A description of these spatial operations and the different algorithms implemented in SpatialHadoop are exposed below.

4.2.1 Range Query

The Range Query (RQ), given one spatial dataset $\mathbb{P}$ and a query area a, finds all spatial objects in the dataset that overlap with a. An example of this spatial query could be to *find all buildings inside the limits of a city*, that is, *buildings* are the objects of the

© 2021 García-García, F.J.

spatial dataset $\mathbb{P}$ and the area bounded by *the limits of the city* is a. The naive method in Hadoop to obtain the result of the query would be to check, one by one, if each spatial object from $\mathbb{P}$ is within the query area a. In the case of SpatialHadoop, and thanks to the use of spatial indexes, the Range Query MapReduce algorithm in SpatialHadoop [Eldawy and Mokbel, 2015] consists of two steps: *global filter* and *local filter*.

The first step uses the *global index* and a *range filtering* function to select only the partitions from $\mathbb{P}$ that need to be processed. The built-in *SpatialFileSplitter* selects only the partitions that overlap with the query area a. Partitions that are completely inside a are part of the query result and do not need further processing. The second step processes partitions that are partially overlapping with a as a refinement step. A *map* function uses the *local index* of the current partition provided by the *SpatialRecordReader* with a local range query algorithm to return the final answer. For instance, it can use an R-tree to check which spatial objects are fully overlapping with a.

If the Range Query is dealing with an index with replication, then the algorithm needs to apply a duplicate avoidance method to remove duplicate spatial objects in each step. For instance, SpatialHadoop computes the intersection of each candidate spatial object with the query area a and checks if the top-left corner is inside the partition boundaries. It is guaranteed that only one partition contains that point because they are disjoint. Furthermore, this technique avoids the need for a *reduce* function.

4.2.2 kNearest Neighbor Query

The kNearest Neighbor Query (kNNQ), given one spatial dataset $\mathbb{P}$, finds the k closest spatial objects (e.g., points) in the dataset to a given query point q. One application case (Accommodation Services), with one spatial dataset $\mathbb{P}$ of locations of hotels and the location q of a conference center, kNNQ could *find the 3 nearest possible hotels to the conference center* in order to select the best hotel ($k = 3$) close to the conference where the user is attending. In Hadoop, a kNNQ algorithm needs to calculate the distance to q of all points in $\mathbb{P}$ and keep only the top-k ones. In [Eldawy and Mokbel, 2015], a kNNQ MapReduce algorithm in SpatialHadoop, that uses different pruning techniques, is presented. Figure 4.2 shows the proposed kNNQ MapReduce algorithm which is composed of the following three steps: *initial answer*, *correctness check* and *answer refinement*.

The previous steps are a pair of MapReduce jobs that calculate the initial result and that are iteratively run again if they do not pass the correctness check until the final answer is obtained. Similar to the Range Query algorithm, a *filtering* function selects the partition in which the query point q is found. Then, the *map* task is responsible for obtaining the *initial answer* by using a local kNNQ algorithm on the points of the selected partition, and in the *reduce* task, the global k nearest neighbors from q are returned. The *correctness check* phase evaluates whether the result obtained is less than k, or there are partitions within the circular range query, centered in q with a radius equal to the k-th largest distance obtained so far. In this case, the *answer refinement* starts by rerunning the previous MapReduce job from the first step, but at this time, more partitions are within the range query, and therefore, they are selected by the *filtering* function. Otherwise, the final result has already been obtained.

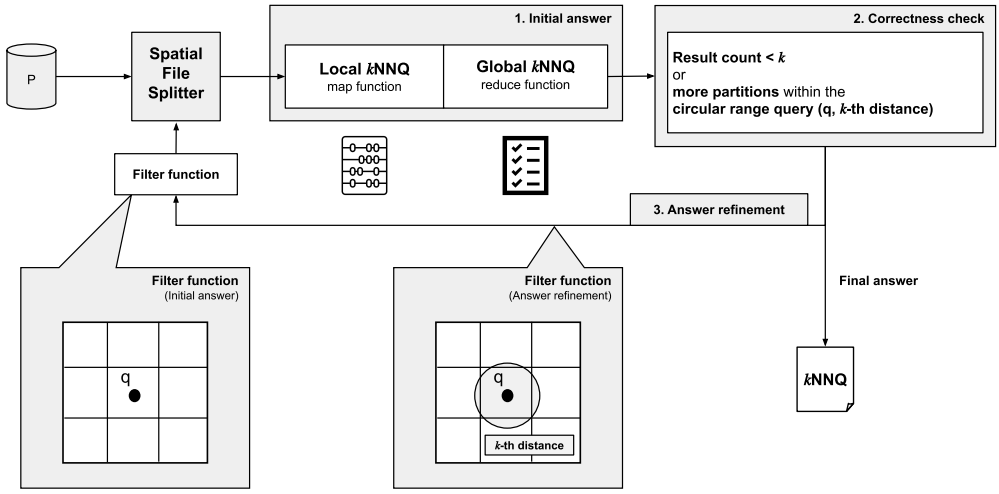


Figure 4.2: Overview of the kNNQ MapReduce algorithm in SpatialHadoop.

4.2.3 Spatial Join Query

The Spatial Join Query (SJQ), given two datasets $\mathbb{P}$ and $\mathbb{Q}$ of spatial objects and a spatial operator θ, finds the set of all pairs (p_i, q_i) where $p_i \in \mathbb{P}, q_i \in \mathbb{Q}$ and $\theta(p_i, q_i) = true$. Moreover, the spatial operator θ could describe different spatial or geometric relations between the spatial objects like *contains, intersects, inside, distance,* etc. For example, a Spatial Join Query could be to *find all urban areas that belong to each state or province*, given a spatial dataset $\mathbb{P}$ of *urban areas* and another spatial dataset $\mathbb{Q}$ of *states and provinces*, using the spatial operator *intersects* as θ. To perform the SJQ of big spatial datasets, SpatialHadoop implements two algorithms (Spatial Join MapReduce and Distributed Join) [Eldawy and Mokbel, 2015]. Furthermore, both approaches uses the *intersects* operator as θ with a plane-sweep implementation.

The *Spatial Join MapReduce* algorithm [Zhang et al., 2009b] is an implementation of the Partition-Based Spatial-Merge (PBSM) join [Patel and DeWitt, 1996] for MapReduce, and it is aimed at non-indexed datasets. Therefore, this algorithm does not take advantage of the spatial indexes of SpatialHadoop. First, this algorithm calculates the MBR of combining the two input datasets using a MapReduce job, and then it divides it using a uniform grid. Next, in the *map* function, the partition to which each spatial object of each of the datasets belongs is calculated, and it is used as the key to group them in the same *reducer*. Finally, in the *reduce* function, the spatial operator θ (e.g., overlaps) is applied to the two received subsets of spatial objects that are part of the same partition.

The *Distributed Join* algorithm [Eldawy and Mokbel, 2015] exploits the spatial index mechanism of SpatialHadoop and it generally consists of three steps: *global join*, *local join*, and *duplicate avoidance*. First, in the *global join* step, pairs of blocks of spatial objects from both datasets are combined. Next, in the *local join* step, the spatial

© 2021 García-García, F.J.

operator θ (e.g., overlaps) is applied to the spatial objects of the joined blocks from the dataset. Finally, in the *duplicate avoidance* step, if the distributed join is dealing with an index with replication, then the algorithm needs to apply a duplicate avoidance method to remove duplicate spatial objects similar to the Range Query. SpatialHadoop presents some variants of the distributed join that depends on whether the datasets are indexed or not. Moreover, a cost model [Belussi et al., 2020a] has been developed for the selection of the best-suited algorithm and variant depending on different properties of the input datasets.

- *Distributed Join with no index.* This variant does not require the input datasets to be indexed, and it is a *Map*-side join implementation of the block nested loop join. Therefore, the *map* function receives pairs of splits resulting from applying the Cartesian product of the splits of the datasets.
- *Distributed Join with index.* Like the previous one, it is a *Map*-side join, but in this case, it requires that both datasets are indexed and therefore works at the partition level. Furthermore, it is an adapted implementation of the Grid File Spatial (GFS) join algorithm [Harada et al., 1990] that has a series of advantages: (1) the *global join* step allows us to apply a *filtering* function that prunes those partition pairs that do not satisfy the spatial operator θ; (2) the *local join* step receives the local indices of the two join partitions, which allows us to optimize the performance of the local plain-sweep join algorithm.
- *Distributed Join with repartition.* Similar to *Distributed Join with index*, it adds a previous MapReduce job in which the distribution of the smallest dataset with the index of the largest dataset is performed. It is mainly designed for the spatial operator *overlaps* to decrease the number of pairs obtained in the *global join*. For instance, those spatial objects or full partitions of the dataset to be repartitioned that fall outside the partitions of the largest dataset are directly dismissed.

4.2.4 Polygon Union Query

The Polygon Union Query (PUQ), given one spatial dataset of polygons $\mathbb{P}$, finds a set of polygons $\mathbb{Q}$ formed by the perimeter of all points found in, at least, one polygon in $\mathbb{P}$, while removing existing inner edges. An example of this spatial query could be to *find the whole available area for agriculture from the areas provided by each state or province*, that is, *areas provided* are the spatial objects of the spatial dataset $\mathbb{P}$ and the resulting total area is $\mathbb{Q}$. Figure 4.3 shows how 3 different polygons, that is, a rectangle, an arrow, and a circle, are combined into their PUQ resulting polygon. In [Eldawy et al., 2013], a PUQ MapReduce algorithm in SpatialHadoop, which exploits spatial partitioning, is presented. The proposed MapReduce algorithm is composed of three steps: *partitioning*, *local union* and *merging*.

First, the *partitioning* step uses a built-in partitioning technique of SpatialHadoop to distribute the polygons to each node. Next, the *local union* step is a *map* function that uses a traditional in-memory polygon union algorithm [de Berg et al., 2008] to get the

local result for a given partition, that is, the set of polygons of the local union. Finally, the *merging* step employs the same union algorithm used in the *map* function to combine the partial results into the final set of polygons. The use of spatial aware partitioning allows each *map* function to receive adjacent polygons, so it can remove more inner edges and return simpler polygons. As a result, the workload of the *merging* step decreases due to the lower number of polygons it receives and the previously distributed cleaning of internal edges.

An enhanced version of the SpatialHadoop algorithm is presented in [Li et al., 2019] that removes the need for the *merging* step. In the *map* function of the *local union* step, a new *pruning* step is added, which removes unrequired line-segments by using the limits of the actual partition. These segments are either not part of the final result or are already generated by another node.

4.2.5 Skyline Query

The Skyline Query (SQ), given one spatial dataset of points $\mathbb{P}$, finds a set of points SKY from $\mathbb{P}$ that are not dominated by any other point of $\mathbb{P}$. A point p_i is said to

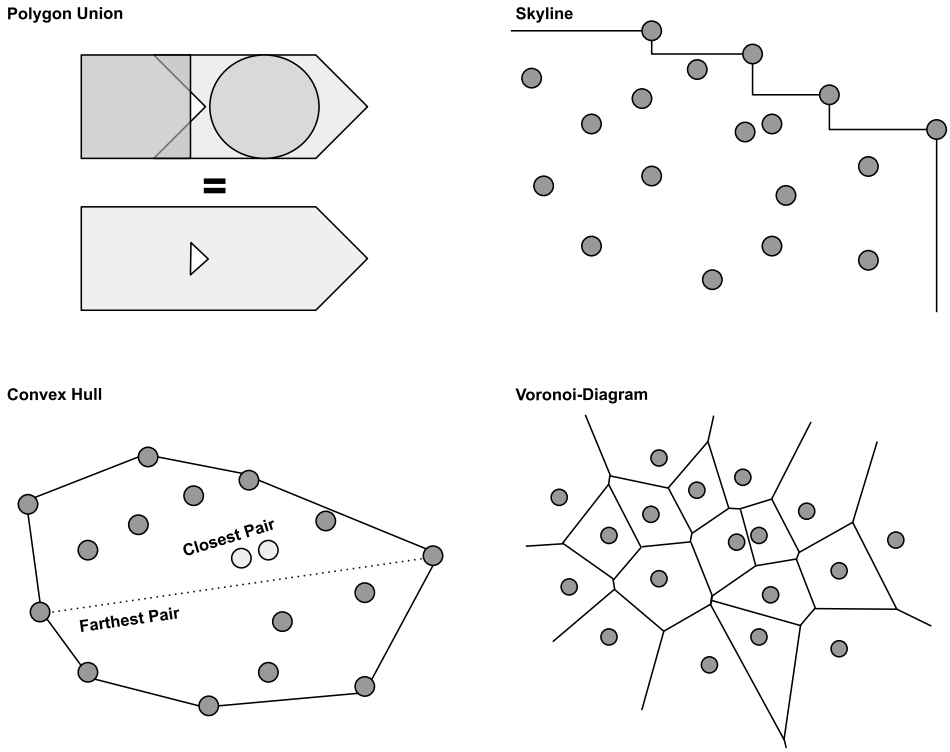


Figure 4.3: Other spatial queries present in SpatialHadoop: Polygon Union, Skyline, Convex Hull, Farthest Pair, Closest Pair and Voronoi-Diagram Queries.

© 2021 García-García, F.J.

dominate another point p_j if each coordinate of p_i is greater than or equal to the same coordinate of p_j, and there is at least one coordinate that is greater in one dimension. An example of this spatial query could be to *find all restaurants that are both cheap and close to a monument*, that is, *restaurants* are the points of the spatial dataset $\mathbb{P}$ with coordinates price and distance to the closest monument, and the resulting set SKY will be those restaurants that are not worse than any other one in both price and distance to a monument. Figure 4.3 shows the different segments of the resulting SQ of a given point dataset. The SQ MapReduce algorithm in SpatialHadoop, that is described in [Eldawy et al., 2013], is composed of the following three steps: *partitioning*, *local skyline* and *global skyline*.

The algorithm follows a similar approach to the Polygon Union Query, described in Section 4.2.4, but using a traditional centralized skyline algorithm for two-dimensional data [Preparata and Shamos, 1985]. Moreover, it utilizes a *filtering* function that prunes those partitions that do not contain points of the final query answer by using the skyline algorithm with their MBRs. Furthermore, an output-sensitive version of the skyline algorithm, which overcomes the memory limitations of doing the *merging* step on a single node, is presented in [Li et al., 2019].

Finally, there are other Skyline Query algorithms in SpatialHadoop with interesting features:

- The skyline algorithm presented in [Pertesis and Doulkeridis, 2015] outperforms the original SpatialHadoop algorithm thanks to the use of a *filtering* function, a *combine* function, and several pruning rules added to different steps.
- In [Kalyvas and Maragoudakis, 2019], the authors propose an alternative algorithm based on added sorting mechanisms that provide better local skylines. Moreover, they present the first SpatialHadoop algorithm in literature for a derived query called *Reverse Skyline Query*.

4.2.6 Convex Hull Query

The Convex Hull Query (CHQ), given a spatial dataset of points $\mathbb{P}$, is the smallest convex polygon $\mathbb{Q}$ that contains all points in $\mathbb{P}$. The points that are part of the result of this query are returned in clockwise order. An example of this spatial query could be to *find the convex hull of all WiFi access points that exist within a city*, in order to get an approximation of the area with WiFi coverage. For instance, *the WiFi access points* are the points of the spatial dataset $\mathbb{P}$ and the resulting convex polygon is $\mathbb{Q}$. The convex hull is very useful as a complement for other spatial processing algorithms like *collision detection* where the approximation could be used, instead of checking point-by-point of the dataset. Figure 4.3 shows the resulting polygon of the CHQ of a given point dataset. A Convex Hull Query MapReduce algorithm in SpatialHadoop that exploits spatial partitioning is described in [Eldawy et al., 2013]. The proposed MapReduce algorithm is composed of three steps: *partitioning*, *local convex hull* and *global convex hull*.

The general plan of the algorithm is similar to that of the previous Skyline Query algorithm described in Section 4.2.5, but using a traditional in-memory convex-hull

algorithm [Andrew, 1979]. Furthermore, it reuses the *filtering* function of the Skyline Query by applying it to the partitions for each of the 4 skylines (*max-max*, *min-max*, *max-min* and *min-min*) of the spatial dataset. Therefore, the partitions that are not part of the skylines are pruned because they do not contribute points to the final result.

Finally, in [Li et al., 2019] the authors propose a new and more efficient algorithm since, in the *local convex hull* step, it performs pruning of all points that are not part of the final convex hull. Therefore, the *global convex hull* step transforms into a simpler *merge* step that performs the union of all received points.

4.2.7 Farthest Pair Query

The Farthest Pair Query (FPQ), given a spatial dataset of points $\mathbb{P}$, is the pair of points (p_i, p_j) such that $p_i \neq p_j$ that have the largest Euclidean distance between them. One of the main properties of this query is that the pair of points are part of the convex hull of $\mathbb{P}$. An example of this spatial query could be to *find the farthest pair of all buildings in a state or province*, in order to get the largest distance between them needed for the design of a communication network. For instance, *the buildings* are the points of the spatial dataset $\mathbb{P}$ and the resulting *farthest pair of buildings* is a pair (p_i, p_j). Figure 4.3 shows how the pair of points of an FPQ of a given point dataset is part of its CHQ. In [Eldawy et al., 2013] a Farthest Pair Query MapReduce algorithm in SpatialHadoop is presented and consists of the following three steps: *partitioning*, *local farthest pair* and *global farthest pair*.

The approach is similar to the previous algorithms with the novelty that the *filtering* function returns pairs of partitions as input from the *map* function that performs the *local farthest pair* step. Moreover, the *filtering* function uses the maximum and minimum distances between the MBRs of the partitions to discard those that do not provide results. Next, the algorithm obtains the local convex hull of each not pruned pair and applies the *rotating calipers* algorithm [Preparata and Shamos, 1985] on the result to determine each *local farthest pair*. Finally, the *global farthest pair* step only has to output the pair of points received in the *reduce* function with the largest distance.

4.2.8 Closest Pair Query

The Closest Pair Query (CPQ), given a spatial dataset of points $\mathbb{P}$, obtains the pair of points (p_i, p_j) such that $p_i \neq p_j$ that have the smallest Euclidean distance between them. Moreover, this is the complementary query to the Farthest Pair Query. An example of this spatial query could be to *find the closest pair of airplanes that are actually flying around the world*, in order to monitor air traffic and avoid possible collisions. For instance, *the airplanes* are the points of the spatial dataset $\mathbb{P}$, and the resulting *closest pair of airplanes* is a pair (p_i, p_j). This query has several differences from kCPQ that make it a less demanding query: (1) It involves a single dataset as input while kCPQ joins two distinct datasets; (2) it uses a fixed value of $k = 1$. Figure 4.3 shows the CPQ of a given point dataset. A Closest Pair Query MapReduce algorithm in SpatialHadoop, based on the classic non-distributed closest pair divide-and-conquer algorithm [Preparata and Shamos, 1985], is described in [Eldawy et al., 2013]. The proposed MapReduce

© 2021 García-García, F.J.

algorithm is composed of three steps: *partitioning*, *local closest pair* and *global closest pair*.

The general schema of the algorithm does not differ much from previous algorithms. The main difference is that if only the closest pair of each of the partitions is used, points that contribute to the final result could be omitted. In particular, if $distCP_i$ is the distance of the closest pair in the $\mathbb{P}_i$ partition, each *map* function must return the points that are at most a $distCP_i$ distance value from the boundaries of $\mathbb{P}_i$ to be able to compare them with points of neighboring partitions in the *reduce* function. Finally, note that for the algorithm to work correctly, partitions must not overlap, so no duplicated points are present.

4.2.9 Voronoi-Diagram Query

The Voronoi-Diagram Query (VDQ), given a spatial dataset of points $\mathbb{P}$, returns the Voronoi-Cells that form the Voronoi-Diagram (VD) that adopts the points of $\mathbb{P}$ as pivots or generators. An example of this spatial query could be to *find the regions associated with each post office*, in order to know which is the area corresponding to each postcode. In this way, any building in the region is closer to its post office than to others. For instance, the *post offices* are the points of the spatial dataset $\mathbb{P}$ and the resulting *regions* build-up VD. The main objective of this query is different from the Voronoi-Diagram based partitioning algorithm presented in Section 3.2: the former obtains a VD from a massive set of pivots by calculating the boundaries of each cell, while the latter obtains the suitable pivots and the corresponding partitions to optimally distribute the points of a big spatial dataset. Figure 4.3 shows the VDQ of a given point dataset. In [Li et al., 2019] a VDQ MapReduce algorithm in SpatialHadoop is presented, and consists of the following four steps: *partitioning*, *local* VD, *pruning* and *merging*.

First, the *partitioning* step uses a built-in partitioning technique of SpatialHadoop, but in this case, it is mandatory to use a disjoint partitioning technique and it is specially optimized for Grid and STR+ partitioning. Next, the *local VD* step is part of a *map* function that uses a traditional divide-and-conquer algorithm [Preparata and Shamos, 1985] to get the local result for a given partition, that is, the set of Voronoi-Cells of the local VD. Following, the *pruning* step identifies final cells which are the output of the algorithm and non-final cells which will be modified during the *merge* step. Finally, the *merging* step employs the same union algorithm used in the *map* function to combine the partial results into the final set of polygons. The *merging* step is divided into two steps (vertical and horizontal merging) to handle a large number of partitions that cannot be carried out by a single machine. On the one hand, the *vertical merging* step is implemented as a *reduce* function, reuses the divide-and-conquer algorithm and the pruning rules to identify final cells. On the other hand, the *horizontal merging* step, which is executed after all *reducers* have finished, is a *CommitJob* function that merges the remaining non-final cells.

4.3 Enhancing SpatialHadoop with Distance-based Queries

Although SpatialHadoop provides several spatial queries and computational geometry algorithms in the *Operations* layer, there are various spatial queries (see Section 2.2) that are not present in this DSDMS. Therefore, this section shows various proposals for efficient distributed (MapReduce) algorithms for the principal distance-based queries (DBQs).

4.3.1 εDistance Range Query

The εDistance Range Query (εDRQ), given one points dataset $\mathbb{P}$, a query point q and a distance threshold ε, finds all points in the dataset $\mathbb{P}$ that fall on the circular shape, centered in q with radius ε. An example of this spatial query could be to *find all parking lots located 2 kilometers from my hotel*. In this case, *parking lots* are the objects of the spatial dataset $\mathbb{P}$, *the hotel* is q and, ε is the distance threshold (2 kilometers).

In [Eldawy and Mokbel, 2015], a generic *range query* operation in SpatialHadoop is proposed. But here, a εDRQ MapReduce algorithm on top of SpatialHadoop has been efficiently implemented [García-García et al., 2016a]. In general, the solution for εDRQ is similar to how Range Query, introduced in 4.2.1, is implemented in SpatialHadoop, except instead of having a generic query area, now we have a circular region defined by the query point q and a distance threshold ε. In Figure 4.4, we can see the operation of the εDRQ MapReduce algorithm in SpatialHadoop, which consists of two steps: *Global εDRQ* and *Local εDRQ*.

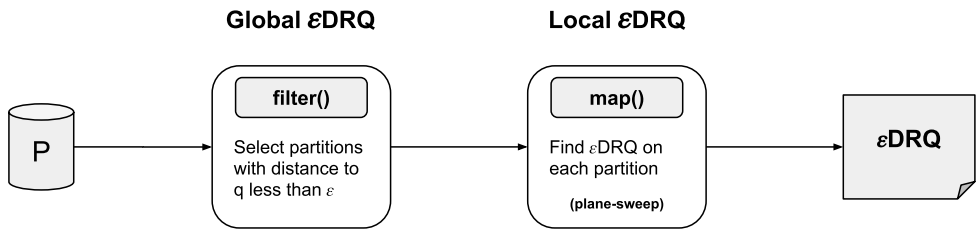


Figure 4.4: Overview of the εDRQ MapReduce algorithm in SpatialHadoop.

The *Global εDRQ* is implemented by a *filtering* function in which the partitions from $\mathbb{P}$, which intersect with the circular region centered at the query point q and with a radius equal to the distance threshold ε, are selected. Next, the *Local εDRQ* consists of a *map*-type task in which, for each selected partition, a *plane-sweep* algorithm is used to select only those points whose distance is smaller than ε. Finally, these points are written in files, obtaining the final query result.

© 2021 García-García, F.J.

4.3.2 kClosest Pairs Query

When two datasets ($\mathbb{P}$ and $\mathbb{Q}$) are combined, the kClosest Pairs Query (kCPQ) discovers the k pairs of points formed from these datasets having the k smallest distances between them (i.e., it reports the top-k pairs from $\mathbb{P} \times \mathbb{Q}$). An example of this spatial query could be to *find the 10 pairs of hotels and subway stations with the shortest distances between them.* In this case, *hotels* are the objects of the spatial dataset $\mathbb{P}$, *subway stations* are the objects from $\mathbb{Q}$ and $k = 10$.

In general, the kCPQ MapReduce algorithm in SpatialHadoop [García-García et al., 2016b, García-García et al., 2018b] is similar to how spatial join query (see Section 4.2.3) is performed [Eldawy and Mokbel, 2015]. This can be described as a generic top-k MapReduce job that takes a specific *plane-sweep* kCPQ algorithm [Roumelis et al., 2016] as a parameter. Therefore, having $\mathbb{P}$ and $\mathbb{Q}$ partitioned by some method (e.g., Grid) into n and m partitions, respectively; and generate $n \times m$ possible pairs of partitions to possibly combine. Then, every suitable pair of partitions (one from $\mathbb{P}$ and one from $\mathbb{Q}$) is sent as the input for the *map* phase. Each *mapper* reads the points from the pair of partitions and performs a plane-sweep (e.g., *Reverse Run* [Roumelis et al., 2016]) kCPQ algorithm between the points inside that pair of partitions. Figure 4.5 shows the three steps of the kCPQ MapReduce algorithm: *Global kCPQ*, *Local kCPQ*, and *Top kCPQ*.

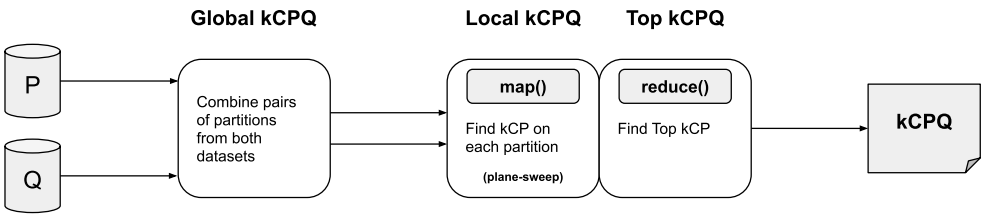


Figure 4.5: Overview of the kCPQ MapReduce algorithm in SpatialHadoop.

First, in the *Global kCPQ* step, pairs of partitions of spatial objects from both datasets are combined. Next, the *Local kCPQ* step aims to find the kCP of the spatial objects of the joined partitions from the datasets with a particular *plane-sweep* kCPQ algorithm. Finally, in the *Top kCPQ* step, a *reducer* examines the candidate pairs from each *mapper* and return the final set of the k closest pairs.

In Algorithm 1 we can see our proposed solution for kCPQ in SpatialHadoop, which consists of a single MapReduce job. The *map* function aims to find the k closest pairs between the local pair of partitions from $\mathbb{P}$ and $\mathbb{Q}$ with a particular *plane-sweep* kCPQ algorithm (*PSKCPQ*). *KMaxHeap* is a max binary heap [Cormen et al., 2009] used to keep record of local selected top-k closest pairs that will be processed by the *reduce* function. The output of the *map* function is in the form of a set of *DistanceAndPair* elements (called $\mathbb{D}$ in Algorithm 1), i.e., pairs of points from $\mathbb{P}$ and $\mathbb{Q}$ and their distances. As in every other top-k pattern, the *reduce* function can be used in the *combiner* to minimize the *shuffle data*. The *reduce* function aims to examine the candidate *DistanceAndPair* elements and return the final set of the k closest pairs. It takes as input a set

of *DistanceAndPair* elements from every *mapper* and the number of pairs (k). It also employs a max binary heap, called *CandidateKMaxHeap*, to calculate the final result. Each *DistanceAndPair* element is inserted into the heap if its distance value is less than the distance value of the heap root. Otherwise, that pair of points is discarded. Finally, candidate pairs that have been stored in the heap are returned as the final result and stored in the output file.

Algorithm 1 kCPQ MapReduce Algorithm

```
1: function MAP(ℙ: set of points, ℚ: set of points, k: number of pairs)
2:     SortX(ℙ)
3:     SortX(ℚ)
4:     KMaxHeap ← PSKCPQ(ℙ, ℚ, k)
5:     if KMaxHeap is not empty then
6:         for all DistanceAndPair ∈ KMaxHeap do
7:             output(null, DistanceAndPair)
8:         end for
9:     end if
10: end function

11: function COMBINE, REDUCE(null, 𝔻: set of DistanceAndPair, k: number of pairs)
12:     Initialize(CandidateKMaxHeap, k)
13:     for all DistanceAndPair ∈ 𝔻 do
14:         Insert(CandidateKMaxHeap, DistanceAndPair)
15:     end for
16:     for all candidate ∈ CandidateKMaxHeap do
17:         output(null, candidate)
18:     end for
19: end function
```

In order to make use of the local indices that SpatialHadoop provides, a version of the kCPQ algorithm using R-trees similarly to Spatial Join Query, described in Section 4.2.3, has been implemented. This new distributed kCPQ algorithm can be applied when both input datasets are indexed through a local R-tree index. Moreover, it follows the same scheme presented in Algorithm 1, consisting of a single MapReduce job whose only difference is the processing performed in the *map* function, keeping the *reduce* function unmodified. In this case, the *map* function applies a *plane-sweep* algorithm over the nodes of the R-trees, as described in [Corral et al., 2004b]. This algorithm consists of traversing both R-trees in a Best-First order, keeping a global min binary heap [Cormen et al., 2009] prioritized by the minimum distance between the considered pairs of MBRs. When dealing with leaf nodes, a *plane-sweep* algorithm is applied to the elements that are contained on them, whereas the δ value is updated appropriately (δ is the distance of the k-th closest pair of points discovered so far). In the case of internal nodes, *plane-sweep* is also applied for processing two internal nodes; the MBR pairs with a minimum distance greater than δ are pruned. We have chosen the Best-First traversal order for the combination of the two R-trees since it is the fastest algorithm for processing of kCPQs according to [Corral et al., 2004b].

4.3.3 εDistance Join Query

The εDistance Join Query (εDJQ) reports all possible pairs of spatial objects from two different spatial datasets ($\mathbb{P}$ and $\mathbb{Q}$), having a distance of each other smaller than a

© 2021 García-García, F.J.

distance threshold ε. An example of this spatial query could be to *find all pairs of hotels and subway stations that are at most 100 meters between them*. In this case, *hotels* are the objects of the spatial dataset $\mathbb{P}$, *subway stations* are the objects from $\mathbb{Q}$ and the value of distance threshold ε is *100 meters*.

The εDJQ can be considered as an extension of the kCPQ, where the distance threshold of the pairs is known beforehand and the processing strategy (e.g., *plane-sweep* technique) is the same as in the kCPQ for generating the candidate pairs of the final result. Therefore, the method for the εDJQ in MapReduce, adapting from kCPQ in SpatialHadoop [García-García et al., 2016b, García-García et al., 2018b], is a *Map*-based join algorithm (Figure 4.6) which is composed of the following two steps: *Global εDJQ* and *Local εDJQ*.

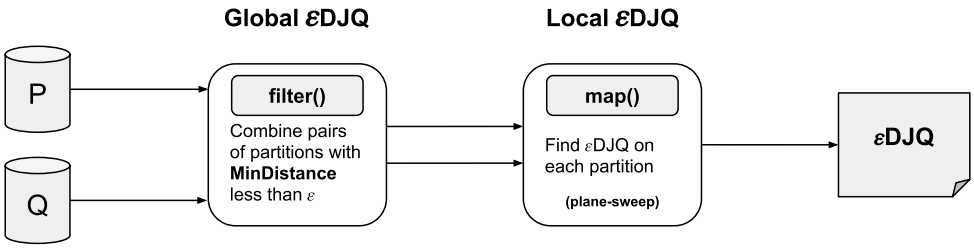


Figure 4.6: Overview of the εDJQ MapReduce algorithm in SpatialHadoop.

Algorithm 2 εDJQ MapReduce Algorithm

```
1: function MAP(ℙ: set of points, ℚ: set of points, ε: threshold distance)
2:     SortX(ℙ)
3:     SortX(ℚ)
4:     Results ← PSεDJQ(ℙ, ℚ, ε)
5:     for all DistanceAndPair ∈ Results do
6:         output(null, DistanceAndPair)
7:     end for
8: end function

9: function filtering(ℂℙ: set of cells, ℂℚ: set of cells, ε: threshold distance)
10:    for all c ∈ ℂℙ do
11:        for all d ∈ ℂℚ do
12:            minDistance ← MinDistance(MBR(c), MBR(d))
13:            if minDistance ≤ ε then
14:                output(c, d)
15:            end if
16:        end for
17:    end for
18: end function
```

First, $\mathbb{P}$ and $\mathbb{Q}$ are partitioned by some method (e.g., Grid) into two sets of partitions, with n and m partitions of points, respectively. Then in the *Global εDJQ* step, every possible pair of partitions is sent as input for the *filtering* function (see Algorithm 2). This function takes combinations of pairs of partitions, in which the datasets of points are partitioned, and a distance threshold ε as input, and it prunes that pairs of cells

that have minimum distances larger than ε. By using SpatialHadoop built-in function *MinDistance*, we can calculate the minimum distance between two partitions (i.e., this function computes the minimum distance between the two MBRs, Minimum Bounding Rectangles, of the two cells).

In the *Local εDJQ* step, each *mapper* (see Algorithm 2) reads the points of a pair of filtered partitions, and performs a *plane-sweep* εDJQ algorithm [Roumelis et al., 2016] (variation of the *plane-sweep* kCPQ algorithm) between the points inside that pair of partitions. The results from all *mappers* are just combined in the *reduce* phase and written into HDFS files, storing only the pairs of points with distance less than ε.

In addition, we can use the local indices provided by SpatialHadoop to obtain improvements in the performance of the previous εDJQ MapReduce algorithm. This new algorithm follows the same scheme of a single MapReduce job, whose only difference is the processing of the distance-based join query that is realized in the *map* function, maintaining the *filtering* function without any modification. In this case, we have locally indexed the data in each partition by R-tree structures that we can use to process the query. The algorithm consists of performing an iterative Depth-First search over the two R-trees. That is, for each pair of internal nodes, one from each index, the *MinDistance* between their MBRs is calculated; if it is larger than ε, then this pair is pruned. Otherwise, the children of the nodes will be checked in the next step, following a depth-first order. When the leaf nodes are reached, the same *plane-sweep* algorithm, as the one without local indices, is applied. We have chosen the iterative Depth-First traversal order for the combination of two R-trees and not the Best-First one because, if ε is large enough, the global min binary heap can grow very quickly and exceed the available main memory and, thus management of secondary memory is needed and the response time of the algorithm execution will be significantly incremented.

4.3.4 kNearest Neighbor Join Query

The kNearest Neighbor Join Query (kNNJQ) is one of the most studied DJQs when two datasets ($\mathbb{P}$ and $\mathbb{Q}$) are combined. This query, given two datasets of points ($\mathbb{P}$ and $\mathbb{Q}$) and a positive number k, finds for each point of $\mathbb{P}$, the k nearest neighbors of this point in $\mathbb{Q}$. One example for this query could be to *find the 10 closest points of interest to each hotel of a city*. In this case, *hotels* form the spatial dataset $\mathbb{P}$, *points of interest* are the spatial objects from $\mathbb{Q}$ and with a value of $k = 10$.

The proposed kNNJQ algorithm in [Nodarakis et al., 2016a], on two datasets $\mathbb{P}$ and $\mathbb{Q}$, consists of four phases of MapReduce jobs: *information distribution*, *primitive computation*, *update lists* and *unify lists*. In the *information distribution* phase, a uniform partitioning of the dataset $\mathbb{Q}$ is performed, and the number of elements from $\mathbb{P}$, which are inside the partitions of $\mathbb{Q}$, is counted. Then, in the *primitive computation* phase, an initial response is provided by calculating the kNNQ for each point p_i of $\mathbb{P}$ with the points of $\mathbb{Q}$ that are in the partition where p_i is located. Once this phase is completed, it is necessary to refine these initial kNN lists for each point of $\mathbb{P}$, if there have been found less than k neighbors, or if there are nearby partitions that overlap with the distance to each k-th nearest neighbor. All this refinement is done in the *update lists* phase, where new non-final kNN lists are obtained. Finally, in the *unify lists* phase, the merging of

© 2021 García-García, F.J.

all kNN lists, resulting from previous phases, is achieved, obtaining the final answer of the query.

To adapt and implement the previous kNNJQ MapReduce algorithm in SpatialHadoop, we have to carry out several extensions and improvements that are detailed below:

1. The *information distribution* phase is implemented using the partitioning methods provided by SpatialHadoop, allowing us to use non-uniform partitions, such as STR, Quadtree, etc., with the different improvements and particularities that they can offer. Figure 4.7 illustrates how the same dataset is partitioned using a uniform-based partitioning technique like Grid (on the left) and using a non-uniform-based partitioning technique like Quadtree (on the right), where the selected partitions are highlighted.

2. The *information distribution* phase is performed only once for each dataset and is reused for further kNNJ queries.

3. SpatialHadoop indices are used in each of these phases to accelerate the processing of the partitions.

4. Finally, an implementation of kNNQ based on a *plane-sweep* algorithm is carried out, which reduces the number of operations and calculations, obtaining a higher performance join operation.

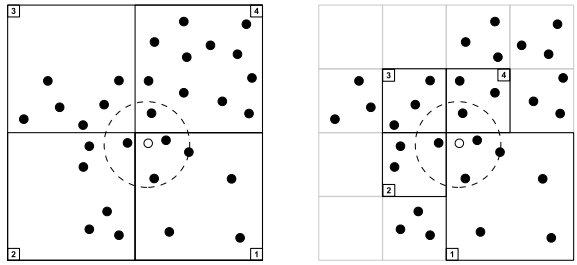


Figure 4.7: Uniform-based partitioning (Grid) vs. Non-uniform-based partitioning (Quadtree) in SpatialHadoop.

Figure 4.8 shows the phases of the proposed kNNJQ MapReduce algorithm [García-García et al., 2020c]: *Bin kNNJ*, *kNNJ on Overlapping Cells* and *Merge Results*. The first phase, *Bin kNNJ* (*information distribution* and *primitive computation* in [Nodarakis et al., 2016a]), that consists of a Bin-Spatial Join of the input datasets in which the join operator is the kNNQ, is accomplished. As described in Algorithm 3, in the *map* function of the *Bin kNNJ* phase, each point $p_i \in \mathbb{P}$ is combined with the partition in which it is located in the dataset $\mathbb{Q}$, so that in the *reduce* function, the *plane-sweep* kNNQ (PSKNNQ algorithm) of that point, with the points of $\mathbb{Q}$ in the same partition, is executed. The result of this phase is a kNN list for each point $p_i \in \mathbb{P}$. Then a completeness

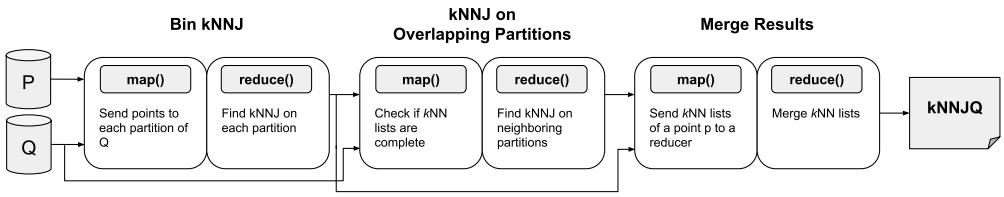


Figure 4.8: Overview of the kNNJQ MapReduce algorithm in SpatialHadoop.

check is made to find which of the previous kNN lists are not final and therefore it is necessary to continue with their processing. As shown in Algorithm 4, for the *kNNJ on Overlapping Partitions* phase (*update lists* in [Nodarakis et al., 2016a]), in the *map* function is checked (using the *GetOverlappedPartitions* function) if the previous kNN lists for each point $p_i \in \mathbb{P}$ contain less than k results (line 26) and also if there are neighboring partitions that overlap with the circular range, centered on $p_i \in \mathbb{P}$ and with radius the distance to the current k-th nearest neighbor (line 30). These points are then sent together with the calculated neighboring partitions to the *reduce* phase, where another *plane-sweep* kNNQ will be performed for each partition. Finally, the *Merge Results* phase (*unify lists* in [Nodarakis et al., 2016a]) consists of collecting the non-final kNN lists of the two previous phases in the *map* function, obtaining the final kNNQ results for each point $p_i \in \mathbb{P}$ in the *reduce* function.

Algorithm 3 Bin kNNJ Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ)
2:     partition ← FindPartition(𝒫^ℚ, p)
3:     output(partition.id, p)
4: end function

5: function REDUCE(partitionId: current partition, PQ: set of points in partition, k: number of neighbors)
6:     P ← GetPointsFromP(PQ)
7:     Q ← GetPointsFromQ(PQ)
8:     for all p ∈ P do
9:         Initialize(kNNList, k)
10:        kNNList ← PSKNNQ(Q, p, k)
11:        output(p, kNNList)
12:    end for
13: end function
```

4.3.5 εDistance Range Join Query

The εDistance Range Join Query (εDRJQ) given two points datasets ($\mathbb{P}$ and $\mathbb{Q}$) and a distance threshold ε, finds, for each point $p_i \in \mathbb{P}$, all points in $\mathbb{Q}$ that fall within the circular shape, centered on p_i with radius ε. This query is also called *spatial range join query*. One example for this query could be to *find the houses with their distances to shopping centers being at most 1500 meters*. In this case, *shopping centers* are the spatial objects from $\mathbb{P}$, *houses* form the spatial dataset $\mathbb{Q}$ and with a value of $\varepsilon = 1500$ meters.

© 2021 García-García, F.J.

Algorithm 4 kNNJ on Overlapping Partitions Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ, k: number of neighbors)
2:     origin ← IsFromPorQ(p)
3:     if origin is from Q then
4:         partition ← FindPartition(𝒫^ℚ, p)
5:         output(partition.id, p)
6:     else
7:         overlappedParts ← GetOverlappedPartitions(𝒫^ℚ, p, k)
8:         for all partition ∈ overlappedParts do
9:             output(partition.id, p)
10:        end for
11:    end if
12: end function

13: function REDUCE(partitionId: current partition, PQ: set of points in partition, k: number of neighbors)
14:    P ← GetPointsFromP(PQ)
15:    Q ← GetPointsFromQ(PQ)
16:    for all p ∈ P do
17:        Initialize(kNNList, k)
18:        kNNList ← PSKNNQ(Q, p, k)
19:        output(p, kNNList)
20:    end for
21: end function

22: function GetOverlappedPartitions(𝒫^ℚ: set of partitions from ℚ, p: point from ℙ, k: number of neighbors)
23:    kNNList ← GetKnnList(p)
24:    nnNumber ← kNNList.size
25:    radius ← GetKthDistance(kNNList)
26:    while nnNumber < k do
27:        radius ← Increase(radius)
28:        nnNumber ← GetNumberOfNeighbors(𝒫^ℚ, p, radius)
29:    end while
30:    overlappedPartitions ← RangeQuery(𝒫^ℚ, p, radius)
31:    return overlappedPartitions
32: end function
```

Note that just as we can formulate and implement the εDJQ as a derived version of kCPQ in which the pruning distance ε is known. Similarly, we can define the εDRJQ based on the kNNJQ algorithm by means of a *Reduce*-based join algorithm, as we can observe in Figure 4.9. Of the three phases discussed above, due to the fact that the ε

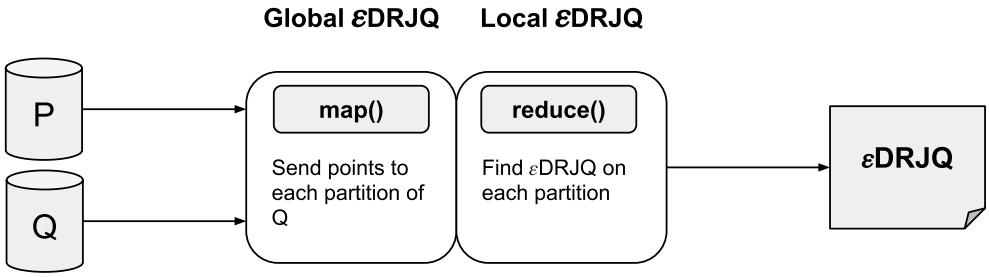


Figure 4.9: Overview of the εDRJQ MapReduce algorithm in SpatialHadoop.

distance is known in advance, the *Bin kNNJ* and *kNNJ on Overlapping Cells* phases are combined into just only one and since we do not have to unify kNN lists, the last phase is not needed to be performed.

After studying the results of εDRJQ and εDJQ, they turn out to be equivalent, i.e., both DJQs report the same result set. The main difference resides in the order of the pairs returned in the final result. While $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports pairs clustered around every point in $\mathbb{P}$ (i.e., for each point $p_i \in \mathbb{P}$, it returns all points in $\mathbb{Q}$ overlapping with a circular shape, centered on p_i with radius ε), $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports unrelated pairs of points (i.e., it returns a sequence of pairs within a distance threshold (ε) from each other). Another difference between these two DJQs is the algorithmic technique used to solve them. While εDRJQ is processed based on multiple executions of εDRQ on $\mathbb{Q}$ for every point in $\mathbb{P}$, the algorithm for solving εDJQ is based on a sort-merge join approach (i.e., it is a *plane-sweep* algorithm between $\mathbb{P}$ and $\mathbb{Q}$).

4.3.6 Reverse kNearest Neighbor Query

The Reverse kNearest Neighbor Query (RkNNQ), given a set of points $\mathbb{P}$ and a query point q, finds the points from $\mathbb{P}$ that have q as one of their k closest points. As an example of this query, we could want to *find the gym that would be affected by the opening of a new one*. In this case, *gyms* form the spatial dataset $\mathbb{P}$, *new gym* is the query object q and $k = 1$.

In the following sections we present two different approaches in order to adapt and implement RkNNQ in SpatialHadoop: *MRSFT* [García-García et al., 2017b], which is based on the multistep *SFT* [Singh et al., 2003] algorithm, and *MRSLICE* [García-García et al., 2019], which is a novel MapReduce version of SLICE [Yang et al., 2014] algorithm.

4.3.6.1 MRSFT - SFT MapReduce algorithm

In general, the parallel and distributed RkNNQ algorithm based on the *SFT* algorithm [Singh et al., 2003] consists of two phases namely *filtering phase* and *verification phase*. Assuming that $\mathbb{P}$ is the dataset to be processed and q is the query point, the basic idea is to have $\mathbb{P}$ partitioned by some method (e.g., Grid) into n cells or partitions of points.

In the *filtering phase*, a MapReduce-based *kNNQ* is executed in order to find every possible candidate point from $\mathbb{P}$. To carry out that, we find the partition from $\mathbb{P}$ where q is located. A first answer for the $kNN(\mathbb{P}, q, K)$ is obtained, and we use the distance from the *k-th* point to q in order to find if there are possible candidate points in other partitions close to q. To ensure an exact result, the value of K must be greater than k ($K \gg k$) as proposed in [Tao et al., 2004], at a magnitude of at least $K = 10 \times d \times k$, where d is the dimensionality of the points in the dataset being examined (e.g., for 2d points, $K = 20 \times k$).

In the *verification phase*, a range query, with a circle centered in q and that distance as radius, is run to finally answer the *kNNQ*. The candidate points with their distance to the query point q are written into HDFS files in order to be the input for the next jobs. At this moment, each candidate point is checked to verify if it is part of the final

© 2021 García-García, F.J.

answer of the query. That is, it finds the number of points that are part of the range query centered on the candidate point and radius the distance to q. If this *number* is less than k, the candidate point is verified to be an RkNN of q. Finally, the results are written into HDFS files, storing only the points coordinates and the distance to q.

4.3.6.2 MRSLICE - SLICE MapReduce algorithm

SLICE is the state-of-the-art RkNNQ algorithm since it is the best for all considered performance parameters in terms of CPU cost [Yang et al., 2015]. Like most of the RkNNQ algorithms, SLICE consists of two phases, namely *filtering phase* and *verification phase*. SLICE improves the filtering power of the six-regions approach [Stanoi et al., 2000], by using its strength of being a cheaper filtering strategy. Moreover, it is important to note that the *filtering phase* dominates the total query processing cost [Yang et al., 2014].

Filtering phase. SLICE divides the space of a set of points $\mathbb{P}$ around the query point q into multiple equally sized regions based on angle division. The experimental study in [Yang et al., 2014] demonstrated that the best performance is achieved when the space is divided into 12 equally sized regions. Given a region R and a point $p \in \mathbb{P}$, we can define the half-space that divides them as $H_{p:q}$. The intersection of this half-space with the limits of the region R allows us to obtain the *upper arc* of p with respect to R ($r^U_{p:R}$) and the *lower arc* of p with respect to R ($r^L_{p:R}$) whose radii meet the condition of $r^U > r^L$. In [Yang et al., 2014], it is shown that a point p' in the region R can be pruned by the point p if p' lies outside its upper arc, i.e., $dist(p', q) > r^U_{p:R}$. Note that a point $p' \in R$ cannot be pruned by p if p' lies inside its lower arc, i.e., $dist(p', q) < r^L_{p:R}$. The *bounding arc* of a region R, denoted as r^B_R, is the k-th smallest upper arc of that region and it is used to easily prune points or set of points. Note that any point p' that lies in R with $dist(p', q) > r^B_R$ can be pruned by at least k points. A point p is called *significant* for the region R if it can prune points inside it, i.e., only if $r^L_{p:R} < r^B_R$. Therefore, SLICE maintains a list of significant points for each region that will be used in the *verification phase*. The following lemmas are used in this phase to reduce the search space by pruning non-significant points.

Lemma 1. *A point $p \in R$ cannot be a significant point of R if $dist(p, q) > 2r^B_R$*

Proof. Shown in [Yang et al., 2014] as Lemma 4. □

Lemma 2. *A point $p \notin R$ cannot be a significant point of R if $dist(M, p) > r^B_R$ and $dist(N, p) > r^B_R$ where M and N are the points where the bounding arc of R intersects the boundaries of R*

Proof. Shown in [Yang et al., 2014] as Lemma 5. □

These lemmas can be easily extended to a complex entity e (i.e., e does not contain any significant point), by comparing $MinDistance(q, e)$ with the bounding arc of each region that overlaps with e.

Verification phase. SLICE tries to reduce the search space by using the following lemma:

Lemma 3. *A point p prunes every point $p' \in R$ for which $dist(p', q) > r^U_{p:R}$ where $0° < maxAngle(p, R) < 90°$*

Proof. Shown in [Yang et al., 2014] as Lemma 1. □

To do this, each point $p \in \mathbb{P}$ is checked against several derived pruning rules: (1) if $dist(q, p) > r^B_R$, p is not part of the RkNNQ answer; (2) if $dist(q, p)$ is smaller than the k-th lower arc of R, p cannot be pruned; and (3) if once the maximum and minimum angles have been calculated of p with respect to q, there is at least one region R with $r^B_R > dist(q, p)$, p can be part of the RkNNQ answer. Once the search space has been reduced, each candidate point is verified as a result of RkNNQ if at most there are k-1 significant points closest to the query object in the region R in which it is located.

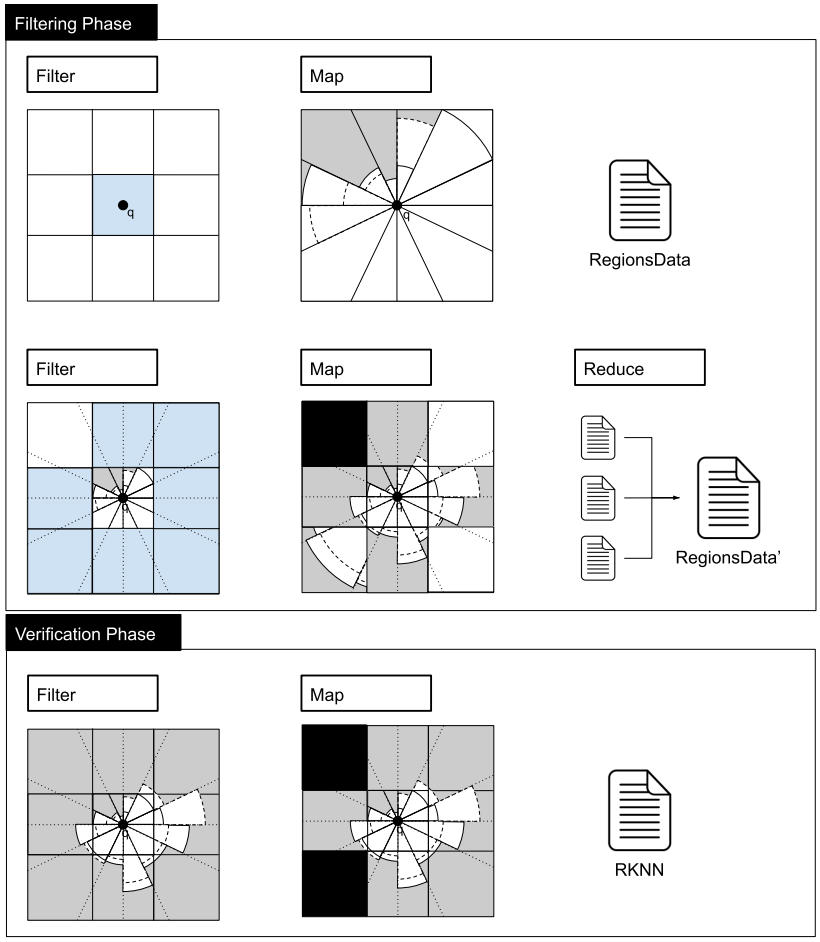


Figure 4.10: Overview of *MRSLICE* algorithm in SpatialHadoop.

© 2021 García-García, F.J.

In general, our distributed *MRSLICE* algorithm for SpatialHadoop based on SLICE algorithm [Yang et al., 2014] consists of three MapReduce jobs:

- **Phase 1**. The *Filtering phase* of SLICE is performed on the partition in which the query object is located.
- **Phase 1.B (optional)**. The filtering process is continued on those partitions that are still part of the search space.
- **Phase 2**. The *Verification phase* is carried out with those partitions that have not been pruned as a result of applying Phases 1 and 1.B.

From Figure 4.10, and assuming that $\mathbb{P}$ is the set of points to be processed and q is the query point, the basic idea is to have $\mathbb{P}$ partitioned by some method (e.g., Grid) into n blocks or partitions of points ($\mathcal{P}^{\mathbb{P}}$ denotes the set of partitions from $\mathbb{P}$). The *Filtering phase* consists of two MapReduce jobs, being optional the second one since in the case of all significant points are found by the first job, the execution of the second job is not necessary. Finally, the *Verification phase* is a MapReduce job that will check if the non-pruned points are part of the RkNNQ answer.

Phase 1: Filtering phase. In the first MapReduce job (Algorithm 5), the *Filter* function selects the partition of $\mathbb{P}$ in which q is found. Then, in the *Map phase*, the *Filtering phase* is applied as described in SLICE. That is, $\mathbb{P}$ is divided into t regions of equal space, and the list of k smallest upper arcs is obtained for each R_i region along with its $r^B_{R_i}$ and its list of significant points, that will be returned as *RegionsData* for further use. To accelerate the *Filtering phase*, an R-tree index is used per partition, and a *heap* is utilized to store the nodes based on their minimum distance to q. As the R-tree nodes are traversed, the *facilityPruned* function from [Yang et al., 2014] is used (Algorithm 5 line 13), pruning the nodes which with the current *RegionsData* do not contain significant points. In the case of leaf nodes, the points are processed by the *pruneSpace* function from [Yang et al., 2014] (Algorithm 5 line 15), which is responsible to update the *RegionsData* information. Finally the k-th lower arc is calculated to be used in the next phase.

Phase 1.B: Filtering phase (optional). The second MapReduce job (Algorithm 6) runs only if the function *Filter* returns some partition. That is, the *facilityPruned* [Yang et al., 2014] function is executed on each of the partitions by comparing its minimum distance to q with the bounding arc of each region R_i with which it overlaps. Note that the upper left partition of $\mathbb{P}$ in Figure 4.10 is in the shaded area, and therefore can be pruned. However, the other partitions can contain significant points, and the *Filtering phase* must be applied to them during the *Map phase*. The result of each of the partitions will be merged on the *Reduce phase* to obtain the k-th upper arcs, bounding arcs, and final significant points (*RegionsData'*).

Algorithm 5 *MRSLICE* Filtering - Phase 1

```
1: function FILTER(P^P: set of partitions from P, q: query point)
2:     return FindPartition(P^P,q)
3: end function

4: function MAP(MBR: Minimum Bounding Rectangle of P, r: root of R-tree of actual partition, q: query
   point, k: number of points, t: number of equally sized regions)
5:     RegionsData.regions ← DivideSpace(MBR,q,t)
6:     RegionsData ← SliceFiltering(r,q,k,RegionsData)
7:     return RegionsData
8: end function

9: function SliceFiltering(r: root of R-tree, q: query point, k: number of points, RegionsData: SLICE
   Regions Data)
10:    Insert(Heap,null,r)
11:    while Heap is not empty do
12:        entry ← Pop(Heap)
13:        if !facilityPruned(entry,q,k,RegionsData) then
14:            if isLeaf(entry) then
15:                pruneSpace(entry,q,k,RegionsData)
16:            else
17:                for all child ∈ entry.children do
18:                    key ← MinDistance(q,child)
19:                    Insert(Heap,key,child)
20:                end for
21:            end if
22:        end if
23:    end while
24:    for all region ∈ RegionsData.regions do
25:        region.boundingArc ← FindkUpperArc(region)
26:    end for
27:    RegionsData.minLowerArc ← ComputeMinLowerArc(RegionsData.regions)
28:    return RegionsData
29: end function
```

Algorithm 6 *MRSLICE* Filtering - Phase 1.B

```
1: function FILTER(P^P: set of partitions from P, q: query point, RegionsData: SLICE Regions Data)
2:     for all p ∈ P^P do
3:         if !facilityPruned(p,q,RegionsData) then
4:             Insert(Result, p)
5:         end if
6:     end for
7:     return Result
8: end function

9: function MAP(r: root of R-tree of actual partition, q: query point, k: number of points, RegionsData:
   SLICE Partition Data)
10:    RegionsData′ ← SliceFiltering(r,q,k,RegionsData)
11:    return RegionsData′
12: end function

13: function REDUCE(RegionsDataArray: Array of SLICE Partition Data)
14:    RegionsData′ ← RegionsDataArray[0]
15:    for all RegionsData ∈ RegionsDataArray do
16:        RegionsData′.P ← Merge(RegionsData.regions, RegionsData′.P)
17:    end for
18:    for all partition ∈ RegionsData′.P do
19:        partition.kUpperArc ← FindkUpperArc(partition)
20:    end for
21:    RegionsData′.minLower ← ComputeMinLower(RegionsData′.P)
22:    return RegionsData′
23: end function
```

© 2021 García-García, F.J.

Algorithm 7 *MRSLICE* Verification - Phase 2

```
1: function FILTER(P^P: set of partitions from P, q: query point, RegionsData: SLICE partition data)
2:     for all p ∈ P^P do
3:         if !USERPRUNED(p,q,RegionsData) then
4:             INSERT(Result, p)
5:         end if
6:     end for
7:     return Result
8: end function

9: function MAP(r: root of R-tree of actual partition, q: query point, k: number of points, RegionsData:
   SLICE Partition Data)
10:    INSERT(Stack,r)
11:    while Stack is not empty do
12:        entry ← POP(Stack)
13:        if !USERPRUNED(entry,q,RegionsData) then
14:            if ISLEAF(entry) then
15:                if ISRKNN(entry,q,k,RegionsData) then
16:                    OUTPUT(entry)
17:                end if
18:            else
19:                for all child ∈ entry.children do
20:                    INSERT(Stack,child)
21:                end for
22:            end if
23:        end if
24:    end while
25: end function

26: function ISRKNN(entry: candidate point, q: query point, k: number of points, RegionsData: SLICE
    Partition Data)
27:    region ← FINDREGION(entry,q,RegionsData)
28:    counter ← 0
29:    for all p ∈ region do
30:        if dist(entry,q) ≤ r^L_{p:R} then
31:            return true
32:        end if
33:        if dist(entry,p) < dist(entry,q) then
34:            counter ← counter + 1
35:            if counter ≥ k then
36:                return false
37:            end if
38:        end if
39:    end for
40:    return true
41: end function
```

Theorem 1 (Completeness)**.** *MRSLICE Filtering returns all significant points.*

Proof. It suffices to show that *MRSLICE Filtering* does not discard significant points. A point p is discarded by *MRSLICE Filtering* only if it is pruned by the *facilityPruned* function by either applying lemma 1 or 2. In any of these cases, it is shown in [Yang et al., 2014] that any point that is not inside the area defined by these lemmas is not a significant point. Points that are discarded can be split into different categories:

Phase 1. Points are pruned in this phase like in the non-distributed SLICE version using Algorithm 5.

Phase 1.B - Partition granularity. Using the *FILTER* function in Algorithm 6, partitions that do not contain any significant point are pruned by applying both lemmas 1 and 2 to the partition as a complex entity.

Phase 1.B - Point granularity. Points are discarded in the *Map Phase* in the same way that in *Phase 1* only on non-pruned partitions.

Phase 1.B - Merging RegionsData. Finally, when merging *RegionsData* in the *Reduce Phase*, both lemmas 1 and 2 are again used to discard non-significant points. □

Phase 2: Verification phase. Finally, a MapReduce job (Algorithm 7) is executed on the partitions that are not pruned by the *Filter* function when applying the pruning rules described above for SLICE, using the *userPruned* function (Algorithm 7 line 3). That is, the algorithm is executed on those partitions that contain some white area. In the *Map phase*, the R-tree, which indexes each partition, is traversed with the help of a *stack* data structure, and the search space is reduced by using the *userPruned* function again. Furthermore, the pruning rules are applied again to the points that are in the leaf nodes, and finally, they are verified if they are part of the final RkNNQ answer. The *isRkNN* function (Algorithm 7 line 15) verifies a candidate point p as part of the answer if there are at most k-1 significant points closer to p than q in the region R_i in which it is located.

Theorem 2 (Correctness)**.** *MRSLICE Verification algorithm returns the correct RkNNQ set.*

Proof. It suffices to show that *MRSLICE Verification* does not (a) discard RkNNQ points, and (b) return non-RkNNQ points. First, the *MRSLICE Verification* algorithm only prunes away those points or/and entries by using the pruning rules derived from lemma 3, by using the information identified by the *MRSLICE Filtering* algorithm, which guarantees no false negatives. Second, every non-pruned point is verified by the *isRkNN* function, which ensures no false positives. We prove that these points are guaranteed to be RkNNQ points by contradiction. Assume a point p returned by *MRSLICE* algorithm is not an RkNNQ point. Then, there exist k significant points closer to p than q, and p is also returned as part of the RkNNQ answer. But then p could not be in the RkNNQ answer since it would have been discarded in line 35 of the *isRkNN* function in Algorithm 7. □

4.4 Extensions and Improvements of DJQs

When it comes to extending and improving the DJQ MapReduce algorithms, we must consider several factors that take into account the characteristics of real-world spatial objects and the execution environment of the DSDMSs. Therefore, we must analyze the factors described in Section 2.2 to deal efficiently with common problems like processing non-points spatial objects, boundary handling, and skewed data management. Furthermore, as a result of this analysis and the study of each distance-based query, several new pruning rules can be designed to reduce the search space and avoid unnecessary distance computations.

In this section, we first expose the extensions of the DJQ MapReduce algorithms to manage other geometric objects different to points. Next, we present new pruning rules for the kCPQ MapReduce algorithm based on the upper bound calculation of the

© 2021 García-García, F.J.

distance value of the k-th closest pair of the joined datasets. Finally, we present several improvements and pruning rules to the kNNJQ MapReduce algorithm to deal with the problems that arise when there are too many objects inside a particular partition, i.e., handling skewed data.

4.4.1 Extensions of the DJQ MapReduce algorithms for processing non-points spatial objects

Usually, real-world datasets are not only limited to points but include other geometric objects, like line-segments, polygons, regions, etc. For instance, a dataset containing the buildings of a city can use polygons, while line-segments can be used to represent roads. Because of this, it is necessary to extend the previously distributed algorithms to be able to process these datasets consisting of more complex spatial objects (**F.1**).

When extending our algorithms, we must modify each of the steps that compose them. Initially, we must take into account that the *replication method* (Grid and STR+) in SpatialHadoop avoids expanding partitions by replicating each object to all overlapping partitions. As a consequence, the query processor has to employ a *duplicate avoidance technique* to account for replicated objects. In our approach, we have used the *reference-point duplicate avoidance technique* [Eldawy and Mokbel, 2015], which consists of selecting a single point of the geometry and discarding the partitions in which the point is not found to avoid duplicates.

Furthermore, to simplify the operations and calculations of distances in the algorithms, the *MBR* (Minimum Bounding Rectangle) that covers the spatial objects will be used. Utilizing the MBR, instead of the exact geometrical representation of the spatial object, reduces its structure to two points (i.e., *min* and *max*), where the most significant spatial object features (position and extension) are maintained (**F.2**). Consequently, the MBR is an approximation widely employed. In this way, the *plane-sweep* algorithms only have to calculate the minimum distance between MBRs without computing complex calculations based on their shapes (e.g., calculate the distance between a convex polygon and a line-segment).

Figure 4.11 illustrates two complex spatial objects (i.e., a lake and a building) with their MBRs and reference points, and shows the minimum distance (MinDistance) between them (two MBRs) is calculated. This process is generally identified as the *filtering step* since it finds all MBRs of spatial objects that verify the query condition. Only in the final phase, the processing of the exact geometry of the spatial objects will be required for obtaining the exact distance values. Commonly known as *refinement step*, this step uses efficient computational geometry algorithms [de Berg et al., 2008] to produce the final query result (e.g., algorithm to compute the distance between two convex polygons).

4.4.2 Improvements for kCPQ in SpatialHadoop

It can be clearly seen that the performance of the proposed solution of the kCPQ MapReduce algorithm (Algorithm 1) will depend on the number of partitions in which the two

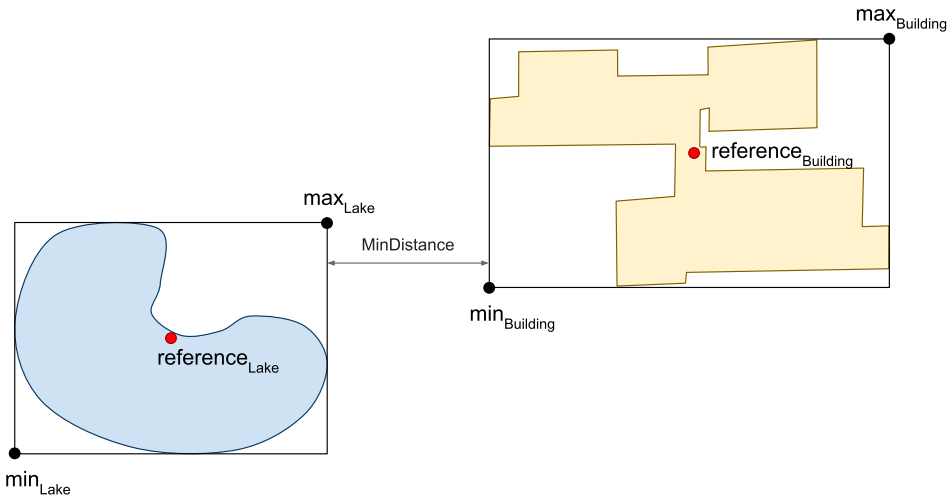


Figure 4.11: Example of two complex spatial objects (Lake vs. Building) with their MBRs, reference points and minimum distance between their MBRs.

sets of points are partitioned. That is, if the set of points $\mathbb{P}$ is partitioned into n partitions (the set $\mathcal{P}^{\mathbb{P}}$) and the set of points $\mathbb{Q}$ is partitioned in m partitions (the set $\mathcal{P}^{\mathbb{Q}}$), then we obtain $n \times m$ combinations of partitions or *map* tasks. Furthermore, we know that plane-sweep-based kCPQ algorithms use a pruning distance value, which is the distance value of the k-th closest pair found so far, to discard those combinations of pairs of points that are not necessary to consider as a candidate of the final query result. As suggested in [García-García et al., 2016b], we need to find in advance an upper bound of the distance value of the k-th closest pair of the joined datasets, called β.

In addition, we can use this β value in combination with the features of the global indexing that SpatialHadoop provides to further improve the pruning phase. Before the *map* phase begins, we exploit the global indices to prune partitions that cannot contribute to the final result. Using SpatialHadoop built-in function *MinDistance*, we can calculate the minimum distance between the two MBRs of the two partitions. That is, if we find a pair of partitions with points that cannot have a distance value smaller than or equal to β, we can prune this combination of pairs of partitions (Rule 1).

Rule 1. Pair of Partitions Pruning
Given two partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$, from $\mathbb{P}$ and $\mathbb{Q}$, respectively, and β is the upper bound of the distance value of the k-th closest pair of the two joined datasets. If MinDistance$(MBR(\mathcal{P}_i^{\mathbb{P}}), MBR(\mathcal{P}_j^{\mathbb{Q}})) > \beta$, then the pair of partitions $(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$ can be pruned, because they do not contain any pair of points with distance smaller than β.

Moreover, the computation of β can be carried out (a) by sampling globally both large datasets and executing a *PSKCPQ* (plane-sweep kCPQ) algorithm over the two samples, or (b) by appropriately selecting a specific pair of partitions to which the two

© 2021 García-García, F.J.

large datasets are partitioned and either (b1) by locally sampling them and executing a *PSKCPQ* algorithm over the two samples, or (b2) by applying an approximate variation of a plane-sweep kCPQ algorithm over the spatial objects of the selected pair. Finally, (a particular case of b1) if both datasets are partitioned employing a Voronoi-Diagram partitioning method, the local computation of β can use some of its properties. In the following subsections, we will see all these methods [García-García et al., 2018b, García-García et al., 2020b].

4.4.2.1 Computing β by Global Sampling

The first method of computing β can be seen in Algorithm 8 (computing β by global sampling algorithm), where we take a small sample from both sets of points ($\mathbb{P}$ and $\mathbb{Q}$) and calculate the k closest pairs using a plane-sweep-based kCPQ algorithm (*PSKCPQ* [Roumelis et al., 2016]) that is applied locally. Then, we set β equal to the distance of the k-th closest pair of the result and use this distance value as input for *mappers*. This β value guarantees that there will be at least k closest pairs if we prune pairs of points with larger distances in every *mapper*. Figure 4.12 shows the general schema of computing β (upper bound of the distance of the k-th closest pair) using global sampling, which is used to filter only pairs of partitions with a minimum distance of their MBRs smaller than or equal to β.

Algorithm 8 Computing β by global sampling Algorithm

```
1: function CALCULATEβ(ℙ: set of points, ℚ: set of points, ρ: global sampling ratio, k: number of pairs)
2:     SampledP ← SAMPLEMR(ℙ, ρ)
3:     SampledQ ← SAMPLEMR(ℚ, ρ)
4:     SORTX(SampledP)
5:     SORTX(SampledQ)
6:     KMaxHeap ← PSKCPQ(SampledP, SampledQ, k)
7:     if KMaxHeap is full then
8:         βDistanceAndPair ← POP(KMaxHeap)
9:         β ← βDistanceAndPair.Distance
10:        OUTPUT(β)
11:    else
12:        OUTPUT(∞)
13:    end if
14: end function

15: function PARTITIONSFILTER(𝒫^ℙ: set of partitions from ℙ, 𝒫^ℚ: set of partitions from ℚ, β: upper
    bound distance)
16:    for all c ∈ 𝒫^ℙ do
17:        for all d ∈ 𝒫^ℚ do
18:            minDistance ← MINDISTANCE(MBR(c), MBR(d))          ▷ MmDist for Voronoi
19:            if minDistance ≤ β then                            ▷ Rule 1
20:                OUTPUT(c, d)
21:            end if
22:        end for
23:    end for
24: end function
```

Moreover, we can further enhance the pruning phase using this β value and the global indexing that SpatialHadoop provides. Before the *map* phase begins, we exploit the global indices to prune partitions that cannot contribute to the final query result. *PARTITIONSFILTER* takes as input each combination of pairs of partitions in which

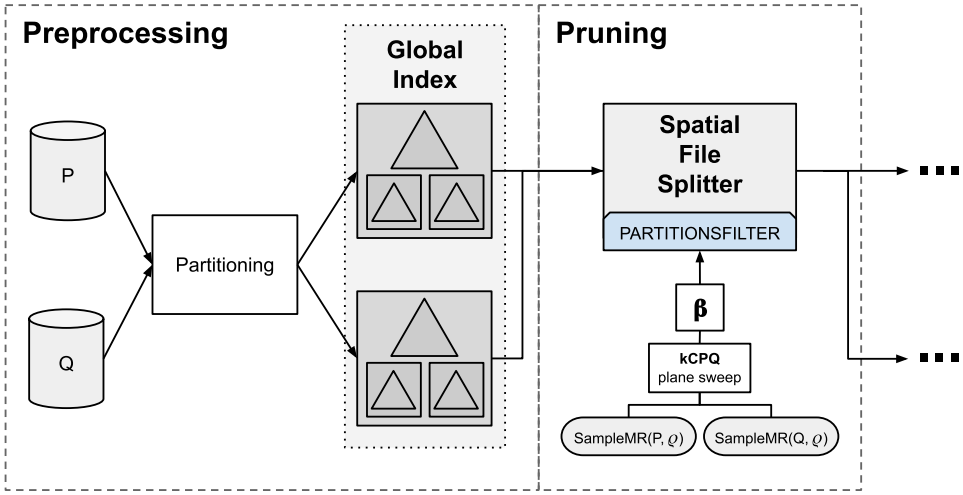


Figure 4.12: Schema for computing β by global sampling.

the input set of points are divided. Then, it uses Rule 1 to prune combinations of pairs of partitions that do not contribute to the final answer. Using different percentages of samples of the input datasets in Algorithm 8, we have obtained results with a significant reduction of execution time as explained later in the performance evaluation (see Section 4.5). Note that to obtain a sample from each dataset, we use a SpatialHadoop built-in MapReduce function, called *SampleMR*, which extracts a percentage of samples (sampling ratio ρ in %, $0.0 < \rho \leq 100.0$) following a sampling Without Replacement (WoR) pattern [Chaudhuri et al., 1999].

4.4.2.2 Computing β by Local Processing

Analyzing the above method for the β calculation, it is clearly observed that the highest time overhead occurs in the execution of the two calls to the *SampleMR* function since they are full-fledged MapReduce jobs. Therefore, to try to improve the previous algorithm and avoid calling the *SampleMR* function, we are looking to take advantage of the information provided by the partitions (global indexes) and other features of SpatialHadoop, and, thus, to make faster the β computation.

Global indices in SpatialHadoop provide the MBR of index partitions, as well as the number of elements contained in them. Thanks to that, we can get an idea of the distribution of data into each partition. To simplify the sampling process, we will find a suitable pair of partitions, that by their characteristics, may contain k closest pairs with a β value as small as possible. Then we can sample locally those partitions without having to execute a MapReduce job (as *SampleMR*).

Since we are looking for the closest k pairs, the most suitable pair of partitions is formed by those with an MBR that contains them, with the highest density of points, and

© 2021 García-García, F.J.

whose intersection area is the largest. The larger the intersection area of two partitions, the larger the probability that points in one set are near points in the other set. If the density is also higher, the distances between points are smaller, and therefore, we will be able to obtain better candidate pairs of partitions. Then, by *Pair Data Density Area Intersection*, $PDDAI(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$, we denote a metric that expresses the *suitability*, based on data density and area intersection, of these two partitions to allocate k closest pairs with as small distances as possible. It is exposed in Definition 4.1.

Definition 4.1. Pair Data Density Area Intersection, $PDDAI(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$
Given two partitions, $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$, $|\mathcal{P}_i^{\mathbb{P}}|$ is the number of elements inside partition $\mathcal{P}_i^{\mathbb{P}}$ (cardinality of $\mathcal{P}_i^{\mathbb{P}}$), $Area(MBR(\mathcal{P}_i^{\mathbb{P}}) \cup MBR(\mathcal{P}_j^{\mathbb{Q}}))$ is the area of the MBR that covers both MBRs of partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$ (union MBR), and $Area(MBR(\mathcal{P}_i^{\mathbb{P}}) \cap MBR(\mathcal{P}_j^{\mathbb{Q}}))$ is the area of the intersection MBR of both MBRs of partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$. Then the Pair Data Density Area Intersection of two partitions, $PDDAI(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$, is defined as
$$PDDAI(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}}) = \frac{|\mathcal{P}_i^{\mathbb{P}}|+|\mathcal{P}_j^{\mathbb{Q}}|}{Area(MBR(\mathcal{P}_i^{\mathbb{P}}) \cup MBR(\mathcal{P}_j^{\mathbb{Q}}))} \times (1 + Area(MBR(\mathcal{P}_i^{\mathbb{P}}) \cap MBR(\mathcal{P}_j^{\mathbb{Q}})))$$

We will select the pair of partitions with the *maximum* value of this metric so that we will have the pair of partitions with the larger combination of density of points and area of intersection. In the case of pairs of partitions that do not intersect, only the data density is taken into count.

4.4.2.2.1 Computing β by Local Sampling

The new method for computing β can be seen in Algorithm 9 (computing β by *local sampling* algorithm), which follows a similar scheme to that of global sampling. There is a new step, the *SELECTPARTITIONS* function, in which the pair of partitions (c and d) having the highest value for the *PDDAI(c,d)* metric is obtained. To do this, the partitions of the two global indices are joined by calculating the *PDDAI* metric for each combination. Then the candidate pair of partitions is sampled by recalculating the sampling ratio ρ since we are dealing with a subset of elements, and we want to obtain the same number of elements as for the case of global sampling. Once the samples are obtained *locally* and verified that they reside in memory, a local plane-sweep-based kCPQ algorithm (*PSKCPQ*) is applied to obtain β. Finally, this value is used in the *PARTITIONSFILTER* function exactly as in Algorithm 8.

4.4.2.2.2 Computing β by Local Approximate Methods

Several approximation techniques (ε-approximate, α-allowance, N-consider and *Time*-consider) have been proposed for distance-based queries using R-trees in [Corral and Vassilakopoulos, 2005]. These techniques can also be used to obtain approximate solutions with a faster execution time, trying to find a balance between computational cost and the accuracy of the query result. N-consider is an approximate technique that depends on the number of points to be combined, and *Time*-consider depends only on the

Algorithm 9 Computing β by local sampling Algorithm

```
1: function SELECTPARTITIONS(P^P: set of partitions from P, P^Q: set of partitions from Q)
2:     maxDensity ← 0
3:     bestPair ← ∅
4:     for all c ∈ P^P do
5:         for all d ∈ P^Q do
6:             pairDensity ← PDDAI(c, d)
7:             if pairDensity > maxDensity then
8:                 maxDensity ← pairDensity
9:                 bestPair ← (c, d)
10:            end if
11:        end for
12:    end for
13:    OUTPUT(bestPair)
14: end function

15: function CALCULATEβ(P: set of points, Q: set of points, ρ: global sampling ratio, k: number of pairs)
16:    localPρ ← CALCULATELOCALRATIO(|P|, ρ)
17:    SampledP ← SAMPLING(P, localPρ)
18:    localQρ ← CALCULATELOCALRATIO(|Q|, ρ)
19:    SampledQ ← SAMPLING(Q, localQρ)
20:    SORTX(SampledP)
21:    SORTX(SampledQ)
22:    KMaxHeap ← PSKCPQ(SampledP, SampledQ, k)
23:    if KMaxHeap is full then
24:        βDistanceAndPair ← POP(KMaxHeap)
25:        β ← βDistanceAndPair.Distance
26:        OUTPUT(β)
27:    else
28:        OUTPUT(∞)
29:    end if
30: end function

31: function PARTITIONSFILTER(P^P: set of partitions from P, P^Q: set of partitions from Q, β: upper
    bound distance)
32:    for all c ∈ P^P do
33:        for all d ∈ P^Q do
34:            minDistance ← MINDISTANCE(MBR(c), MBR(d))          ▷ Computing MinDistance
35:            if minDistance ≤ β then                            ▷ Rule 1
36:                OUTPUT(c, d)
37:            end if
38:        end for
39:    end for
40: end function
```

query processing time. On the other hand, ε-approximate and α-allowance are distance-based approximate techniques and can be used for adjustment of quality of the query result (kCPQ). For this reason, we will consider them as candidates for application in our problem. Since $\varepsilon \geq 0$ values are unlimited, according to the conclusions of [Corral and Vassilakopoulos, 2005, Gao et al., 2015], it is not easy to adjust the β value (upper bound of the distance value of k-th closest pair). For this reason, here we will choose the α-allowance technique, where α is a bounded positive real number ($0 \leq \alpha \leq 1$). With this approximate method, we can easily adjust the balance between the execution time of the kCPQ algorithm and the accuracy of the final query result. Notice that this α-allowance technique can be easily transformed to the ε-approximate technique with $\alpha = 1/(1+\varepsilon)$ [Gao et al., 2015].

According to [Corral and Vassilakopoulos, 2005], we can apply the α-allowance approximate technique in all plane-sweep-based kCPQ algorithms ($PSKCPQ$) [Roumelis

© 2021 García-García, F.J.

et al., 2014, Roumelis et al., 2016] and the three sliding variants (Strip, Window, and Semi-Circle) to adjust the final query result. It can be carried out by multiplying δ by $(1 - \alpha)$, giving rise to $\alpha PSKCPQ$ since it is a distance-based approximate technique, and δ is the distance value of the k-th closest pair found so far, during the processing of the query algorithm. Analyzing the α parameter in this approximate technique: when $\alpha = 0$, we will get the normal execution of the plane-sweep $PSKCPQ$ algorithm; when $\alpha = 1$, we will invalidate the δ value (it will always be 0) and no pair of points will be selected for the result; finally, when $0 < \alpha < 1$, we can adjust the strip sizes, the window, and the semi-circle over the sweeping axis since all of them depend on the δ value. Therefore, the smaller α value, the larger the upper bound of the δ value (i.e., more points will be considered and fewer points will be discarded); on the other hand, the larger α value, the smaller the upper bound of the δ value (i.e., fewer points will be considered and more points will be discarded).

The schema to compute β by using the α-allowance approximate technique with a plane-sweep-based kCPQ algorithm (αPSKCPQ) is very similar to the schema of computing β by local sampling illustrated in the right diagram of Figure 4.13. The essential difference is that sampling is not used in the selected pair of partitions, and all points from them are combined by the $\alpha PSKCPQ$ algorithm, obtaining a β value in a faster way if the α value is large enough.

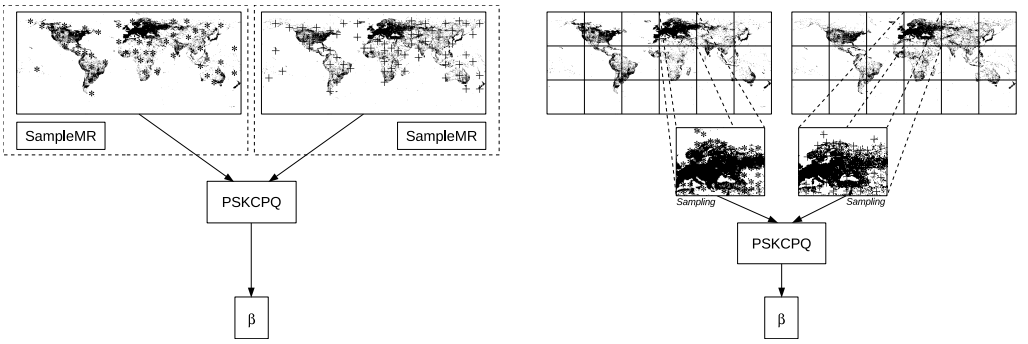


Figure 4.13: Schema for computing β. Global sampling (left) vs. local sampling (right), with Grid partitioning technique.

The adaptation of the previous Algorithm 9 to local approximate is straightforward. The $CALCULATE\beta$ function no longer accepts ρ as a parameter since we do not perform a sampling of the input datasets, but for each set, we get several elements that allow us to work within the main memory. Furthermore, we have a new α parameter, and the function $PSKCPQ$ is replaced by the new $\alpha PSKCPQ$ function that takes this new parameter for the adjustment of the approximate technique. The subsequent steps of the algorithm remain unmodified.

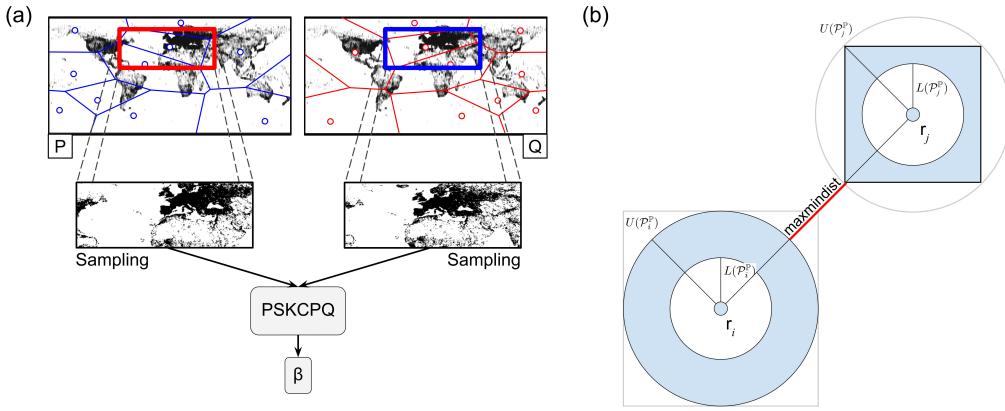


Figure 4.14: Computation of β, using Voronoi-Diagram based partitioning by sampling locally from both datasets (a), and partition refinement by its *MBR*, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ properties and maximum minimum distance calculation (b).

4.4.2.3 Computing β using Voronoi-Diagram based partitioning

Using Voronoi-Diagram based partitioning, as shown in Figure 4.14 (a), we can improve the kCPQ MapReduce algorithm by modifying its local sampling β computation and the *filter function*. For the computation of β, the most appropriate partitions, where an initial kCPQ is performed, are those whose pivots are closer to each other and have both the higher density of points and area of intersection (*PDDAI*). Figure 4.14 (b) shows that for each partition $\mathcal{P}_i^{\mathbb{P}}$ of this partitioning technique, we have both its $MBR(\mathcal{P}_i^{\mathbb{P}})$ and its $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ values, allowing us to detect areas of the former in which there are no points.

For the *PARTITIONSFILTER* two new distances metric can be used: the minimum distance between two pivots from two different partitions $minDist(r_i, r_j)$ and the maximum minimum distance between two partitions $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$. They are exposed in Definition 4.2 and Definition 4.3, respectively. Therefore, as shown in Figure 4.14 (b), this function prunes pairs of partitions which have $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$ larger than β, as we can see in the pruning Rule 2.

Definition 4.2. Minimum distance between two pivots, $minDist(r_i, r_j)$
Given two pivots, $r_i \in \mathcal{R}^{\mathbb{P}}$ and $r_j \in \mathcal{R}^{\mathbb{Q}}$ $i \neq j$ that generate two partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$, the minimum distance between two pivots, $minDist(r_i, r_j)$, is defined as
$minDist(r_i, r_j) = dist(r_i, r_j) - U(\mathcal{P}_i^{\mathbb{P}}) - U(\mathcal{P}_j^{\mathbb{Q}})$

Definition 4.3. Maximum minimum distance between two partitions, $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$
Given two partitions, $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$ $i \neq j$, the maximum minimum distance between two partitions, $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$, is defined as
$MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}}) = max\{minDistance(MBR(\mathcal{P}_i^{\mathbb{P}}), MBR(\mathcal{P}_j^{\mathbb{Q}})), minDist(r_i, r_j)\}$

© 2021 García-García, F.J.

Rule 2. Pair of Voronoi Partitions Pruning
Given two partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$ $i \neq j$, from $\mathbb{P}$ and $\mathbb{Q}$, respectively, and β is the upper bound of the distance value of the k-th closest pair of the two joined datasets. If $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}}) > \beta$, then the pair of partitions $(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$ can be pruned, because they do not have any pair of points with distance smaller than β.

Rule 2 allows us to prune combinations of partitions from $\mathbb{P}$ and $\mathbb{Q}$, reducing the number of *map* tasks that the kCPQ MapReduce algorithm needs to perform to get the final query result.

Algorithm 10 shows the complete adaptation of the local sampling β computation when using Voronoi-Diagram based partitioning. Note that the processing scheme remains unmodified, but you get higher precision in the calculation of PDDAI (line 6) and the *PARTITIONSFILTER* function with the use of *MmDist* (line 34).

Algorithm 10 Computing β using Voronoi-Diagram based partitioning

```
1: function SELECTPARTITIONS(P^P: set of partitions from P, P^Q: set of partitions from Q)
2:     maxDensity ← 0
3:     bestPair ← ∅
4:     for all c ∈ P^P do
5:         for all d ∈ P^Q do
6:             pairDensity ← PDDAI(c, d)
7:             if pairDensity > maxDensity then
8:                 maxDensity ← pairDensity
9:                 bestPair ← (c, d)
10:            end if
11:        end for
12:    end for
13:    OUTPUT(bestPair)
14: end function

15: function CALCULATEβ(P: set of points, Q: set of points, ρ: global sampling ratio, k: number of pairs)
16:    localPρ ← CALCULATELOCALRATIO(|P|, ρ)
17:    SampledP ← SAMPLING(P, localPρ)
18:    localQρ ← CALCULATELOCALRATIO(|Q|, ρ)
19:    SampledQ ← SAMPLING(Q, localQρ)
20:    SORTX(SampledP)
21:    SORTX(SampledQ)
22:    KMaxHeap ← PSKCPQ(SampledP, SampledQ, k)
23:    if KMaxHeap is full then
24:        βDistanceAndPair ← POP(KMaxHeap)
25:        β ← βDistanceAndPair.Distance
26:        OUTPUT(β)
27:    else
28:        OUTPUT(∞)
29:    end if
30: end function

31: function PARTITIONSFILTER(P^P: set of partitions from P, P^Q: set of partitions from Q, β: upper
    bound distance)
32:    for all c ∈ P^P do
33:        for all d ∈ P^Q do
34:            minDistance ← MMDIST(c, d)                    ▷ Computing MmDist
35:            if minDistance ≤ β then                       ▷ Rule 2
36:                OUTPUT(c, d)
37:            end if
38:        end for
39:    end for
40: end function
```

4.4.3 Improvements for kNNJQ in SpatialHadoop

In this section, we first present a *repartitioning technique* to improve the distributed kNNJQ algorithms to deal with skewed data. Next, we adapt the distance metrics, and pruning rules [Kuhlman et al., 2017] for kNNJQ MapReduce algorithm in SpatialHadoop. Finally, we incorporate into SpatialHadoop the *less data* technique [Moutafis et al., 2019b] to try to move as few data as possible between computing nodes.

4.4.3.1 Improvements for processing skewed data

When MapReduce tasks are performed, a problem that usually appears is the so-called *skewed data*. In general terms, this problem consists in that some partitions have a higher number of elements than the rest of them, and therefore it causes that there are tasks that take a long time to be executed and a delay in obtaining the final result can be derived (**F.3**). Furthermore, the partitioning techniques in SpatialHadoop and other systems are usually based on getting partitions close to the underlying filesystem block size (**F.5**) that has been established in the corresponding data cluster. However, DJQ MapReduce algorithms, like kNNJQ, can produce combinations of partitions with a very large number of elements that would delay the obtaining of results and increase the main memory used (**F.4**).

The proposed improvements aim, from a set of data already partitioned by SpatialHadoop (e.g., Grid, Quadtree), to repartition, if necessary, each local partition to solve the aforementioned problem. To this end, a kind of double index is created by having the original global index plus a sub-index for each of the partitions when a certain number of elements is exceeded, and they need to be repartitioned. For instance, we can have a dataset partitioned by Quadtree into 12 partitions, and then each partition is split into a Grid of 4×4 partitions. To create this index, we have to take into account the factors **F.1**, **F.2**, **F.3** and **F.4** since SpatialHadoop has also considered the factor **F.5** for the initial partitions and because we will not save the resulting partitions as new HDFS files. This *repartitioning technique* is used mainly for kNNJQ and εDRJQ in SpatialHadoop, although they can be applied to other distributed DJQ algorithms and DSDMSs. Figure 4.15 shows the new phase (*Repartitioning*) of the proposed kNNJQ MapReduce algorithm in SpatialHadoop. The *Repartitioning* phase uses an existing partitioning technique to subdivide the largest and/or densest partitions from dataset $\mathbb{Q}$ and saves the information for further use in subsequent phases. Note that, in [Moutafis et al., 2019b], repartitioning is not performed since the Quadtree-based partitioning in Hadoop is done completely under the control of the kNNJQ algorithm of [Moutafis et al., 2019b], and is not limited by the file-system block size (unlike the partitioning techniques provided by SpatialHadoop).

We have implemented two types of repartitioning techniques [García-García et al., 2020c], one based on a Grid structure and another based on a recursive decomposition of space by Quadtree.

The *Grid-based repartitioning* method divides the original partition into as many *rows* and *columns* as necessary so that each cell or partition has at most L elements. In our experiments, we have used $num\_rows = num\_columns = \sqrt{(\frac{num\_elements}{L})}$, where $num\_elements$ is the number of elements in the original partition, and L is a maximum

© 2021 García-García, F.J.

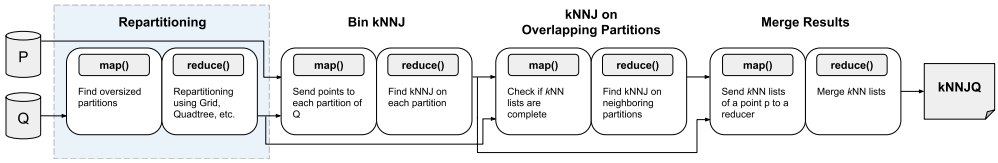


Figure 4.15: Repartitioning phase in the kNNJQ MapReduce algorithm in SpatialHadoop.

number of elements (e.g., in our experiments $L = 50000$). For kNNJQ, this repartitioning is done in the *Bin Join* phase, previously described in Algorithm 4 and illustrated in Figure 4.9, where, during the *map* phase, the elements of both sets are distributed based on a formula that determines the new partition they belong. This distribution is the great advantage of Grid partitioning since no previous preprocessing is needed to divide a partition into a certain number of rows and columns (i.e., sub-partitions). Then, in the *reduce* phase, a count of the elements of the largest set that belongs to each created sub-partition is performed. This way, the next phase can use the recently created index to obtain the partitions that overlap with the partial results. These sub-partitions are smaller than the original partition, and therefore candidates from calculations of kNNJQ will be pruned. However, even if a limit of elements has been established, it is impossible to know if any of the sub-partitions will overcome it since we do not know a priori how the elements are distributed.

The *Quadtree-based repartitioning* is a data-driven technique widely used in many spatial applications. Since this repartitioning method is based on how the data are distributed and, hence, a simple formula that splits the partition into $rows \times columns$ is not enough, as it is in the case of the *Grid*-based repartitioning, a new task must be performed. As shown in Algorithm 11, this is a MapReduce job that performs a repartitioning of each of the partitions on the initial index. To do this, in the *map* phase, it uses a maximum number of elements L and a data sampling process to have a representative set of how the elements are distributed (r is a sample ratio) and reduce

Algorithm 11 Quadtree-based repartitioning Algorithm

```
1: function MAP(p: point from ℚ, 𝒫^ℚ: set of partitions from ℚ, r: sample ratio)
2:     partitionId ← FindPartition(𝒫^ℚ, p)
3:     if Random ≤ r then
4:         output(partitionId, p)
5:     end if
6: end function

7: function REDUCE(partitionId: current partition, Q: set of points in partition, L: max number of
   elements, r: sample ratio)
8:     Initialize(Quadtree, L × r)
9:     for all q ∈ Q do
10:        InsertInto(Quadtree, q)
11:    end for
12:    output(partitionId, Quadtree)
13: end function
```

Algorithm 12 Range Query with repartitioning Algorithm

```
1: function RangeQueryWithRePartitioning(circle: circular region, P^Q: set of partitions from Q,
   Quadtrees: set of quadtrees for each P^Q)
2:     Initialize(SelectedParts)
3:     for all c ∈ P^Q do
4:         if Intersects(circle, c) then
5:             InsertInto(SelectedParts, c)
6:         end if
7:     end for
8:     Initialize(SelectedSubParts)
9:     for all c ∈ SelectedParts do
10:        Quadtree ← FindQuadtree(Quadtrees, c)
11:        SubParts ← Intersects(Quadtree, circle)
12:        InsertInto(SelectedSubParts, SubParts)
13:    end for
14:    return SelectedSubParts
15: end function
```

the creation time of the subsequent Quadtree. As we will see in the experimental section: $L = 100000$ and $r = 2\%$. Finally, in the *reduce* phase, it inserts the sampled elements in a Quadtree per partition that will form part of the new sub-index. Once the repartitioning is done, the algorithm behaves in the same way as for the repartitioning based on Grid. In Algorithm 12, the new range query method *RangeQueryWithRePartitioning* for selecting partitions overlapping with the circular region centered at the query point q and with a radius equal to the distance threshold δ, is shown. Initially, the global index is used to select those partitions that overlap with the region, and subsequently, the corresponding Quadtree is used to obtain the sub-partitions that overlap with it. Note that in this way, more candidates are pruned, and therefore the search in the spatial dataset is also reduced.

4.4.3.2 Using Voronoi-Diagram based partitioning for kNNJQ

We have designed and implemented several improvements for the kNNJQ in SpatialHadoop using Voronoi-Diagram based partitioning [García-García et al., 2020b]. As shown in Figure 4.16, this partitioning technique can be incorporated into the proposed kNNJQ MapReduce algorithm in two ways: (a) performing the *initial Partitioning* process of the datasets in the *Preprocessing* step (see Figure 4.1), and/or (b) subdividing the partitions from $\mathbb{Q}$ in the *Repartitioning phase* individually, and then, using its properties on the *kNNJ on Overlapping Partitions* phase. With the first one, we can take advantage of the characteristics of this technique globally, using the default parameters given by SpatialHadoop, in the same way that it is done for any built-in query. For the second way, we can accelerate the kNNJQ processing by decomposing the initial partitioning, by using the Voronoi-Diagram based partitioning technique, in smaller partitions given a maximum number of elements to solve skew data problems (*Repartitioning* phase) and reduce the number and size of the tasks of the *Bin kNNJ* and *kNNJ on Overlapping Partitions* phases. Furthermore, when calculating the overlapping partitions, the coordinates of each pivot r_i and the $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ values can be used to get better performance and accuracy than using only the *MBR* of each partition $\mathcal{P}_i^{\mathbb{P}}$, $MBR(\mathcal{P}_i^{\mathbb{P}})$. Figure 4.16 (b) shows that only the shaded part can contain points within

© 2021 García-García, F.J.

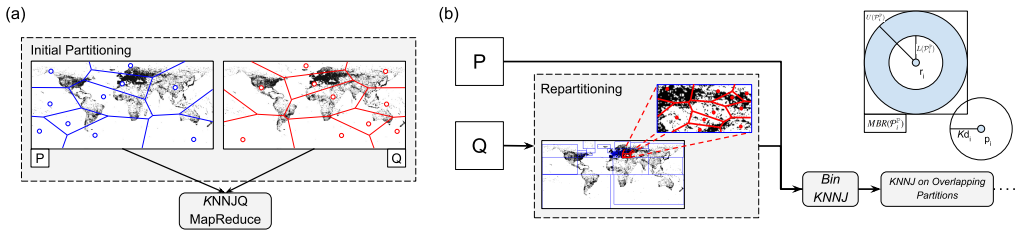


Figure 4.16: Voronoi-Diagram based partitioning on the *initial partitioning* of the datasets (a) and in the *repartitioning* and *kNNJ on Overlapping Partitions* phases (b).

the $MBR(\mathcal{P}_i^{\mathbb{P}})$, and therefore there is no overlap with the circle centered in p_i and the distance of the current k-th nearest neighbor as radio.

Furthermore, we can exploit the properties of Voronoi-Diagram based partitioning and adapt the distance metrics and pruning rules [Kuhlman et al., 2017] for kNNJQ MapReduce algorithm in SpatialHadoop.

The points inside each Voronoi-Cell V_i are denoted as $V_i.core = \{p : p \in V_i\}$. The *support* set of a Voronoi-Cell V_i, called $V_i.support$, contains at least all data points that satisfy the following two conditions:

1. $\forall q \in V_i.support$, $q \notin V_i.core$, and

2. there exists at least one point $p \in V_i.core$ such that $q \in kNN(V_i, p, k)$.

The $V_i.support$ must be sufficient to guarantee that the kNN of all core points, in each cell V_i, can be found among $V_i.core$ and $V_i.support$.

Since *support* points must be duplicated, they are considered multiple times in a kNNQ. Therefore, a large number of support points increases the computation costs per partition since many more points must be searched. To minimize the number of support points, in [Kuhlman et al., 2017], two distance metrics and two pruning rules are defined.

The *core-distance* of a given Voronoi-Cell V_i represents the maximum distance from a *core* point p of V_i to its k-th nearest core neighbor q. It defines an upper bound on the distance between any core point of V_i and the possible support points. That is, given a point q outside V_i, it is guaranteed not to be a support point of V_i if its distance to any core point of V_i is larger than the $corDist(V_i)$.

Definition 4.4. core-distance of V_i, $corDist(V_i)$, [Kuhlman et al., 2017]
$corDist(V_i) = max(dist(p, q))\ \forall p, q \in V_i.core$ *where* $q \in kNN(V_i, p, k) \in V_i.core$

The *support-distance* takes the pivot r_i of cell V_i into consideration, and it represents the maximum distance of a possible support point of V_i to the pivot r_i of V_i.

Definition 4.5. support-distance of V_i, $supDist(V_i)$, [Kuhlman et al., 2017]
$supDist(V_i) = max(dist(p, r_i) + dist(p, q))\ \forall p, q \in V_i.core$
where $q \in kNN(V_i, p, k) \in V_i.core$

Now, we remind the two pruning rules proposed in [Kuhlman et al., 2017] at Voronoi-Cell and point levels. The first pruning rule, Rule 3, which is applied in the *map* function of the *kNNJ on Overlapping Partitions* phase, avoids unnecessary data duplication (Algorithm 13, line 4) and also reduces the number of Voronoi-Cells each point must be checked against when mapping points to support sets (Algorithm 13, line 22).

Rule 3. *Support Cell Granularity Pruning*, [Kuhlman et al., 2017]
Given two Voronoi-Cells V_i, V_j and their corresponding pivots r_i, r_j, $i \neq j$. If the $supDist(V_i) \leq dist(r_i, r_j)/2$, then V_j does not contain any support points of V_i.

The second one, Rule 4, allows us to prune, in the *map* phase of the *KNNJ on Overlapping Partitions* phase (Algorithm 13, line 7), the points of the support cells that are not part of any partial kNN list. This allows us to reduce, even more, the shuffled data (fewer points are transferred to the *reduce* phase) and the complexity of the final kNN calculation for each point (the size of the set of support points is smaller).

Rule 4. *Support Point Granularity Pruning*, [Kuhlman et al., 2017]
Given any point $p \in V_i$, $q \in V_j$, $i \neq j$, and $HP(V_i, V_j)$ is the HyperPlane boundary between Voronoi-Cells V_i and V_j. If $dist(q, HP(V_i, V_j)) \geq corDist(V_i)$, then $q \notin kNN(V_j, p, k)$.

That is, Rule 4 allows us to prune points within the support cells that have not been already discarded by Rule 3. Furthermore, according to [Hjaltason and Samet, 2003], the following lower bound can be used in place of the exact value of $dist(q, HP(V_i, V_j))$ in pruning Rule 4.

Definition 4.6. *Lower bound of $dist(q, HP(V_i, V_j))$*, [Kuhlman et al., 2017]
Given two Voronoi-Cells V_i and V_j and their corresponding pivots r_i, r_j, $i \neq j$, and a point $q \in V_j$, $dist(q, HP(V_i, V_i)) \geq \frac{dist(q,r_i) - dist(q,r_j)}{2}$

Thanks to the Definition 4.6, we can use a lower bound whose calculation is less complex than $dist(q, HP(V_i, V_j))$ leading to the pruning Rule 5, which reduces the calculation time, preventing it from penalizing the application of this pruning rule.

Rule 5. *Support Point Granularity Pruning by a Lower bound*, [Kuhlman et al., 2017]
Given any point $p \in V_i$, $q \in V_j$, $i \neq j$. If $\frac{dist(q,r_i) - dist(q,r_j)}{2} \geq corDist(V_i)$, then $q \notin kNN(V_j, p, k)$.

4.4.3.3 Less Data Technique

The *less data* technique [Moutafis et al., 2019b] can be used in our kNNJQ MapReduce algorithm to reduce the size of the shuffled data and the size of the output data of the *kNNJ on Overlapping Partitions* phase. Moreover, applying this technique in our kNNJQ MapReduce algorithm, each computing node will calculate and return a kNN list for every query point in the *Bin kNNJ* phase, based on its local data. Then some additional phases are needed to exchange data among nodes and find possible misses of nearer neighboring points while trying to move as less data as possible between nodes.

© 2021 García-García, F.J.

Algorithm 13 Improved kNNJ on Overlapping Partitions Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ, k: number of neighbors)
2:     origin ← IsFromPorQ(p)
3:     if origin is from Q then
4:         filteredParts ← PrunePartitions(𝒫^ℚ, p, k)                    ▷ Rule 3
5:         partition ← FindPartition(filteredParts, p)
6:         if partition is not NULL then
7:             if PrunePoint(filteredParts, p) == false then             ▷ Rule 5
8:                 output(partition.id, p)
9:             end if
10:        end if
11:    else
12:        overlappedParts ← GetOverlappedPartitions(𝒫^ℚ, p, k)
13:        for all partition ∈ overlappedParts do
14:            output(partition.id,p)
15:        end for
16:    end if
17: end function

18: function GetOverlappedPartitions(𝒫^ℚ: set of partitions from ℚ, p: point from ℙ, k: number of neighbors)
19:    kNNList ← GetKnnList(p)
20:    nnNumber ← kNNList.size
21:    radius ← GetKthDistance(kNNList)
22:    supParts ← GetSupportPartitions(𝒫^ℚ, p, radius)                  ▷ Rule 3
23:    nnNumber ← GetNumberOfNeighbors(supParts, p, radius)
24:    while nnNumber < k do
25:        supParts ← GetSupportPartitions(supParts, p, radius)
26:        nnNumber ← GetNumberOfNeighbors(supParts, p, radius)
27:    end while
28:    return supParts
29: end function
```

In the original algorithm, every point, which is still not finished, is moved to its *reducer* of the *kNNJ on Overlapping Partitions* phase with its kNN list. Therefore, it adds a significant load to the network, especially for large k values. We decided to replace the kNN list with the distance to the k-th neighbor as a bound, which is the only info needed in the *reducer*. The partial kNN lists will be finally merged on the last *Merge Results* phase.

Continuing with the idea of reducing the size of the data that is handled in the different phases of the algorithm, the pruning Rule 6 allows us to determine which of the kNN lists have turned out to be final.

Rule 6. *Final kNN List Pruning*
Given any point $p \in V_i$, $q \in kNN(V_i, p, k)$. *If* $\frac{dist(r_i,r_j)}{2} \geq dist(r_i,p) + dist(p,q)\ \forall$ $V_j \cap V_i.support \neq \varnothing$, *then* $kNN(V_i, p, k)$ *is final.*

Therefore, with this new pruning rule (Rule 6), we can split the output of the *reducers* of the *Bin kNNJ* phase into different group of files (final kNN lists and non-final kNN lists) thus reducing the input data size of the *kNNJ on Overlapping Partitions* and *Merge Results* phases. As a consequence of this reduction in the input data, the size of the shuffled data between the *map* and *reduce* tasks of these phases is also considerably smaller.

Finally, Figure 4.17 shows the differences in the flow of data of the kNNJQ MapReduce algorithm. Note that all final kNN lists are written directly to the output just after

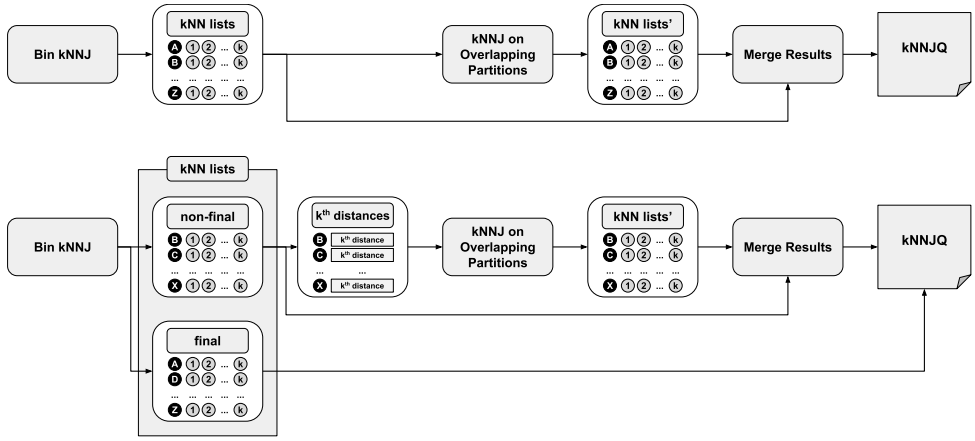


Figure 4.17: kNNJQ MapReduce algorithm (top) vs. the use of the *less data* technique (bottom).

the *Bin kNNJ* phase. Besides, the input of the *kNNJ on Overlapping Partitions* phase is reduced in size to a list of pairs (i.e., point and distance of the k-th nearest neighbor found so far) of the non-final kNN lists.

4.5 Performance Evaluation

This section presents the results of an extensive experimental study aiming at measuring and evaluating the efficiency of the DBQs algorithms in SpatialHadoop and their extensions and improvements proposed in Sections 4.3 and 4.4. In particular, Subsection 4.5.1 describes the experimental settings for this performance study in SpatialHadoop. Following subsections study each DBQ MapReduce algorithm (i.e., εDRQ, kNNQ, RkNNQ, εDRJQ, kCPQ, kNNJQ and εDJQ) to test their efficiency, scalability and the effects of several extensions and improvements.

4.5.1 Experimental Setup

For the experimental evaluation, we have used real and synthetic (clustered) datasets of 2d points to test our *DBQ* MapReduce algorithms in SpatialHadoop. For real-world datasets we have used datasets from OpenStreetMap[1] already described in Section 3.6: *LAKES* (L), *PARKS* (P), *ROADS* (R), *BUILDINGS* (B) and *ROAD_NETWORKS* (RN). Remember that to generate datasets of points from non-point spatial objects, we have considered the *center* of each MBR and the *centroid* of each polygon. For spatial objects experiments, we have used the datasets unmodified (i.e., polygons and line-strings).

[1] `http://spatialhadoop.cs.umn.edu/datasets.html`

© 2021 García-García, F.J.

SpatialHadoop requires the datasets to be partitioned and indexed before invoking any spatial operations. For instance, the running times needed for the *Preprocessing* phase using a *Quadtree* partitioning technique are 94 sec for *LAKES*, 103 sec for *PARKS*, 150 sec for *ROADS*, 175 sec for *BUILDINGS* and 1053 sec for *ROAD_NETWORKS*. Spatial data are indexed and stored on HDFS and for the subsequent execution of spatial queries, they are already available.

For synthetic datasets, we have created *clustered data* since data in the real world are often clustered or correlated. In particular, real spatial data may follow a distribution similar to the clustered one. We have generated several files of different sizes using our own generator of clustered distributions, implemented in SpatialHadoop and with a similar format to the real data. The dataset sizes are 25M (5.4 GB), 50M (10.8 GB), 75M (16.2 GB), 100M (21.6 GB) and 125M points (27 GB), with 2500 clusters in each dataset (uniformly distributed in the range [(-179.7582155, -89.96783429999999) - (179.84404100000003, 82.51129005000003)]), which is the MBR of *BUILDINGS*), where, for a set having N points, *N/2500* points are gathered around the center of each cluster, according to Gaussian (normal) distribution with mean 0.0 and standard deviation 0.2, as in [Eldawy et al., 2013]. For example, for an artificial dataset of 100M of points, we have 2500 clusters uniformly distributed, and for each cluster, we have generated 40000 points according to Gaussian distribution (mean = 0.0, standard deviation = 0.2). In Figure 4.18, we can observe a small area of a clustered dataset. We made 5 combinations of synthetic datasets by combining two separate instances of datasets, for each of the above 5 cardinalities (i.e., $25MC1 \times 25MC2$, $50MC1 \times 50MC2$, $75MC1 \times 75MC2$, $100MC1 \times 100MC2$ and $125MC1 \times 125MC2$).

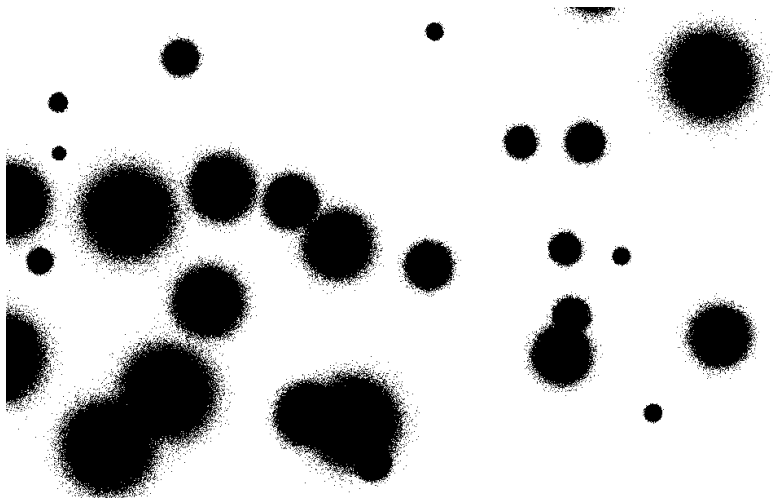


Figure 4.18: Synthetic dataset. Small area from a clustered dataset.

The main measure of performance in our experiments is the total *execution time* (i.e., running time or response time) in seconds (sec), and represents the time spent

for the execution of each distributed DBQ algorithm in SpatialHadoop. Moreover, the *shuffled data*, that describes the amount of information produced in the *mapper* tasks and moved to the nodes where the *reducer* tasks will run, shown in Gigabytes (GB), is also used as a performance metric in our experiments, to acquire more information on the behavior of the different phases of kNNJQ in SpatialHadoop.

Moreover, all experiments were conducted on the same OpenStack cluster of 12 nodes, described in Section 3.6. We used the latest code available in the repositories of SpatialHadoop[2], with the addition of our open-source DJQ MapReduce algorithms[3].

4.5.2 εDRQ experiments

In this section, we present the most representative results of our experimental evaluation of the εDRQ MapReduce algorithm [García-García et al., 2016a]. We have used both synthetic (Uniform) and real 2d point datasets to test our εDRQ algorithm in SpatialHadoop. On the one hand, for synthetic datasets, we have generated several files of distinct sizes (64MB, 128MB, 256MB, 512MB, and 1024MB) using SpatialHadoop built-in uniform generator [Eldawy and Mokbel, 2015]. On the other hand, for real datasets, we have sampled the *BUILDINGS* dataset described in Section 4.5.1, with several % values (25%, 50%, 75% and 100%). For these experiments, the *query point* is located at the center of the Minimum Bounding Rectangles (*MBRs*), to which the datasets are partitioned by SpatialHadoop. For example, the MBR of the synthetic datasets is [(0,0)-(1000000,1000000)], and the query point is at (500000,500000). Finally, Table 4.1 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	Values (default)
Uniform dataset size (MB)	64, 128, 256, 512, 1024
BUILDINGS, % of samples	25, 50, 75, (100)
Distance threshold, ε	20, (40), 75, 100, 200
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Grid, (STR)

Table 4.1: Configuration parameters used in our εDRQ experiments.

4.5.2.1 The effect of the increment of the dataset size

Our first experiment is to examine the effect of the dataset size using a fixed $\varepsilon = 40$. As we expected for uniform datasets (Figure 4.19, left chart), the execution times are almost linear because the number of partitions that pass the filtering phase is less than the number of *map* tasks. However, for the experiments with real datasets (Figure 4.19, right chart), the execution time varies due to the partitioning performed on the

[2] `https://github.com/aseldawy/spatialhadoop2`
[3] `https://github.com/acgtic211/spatialhadoop2/tree/DJQ`

© 2021 García-García, F.J.

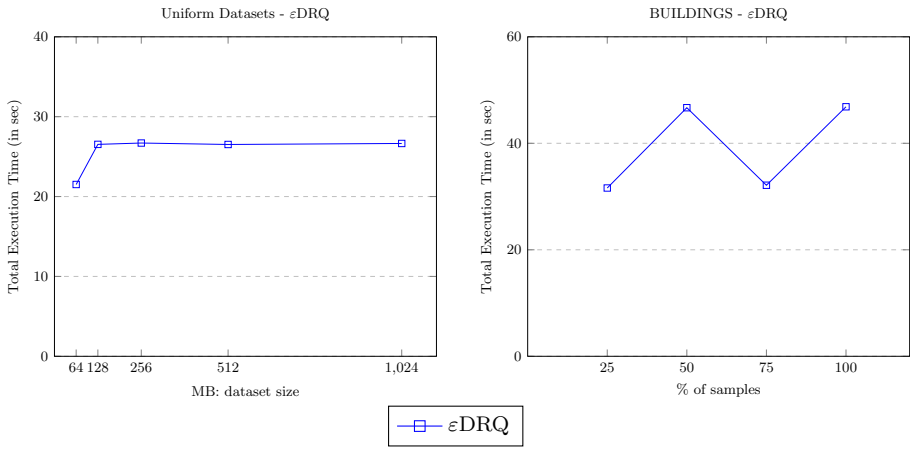


Figure 4.19: εDRQ cost, total execution time vs. dataset size for uniform (left) and % of samples of *BUILDINGS* (right).

data since, in a Grid-based partitioning, the number of cells depends on the size of the dataset and the number of partitions depends on the applied partitioning technique. For example, the number of points to consider for the *BUILDINGS* dataset sampled at 50% and 100% ratio is the same, being more than at 25% and 75%. And as we expected, the number of items returned by the query increases by the same percentage as data grows.

4.5.2.2 The effect of the increment of ε values

The second experiment studies the effect of two spatial partitioning techniques (Grid and STR) included in SpatialHadoop and ε value. As shown in Figure 4.20, left graph, the choice of a partitioning technique does not greatly affect the execution time, showing similar behavior. Using *Grid* partitioned files, the execution time increases near linearly until, almost, every point is selected, and then it grows more slowly. As *Grid* partitioning is based on a uniform structure, the increment of ε values increases the number of selected partitions evenly. However, since *STR* partitioned files are non-uniform, the result is a stepped graph. For example, when ε value is 75, more partitions are selected, and the execution time increases suddenly.

4.5.2.3 Speedup of the algorithm

The third experiment aims to measure the speedup of the εDRQ MapReduce algorithms, varying the number of computing nodes (η). Figure 4.20, right chart, shows the impact of ranging η from 1 to 12 on the performance of εDRQ MapReduce algorithm, for $BUILDINGS$ with a fixed value of $\varepsilon = 40$. From this figure, it could be concluded that the performance of our approach has a direct relationship with the number of computing nodes. It could be deduced that better performance would be obtained if more computing nodes are added. But, when the number of computing nodes exceeds the number of *map* tasks, no improvement for that individual job is obtained.

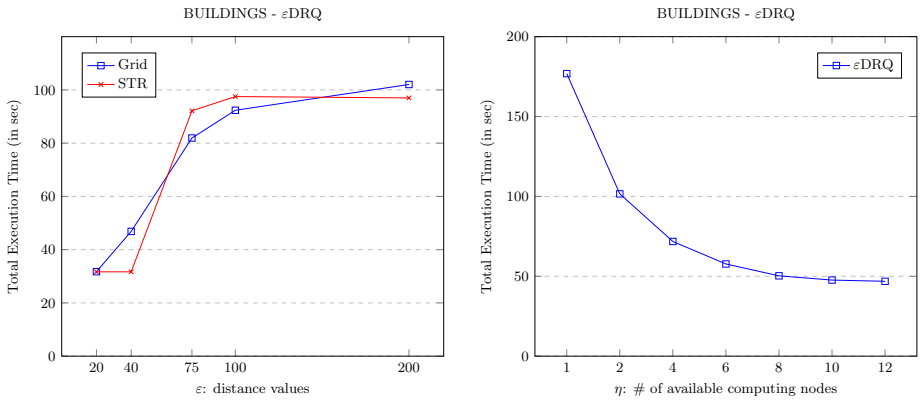


Figure 4.20: εDRQ cost, total execution time vs. ε value (left) and number of computing nodes η (right).

4.5.2.4 Conclusions from the experimental results

By analyzing the previous experimental results, we can extract several conclusions that are shown below:

1. We have experimentally demonstrated the *efficiency* (in terms of total execution time) and the *scalability* (in terms of ε values, sizes of datasets, and the number of computing nodes (η)) of the proposed MapReduce algorithm for εDRQ.

2. For uniform distributed datasets, the execution time variation is small, while for real datasets, the execution times depend on the number of selected partitions in the *filtering* phase.

3. The larger the ε values, the higher the number of spatial objects to be checked, and so the execution time grows. On the one hand, *Grid* partitioning shows a near-linear increase due to the regular division of the space. On the other hand, *STR* partitioning has non-regular partitions, obtaining a stepped increment.

4. εDRQ shows better performance when the number of computing nodes (η) grows, but if there are not enough tasks available for a specific number of nodes, no performance improvements are obtained.

4.5.3 kCPQ experiments

In this section, we present the most representative results of our experimental evaluation of the kCPQ MapReduce algorithm [García-García et al., 2016b, García-García et al., 2018b, García-García et al., 2020c, García-García et al., 2020b]. We have used synthetic (clustered) and real 2d point datasets to test our kCPQ algorithm in SpatialHadoop. For synthetic clustered datasets, we have generated several files of different sizes (25M, 50M, 75M, 100M, and 125M points) using the process detailed in Section 4.5.1. For real datasets, we have used the following: *LAKES*, *PARKS*, *ROADS*, *BUILDINGS* and *ROADS_NETWORKS*. Moreover, to perform a kCPQ experiment with two big spatial

© 2021 García-García, F.J.

datasets (one of them is *BUILDINGS* with 115M points), we have created a new big quasi-real dataset from *LAKES* (8.4M), with a similar quantity of points. The creation process is as follows: taking one point of *LAKES*, p, we generate 15 new points gathered around p (i.e., the center of the cluster), according to the Gaussian distribution described above, resulting in a new quasi-real dataset, *CLUS_LAKES*, with around 126M of points (27.5 GB). This dataset has the same shape as *LAKES*, but with more dense areas all over the world. Finally, Table 4.2 summarizes the configuration parameters used in our experiments (sampling ratio values express % of the whole datasets). Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	**Values (default)**
# of closest pairs, k	1, 10, (10^2), 10^3, 10^4, 10^5
α-allowance, α	0.0, 0.25, 0.50, (0.75), 0.85, 0.95
Sampling ratio, ρ	0.005, 0.01, 0.05, (0.1), 0.5, 1, 5, 10
% Dataset, δ	25, 50, 75, (100)
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Grid, (STR), Quadtree, Hilbert
PS algorithms	Classic, (Reverse Run)
PS improvements	Strip, Window, (Semi-Circle)
Sampling (Voronoi)	k-means++
Pivot selection (Voronoi)	Random, (k-means), OPTICS
% $\mathbb{P}$ area, γ	25, 50, 75, (100)

Table 4.2: Configuration parameters used in our kCPQ experiments.

4.5.3.1 The effect of applying β computation

Our first experiment is to examine the use of β distance value (the upper bound of the distance value of the k-th closest pair) for kCPQ MapReduce algorithms in SpatialHadoop (computed by global sampling (Algorithm 8), by local sampling (Algorithm 9) or by using the α-allowance approximate technique).

As shown in Figure 4.21, left chart, for the real datasets $LAKES \times PARKS$ combination using the Grid partitioning technique with various sampling ratios (ρ) and a fixed $k = 10^2$ ($k = 100$), the execution time is almost constant for the three methods. This trend in the results is mainly due to the fact that there is a trade-off between the time of sampling and β calculation with the one from the individual MapReduce tasks. With a larger sampling ratio ρ, a better β is obtained, which in turn improves the final *PSKCPQ* execution time. However, increasing the value of ρ also increases the time to calculate β. The use of β values accelerates the answer of the kCPQ, and using the method of *local sampling* reduces the response time by around 22 times; whereas, for the *global sampling*, the reduction is around 4 times faster than without β computation. This means that the use of local sampling shortens significantly the execution time because selecting a suitable pair of partitions for each dataset and using sampling over them reduce the computed β values and increase the power of pruning when passed to

the *mappers*. For instance, for a sampling ratio (ρ) equal to 0.1%, the β values obtained by global sampling is 0.0144191, whereas by local sampling it is 0.0054841.

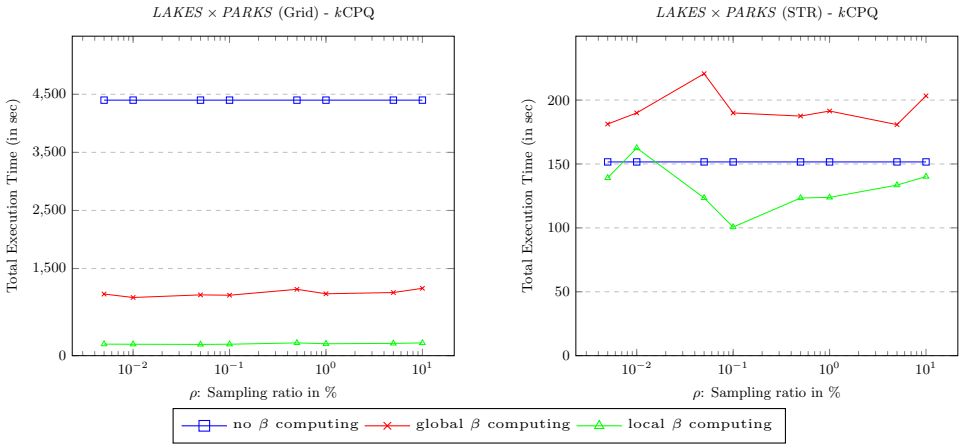


Figure 4.21: kCPQ cost without and with β computation ($LAKES \times PARKS$), varying the sampling ratio ρ.

In the right chart of Figure 4.21, we see a different behavior if we apply the STR partitioning technique for the same datasets. We observe that the use of global sampling for the computation of β is more expensive than without β values in the *preprocessing* phase; this is due to the fact that with the dataset sizes and the used partitioning technique (STR), the time spent to perform the MapReduce sampling jobs (SampleMR) produces an overhead much higher than the obtained improvement in response time. On the other hand, the use of local sampling to get the kCPQ is faster than the other two alternatives because the time required to perform the local sampling is very small, and the use of β improves the time of the individual *map* tasks. Moreover, a similar trend is observed between global and local sampling, which confirms that the improvement comes actually from reducing as much as possible the time required to obtain β. Finally, when comparing both charts in Figure 4.21, STR outperforms Grid since STR is a partitioning technique based on the data distribution; therefore, partitions with more uniform numbers of elements are produced, improving distance-based pruning of pairs of partitions and load balancing between nodes. However, Grid partitioning uses uniform division of the space without taking into account the global data density; therefore, it produces some partitions with much more elements than others, and certain *map* tasks delay the total response time of the query. Note that we have chosen for this first experiment the Grid and STR partitioning techniques because they are used in [Eldawy and Mokbel, 2015] for performance comparison of the spatial queries and, Grid is the simplest (uniform grid of $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid cells, where n is the desired number of partitions) and STR corresponds to R-trees which are widely used.

Figure 4.22 illustrates the same type of experiment (reporting the total execution time), but now for the $BUILDINGS \times PARKS$ combination. In the left chart, we

© 2021 García-García, F.J.

can see the same trend for Grid partitioning as in Figure 4.21, where the *preprocessing* phase for computing β with local sampling is 2.7 times faster than using global sampling (whereas without the *preprocessing* phase needed around 21900 seconds and it is not depicted in the figure). In the right chart, STR is faster than Grid (e.g., for $\rho = 0.1\%$ and global sampling, STR is 2.7 times faster than Grid), and local sampling is also 80 seconds faster than global sampling for computing β for the same reasons explained previously. Notice that, without the computation of β, around 2900 seconds to carry out the kCPQ are needed (not depicted in the figure). Again, comparing both charts, STR outperforms Grid according to the same reasons exposed above.

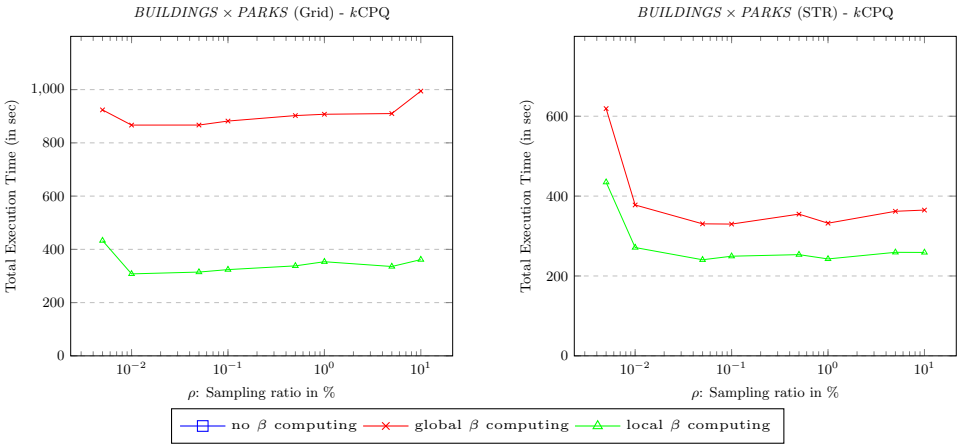


Figure 4.22: kCPQ cost without and with β computation ($BUILDINGS \times PARKS$), varying the sampling ratio ρ.

From these experiments, we can conclude that the use of *local sampling* for computing β (Algorithm 9) generates smaller β values (e.g., $BUILDINGS \times PARKS$ (STR) and $\rho = 0.1\%$, the β value obtained by global sampling is 0.00211, whereas for local sampling β is 0.00050) and then, this is more effective than global sampling when β is used in the *mappers*. Moreover, the partitioning technique is a determinant factor for this kind of distance-based join; in particular, STR outperforms Grid in all cases. Finally, the value of ρ (sampling ratio) is an important parameter to be considered, and we have to find a trade-off between the time of sampling and the value of β computation (the smaller β value, the larger the time of sampling). Therefore, we have chosen $\rho = 0.1\%$ as the value for the remaining experiments due to its excellent results.

Interesting results are also shown in Table 4.3, where all possible pairs of partitions are shown, considering various percentages (δ) of the datasets ($BUILDINGS \times CLUS\_LAKES$ (STR)) and, with (GS $\equiv$ using global sampling and LS $\equiv$ using local sampling) or without using the computation of β for $k = 100$ (for other k values the percentage of reduction is similar). We can extract three interesting conclusions from this table: (1) with the use of β, we significantly reduce the number of possible pairs of partitions to be joined (e.g., using the complete datasets, only 120 out of 1260 possible

pairs of partitions are considered); (2) the β value returned by global or local sampling is not so determinant for the reduction of the number of pairs of partitions to be combined (i.e., a smaller β value does not imply reducing the number of considered pairs of partitions) as we can see in the two right columns; and (3) the percentage of datasets to be joined is related to the number of considered pairs of partitions when a β value is applied for the STR partitioning technique (e.g., the 75%, 50%, and 25% of 120 are very close to 85, 55 and 32).

δ (%)	**Without** β	β **GS**	β **LS**
25	120	32	**32**
50	315	55	**55**
75	672	85	**84**
100	1260	120	**120**

Table 4.3: kCPQ cost, number of considered pairs of partitions without or with (global sampling (GS) or local sampling (LS)) β computation.

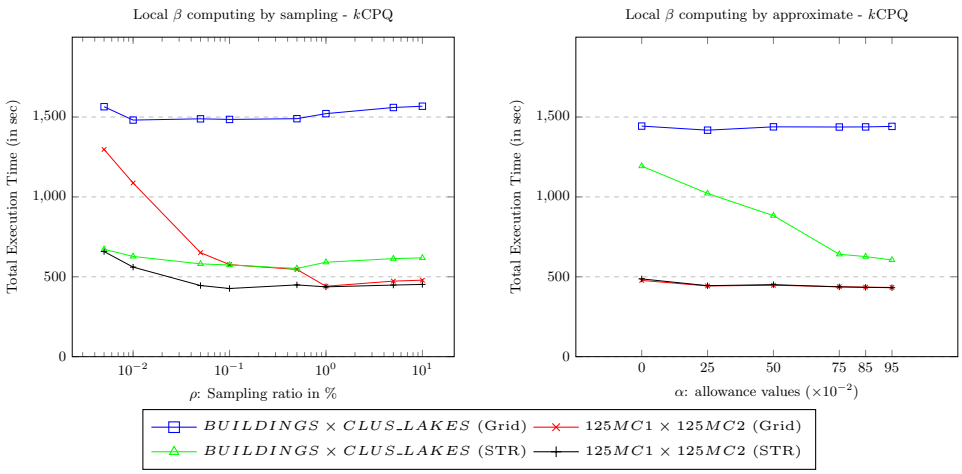


Figure 4.23: kCPQ cost using local sampling (left) and α-allowance approximate (right) technique for β computation.

In Figure 4.23, we study the behavior of the kCPQ MapReduce algorithm in SpatialHadoop, concerning the total execution time when β is computed locally from a suitable pair of partitions by local sampling or by using the α-allowance approximate technique for the combination of the largest datasets (real and synthetic) and by using two partitioning techniques (Grid and STR). In the left chart, we can see the trends for several sampling ratios (ρ). Again the STR partitioning reduces the response time significantly for real datasets (2.6 times faster when $\rho = 0.1\%$) with respect to Grid, but for the combination of synthetic data, the reduction is smaller (1.3 times faster when

© 2021 García-García, F.J.

$\rho = 0.1\%$); even for $\rho = 1.0\%$, $\rho = 5.0\%$ and $\rho = 10.0\%$, the execution times are almost the same. Moreover, notice that, when ρ is larger than 0.5%, the execution time with local sampling is increased slightly since the time needed to compute β increases with the increment of the sampling ratio. In the right chart, we can see the effect of applying the $\alpha PSKCPQ$ algorithm to the two selected partitions for computing β, by using various α values (0.0, 0.25, 0.50, 0.75, 0.85 and 0.95) to report the results of kCPQ. The response time is stable for all α values when the partitioning technique is Grid (real and synthetic) and STR (synthetic), but for $BUILDINGS \times CLUS\_LAKES$ (STR), the reduction from $\alpha = 0.95$ to $\alpha = 0.0$ is around 580 sec. Taking into account this result, we can deduce that the use of this approximate technique is useful for computing β, using high values of α. Moreover, for this case, the difference between $\alpha = 0.75$, $\alpha = 0.85$ and $\alpha = 0.95$ is very small. This behavior could be due to the fact that at the beginning of the $\alpha PSKCPQ$ processing, this algorithm gets a small β value quickly, and then it is executed very fast. Finally, if we compare both charts of Figure 4.23, we can conclude that both techniques are very suitable to compute β and get the result of kCPQ in SpatialHadoop very fast, in particular when $\rho \in [0.1\%, 1.0\%]$ and $\alpha \in [0.75, 0.95]$.

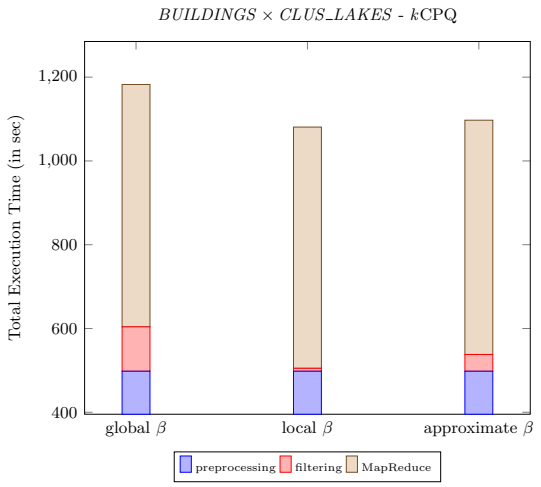


Figure 4.24: kCPQ cost, total execution time of different phases in the execution of kCPQ MapReduce algorithm.

Figure 4.24 shows the time spent in each phase that processing of the kCPQ in SpatialHadoop is split, when the three approaches to compute β are applied in the *filtering* step according to Figure 4.1. The configuration for this experiment is $BUILDINGS \times CLUS\_LAKES$, STR, $\rho = 0.1$, $k = 100$. The three phases are: *preprocessing*, *filtering* (*Global kCPQ*) and *MapReduce* (*Local kCPQ* and *Top kCPQ*). The time spent in the *preprocessing* phase (STR) is the same for the three bars (498 sec), whereas the times spent for the *filtering* phase are different, depending on the technique (global sampling, local sampling or approximate) applied for computing β. By using the local sampling, we get the smallest time spent (7 sec), next the approximate (40 sec) and the largest

execution time is for global sampling (106 sec). When the *filtering* phase is ended, a β value is passed to the next phase; the smaller the β value, the faster the next phase (*MapReduce*). With this in mind, the execution time spent in the last phase for the three techniques are: *global* $\beta = 578.498$ sec ($\beta = 0.00157$), *local* $\beta = 575.854$ sec ($\beta = 0.00062$) and *approximate* $\beta = 559.254$ sec ($\beta = 0.00013$).

The main conclusions that we can extract for this experiment are: (1) the use of small β values accelerates the answer of the kCPQ; (2) local sampling needs less time, generates smaller β values and is more effective than global sampling; (3) both local sampling and the α-allowance approximate technique are very suitable to compute β; and (4) STR outperforms Grid since STR is a partitioning technique based on the data distribution. For these reasons, we have chosen the *local sampling* as the default β calculation technique for the rest of the experiments.

4.5.3.2 Comparison of different plane-sweep algorithms and the use of local indices

This experiment aims to find the comparison of two plane-sweep-based kCPQ algorithms (*Classic* and *Reverse Run*) and an improvement (*Sliding Strip*, *Window*, or *Semi-Circle*) that has the best performance. As we can see in Table 4.4, the total execution times obtained do not show significant improvements between the plane-sweep algorithms and variants. This is due to various factors, such as reading disk speed, network delays, consumed time for each task, etc. As shown in this table, the difference between them is not quite significant (mainly for $LAKES \times PARKS$ ($L \times P$)), being the *Semi-Circle Reverse Run* algorithm the fastest in all cases, and the *Classic Strip* the slowest variant (with the highest execution time). This is because the *Reverse Run* algorithm has been specifically designed to reduce the number of distance computations [Roumelis et al., 2014, Roumelis et al., 2016]. For this reason, we have chosen the *Semi-Circle Reverse Run* as the plane-sweep kCPQ algorithm for all our experiments.

kCPQ Algorithm	$L \times P$	$B \times P$
Classic Strip	126.871	293.852
Classic Window	124.661	283.441
Classic Semi-Circle	121.263	267.171
Reverse Strip	123.013	276.398
Reverse Window	121.768	230.390
Reverse Semi-Circle	120.648	229.226
Local indices (R-tree)	147.023	318.450

Table 4.4: kCPQ cost, total execution time (in seconds) spent by each kCPQ algorithm, plane-sweep without indices and with local indices (R-tree).

Finally, since our kCPQ algorithm in SpatialHadoop supports local indices (R-trees), we have compared it with the plane-sweep algorithms (without indices). The execution time of local indices (R-tree), shown in the last row of Table 4.4, is higher than the running time of all plane-sweep-based algorithms. The reason why the use of local indices

© 2021 García-García, F.J.

slows down the algorithm is the fragmentation of the data produced by the R-tree's own structure. When no local indices are used, all elements present in the corresponding partitions are loaded into main memory, and then the appropriate plane-sweep-based kCPQ algorithm is performed. However, when using R-tree structures, the spatial data objects are stored in the leaves, and their number is determined by the degree of the tree. This degree for the node size and the configuration used for the experiments is 26 (suggested by [Eldawy and Mokbel, 2015]). When, finally, it is necessary to compare leaf nodes, multiple *PSKCPQ* algorithms with small quantities of spatial objects are performed. The sum of the execution times of these tasks becomes greater than working with all data in the partitions directly in main memory. We can see this behavior with the $BUILDINGS \times PARKS$ ($B \times P$) combination, where *Reverse Run Semi-Circle* is around 30% faster than using the local indices (R-trees).

The main conclusions that we can extract from this experiment are: (1) the *Semi-Circle Reverse Run* algorithm is the fastest *PSKCPQ* algorithm, and (2) the use of local indices (R-trees) does not increase the performance of the algorithm because of the time spent by multiple small *PSKCPQ* executions.

4.5.3.3 The effect of using different spatial partitioning techniques

In this experiment, we evaluate the effect of choosing a partitioning technique with the proposed kCPQ MapReduce algorithm. In [Eldawy et al., 2015], the most important conclusions for distributed join processing, using the *overlap* spatial predicate, are the following: (1) the smallest running time is obtained when the same partitioning technique is used in both datasets (except for Z-curve, which reports the worst running times), and (2) the Quadtree outperforms all other techniques for the running time since it minimizes the number of overlapping partitions between the two data files by employing regular space partitioning. Therefore, we experiment with the kCPQ MapReduce

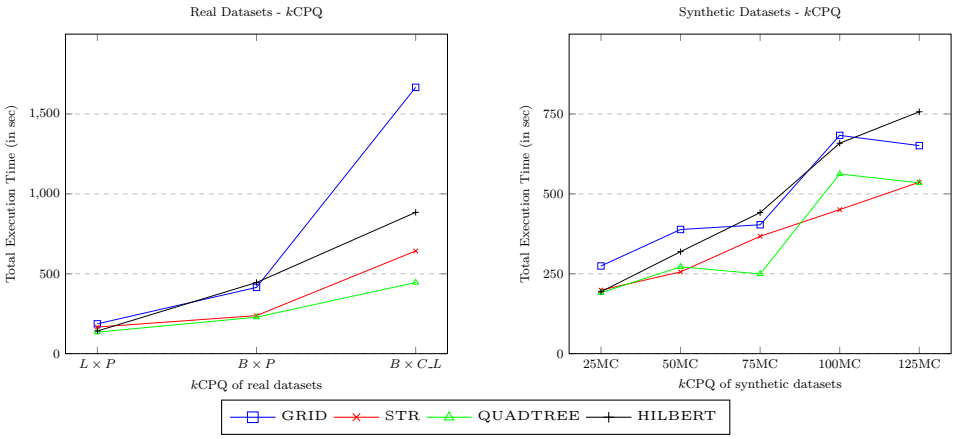


Figure 4.25: kCPQ cost, total execution time of different partition techniques, combining real (left) and synthetic datasets (right).

algorithm with both datasets partitioned with the same technique from the following: Grid, STR, Quadtree, and Hilbert-Curve.

As shown in the left chart of Figure 4.25, for the kCPQ performance of real datasets ($LAKES \times PARKS$, $BUILDINGS \times PARKS$ and $BUILDINGS \times CLUS\_LAKES$), the choice of a partitioning technique affects the execution time. For instance, Quadtree is the fastest (445 sec), the STR is the second (642 sec), the third is Hilbert (884 sec), and the slowest is the Grid (1667 sec), for the combination of the biggest real datasets: $BUILDINGS \times CLUS\_LAKES$ ($B \times C\_L$). Moreover, the influence of the partitioning technique is less for the combination of the smallest datasets, $LAKES \times PARKS$ ($L \times P$), where the execution times are almost the same (e.g., Quadtree is only 32 sec faster than STR). The behavior for synthetic datasets is different (see the right chart of Figure 4.25) due to the nature of the data distribution (uniform distribution of the centers of the clusters) and the type of partitioning technique (replication-based and distribution-based). The trends of replication-based techniques (Quadtree and Grid) are very similar, as in the case of distribution-based (STR and Hilbert). Moreover, for the combination of the biggest synthetic datasets, $125MC1 \times 125MC2$ (125M), the fastest partitioning technique is Quadtree (534 sec), and STR has a very close running time (only 2 sec slower), Grid takes 651 sec, and Hilbert is the slowest with 757 sec. Note that, a label like *25MC* on X-axis of the chart for synthetic datasets indicates the combination $25MC1 \times 25MC2$.

4.5.3.4 The effect of the increment of k values

This experiment studies the effect of increasing the k value for the combination of the biggest datasets (real and synthetic). The left chart of Figure 4.26 shows that the total execution time for real datasets ($BUILDINGS \times CLUS\_LAKES$) grows slowly, as the number of results to be obtained (k) increases, until $k = 10^4$; but for $k = 10^5$, the increment is larger, mainly for STR (around 850 sec). The *Quadtree* reports the

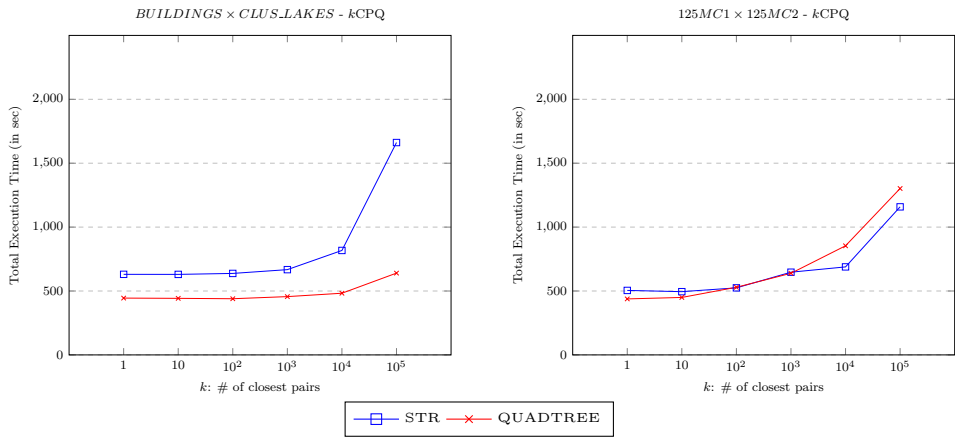


Figure 4.26: kCPQ cost, total execution time vs. k values.

© 2021 García-García, F.J.

best execution times, even for large k values (e.g., $k = 10^5$). These results mean that the Quadtree is less affected by the increment of k because Quadtree employs regular space partitioning, depending on the concentration of the points. For the combination of synthetic datasets ($125MC1 \times 125MC2$), in the right chart, for small k values, the Quadtree is slightly faster than STR, but for larger k values, the roles are swapped, and STR is faster than Quadtree.

The main conclusions that we can extract for this experiment are: (1) the Quadtree again satisfies kCPQ in the fastest way, mainly for real datasets, and (2) the higher the k values, the greater the possibility that pairs of partitions are not pruned, more *map* tasks could be required, and more total execution time is needed.

4.5.3.5 The effect of extending the algorithm for non-points spatial objects

In the following experiments, we analyze the behavior of the kCPQ algorithm in SpatialHadoop when the extension for processing non-points spatial objects is applied (see Section 4.4.1). Therefore, we will vary different parameters, such as dataset size, type of spatial object, partitioning technique, and the k value.

In Figure 4.27, the chart on the left shows the $kCP(\mathbb{P}, \mathbb{Q}, k)$ for point datasets (where $\mathbb{P} \times \mathbb{Q} \equiv LAKES \times PARKS$ ($L \times P$), $PARKS \times ROADS$ ($P \times R$), $ROADS \times BUILDINGS$ ($R \times B$) and $BUILDINGS \times ROAD\_NETWORKS$ ($B \times RN$)) respect to the execution time for a fixed $k = 100$. The first conclusion is that the execution times grow as dataset size increases. The best partitioning technique is *Quadtree*, which is approximately 15% faster than *STR*. Moreover, for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) Quadtree is the fastest, e.g., for $B \times RN$ Quadtree is 12% (174 sec) faster than STR.

In Figure 4.27, the chart on the right shows the kCPQ performance for real spatial object datasets ($L \times P$: *polygons* $\times$ *polygons*, $P \times R$: *polygons* $\times$ *line-strings*, $R \times B$: *line-strings* $\times$ *polygons* and $B \times RN$: *polygons* $\times$ *line-strings*) with respect to the execution time. A trend similar to the chart on the left (for points) is observed for the combination of small-to-medium dataset sizes. *Quadtree* is the fastest partitioning technique for the largest dataset combinations (it is slightly better than *STR*), and the *Grid* is the slowest. *Quadtree* outperforms all other partitioning techniques with respect to the running time since it minimizes the number of overlapping partitions between the two files for a DJQ by employing regular space partitioning. Moreover, comparing both charts in Figure 4.27, it may be seen that, when a kCPQ is executed between two datasets of spatial objects, it is more costly than when the two datasets are points, although the trend is very similar. This is because the computation of the distance between two spatial objects (e.g., between two polygons or between a polygon and a line-string) is more costly than computing the distance between a pair of points. It should also be borne in mind that the size of the datasets of spatial objects is larger than the size of point datasets and are, therefore, more costly to retrieve (read) and process. In addition, the distances between spatial objects are much smaller because some objects occupy a certain area with respect to the centroids that do not have any. This reduction in the distance values between spatial objects produces a smaller pruning bound by the plane-sweep kCPQ algorithm, which discards fewer elements in each step of the algorithm.

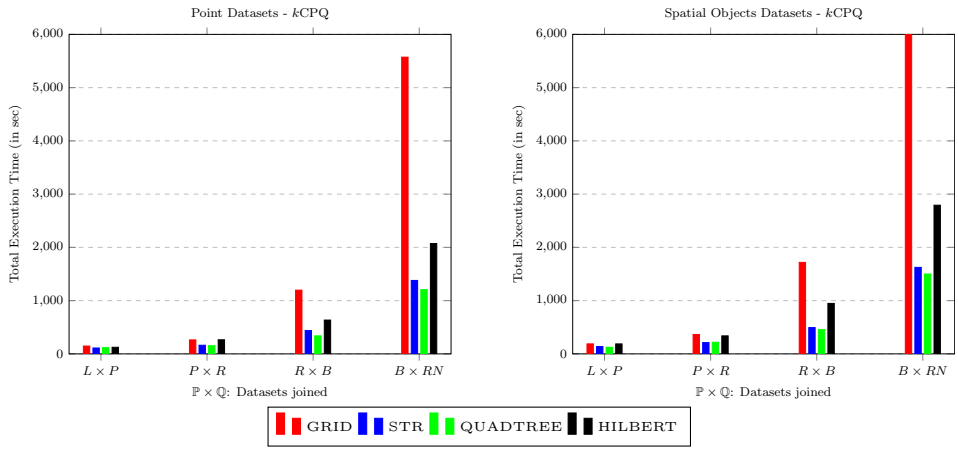


Figure 4.27: kCPQ cost, total execution time of different partitioning techniques, joining points (left) and non-points spatial objects (right).

Figure 4.28 shows the effect of increasing k value in the combination of the biggest datasets ($BUILDINGS \times ROAD\_NETWORKS$) for kCPQ. The first conclusion that can be drawn is that the total execution time grows slowly as the number of results to be found (k) increased for both types of spatial objects. *Quadtree* has very stable execution times, even for large k values (e.g., $k = 10^5$) and when the sets of spatial objects (*polygons* $\times$ *line-strings*) are joined. This means that the *Quadtree* is less affected by the increase of k because *Quadtree* employs a regular space partitioning technique depending on the concentration/density of spatial objects. In Figure 4.28 all algorithms show a slight deviation for the highest values of k.

The main conclusions that we can draw for this experiment are: (1) the kCPQ between two datasets of spatial objects (non-points) has more cost than when joining points because the final distance calculations need more time and the dataset size is higher; (2) the distances between spatial objects are smaller, and so the pruning bound of the *PSKCPQ* is less effective; and (3) the Quadtree again gets the best performance for kCPQ, for all types of spatial datasets.

4.5.3.6 Using Voronoi-Diagram based partitioning

The following experiment aims to measure the effect of using Voronoi-Diagram based partitioning (see Section 3.4) for the kCPQ MapReduce algorithm in SpatialHadoop. Therefore, we will compare this partitioning technique with the *Quadtree* (Q) built-in partitioning technique which has shown to obtain the best performance results for kCPQ. We will consider the k-means++ sampling (the winner in the sampling experiments of Section 3.6.2.1), and the three pivot selection techniques: *random selection* ($Voronoi_{kR}$, V_{kR}), *k-means selection* ($Voronoi_{kk}$, V_{kk}) and *OPTICS selection* ($Voronoi_{kO}$, V_{kO})).

In Figure 4.29, left chart, the kCPQ for a fixed $k = 100$ and for real spatial datasets ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) is shown with respect to the execution time for the

© 2021 García-García, F.J.

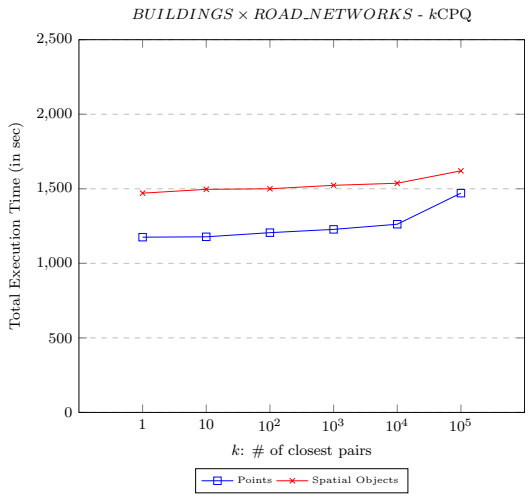


Figure 4.28: kCPQ cost (Quadtree-based partitioning), total execution time vs. k values.

different partitioning techniques (*Voronoi*$_{kk}$, *Voronoi*$_{kR}$, *Voronoi*$_{kO}$ and *Quadtree*). We can observe that the execution times in all partitioning techniques grow almost linearly as the size of the datasets is increased, except *Voronoi*$_{kO}$ that for $P \times R$ the time is very high due to mainly the high *preprocessing* cost. For kCPQ, the best partitioning technique is *Quadtree*, which is approximately 18% faster than *Voronoi*$_{kk}$. Moreover, for the combinations of $L \times P$ and $P \times R$, *Voronoi*$_{kk}$ is slightly faster than *Quadtree* (e.g., for $L \times P$ *Voronoi*$_{kk}$ is 14 sec faster than Quadtree), but for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) *Quadtree* is the fastest, e.g., for $B \times RN$ *Quadtree* is 18% (254 sec) faster than *Voronoi*$_{kk}$. That is, *Voronoi*$_{kk}$ exhibits smaller runtime values for smaller dataset sizes since it produces a slightly larger number of partition combinations (e.g., 24 vs. 23 partition pairs for $L \times P$) that are better distributed in tasks for our cluster of nodes. But for bigger dataset sizes, *Quadtree* is the fastest for kCPQ since it minimizes the number of partitions for each dataset and the number of pairs of partitions that overlap between them. For instance, for the combination of $B \times RN$, *Quadtree* obtains $78 \times 430 = 33540$ possible pairs of partitions, remaining 711 pairs of partitions (2%) after applying the Rule 1, with a total execution time of 1220 sec. In the case of *Voronoi*$_{kk}$, it generates $81 \times 512 = 41472$ pairs of partitions, remaining 1191 pairs of partitions (2.8%) after applying Rule 2, with a total execution time of 1474 sec, that is slightly higher than for Quadtree due to the increase on the number of *map* tasks. Finally, *Voronoi*$_{kO}$ shows the worst results, noting that the indexing time of *Voronoi*$_{kO}$ is much higher and the number of partitions is smaller. Figure 4.29, right chart, shows the effect of increasing the k value for the combination of the biggest datasets ($BUILDINGS \times ROAD\_NETWORKS$) for kCPQ. This experiment shows that the total execution time grows slowly as the number of results to be obtained (k) increases. All partitioning techniques report very stable execution times, even for large

k values (e.g., $k = 10^5$), although, we can see that *Quadtree* still exhibits the best performance (the lowest execution times).

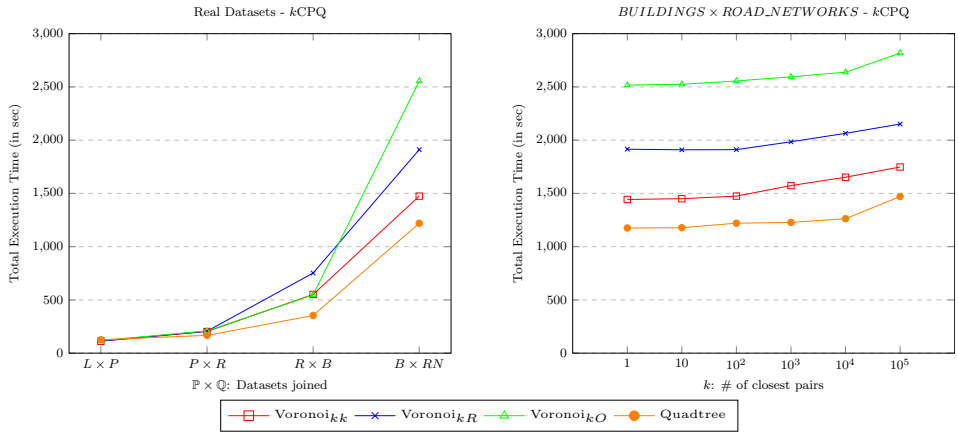


Figure 4.29: kCPQ cost, total execution time of different partitioning techniques, joining real datasets (left) and varying the k values (right).

4.5.3.7 Extensibility varying the ℙ dataset area

In this experiment, we evaluate the extensibility of the proposed kCPQ MapReduce algorithm, considering different percentages (γ) of the $\mathbb{P}$ dataset area and keeping $\mathbb{Q}$ fixed. In this experiment, we compare our best approach using the Voronoi-Diagram based partitioning technique ($Voronoi_{kk}$) to Quadtree. We aim to assess the performance of this DJQ when the amount of data is massive, varying the smallest dataset ($\mathbb{P}$) by executing a *Window Query* centered on the MBR of $\mathbb{P}$ with a percentage (γ) of the original MBR. In the case of *ROADS* and the γ values of 25%, 50%, 75%, and 100%, we have obtained a percentage of points of 2%, 27%, 70%, and 100% from the original dataset $\mathbb{P}$.

Figure 4.30 shows that $Voronoi_{kk}$ presents smaller execution times when the size of the datasets is smaller since the pruning rule with *MmDist* works better, and there is still a higher number of partitions. However, as shown in the experiments of Section 4.5.3.6, *Quadtree* minimizes the number of partitions and therefore obtains better results for high γ values.

4.5.3.8 Speedup of the algorithm

This final experiment aims to measure the speedup of the proposed kCPQ MapReduce algorithm, varying the number of computing nodes (η). To evaluate the scalability performance, we compare our best approach using the Voronoi-Diagram based partitioning technique ($Voronoi_{kk}$) to the same MapReduce algorithm using the *Quadtree* partitioning scheme.

© 2021 García-García, F.J.

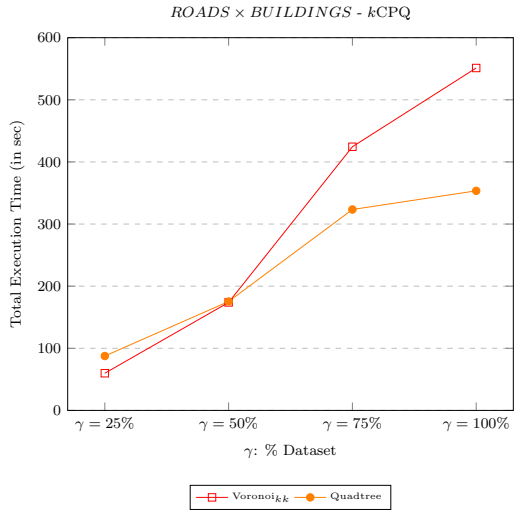


Figure 4.30: kCPQ cost, total execution time for the combination of $ROADS \times BUILDINGS$, considering different γ (%) values for $k = 100$.

Figure 4.31 shows the impact of considering a different number of computing nodes on the performance of the kCPQ MapReduce algorithm for $BUILDINGS \times PARKS$ and using default configuration values. From this chart, we can conclude that the performance of our approach has a direct relationship with the number of computing nodes. It could also be deduced that better performance would be obtained if more

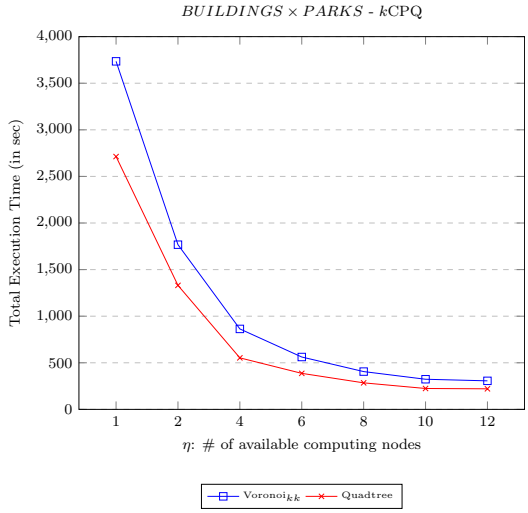


Figure 4.31: kCPQ cost with respect to the number of computing nodes η (Speedup).

computing nodes are added, but when the number of computing nodes exceeds the number of *map* tasks, no improvement is obtained. Finally, we can gather that *Quadtree* exhibits lower execution times than $Voronoi_{kk}$.

4.5.3.9 Conclusions from the experimental results

We have experimentally demonstrated the *efficiency* (in terms of total execution time) and the *scalability* (in terms of k values, sizes of datasets and number of computing nodes (η)) of the proposed kCPQ MapReduce algorithm in SpatialHadoop. From the previous experimental results, we can extract the following conclusions:

1. The initial kCPQ MapReduce algorithm, described in Section 4.3.2, is significantly improved by the three methods for the computation of an upper bound distance β from Section 4.4.2. More specifically, the local computation methods, based either on sampling or the α-allowance approximate technique, have shown the best improvements in the efficiency of the kCPQ algorithm.

2. Alternative plane-sweep-based algorithms (*Classic* and *Reverse Run*) in the MapReduce implementation have similar performances in terms of execution time (*Reverse Run* is slightly faster), although they are faster than using local indices (R-trees) in each *map* task.

3. The *Quadtree* partitioning technique improves the efficiency of the kCPQ MapReduce algorithms significantly. This is due to the regular division of the space, according to the data distribution (the densities of the partitions depend on the concentration of points) [Eldawy et al., 2015].

4. The larger the k, the higher the probability that pairs of partitions are not pruned, more *map* tasks will be needed, and higher total execution time is spent in reporting the final query result.

5. When combining spatial objects (non-points), the running time is slightly higher than for points but following similar trends.

6. *Quadtree* also outperforms all Voronoi-Diagram based partitioning techniques concerning the execution time for the kCPQ, although $Voronoi_{kk}$ (V_{kk}) technique presents slightly better performance for the combinations of the smallest datasets.

7. In the experiments varying the γ values (extensibility), if the size of the MBR of $\mathbb{P}$ is smaller compared to $\mathbb{Q}$, $Voronoi_{kk}$ presents a slightly better behavior than *Quadtree* for kCPQ. However, for medium and large γ values, *Quadtree* gets the best performance.

8. The larger the number of computing nodes (η), the faster the kCPQ MapReduce algorithm is, but when η exceeds the number of *map* tasks, no improvement for the whole job is obtained.

© 2021 García-García, F.J.

4.5.4 εDJQ experiments

This section collects the most representative results of several experiments over our εDJQ MapReduce algorithm [García-García et al., 2018b, García-García et al., 2020c]. For this performance evaluation, we have used the same datasets from the kCPQ experiments. For instance, we have used synthetic clustered datasets of distinct sizes (25M, 50M, 75M, 100M and 125M points), real datasets (*LAKES*, *PARKS*, *ROADS*, *BUILDINGS* and *ROADS_NETWORKS*) and the quasi-real dataset (*CLUS_LAKES*). Table 4.5 summarizes the configuration parameters used in our experiments, note that ε represents a distance threshold. Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	Values (default)
point distance, ε ($\times 10^{-4}$)	2.5, 5, 7.5, 12.5, (25), 50
non-point distance, ε ($\times 10^{-5}$)	7.5, 10, 25, 50, 75, (100)
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Grid, (STR), Quadtree, Hilbert
PS algorithms	(Classic), Reverse Run
PS improvements	Strip, Window, (Semi-Circle)

Table 4.5: Configuration parameters used in our εDJQ experiments.

4.5.4.1 Comparison of different plane-sweep algorithms and the use of local indices

For the εDJQ, we have designed and executed similar experiments to those in Section 4.5.3.2 for the kCPQ to detect which is the best variant of the plane-sweep algorithms. Table 4.6 shows these results, and we can observe that the *Strip* variant of *Classic* and *Reverse Run* is the slowest, but *Window* and *Semi-Circle* have very close execution times, and the *Classic Semi-Circle* is slightly the fastest. Moreover, as for the kCPQ, we have adapted the distributed join algorithm [Eldawy and Mokbel, 2015] to implement a distributed εDJQ algorithm using the *Classic* plane-sweep technique in each combination of pairs of nodes of local R-trees. As shown in the last row of Table 4.6, its total execution time is much higher than one of the plane-sweep-based algorithms without using local indices (R-trees). The justification is very similar to that of the kCPQ, in which the use of a single plane-sweep algorithm for the entire partition is favored over multiple accesses and executions of plane-sweep-based εDJQ algorithm on pairs of R-tree leaves. Finally, we can highlight that for the smallest datasets combination, $LAKES \times PARKS$ ($L \times P$), the *Classic Semi-Circle* is around 23 times faster than using the local indices, while for the join of larger datasets, $BUILDINGS \times PARKS$ ($B \times P$), *Classic Semi-Circle* is around 25 times faster.

εDJQ Algorithm	$L \times P$	$B \times P$
Classic Strip	275.701	2798.069
Classic Window	98.024	418.473
Classic Semi-Circle	91.923	391.612
Reverse Strip	268.777	2506.165
Reverse Window	99.150	437.814
Reverse Semi-Circle	98.981	434.038
Local indices (R-tree)	2129.338	9748.563

Table 4.6: Total execution time (in sec) spent by each εDJQ algorithm, plane-sweep without indices and with local indices (R-tree).

4.5.4.2 The effect of using different spatial partitioning techniques

This experiment studies the effect of choosing a partitioning technique for the proposed εDJQ MapReduce algorithm. Similar to kCPQ, this choice affects the execution time of εDJQ, regardless of whether the datasets are real or synthetic. For instance, for real datasets (see the left chart of Figure 4.32), for the combination of large datasets, $LAKES \times PARKS$ ($L \times P$), Hilbert partitioning is slightly faster than the other techniques (e.g., it is 11 sec faster than STR, which is the second), but for $BUILDINGS \times PARKS$ ($B \times P$), Quadtree is the fastest (82 sec faster than the second, STR), and for the largest datasets combination, $BUILDINGS \times CLUS\_LAKES$ ($B \times C\_L$), STR is the fastest (324 sec faster than Quadtree). From these results with real data, we can conclude that the bigger the datasets, the better the performance of STR for εDJQ. The behavior for synthetic datasets is not so different (see the right chart

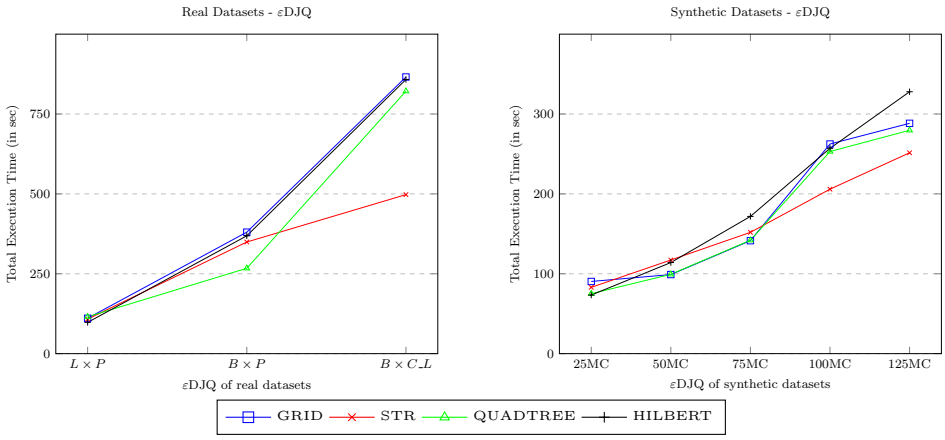


Figure 4.32: εDJQ cost, total execution time of different partition techniques, combining real (left) and synthetic datasets (right).

© 2021 García-García, F.J.

of Figure 4.32), although the data distribution is distinct. In the same way, the trends of replication-based techniques (Quadtree and Grid) are very similar to those of kCPQ, as the case for distribution-based (STR and Hilbert), with small gaps between them. Moreover, for the combination of the smallest synthetic datasets, $25MC1 \times 25MC2$ (25M), again Hilbert is slightly the fastest (only 2 sec faster than Quadtree). The Quadtree is the fastest for the combination of $50MC1 \times 50MC2$ (50M) and $75MC1 \times 75MC2$ (75M), while STR is the fastest for the biggest synthetic datasets (e.g., for $125MC1 \times 125MC2$ (125M), STR is 28 sec faster than Quadtree, which is the second). Similarly to real datasets, we can conclude for synthetic data that the bigger the datasets, the better the performance of STR for εDJQ. Also note that, when we write on the X-axis of the chart for synthetic datasets $25MC$, we mean $25MC1 \times 25MC2$.

Lastly, we should highlight the excellent behavior of *Quadtree* partitioning technique, which reports the lower execution times in most of the cases (mainly for real datasets), as in the previous kCPQ experiments and in [Eldawy et al., 2015] for distributed overlap join.

4.5.4.3 The effect of the increment of ε values

In this experiment, we study the effect of increasing the ε value in the εDJQ MapReduce algorithm for the combination of the biggest datasets (real and synthetic). As shown in the left chart of Figure 4.33, the total execution time for real datasets ($BUILDINGS \times CLUS\_LAKES$) grows as ε increases. Both partitioning techniques (Quadtree and STR) have similar performance for all ε values, except for $\varepsilon = 50 \times 10^{-4}$, where STR outperforms Quadtree (i.e., STR is 295 sec faster). For the combination of synthetic datasets ($125MC1 \times 125MC2$) in the right chart, for small ε values both techniques (Quadtree and STR) have similar performance, but for larger ε values Quadtree is faster than STR (e.g., Quadtree is 65 sec faster for $\varepsilon = 25 \times 10^{-4}$).

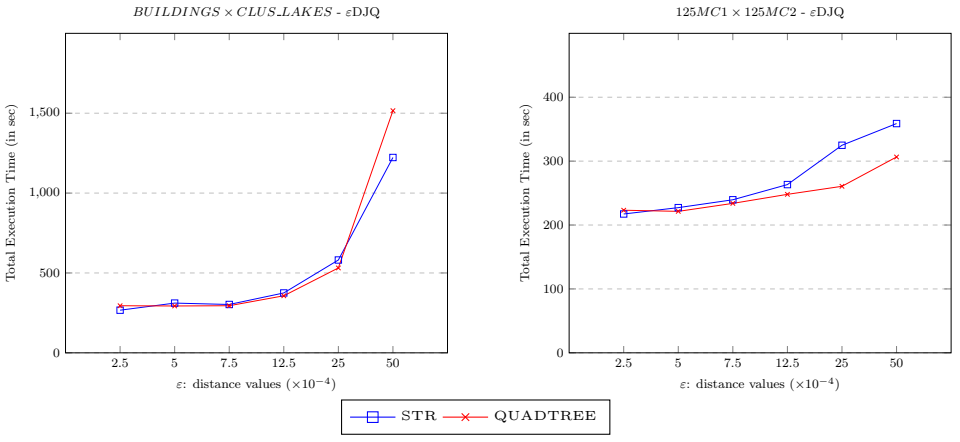


Figure 4.33: εDJQ cost, total execution time vs. ε values.

Similar conclusions to the kCPQ performance can be extracted for the εDJQ: (1) the Quadtree outperforms STR for the εDJQ mainly for synthetic datasets (for real datasets, except for large ε values) and (2) the higher the ε values, the greater the possibility that pairs of partitions are not pruned, more *map* tasks are needed and more total execution time is required.

4.5.4.4 The effect of extending the algorithm for non-points spatial objects

In this experiment, we analyze the performance of the εDJQ algorithm in SpatialHadoop when applying the extension for processing non-points spatial objects (see Section 4.4.1). Consequently, we will study the effect of varying different parameters, such as dataset size, type of spatial object, partitioning technique, and ε value.

Figure 4.34 shows the $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ performance for point datasets (where $\mathbb{P} \times \mathbb{Q} \equiv LAKES \times PARKS$ ($L \times P$), $PARKS \times ROADS$ ($P \times R$), $ROADS \times BUILDINGS$ ($R \times B$) and $BUILDINGS \times ROAD\_NETWORKS$ ($B \times RN$)) respect to the total execution time for a fixed $\varepsilon = 0.001$ (100×10^{-5}). As for kCPQ, the choice of partitioning technique clearly affected the εDJQ execution time, and again *Quadtree* performance is the best for point datasets (slightly better than *STR*) as seen in the chart on the left. For the combinations of the biggest datasets ($R \times B$ and $B \times RN$) *Quadtree* is the fastest, e.g., for $B \times RN$ *Quadtree* is 8% (91 sec) faster than *STR*.

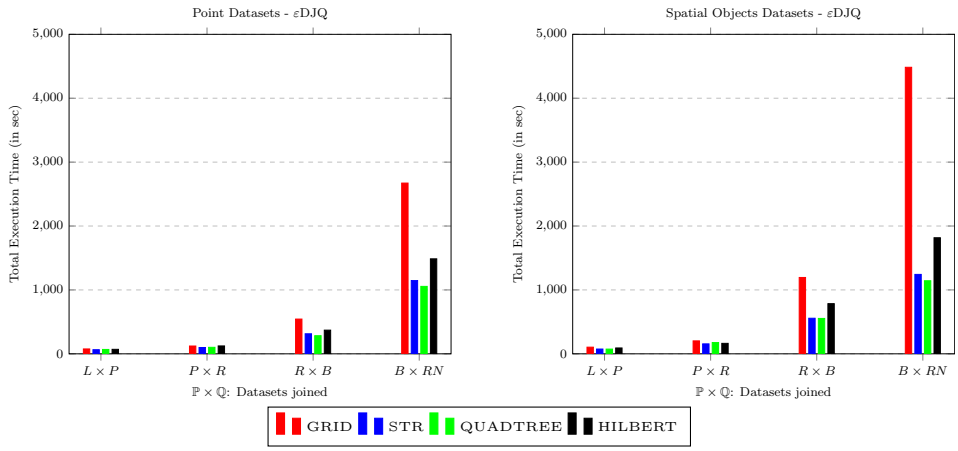


Figure 4.34: εDJQ cost, total execution time of different partitioning techniques for points (left) and non-points spatial objects (right).

In the chart on the right of Figure 4.34, the results of εDJQ for real spatial object datasets are shown with respect to the total execution time ($\varepsilon = 0.001$). The trend is similar to the left-hand chart, where the *STR* partitioning technique is the fastest in all cases (slightly faster than *Quadtree*, except for $B \times RN$), and again the Grid is the slowest. For example, *Quadtree* is 9% (100 sec) faster than *STR* in the combination of the biggest datasets. Therefore, the conclusion is that the bigger the datasets, the better the performance of *Quadtree* for εDJQ. A comparison of the two charts in Figure 4.34

© 2021 García-García, F.J.

shows that for the same ε value, the εDJQ between two datasets of spatial objects is more expensive than when the two datasets are points (the same as for kCPQ), although the trend is very similar. The reason is that the computation of the distance between spatial objects is more costly than the distance between points, and because, just as for kCPQ, the distances between spatial objects are smaller, returning more results for the same ε.

Figure 4.35 shows the total execution time grows as ε is increased. At higher ε values, execution times started to increase due to the increment in the number of elements in the results. If εDJQ behavior in joining points and spatial objects is compared in SpatialHadoop, it may be seen that when a εDJQ is executed between two point datasets, the execution time is smaller than when the two datasets are spatial objects for small and medium ε values. However, for higher ε values ($\varepsilon >= 75 \times 10^{-5}$), its performance is worse, even though the calculation of the distance between spatial objects is more expensive than for points. The main reason is that the resulting partitions from *Quadtree* partitioning are different for each type, and in the case of spatial objects, they tend to contain fewer elements. Therefore, in this particular case, the workload is more balanced, and there are less skewed data when dealing with spatial objects than with points for higher ε values.

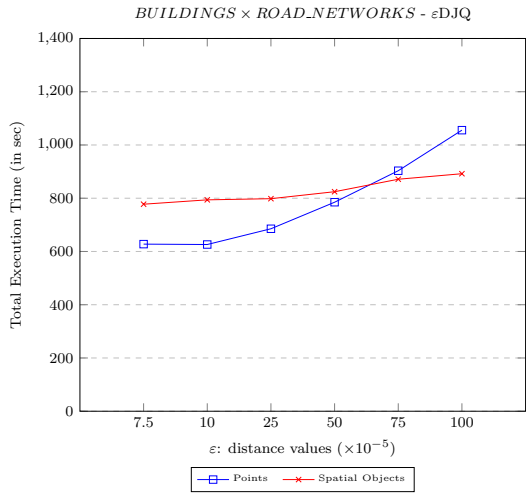


Figure 4.35: εDJQ cost (Quadtree), total execution time vs. ε values.

The main conclusions that we can extract for this experiment are: (1) the εDJQ between two datasets of spatial objects has more cost than when joining points because the final distance calculations need more time and the dataset size is higher; (2) the distances between spatial objects are smaller, so the query returns more results for the same ε; and (3) although the *STR* partitioning technique has slightly better results for smaller datasets, the *Quadtree* gets the best performance when the biggest datasets are combined.

4.5.4.5 Speedup of the algorithm

This experiment aims to measure the speedup of the εDJQ MapReduce algorithm varying the number of computing nodes (η). We have used the *Quadtree* as the partitioning technique, even though STR has a very similar trend. Figure 4.36 shows the impact of changing the number of computing nodes on the performance of εDJQ MapReduce algorithm, for $BUILDINGS \times PARKS$ with the default configuration values. From this chart, we can conclude that the performance of our approach has a direct relationship with the number of computing nodes. It could also be deduced that better performance would be obtained if more computing nodes are added. However, when the number of computing nodes exceeds the number of *map* tasks, no improvement for the whole job is obtained.

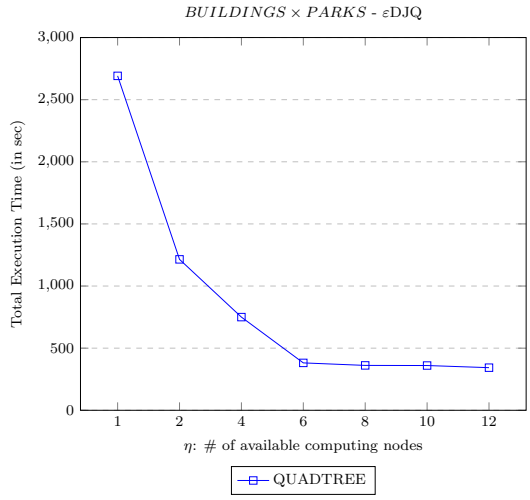


Figure 4.36: εDJQ cost with respect to the number of computing nodes η (Speedup).

4.5.4.6 Conclusions from the experimental results

We have experimentally demonstrated the *efficiency* and the *scalability* of the proposed εDJQ MapReduce algorithm in SpatialHadoop. The most relevant conclusions that we can draw from the previous experiments are the following:

1. The use of plane-sweep-based algorithms (either *Classic* or *Reverse Run* version) for the εDJQ in SpatialHadoop allows lower execution times than utilizing local indices (R-trees).

2. The larger the ε value, the higher the probability that pairs of partitions are not pruned, more *map* tasks will be needed, and higher total execution time is spent in reporting the final result.

3. The use of the *Quadtree* partitioning technique improves the performance of the εDJQ algorithm. Although the *STR* partitioning technique has slightly better

© 2021 García-García, F.J.

results for smaller datasets, the *Quadtree* gets the best performance when the biggest datasets are joined. For instance, its regular division of the space minimizes the number of overlapping partition pairs when increasing the ε value.

4. When combining spatial objects, the computation of the final distance calculations increases the total execution time, and the fact that the distance between spatial objects is less than the distance between points makes the query returns more final results for the same ε. However, the performance follows similar behavior for both types of spatial objects (points and non-points).

5. εDJQ shows better performance when the number of computing nodes (η) is increased, but if there are not enough tasks available for a specific number of nodes, no performance improvements are obtained.

4.5.5 kNNJQ experiments

In this experimental section, we expose the most significant results of our performance evaluation of the kNNJQ MapReduce algorithm [García-García et al., 2020b, García-García et al., 2020c]. We have used only real-world datasets because this type of data is more realistic, and the conclusions that we can infer from their experimental results are more representative than if we use synthetic ones. We have utilized the following real datasets (described in Section 4.5.1): *LAKES* (L), *PARKS* (P), *ROADS* (R), *BUILDINGS* (B) and *ROAD_NETWORKS* (RN). Again, we have transformed spatial objects to points by using the center of each MBR and the centroid of each polygon.

To study the performance of the kNNJQ MapReduce algorithm, we have utilized the *Quadtree* as the global partitioning technique due to the excellent results reported in all join operations [Eldawy et al., 2015, García-García et al., 2018b]. To test the improvements related to the use of *repartitioning* techniques, we have employed *Grid* and *Quadtree* partitioning methods. When Voronoi-Diagram based partitioning is applied, the same technique will be used for both *Global partitioning* and *Repartitioning* phases.

Table 4.7 summarizes the configuration parameters employed in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	Values (default)
# of nearest neighbors, k	1, (10), 25, 50, 75, 100
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	(Quadtree), Voronoi
Repartitioning technique	None, Grid, (Quadtree), Voronoi
Sampling (Voronoi)	k-means++
Pivot selection (Voronoi)	Random, (k-means), OPTICS
% $\mathbb{P}$ area, γ	25, 50, 75, (100)

Table 4.7: Configuration parameters used in our kNNJQ experiments.

4.5.5.1 The effect of using repartitioning techniques

The first experiment for the kNNJQ MapReduce algorithm studies the effect of using *repartitioning* techniques (see Section 4.4.3.1). For this aim, we measure the variation of two parameters, such as the dataset sizes to be joined (scalability) and k values. Also, note that $L = 100000$ and $r = 2\%$ for *Quadtree*-based repartitioning and $L = 50000$ for the *Grid*-based repartitioning techniques. Remember that L represents the maximum number of elements in each partition, and r is the sample ratio in the sampling process of the *Quadtree*-based repartitioning. In the chart on the left, in Figure 4.37, the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query is shown with respect to the execution time and $k = 10$, where $\mathbb{P} = LAKES$ has been fixed as the smallest dataset and the others as $\mathbb{Q}$ ($PARKS$, $ROADS$, $BUILDINGS$ and $ROAD\_NETWORKS$), resulting in the following combinations of $\mathbb{P} \times \mathbb{Q}$: $L \times P$, $L \times R$, $L \times B$ and $L \times RN$. The most important conclusion that can be arrived at from this chart is that the *Quadtree*-based repartitioning technique is the fastest, next is *Grid*-based, whereas the worst alternative is not to use any repartitioning technique (mainly when joining the biggest datasets). For example, for $L \times P$, *Quadtree* is 1.8 times faster than *Grid* and for $L \times RN$, *Quadtree* is 4.8 times faster. Another important result is that *Quadtree*-based repartitioning technique is quite stable, with increase in size of $\mathbb{Q}$ dataset for a fixed $k = 10$. For instance, from $L \times P$ to $L \times RN$, the increment is 53.4% (1491 sec) when the increment of $ROAD\_NETWORKS$ (717M) with respect to $PARKS$ (10M) is huge in terms of the number of points. Another conclusion is that *Quadtree* has quite stable execution times with a sub-linear increment as the size of the dataset ($\mathbb{Q}$) grows. This excellent behavior of SpatialHadoop with *Quadtree*-based repartitioning technique is because its repartitioning technique deals with skewed data very well.

The chart on the right, in Figure 4.37, shows the effect of increasing the value of k for the combined datasets ($LAKES \times PARKS$). We can observe that the use of

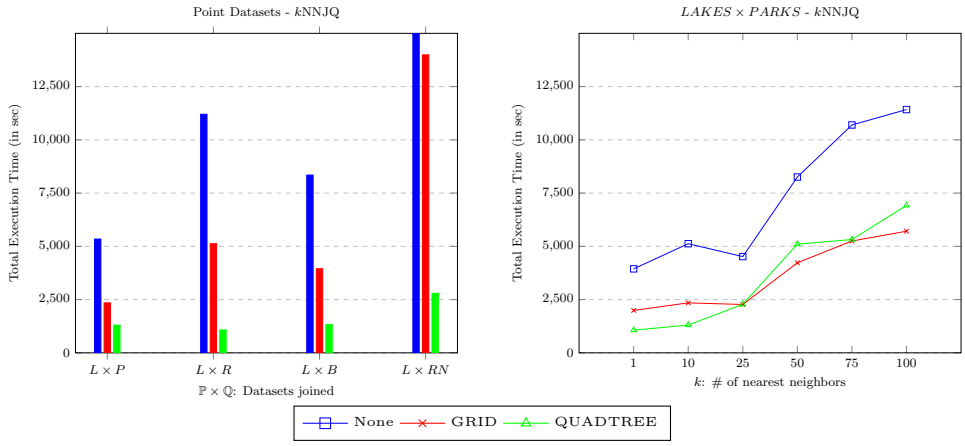


Figure 4.37: kNNJQ cost, total execution time of different datasets combinations (left) and varying the k values (right).

© 2021 García-García, F.J.

repartitioning techniques improves the kNNJQ performance in SpatialHadoop highly. Moreover, we can see that for small k values ($k \leq 25$), *Quadtree* is faster than *Grid*, but when k is large ($k > 25$) *Grid* takes a similar or even shorter time to report the query result. This behavior is because the repartitioning techniques produce different types of partition subsets. In the case of *Grid*, it is a uniform distribution where all partitions are the same size, whereas for *Quadtree*, it is a regular space partitioning technique based on the concentration of spatial objects, and therefore, it generates different sized partitions. In an algorithm like kNNJQ, an increase in k augments the possibility of selecting more partitions overlapping with the ranges of distances found in the *Bin kNN Join* phase. Therefore, the same point must be compared in more than one partition, increasing the size of shuffled data and having a partial kNN list for each partition that must be combined in the last phase of the algorithm. These results suggest that as k increases, the number of overlapping partitions increases to a greater extent and more suddenly for *Quadtree*-based repartitioning than for *Grid*-based.

The following experiment with the kNNJQ MapReduce algorithm compares the *Grid*-based and *Quadtree*-based repartitioning techniques in SpatialHadoop by evaluating the cost, in total execution time and shuffled data in each of the phases in the query algorithm. In Figure 4.38, the chart on the left, the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query for the combination of the $LAKES \times PARKS$ datasets is shown for each repartitioning technique and fixing $k = 10$. We can observe that SpatialHadoop with the *Quadtree*-based repartitioning technique has the best performance. *Grid* is much slower, especially in the *kNNJ on Overlapping Cells* phase. This is because the *Quadtree* partitions the data better since it takes into account its skewed distribution, so after the *Bin kNNJ* phase, there are more final kNN lists, and therefore, the processing time for the next phase is shorter. The *kNNJ on Overlapping Cells* phase is usually more costly if the number of final kNN lists from the previous phase is smaller because, during the range query

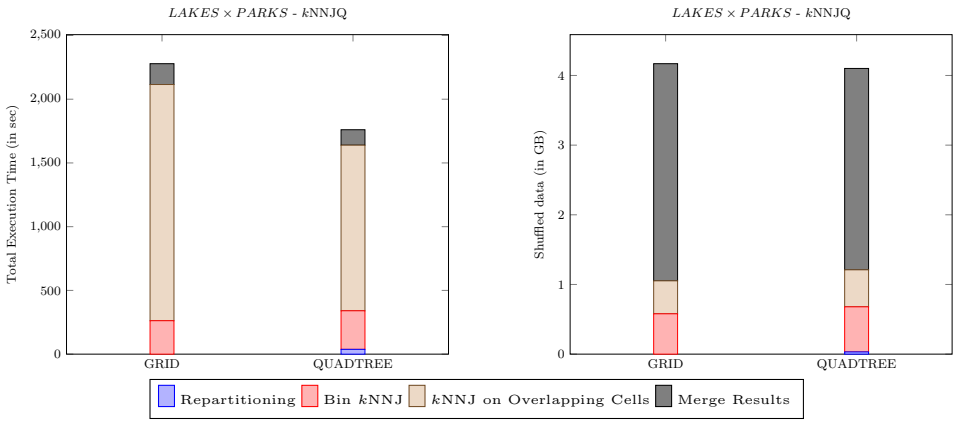


Figure 4.38: kNNJQ cost per phase considering different repartitioning techniques on the combination of the smallest datasets. Total execution time in sec (left) and shuffled data in GBytes (right).

on the nearby partitions, the number of partitions to be searched for kNN candidates grows. Finally, the execution time required for *Quadtree*-based repartitioning technique in the *Repartitioning* phase is very short (2% over the total time) compared to the saved time (28% faster than *Grid*-based repartitioning technique).

The chart on the right, in Figure 4.38, shows the results of the same query and parameters as in the previous experiment, but, in this case, considering the amount of shuffling data exchanged in the different MapReduce phases of the *Grid*-based and *Quadtree*-based repartitioning techniques for kNNJQ in SpatialHadoop. On the one hand, we can observe that the difference between the *Bin kNNJ* and *kNNJ on Overlapping Cells* phases is almost negligible. On the other hand, there are more shuffled data in the *Merge Results* phase for *Grid* than for *Quadtree*. This confirms that the *Grid*-based repartitioning technique generated more partial kNN lists, and therefore, the *Merge Results* phase must process all of them for the final query result. In addition, *Grid* has to process more data, and as a consequence, more time is spent in the *Merge Results* phase, as we can see in Figure 4.38, left-hand chart.

Continuing with the above experiment, Figure 4.39 shows the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query for the combination of the datasets ($LAKES \times ROAD\_NETWORKS$) for each repartitioning technique and for a fixed $k = 10$. The left chart of Figure 4.39 shows the time consumed by the different phases of the algorithm. The first conclusion would be that the differences are greater than for the kNNJQ with smaller datasets. The widest time difference between *Grid* and *Quadtree* appears in the *kNNJ on Overlapping Cells* phase (10 times slower for *Quadtree*). The main reason is that the *Grid*-based repartitioning technique leaves fewer final kNN lists after the *Bin kNNJ* phase, and so, the algorithm generates more partitions than *Quadtree*, and therefore, requires more tasks. Moreover, *Grid* may have problems with skewed data because its uniform partitioning does not take into account the skewed distribution of the data, which could also generate parti-

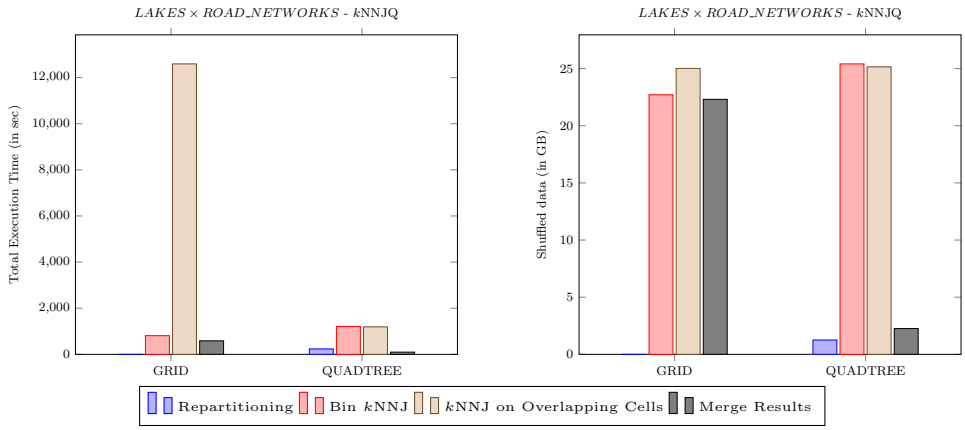


Figure 4.39: kNNJQ cost per phase considering different repartitioning techniques on the combination with the biggest dataset. Total execution time in sec (left) and shuffled data in GBytes (right).

© 2021 García-García, F.J.

tions with many more spatial objects inside. As a consequence of this increment in the number of partial results, the *Merge Results* phase also requires more time to return the final result of the query. Finally, in the *Repartitioning* phase, the execution time required by *Quadtree*-based repartitioning technique is barely 8.5% (234 sec) over the total time, in comparison with the saved time (5 times faster than *Grid*). The right-hand chart in Figure 4.39 shows the cost in shuffled data corresponding to the previous execution times. With the *Quadtree*-based repartitioning technique, there is a bit more shuffled data in the *Bin kNNJ* phase than with *Grid* since there are more partitions. The following *kNNJ on Overlapping Cells* phase presents practically the same values because despite having more final kNN lists, the data must be sent for the largest dataset since it is unknown in advance whether they will be used in the *reduce* part of that phase. Finally, in the *Merge Results* phase, 9.8 times more information is exchanged with *Grid* than with *Quadtree* since more kNN lists are generated in the previous phase.

The chart on the left, in Figure 4.40, shows the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query executed for the combinations of $\mathbb{P} \times \mathbb{Q}$: $L \times P$, $L \times R$, $L \times B$ and $L \times RN$, and the shuffled data cost in Gigabytes for a fixed $k = 10$. The first conclusion is that the shuffled data for both techniques (*Grid* and *Quadtree*) grow as the size of the datasets increases. *Grid* values are a little higher than *Quadtree* for all combinations of datasets because it usually produces fewer final kNN lists for that fixed k. That is, with *Quadtree*-based repartitioning technique, the shuffled data values are lower for all dataset sizes, despite pre-indexing in the *Repartitioning* phase. The right-hand chart, in Figure 4.40, shows the effect of the increment of k value for the combination of the $LAKES \times PARKS$ datasets. For small / medium k values ($k \leq 50$), the shuffled data cost is lower for *Quadtree* than *Grid*, but when k is large ($k > 50$) *Grid* exchanges fewer data to return the same result of the query. As mentioned above, in an algorithm such as kNNJQ, as the value of k increases, the possibility that the number of overlapping partitions also

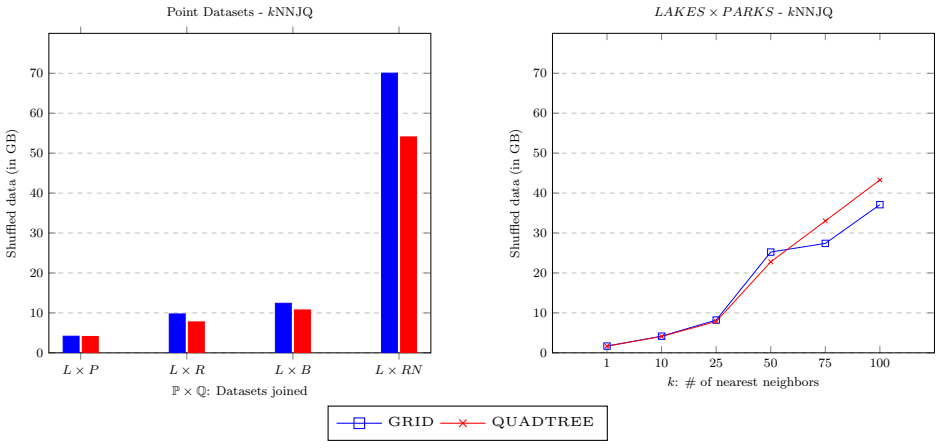


Figure 4.40: kNNJQ cost (shuffled bytes) considering different datasets (left) and varying the k values (right).

increases, and thereby, the shuffled data size of the algorithm. This increment depends on the morphology of the underlying partitioning technique. *Grid*-based repartitioning technique shows more stable values of shuffled data than *Quadtree*-based, because its partitioning is uniform, and all partitions are the same size and shape, as shown in the right-hand chart in Figure 4.40. *Quadtree*-based repartitioning technique presents sharper changes because its partitioning is not uniform, partitions have different sizes and shapes, and therefore, when the distances increase in the range queries, the number of selected partitions does not increase uniformly.

The main conclusions that we can extract for this experiment are: (1) the use of repartitioning techniques accelerates the answer of the kNNJQ; (2) especially the *Quadtree*-based repartitioning technique obtains the best execution times since it is very good at handling skewed data; (3) as k increases, the number of overlapping partitions and kNN lists, also increases, shuffling more data and incrementing the *Merge Results* phase execution time; and (4) the shuffled data for both repartitioning techniques (*Grid* and *Quadtree*) grow as the size of the datasets increases. For these reasons, we have chosen the *Quadtree* as the default repartitioning technique for the rest of the kNNJQ experiments in SpatialHadoop.

4.5.5.2 The effect of using Voronoi-Diagram based partitioning

The aim of this experiment is to study how the use of Voronoi-Diagram based partitioning technique affects the kNNJQ MapReduce algorithm performance. To this end, we will use k-means++ as the partition-based sampling method since this is the best performing technique for kNNJQ processing to find a small but representative profile from big spatial datasets (see Section 3.6.2.1). Therefore, this experiment compares the three *pivot selection* techniques (**R**andom, **k**-means and **O**PTICS) with *k-means++* as the sampling method and *Quadtree* (Q) for the kNNJQ in SpatialHadoop, based on the execution time, in each of the phases. They are denoted as $Voronoi_{kk}$ (V_{kk}), $Voronoi_{kR}$ (V_{kR}) and $Voronoi_{kO}$ (V_{kO}).

In Figure 4.41, left chart, the kNNJQ for the combination of different datasets ($L \times P$, $L \times R$, $L \times B$ and $L \times RN$) is shown for each *pivot selection* technique and for a fixed $k = 10$. We can observe that $Voronoi_{kk}$ exhibits the best performance in all cases. Moreover, *Quadtree* is much slower, especially in the *kNNJ on Overlapping Partitions* phase. This result comes from the fact that, with the three *Voronoi* variants, every point of $\mathbb{P}$ is assigned to $\mathbb{Q}$'s partition that contains at least k elements, so after the *Bin kNNJ* phase, there are more final kNN lists and therefore the processing time of the next phase is reduced. Note that the *kNNJ on Overlapping Partitions* phase is usually more expensive if the number of final kNN lists from the previous phase is lower because, when executing the range query on the nearby partitions, there is a large growth of the number of partitions to search for kNN candidates. Notice the high execution time needed for $L \times RN$ using V_{kO}, this is because the OPTICS algorithm does not generate a fixed number of clusters, but it depends strongly on the data distribution (and the number of clusters is less than k). In this figure, we can also highlight the reduction of the differences in execution time between the four partitioning techniques with the combination of the largest dataset, $L \times RN$, mainly because the *Quadtree* technique

© 2021 García-García, F.J.

returns more final kNN lists. As the volume and size of $\mathbb{Q}$ are much greater, the volume of points in $\mathbb{P}$ that fall into partitions of $\mathbb{Q}$ is also greater, obtaining final results that reduce the execution time of the kNNJQ. Another conclusion that can be drawn from the results is that *Quadtree* is the fastest while *Voronoi*$_{kk}$ is slower for the *Repartitioning* phase. This behavior is due to the use of an algorithm based on *k-means* that makes the total execution time increase slightly, in the same way to the *Indexing* time in the experiments of Section 3.6.2.2. However, this preprocessing technique obtains the best results due to the good handling of the skewed data (e.g., the time spent in the *Bin kNNJ* phase is the smallest).

Moreover, similar performance can be observed in Figure 4.41, right chart, where, as the k value is increased for the combination of the datasets, $LAKES \times ROADS$, the execution time of the *kNNJ on Overlapping Partitions* phase is also higher. Besides, we have to emphasize the high execution time needed for $k = 75$ using V_{kR}, mainly due to the random nature of this pivot selection technique. Notice that the increase of the *Repartitioning* phase time for *Voronoi*$_{kk}$ is less than that shown in the *Indexing* process (see Section 3.6.2.2). This reduction is because the former is done within each partition using a MapReduce job, while the latter runs in the master node. Finally, in the *Merge Results* phase, we can see how *Quadtree* exchanges more information than both *Voronoi* variants since in the previous phase, more kNN lists have been generated for all dataset combinations.

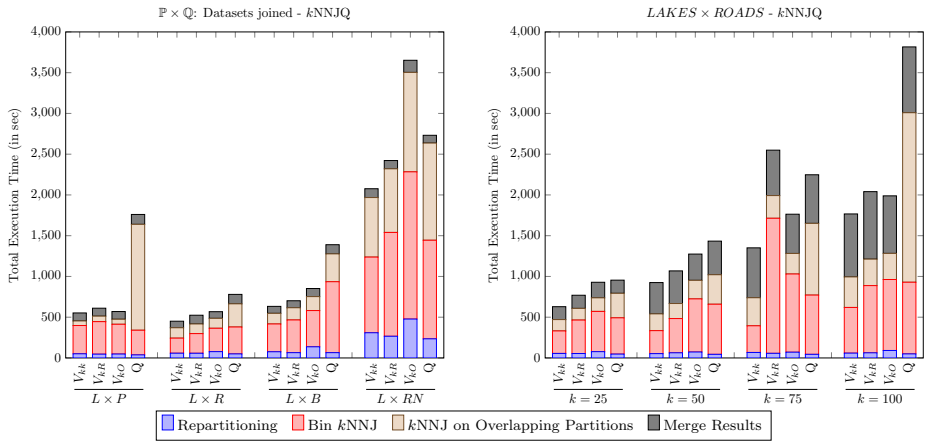


Figure 4.41: kNNJQ cost, total execution time of different partitioning techniques for several datasets combinations (left) and varying the k values (right).

The main conclusions that we can draw from this experiment are: (1) *Voronoi*$_{kk}$ shows the best performance for kNNJQ in SpatialHadoop; (2) *Voronoi*$_{kk}$ generates more final kNN lists after the *Bin kNNJ* phase reducing the execution time in the next phase; (3) *Quadtree* exhibits higher execution times although it is the fastest for the *Repartitioning* phase while *Voronoi*$_{kk}$ is slower; and (4) the differences in execution time between the partitioning techniques based on Voronoi-Diagrams and Quadtree are reduced when joining with the largest $\mathbb{Q}$ dataset, i.e., $L \times RN$.

4.5.5.3 The effect of the improvements

This experiment compares the best variant of Voronoi-Diagram based partitioning technique for kNNJQ MapReduce algorithm designed so far (V_{kk}), with the enhanced version including all improvements proposed in Section 4.4.3 (V_{kkI}), considering the total execution time in each of the phases. In Figure 4.42, left chart, the kNNJQ for the combination of different datasets ($L \times P$, $L \times R$, $L \times B$ and $L \times RN$) is shown for a fixed $k = 10$. We can observe that the $Voronoi_{kkI}$ exhibits the best performance in all cases. The main reason is the reduction in the execution times of phases 3 (*kNNJ on Overlapping Partitions*) and 4 (*Merge Results*) accomplished by using the improvements. For instance, the *pruning rules* (3 and 5) that eliminate points from the dataset $\mathbb{Q}$ that are not part of the final query result and the *less data* technique that decreases the size of the input set (only those points of $\mathbb{P}$ that have not finished) as well as the size of the shuffled data between the MapReduce phases.

Moreover, the right chart of Figure 4.42 shows a similar behavior where, as the k value is increased for the combination of the datasets, $LAKES \times ROADS$, the execution time of the V_{kkI} increases less than for V_{kk}. Also, this time difference grows with the increment of the k value, due mainly to the increase in the size of the partial results (kNN lists). In the improved version, V_{kkI}, only non-final kNN lists are used in phases 3 and 4, causing that when k increases, the non-improved version works with more intermediate data.

These time differences are even larger when the size of the smallest dataset increases, as can be seen in Figure 4.43, left chart. For the combination $ROADS \times BUILDINGS$ (72M points $\times$ 115M points), we observe how the execution times are higher for the unimproved version (V_{kkI} 2860 sec vs. V_{kk} 3746 sec), especially in the phases 3 (*kNNJ on Overlapping Partitions*) and 4 (*Merge Results*). Furthermore, this behavior is explained with Figure 4.43, right chart, which shows that the size of the shuffled data of these

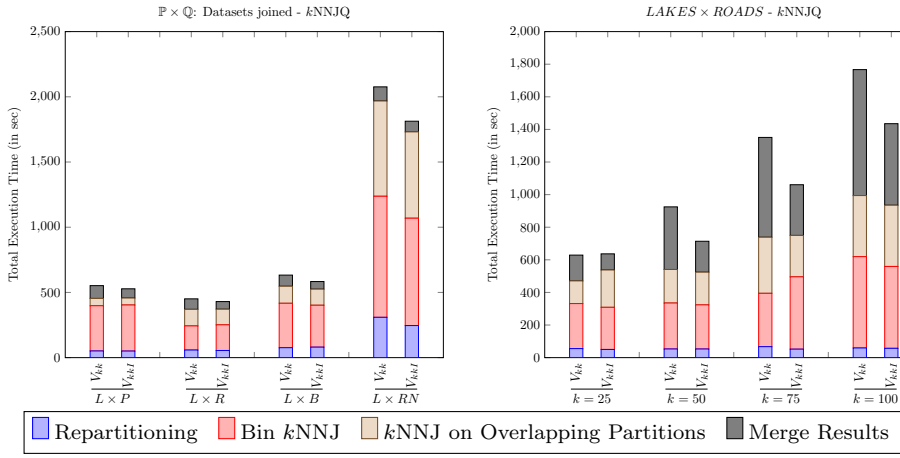


Figure 4.42: kNNJQ cost, total execution time considering the improvements for datasets combinations (left) and varying the k values (right).

© 2021 García-García, F.J.

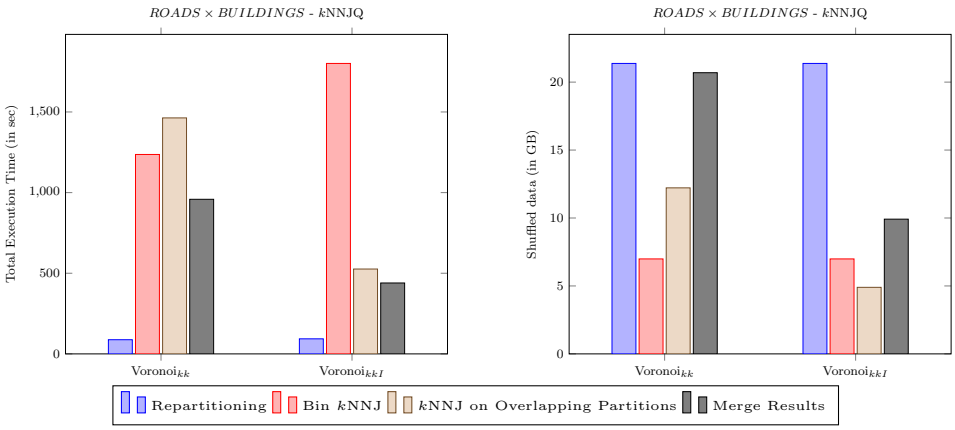


Figure 4.43: kNNJQ cost per phase considering the improvements on the combination $ROADS \times BUILDINGS$. Total execution time in sec (left) and shuffled data in GBytes (right).

phases is greater than double for the non-improved version. Moreover, notice that the calculation of the Rule 6 increases the execution time of the *Bin kNNJ* phase, although it is worth it for the best-obtained results.

The main conclusions that we can deduce for this experiment are the following: (1) $Voronoi_{kkI}$ is the clear winner for the execution time, especially in the *kNNJ on Overlapping Partitions* and the *Merge Results* thanks to the use of *prunning rules* and *less data* technique; (2) growing the k value, the execution time of the V_{kkI} increases less than for V_{kk} because there is less intermediate data to be processed thanks to the *less data* technique and; (3) the application of the Rule 6 gets higher execution times for the *Bin kNNJ* phase, but its results accelerate later phases.

4.5.5.4 Extensibility varying the $\mathbb{P}$ dataset area

In this experiment, as for kCPQ, we evaluate the extensibility of the proposed kNNJQ MapReduce algorithm, considering different percentages (γ) of the $\mathbb{P}$ dataset and keeping $\mathbb{Q}$ fixed. Similar to the kCPQ, we aim to assess the performance of the kNNJQ when the amount of data is massive, varying the smallest dataset ($\mathbb{P}$) by executing a *Window Query* centered on the MBR of $\mathbb{P}$ with a percentage (γ) of the original MBR. In the case of *ROADS* and the γ values of 25%, 50%, 75%, and 100%, we have obtained a percentage of points of 2%, 27%, 70%, and 100% from the original dataset $\mathbb{P}$.

In Figure 4.44 it is shown that, for kNNJQ, when the size of the data is small (γ = 25%), *Quadtree* works better because the cost of the calculation of rules is almost insignificant for the pruning data (very few points removed from the dataset). As the size of the query window increases, the time differences also increase for $Voronoi_{kkI}$, because although the running time of the *Bin kNNJ* phase is slightly higher for the calculation of the rules, the execution times of phases 3 and 4 decrease considerably

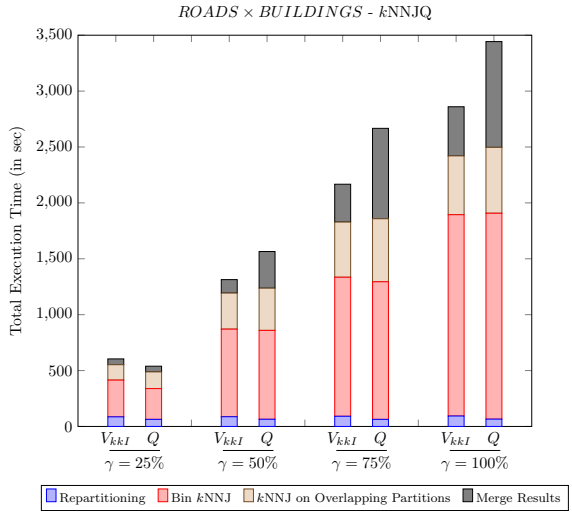


Figure 4.44: kNNJQ cost, total execution time for the combination $ROADS \times BUILDINGS$, considering different γ values (% Dataset $\mathbb{P} = ROADS$) and $k = 10$.

thanks to the fact that the size of the input data (shuffled data) through the use of pruning rules decreases.

4.5.5.5 Speedup of the algorithm

This final experiment intends to measure the speedup of the proposed kNNJQ MapReduce algorithm changing the number of computing nodes (η). To assess the scalability performance, we compare our best approach using the Voronoi-Diagram based partitioning technique ($Voronoi_{kkI}$) to the same MapReduce algorithms using the *Quadtree* partitioning method.

Figure 4.45 shows the influence of varying the number of computing nodes on the performance of kNNJQ MapReduce algorithm, for $LAKES \times PARKS$ with the default configuration values. From this chart, as for kCPQ, we can conclude that the performance of our approach has a direct relationship with the number of considered computing nodes. It could also be inferred that better performance would be obtained if more computing nodes are added to the cluster, but when this number (η) exceeds the number of *map* tasks, there is no improvement. $Voronoi_{kkI}$ is still showing a better behavior than *Quadtree*.

4.5.5.6 Conclusions from the experimental results

We have experimentally demonstrated the *efficiency* and the *scalability* of the proposed kNNJQ MapReduce algorithm in SpatialHadoop. By analyzing the previous experimental results, we can derive the following conclusions:

© 2021 García-García, F.J.

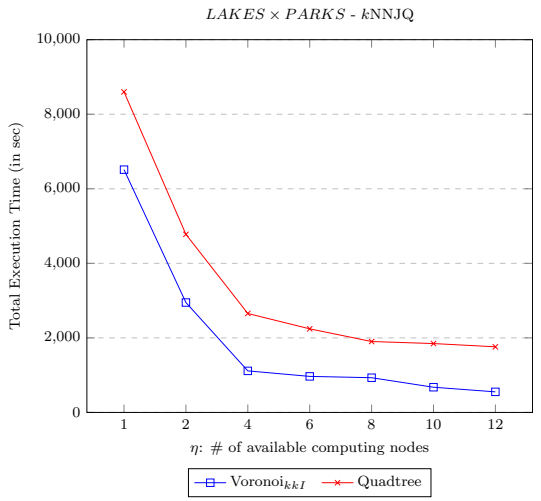


Figure 4.45: kNNJQ cost with respect to the number of computing nodes η (Speedup).

1. The use of *repartitioning* techniques in SpatialHadoop considerably reduces the total execution times and shuffled data, mainly when large datasets are joined in kNNJQ. This indicates that the use of repartitioning techniques is a good policy for MapReduce algorithms based on phases.

2. Considering the k-means++ sampling, we have compared three clustering algorithms (Random, k-means, and OPTICS) for the pivot selection. The *Repartitioning* execution time for V_{kR} is the smallest and grows almost linearly as the size of the datasets, while, for V_{kk}, this increment is larger due to the use of this clustering algorithm. The use of OPTICS, V_{kO}, is the slowest. But V_{kk} exhibits the best global performance in all cases because this combination of k-means algorithms indexes the data appropriately for the next phases in the kNNJQ MapReduce algorithm. Furthermore, the time consumed by k-means algorithm in the *Repartitioning* phase (it is a MapReduce job) is compensated by the gain in performance in subsequent phases of the query processing.

3. V_{kk} is also faster than *Quadtree*, because it deals better with skewed data and it gets more final results in the *Bin kNNJ* phase.

4. The improved version of V_{kk}, V_{kkI}, has been designed to decrease considerably the total execution time, especially in the *kNNJ on Overlapping Partitions* and *Merge Results* phases, by reducing the size of the input data, the shuffled data and the data that is handled in the kNN computation through the use of different pruning rules.

5. In the experiments of varying the γ values (extensibility), if the size of the MBR of $\mathbb{P}$ is very small compared to $\mathbb{Q}$, *Quadtree* presents a better behavior than V_{kkI}

due to lower efficiency of the pruning rules. But when the MBR is large enough, then V_{kkI} shows better performance than *Quadtree*.

6. *Voronoi*$_{kkI}$ outperforms *Quadtree* when the number of computing nodes (η) is increased, but if there are not enough tasks available for a certain value of nodes, no performance improvements are obtained.

4.5.6 εDRJQ experiments

This section shows the experimental results of our εDRJQ MapReduce algorithm [García-García et al., 2020c]. Given that εDRJQ and εDJQ are equivalent (see Section 4.3.5), we have used the same real datasets from the εDJQ experiments to compare both algorithms. For instance, we have used the following real-world 2d point datasets: *LAKES*, *PARKS*, *ROADS*, *BUILDINGS* and *ROADS_NETWORKS*. Moreover, we have set the *Quadtree* partitioning technique and the *Classic Semi-Circle* plane-sweep algorithm as default parameters, being the best configuration for the εDJQ performance. Table 4.8 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	Values (default)
Distance threshold, ε ($\times 10^{-5}$)	7.5, 10, 25, 50, 75, (100)
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Quadtree
PS algorithms	Classic
PS improvements	Semi-Circle

Table 4.8: Configuration parameters used in our εDRJQ experiments.

4.5.6.1 Comparison with εDJQ

The following experiment compares the performance in SpatialHadoop for εDRJQ and εDJQ. Figure 4.46 shows the execution times of both queries, with the joined datasets ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) and $\varepsilon = 0.001$. First, the main conclusion is that εDJQ is the clear winner for the total execution time when varying the joined datasets, especially for the largest combination $B \times RN$ (the total execution time of εDRJQ is double εDJQ). However, when small-to-medium datasets are joined ($L \times P$, $P \times R$ and $R \times B$), εDRJQ execution times are only slightly higher. The main reason is that εDRJQ is a *Reduce*-based Join algorithm, and time is consumed by having to perform data shuffling and sorting between the *map* and *reduce* phases. Therefore, this time especially increases for the $B \times RN$ where the number of combinations of partitions and spatial objects is considerably higher. However, εDJQ is a *Map*-based algorithm, so it does not shuffle data, and the final query result is already at the end of the *map* phase.

© 2021 García-García, F.J.

As shown in the right-hand chart of Figure 4.46, total execution time grows as ε increases. As concluded in Section 4.5.4 for εDJQ, the execution time of εDRJQ increases for larger ε values since more elements participate in the final results. The performance trend is similar for both queries, but εDJQ is faster for any ε value. In addition, the total execution time grows faster for εDRJQ because the size of the data exchanged between the *map* and *reduce* phases also increases.

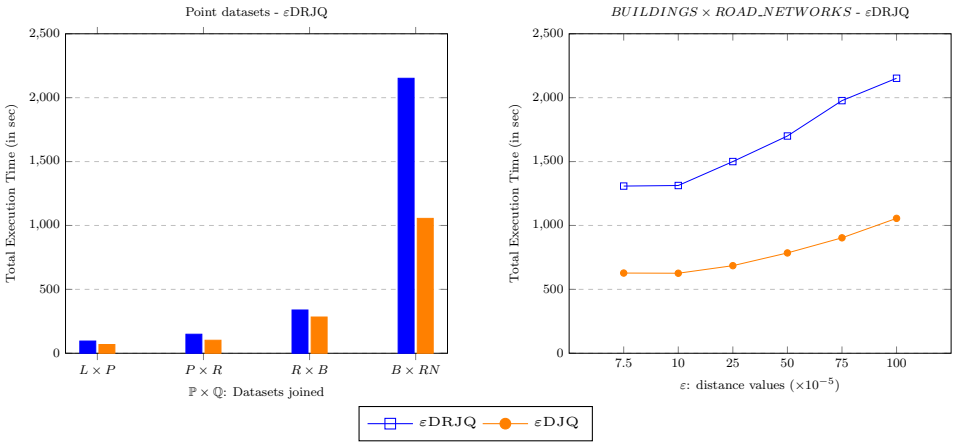


Figure 4.46: εDRJQ cost, total execution time considering different datasets combinations (left) and varying the ε values (right).

4.5.6.2 Speedup of the algorithm

Our last experiment measures the speedup of the proposed εDRJQ MapReduce algorithm, varying the number of computing nodes (η). As shown in Figure 4.47, the execution times for εDRJQ, and as for kCPQ and εDJQ, has shorter values than kNNJQ, which is based on it and both follow a processing scheme of multiple executions. Finally, we can again conclude that a higher number of computing nodes increments performance. However, notice that there is practically no improvement after $\eta = 6$, mainly due to some tasks that take longer due to skewed data problems.

4.5.6.3 Conclusions from the experimental results

The most relevant experimental conclusions of εDRJQ are the following:

1. We have experimentally demonstrated the *efficiency* and the *scalability* of the proposed distributed algorithm for εDRJQ in SpatialHadoop.

2. The larger the ε values, the higher the number of spatial objects checked, so the total execution time increases.

3. εDRJQ is slower than εDJQ because it is a *Reduce*-based Join algorithm, so it needs more time to shuffle data between the *map*, and *reduce* phases.

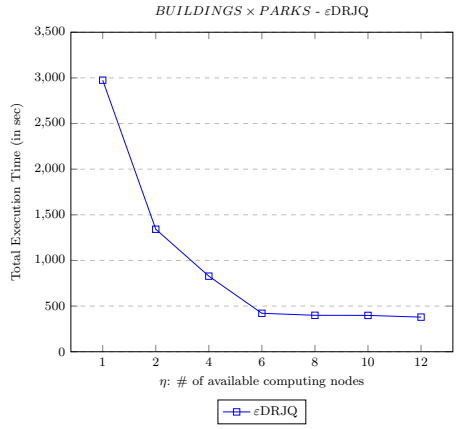


Figure 4.47: εDRJQ cost with respect to the number of computing nodes η (Speedup).

4. The total execution time of εDRJQ grows faster than εDJQ as the ε value increases, due to the high quantity of data shuffling since the first is a *Reduce*-based Join algorithm and the second uses a *Map*-based Join algorithm.

5. εDRJQ shows better performance when the number of computing nodes (η) is increased, but if there are not enough tasks available for a specific number of nodes, no performance improvements are obtained.

4.5.7 Reverse k Nearest Neighbors experiments

In this section, we present the most representative results of our experimental evaluation and comparison of RkNNQ algorithms in SpatialHadoop [García-García et al., 2019]. We have used real-world 2d point datasets from Section 4.5.1 to test our RkNNQ algorithms, that is, *MRSFT* and *MRSLICE* algorithms in SpatialHadoop. Furthermore, all datasets have been previously partitioned by using the STR partitioning technique with a local R-tree index per partition. Finally, to get a representative execution time, a random sample of 100 points from the smallest dataset (*LAKES*) has been obtained, and the average of the execution time of the RkNNQ of these points have been calculated since this query mainly depends on the location of the query point concerning the dataset.

Parameter	Values (default)
Number of regions, t	6, (12), 18, 24, 30
# reverse nn, k	1, 5, (10), 15, 20, 25, 50
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Grid, (STR)

Table 4.9: Configuration parameters used in our RkNNQ experiments.

© 2021 García-García, F.J.

Table 4.9 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

4.5.7.1 The effect of the number of regions

The first experiment of RkNNQ aims to examine the best t value (number of regions) for *MRSLICE*, using the *BUILDINGS* dataset and the k values of 10 and 50. In Figure 4.48 we can see that there is a little difference in the results obtained when the t value is varied, especially for $k = 10$, being greater differences when a larger k value is used (i.e., $k = 50$). On the one hand, for $k = 10$, smaller values of t get faster times (e.g., $t = 6$ has an execution time of 67 sec which is 4 sec faster than $t = 12$). On the other hand, for $k = 50$, $t = 12$ gets the smallest execution time (221 sec) and for $t < 12$ and $t > 12$, the execution time increases. Although there are no large differences, the value of t that shows better performance for both k values is $t = 12$, reaching the same conclusion as in [Yang et al., 2014] but now in a distributed environment. From now on, we will use $t = 12$ in all our experiments.

4.5.7.2 The effect of the increment of the dataset size

Our second experiment of RkNNQ studies the scalability of the RkNNQ MapReduce algorithms (*MRSLICE* and *MRSFT*), varying the dataset sizes. As shown in Figure 4.49 for the RkNNQ of real datasets (*LAKES*, *PARKS*, *ROADS*, *BUILDINGS* and *RN*) and a fixed $k = 10$. The execution times of *MRSLICE* are much lower than those from *MRSFT* (e.g., it is 477 sec faster for the largest dataset *RN*) thanks to how the reduced search space and the limited number of MapReduce jobs. Note that, for *MRSFT*, at least $k * 20 + 1$ jobs are executed, while for the case of *MRSLICE*, 3 jobs are launched at most. In both algorithms, the execution times do not increase too much, showing

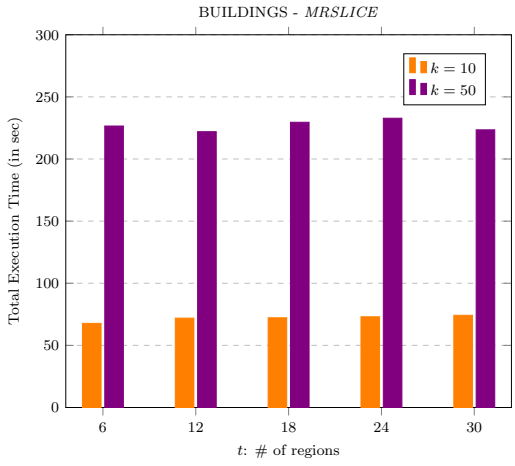


Figure 4.48: *MRSLICE* total execution times considering different t values.

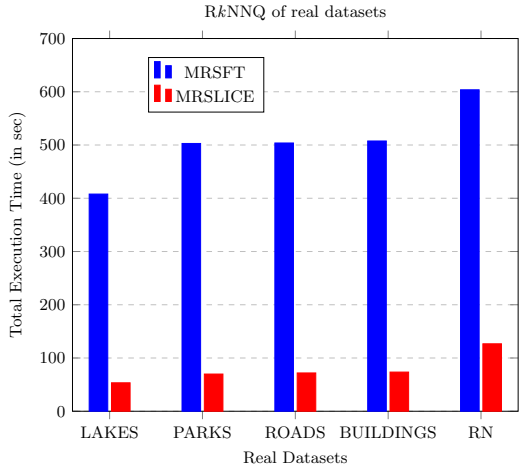


Figure 4.49: RkNNQ total execution times considering different datasets.

quite stable performance, mainly for *MRSLICE*. This is due to the indexing mechanisms provided by SpatialHadoop that allow fast access to only the necessary partitions for query processing. Furthermore, this behavior shows that the number of candidates for *MRSLICE* is almost constant (the expected number of candidates is less than $3.1 * k$ as stated in [Yang et al., 2014]), only showing a visible increment in the execution time for the *RN* dataset, due to the increase in the density of partitions and its distribution causes the need to execute the optional job (phase 1.b) of the *Filtering* phase.

4.5.7.3 The effect of the increment of k values

This experiment aims to measure the effect of increasing the k value for the dataset ($BUILDINGS$). The left chart of Figure 4.50 shows that the total execution time grows as the value of k increases, especially for *MRSFT*. This is because as the value of k increases, the number of candidates $k*20$ also grows, and for each of them, a MapReduce job is executed. On the other hand, *MRSLICE* limits the number of MapReduce jobs to 3, obtaining a much smaller increment and more stable results since the disk accesses are reduced significantly by traversing the index of the dataset a small number of times. Note that the small increment in the execution times when $k = 25$, mainly because when reaching a certain k value, the result of the first job of the *Filtering phase* is not definitive, and it has been necessary to execute the optional job (phase 1.b). In this case, the number of involved partitions in the query increases as well. Finally, the execution time for $k = 50$ increases slightly.

4.5.7.4 Speedup of the algorithms

This experiment studies the speedup of the RkNNQ MapReduce algorithms, varying the number of computing nodes (η). The right chart of Figure 4.50 shows the impact of using

© 2021 García-García, F.J.

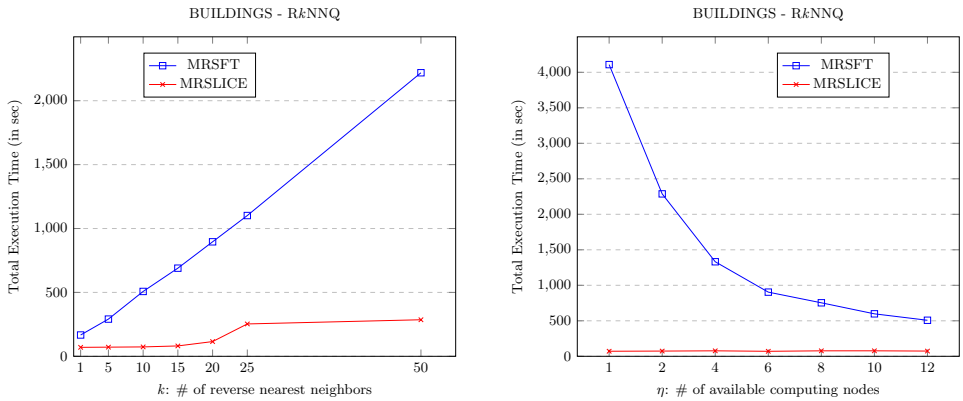


Figure 4.50: RkNNQ cost, total execution time vs. k values (left) and vs. number of computing nodes η (right).

several computing nodes on the performance of RkNNQ MapReduce algorithms, for *BUILDINGS* with a fixed value of $k = 10$. From this chart, we can deduce that *MRSFT* would obtain better performance if more computing nodes are added. *MRSLICE* is still outperforming *MRSFT* and is not affected, despite reducing the number of available computing nodes. This is because *MRSLICE* is an algorithm in which both the number of partitions involved in obtaining the query result and the number of MapReduce jobs are minimized. That is, depending on the location of the query point q and the k value, the number of partitions is usually one, and varying the number of computing nodes does not affect the execution time. However, the use of the computing resources of the cluster is quite small, which allows the execution of several RkNNQs in parallel, taking advantage of the distribution of the dataset into the cluster nodes. On the other hand, *MRSFT* executes several *kNNQs* in parallel, using all computing nodes completely for large k values.

4.5.7.5 Conclusions from the experimental results

We can summarize the following conclusions from the previous experimental study:

1. We have experimentally demonstrated the *efficiency* and the *scalability* of the proposed *MRSLICE* algorithm for RkNNQ and we have compared it with *MRSFT* algorithm in SpatialHadoop.

2. As stated in [Yang et al., 2014], the value of t (the number of equally sized regions in which the dataset is divided) that shows the best performance is 12.

3. *MRSLICE* outperforms *MRSFT* several orders of magnitude (around five times faster), thanks to its pruning capabilities and the limited number of MapReduce jobs.

4. The larger the k values, the greater the number of candidates to be verified, but for *MRSLICE* the number of jobs and partitions involved are quite restricted, and the total execution time increases considerably less than for *MRSFT*.

5. The use of computing nodes by *MRSLICE* is small, allowing the execution of several queries in parallel, unlike *MRSFT* that can leave the cluster busy.

4.6 Conclusions

This section summarizes the main conclusions of this chapter. First, it focuses on the spatial query processing in SpatialHadoop with a detailed description of its features and tools. The use of the *SpatialFileSplitter* and the *SpatialRecordReader* allows us to obtain better performance on spatial operations over Hadoop. Next, we have described the available spatial queries in the *Operations* layer of SpatialHadoop. They are organized as spatial operations (range, $k$nearest neighbor and spatial join queries) and computational geometry algorithms (polygon union, skyline, convex hull, farthest pair, closest pair, and Voronoi-Diagram). Moreover, we have designed and implemented new DBQ MapReduce algorithms in SpatialHadoop: εDRQ, kNNQ, RkNNQ, εDRJQ, kCPQ, kNNJQ and εDJQ. Then we have detailed several extensions and improvements of the previous spatial query algorithms with the use of non-points spatial objects, several new pruning rules, and the use of the Voronoi-Diagram based partitioning technique. Finally, the execution of an extensive set of experiments on synthetic and real-world datasets has demonstrated that our DBQ MapReduce algorithms are efficient, robust, and scalable for parameters such as dataset sizes, k, ε, number of computing nodes (η) and others. Furthermore, these improvements have considerably enhanced the performance of the distributed algorithms, especially for the computation of a β upper-bound for kCPQ and the combination of Voronoi-Diagram based partitioning, *less data* technique and pruning rules for kNNJQ.

© 2021 García-García, F.J.

Chapter 5

Spatial Query Processing in LocationSpark

© 2021 García-García, F.J.

Chapter 5

Spatial Query Processing in LocationSpark

Contents

In this chapter, the structure and spatial operations in LocationSpark are detailed. First, in Section 5.1, the general spatial query processing scheme in LocationSpark is presented together with the different features and tools that it provides to add spatial capabilities to Spark. Next, the spatial queries already supported by LocationSpark are exposed in Section 5.2. Moreover, new spatial queries, extensions and improvements implemented in LocationSpark are described in Section 5.3. Finally, a performance evaluation of several spatial query algorithms and a comparison with their equivalents in SpatialHadoop is presented in Section 5.4.

5.1 LocationSpark for Spatial Query Processing

LocationSpark [Tang et al., 2016, Tang et al., 2020] is a library in Spark that provides an API for spatial query processing and optimization based on Spark's standard dataflow operators. LocationSpark is implemented on top of *Resilient Distributed Datasets* (RDDs); these key components of Spark are fault-tolerant collections of elements that can be operated in parallel. LocationSpark is a library for Spark and provides the *Class LocationRDD* for spatial operations [Tang et al., 2016]. It is an efficient in-memory distributed spatial query processing system (Spark-based spatial analytics system [Pandey et al., 2018]). LocationSpark provides several optimizations to enhance Spark for managing spatial data, and, as shown in Figure 5.1, it is organized by layers:

- *Memory Management.* In this layer for spatial data, LocationSpark uses the cache capabilities of Spark to dynamically cache frequently accessed data into memory and store the less frequently used data on disk.
- *Spatial Index.* LocationSpark builds two levels of spatial indexes (global and local). To build a *global index*, LocationSpark samples the underlying dataset to learn the data distribution in space and populates a grid or a region Quadtree index. In addition, each data partition has a *local index* (e.g., a grid local index, an R-tree, a variant of the Quadtree, or an IR-tree). LocationSpark adopts a new *Spatial Bitmap Filter* to reduce the communication cost when dispatching queries to their overlapping data partitions, termed *sFilter*. This structure is an in-memory variant of a Quadtree with the leaf nodes indicating whether a region contains data items. Therefore, this information can be used to speed up query processing by avoiding needless communication with data partitions that do not contribute to the query answer.
- *Query Executor.* In this layer, LocationSpark evaluates the runtime and memory usage trade-offs from various alternatives and chooses the best execution plan to run on each slave node. LocationSpark has a new layer, termed *Query Scheduler*, with an automatic skew analyzer and a plan optimizer to mitigate query skew.
- *Query Scheduler.* LocationSpark analyzes and mitigates skew queries with an automatic skew analyzer and a plan optimizer to be applied in the query execution

© 2021 García-García, F.J.

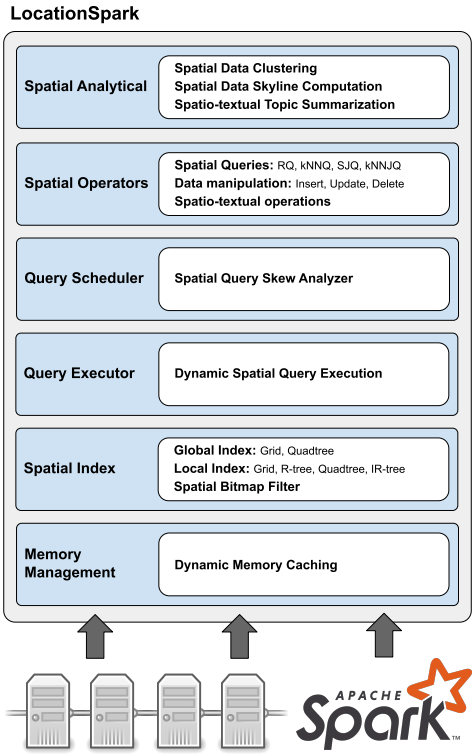


Figure 5.1: Architecture of LocationSpark by layers.

plan. It uses a cost model to analyze the skew to be used by the spatial operators and a plan generation algorithm to construct a load-balanced query execution plan. After the plan generation, local computation nodes select the proper algorithms to improve their local performance based on the available spatial indexes and the registered queries on each node.

– *Spatial Operators.* This layer provides support for spatial querying and spatial data updates. It provides a rich set of spatial queries, including spatial range query, kNNQ, spatial join, and kNNJQ. Moreover, it supports data updates and spatio-textual operations.

– *Spatial Analytical.* Finally, and due to the importance of spatial data analysis in this type of DSDMSs, LocationSpark provides spatial data analysis functions, including spatial data clustering, spatial data skyline computation, and spatio-textual topic summarization.

For processing spatial queries, LocationSpark builds a distributed spatial index structure for in-memory spatial data. As we can see in Figure 5.2, for spatial join queries,

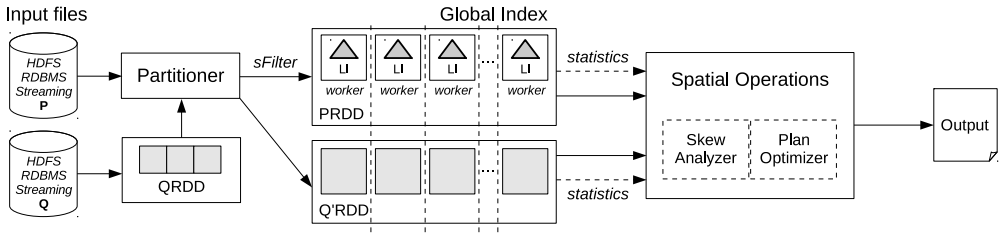


Figure 5.2: Spatial query processing in LocationSpark.

given two datasets $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P}$ is partitioned into N partitions based on a spatial index criteria (e.g., N leaves of a R-tree) by the *Partitioner* leading to the *PRDD* (Global Index). The *sFilter* determines whether a point is contained inside a spatial range or not. Next, each *worker* has a local data partition $\mathbb{P}_i$ $(1 \leq i \leq N)$ and builds a Local Index (**LI**). *QRDD* is generated from $\mathbb{Q}$ by a member function of RDD (*Resilient Distributed Dataset*) natively supported by Spark, that forwards such point to the partitions that spatially overlap it. Now, each point of $\mathbb{Q}$ is replicated to the partitions that are identified using the *PRDD* (Global Index), leading to the *Q'RDD*. Then a post-processing step (using the *Skew Analyzer* and the *Plan Optimizer*) is performed to combine the local results to generate the final output.

5.2 Spatial Queries supported by LocationSpark

LocationSpark supports four types of spatial query predicates [Tang et al., 2016, Tang et al., 2020]: spatial range search, kNN search (kNNQ), spatial range join and kNNJQ.

The *spatial range search* is a generic spatial query where one dataset and a spatial range area (e.g., rectangle or circle) are involved. First, the global index is used to find the overlapping partitions, and then, for each of them, a local spatial range search exploits the local index to speed up the query.

The *kNN search* in LocationSpark consists of three steps similar to the approach implemented in SpatialHadoop. Firstly, the partition where the query point belongs is located and, a kNNQ in that partition is calculated. Next, a range search is carried out on the overlapping partitions by the circle region centered at the query point with a radius the distance to the k-th nearest neighbor. Finally, the points of the query range are combined with the initial kNN result to obtain the final result of the query.

For the *spatial range join*, there are two algorithms in LocationSpark [Tang et al., 2020]. The first one is an *indexed nested-loops* join algorithm, where the spatial index from the largest dataset (points) is repeatedly traversed by a range query for each item from the smallest dataset (query points). Note that it is the naive version of the εDRJQ algorithm in LocationSpark [García-García et al., 2020c]. The second is a *block-based* algorithm using a parallel tree traversal, i.e., it builds two spatial indexes over both the input datasets and performs a depth-first search over both trees simultaneously.

© 2021 García-García, F.J.

This algorithm is the first approach of the εDJQ algorithm in LocationSpark [García-García et al., 2020c]. The query execution plan for spatial range join is shown in [Tang et al., 2020]. It should be noted that the execution plan of the spatial range join treats separately the partitions that can cause skew by repartitioning the latter to obtain better performance, and because of that, a merge step to unify the results is also needed.

Similar to the spatial range join, for the *kNNJQ* there are two available algorithms, that is, an *indexed nested-loops* and a *block-based* algorithm. For the first case, similar to the spatial range join, *indexed nested-loops* algorithm can be applied to kNNJQ, where it computes the set of kNNQ for each query point in the outer dataset. An index is built on the inner dataset. For the second case, the *block-based* algorithm partitions both datasets (query points and points) in two different set of partitions and find the kNN candidates for query points in the same partition. Then, the post-processing refine step computes kNNQ for each query point in the same partition. Notice that in [Pandey et al., 2018], a performance comparison between *Simba* [Xie et al., 2016] and *LocationSpark* determined that the *indexed nested-loops* version was the winner for total running time.

5.2.1 kNearest Neighbor Join Query

We know the kNearest Neighbor Join Query (kNNJQ) retrieves for each point of one dataset, k nearest points in the other dataset. That is, given two datasets of points ($\mathbb{P}$ and $\mathbb{Q}$ where $|\mathbb{P}| > |\mathbb{Q}|$) and a positive number k, the kNNJQ finds for each point of $\mathbb{Q}$, k nearest neighbors of this point in $\mathbb{P}$.

As shown in Section 4.3.4, the kNNJQ algorithm in SpatialHadoop, adapting the scheme presented in [Nodarakis et al., 2016a], have the following phases: (1) the *Repartitioning* phase, that redistributes the input datasets to deal with skew problems, (2) the *Bin kNNJ* phase, that founds the initial kNNJ answer, (3) the *kNNJ on Overlapping partitions* phase, that refines the previous results, and (4) *Merge results* phase, where the results from previous phases are combined.

The kNNJQ in LocationSpark consists of similar phases to the ones designed for SpatialHadoop but following the scheme from the spatial range join in LocationSpark. That is, when calculating the individual kNNQ, each element of the query dataset $\mathbb{Q}$ finds its individual kNNQ using the spatial index from the largest dataset $\mathbb{P}$. Figure 5.3 shows the *Spark DAG* (Directed Acyclic Graph) or *Execution Plan* of the *indexed nested-loops* kNNJQ in LocationSpark. In Stage 1, the dataset $\mathbb{P}$ and $\mathbb{Q}$ are partitioned, using

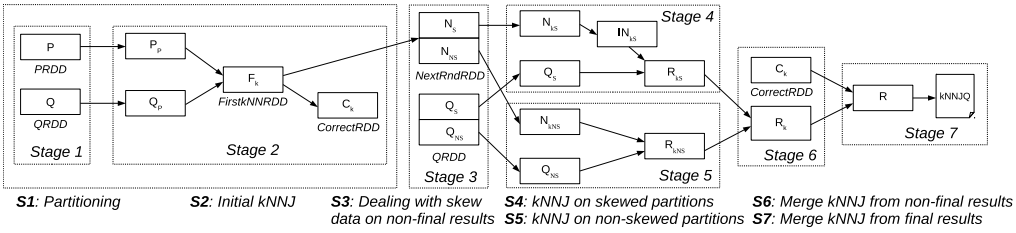


Figure 5.3: Execution Plan for kNNJQ in LocationSpark.

the same spatial partitioner, and the largest dataset $\mathbb{P}$ is indexed, according to a given spatial index method. In Stage 2, the initial kNN lists (*FirstkNNRDD*) are calculated, using a nested loop-based algorithm on Quadtrees, for each point in the partitions where it is located. Stage 3 collects those points where a final answer has not been obtained (*NextRndRDD*), and then a spatial range join is performed in Stages 4 and 5 using the distance to the k-th nearest neighbor. Note that these two last stages, as mentioned before, exist since spatial range join treats in parallel the partitions that present skew, and therefore are repartitioned, of those that do not. Finally, in Stage 7, the results of the correct initial kNNs lists (*CorrectRDD*) are combined with the distances to the points from the range join merged in previous Stage 6.

5.3 Enhancing LocationSpark with Distance-based Queries

As seen in the previous section, LocationSpark provides several spatial queries through the *Spatial Operators* layer. However, this DSDMS does not include some of the most studied DBQs (see Section 2.2). Consequently, this section shows several proposals for efficient in-memory distributed (Spark) algorithms to enrich the capabilities of the *Spatial Operators* layer.

5.3.1 kClosest Pairs Query

The kClosest Pairs Query (kCPQ) is known to find the k pairs of spatial objects from two datasets having the k smallest distances from all possible combinations. Therefore, when joining two datasets ($\mathbb{P}$ and $\mathbb{Q}$), the kCPQ gets the top-k pairs from $\mathbb{P} \times \mathbb{Q}$ with the *MinDistance* between their elements as the chosen spatial operator. Section 4.3.2 detailed the kCPQ in SpatialHadoop as a generic top-k MapReduce job that uses a selected *plane-sweep* kCPQ algorithm [Roumelis et al., 2016] as the spatial operator. Moreover, a modified algorithm with the computation of an upper-bound (β) (see Section 4.4.2) improved the performance significantly. Therefore, the final scheme of the kCPQ algorithm in SpatialHadoop consists of the following four steps: (1) β *computation* step, where an upper-bound of the kCPQ is found; (2) *Global kCPQ* step, that selects those pairs of partitions with a *MinDistance* between their MBRs less than β; (3) *Local kCPQ* step, that finds the kCPQ of a selected pair of partitions; and (4) *Top kCPQ* step, that obtains the final answer from the partial results.

Assuming that $\mathbb{P}$ is the largest dataset to be combined and $\mathbb{Q}$ is the smallest one, we could follow the previous scheme of the kCPQ in SpatialHadoop and the ideas presented in [Tang et al., 2020] to design and implement a kCPQ in-memory algorithm in LocationSpark. Consequently, we can describe its *Execution Plan* as follows. Stage 1 partitions the two input datasets according to a given spatial index schema. Next, Stage 2 adds statistic data to each partition, $S_{\mathbb{P}}$ and $S_{\mathbb{Q}}$, and they are combined by pairs, $S_{\mathbb{PQ}}$. In Stage 3, the partitions from $\mathbb{P}$ and $\mathbb{Q}$ with the largest density of spatial objects, $\mathbb{P}_\beta$ and $\mathbb{Q}_\beta$, are selected to be combined by using a plane-sweep kCPQ algorithm [Roumelis et al., 2016] to compute an upper bound of the distance value of the k-th closest pair

© 2021 García-García, F.J.

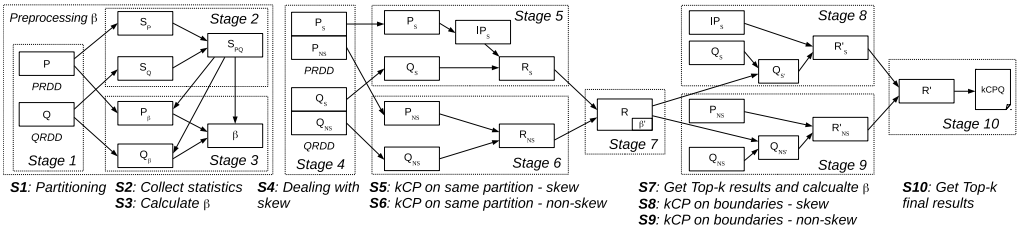


Figure 5.4: Execution Plan for kCPQ in LocationSpark.

(β). In Stage 4, the combination of all possible pairs of partitions from $\mathbb{P}$ and $\mathbb{Q}$, $S_{\mathbb{PQ}}$, is filtered according to the β value (i.e., only the pairs of partitions with minimum distance between the MBRs of the partitions is smaller than or equal to β are selected), giving rise to $FS_{\mathbb{PQ}}$, and all pairs of filtered partitions are processed by using a plane-sweep kCPQ algorithm. Finally, the results are merged to get the final output. With the previous *Execution Plan* and increasing the dataset sizes, the execution time increases considerably due to skew and shuffle problems. These problems arise mainly by the use of the *MapPartitions* transformation from Spark that creates a *wide dependency* and the redistribution of the data between workers. To solve it, we employ the same partitions for both datasets allowing the use of the *ZipPartitions* transformation, which creates a *narrow dependency* and reduces the data shuffling. Moreover, we modify Stage 4 with the query plan used for dealing with skewed data shown in [Tang et al., 2020], leaving the final plan as shown in Figure 5.4.

Stages 1, 2, and 3 are still present to calculate the β value that accelerates the local pruning phase on each partition. In Stage 4, using the *Query Plan Scheduler*, $\mathbb{P}$ is partitioned into $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{P}_{\mathbb{NS}}$ being the partitions that present and do not present skew, respectively. The same partitioning is applied to $\mathbb{Q}$. In Stage 5, a plane-sweep kCPQ algorithm [Roumelis et al., 2016] is applied between spatial objects of $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{Q}_{\mathbb{S}}$, that are in the same partition, and likewise for $\mathbb{P}_{\mathbb{NS}}$ and $\mathbb{Q}_{\mathbb{NS}}$ in Stage 6. These two stages are executed independently, and the results are combined in Stage 7. Finally, it is still necessary to calculate if there is any missing candidate for each partition found on the boundaries of that same partition in the other dataset. To do this, we use β', which is the maximum distance from the current set of candidates as a radius of a range filter with the center in each partition to obtain possible new candidates on those boundaries. The kCPQ calculation for each partition with its candidates is executed in Stages 8 and 9, and these results are combined in Stage 10 to obtain the final answer of the query. Notice that, with these changes, the execution plan of kCPQ is very similar to the one from kNNJQ, with the only difference that instead of maintaining different kNN lists for each spatial object, it only takes care of a single global kNN list.

5.3.2 εDistance Join Query

Previously, we have defined that the εDistance Join Query (εDJQ) joins two spatial datasets ($\mathbb{P}$ and $\mathbb{Q}$) by returning all possible pairs of spatial objects having a distance

smaller than a distance threshold ε. In addition, Section 4.3.3 details the εDJQ algorithm in SpatialHadoop that follows a two-step scheme: (1) *Global εDJQ* step, that prunes pairs of partitions having a *MinDistance* between their MBRs higher than ε, and (2) *Local εDJQ* step, that uses a plane-sweep algorithm to return the pairs of spatial objects of the εDJQ for each non-pruned pair of partitions.

The *Execution Plan* for εDJQ in LocationSpark is a variation of the kCPQ one, where the β and β' computation stages (Stages 2, 3 and 7) are removed, since $S_{\mathbb{PQ}}$ is now filtered by ε (i.e., $\beta = \beta' = \varepsilon$), which is the threshold distance known beforehand. First, in Stage 1, $\mathbb{P}$ is redistributed in $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{P}_{\mathbb{NS}}$ by applying the *Query Plan Scheduler* to query partitions that present and do not present skew separately. Like in kCPQ, this partitioner is applied to $\mathbb{Q}$. Next, Stages 2 and 3 execute a plane-sweep εDJQ algorithm [Roumelis et al., 2016] between spatial objects of the same partition in $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{Q}_{\mathbb{S}}$ for skewed data, and in $\mathbb{P}_{\mathbb{NS}}$ and $\mathbb{Q}_{\mathbb{NS}}$ for non-skewed data. Then, it is still needed to search for possible pairs of spatial objects on the boundaries of the partitions of one dataset in the other. Therefore, Stages 4 and 5 use a range, with the center in each partition and ε as the radius to obtain these candidates. To get the refined answer for each partition, a plane-sweep εDJQ algorithm is applied again. Finally, Stage 6 unifies the results from Stages 2, 3, 4, and 5 in the final query output.

As explained in Section 4.3.5, the εDJQ and εDRJQ are equivalent and report the same results, but in a different order. Besides, the main difference between both *MapReduce* algorithms in SpatialHadoop is that εDJQ is a *Map*-based Join algorithm derived

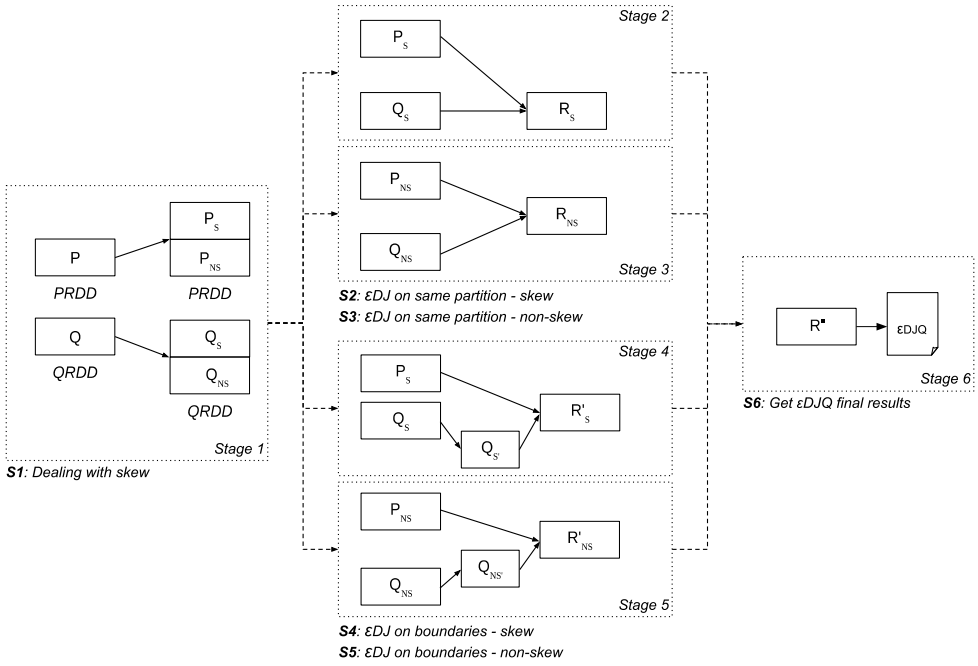


Figure 5.5: Execution Plan for εDJQ in LocationSpark.

© 2021 García-García, F.J.

from kCPQ, while εDRJQ is a *Reduce*-based Join algorithm that comes from kNNJQ. Therefore, the *Execution Plan* for εDRJQ in LocationSpark is a simplification of the kNNJQ one in which a pruning distance ε known beforehand is used. However, and as mentioned before, the algorithms kCPQ and kNNJQ follows a similar processing scheme in LocationSpark, and consequently, the *Execution Plans* of εDJQ and εDRJQ do not show differences since they are based on kCPQ and kNNJQ, respectively.

5.4 Performance Evaluation

This section details the results of an experimental comparison [García-García et al., 2020c] between LocationSpark and SpatialHadoop by measuring and evaluating the efficiency of the several DBQ algorithms presented in this chapter and Chapter 4. In particular, Subsection 5.4.1 describes the experimental settings. Subsection 5.4.2 shows all experiments comparing kCPQ and εDJQ, taking into account several performance parameters. Subsection 5.4.3 shows all experiments for kNNJQ, paying special attention to how the *Quadtree*-based repartitioning technique in SpatialHadoop compares to the in-memory processing of LocationSpark. Subsection 5.4.4 compares εDRJQ in both DSDMSs, regarding scalability of the datasets to be combined and ε values. Subsection 5.4.5 shows the speedup of the proposed distributed DJQ algorithms in both DSDMSs, varying the number of computing nodes in the cluster. Finally, in Subsection 5.4.6 a summary of the most relevant conclusions from the experimental results is reported.

5.4.1 Experimental Setup

For the experimental comparison, we have used the implementations of SpatialHadoop[1] and LocationSpark[2], with the addition of our open-source DJQ algorithms, that can be downloaded from our SpatialHadoop[3] and LocationSpark[4] forks. Moreover, we have used real-world 2d points and geometric datasets to compare and analyze our distributed DJQ algorithms in SpatialHadoop and LocationSpark. Therefore, we have used the same datasets from OpenStreetMap already used in Section 3.6 and 4.5: *LAKES* (L), *PARKS* (P), *ROADS* (R), *BUILDINGS* (B) and *ROAD_NETWORKS* (RN). On the one hand, we have used the datasets as-is for spatial objects experiments (i.e., polygons and line-strings). On the other hand, for 2d points experiments, we have considered the *center* of each MBR and the *centroid* of each polygon as in previous chapters.

To accomplish a fair comparison between both DSDMSs, we must take into account the most suitable partitioning technique to execute the DJQs. In [Pandey et al., 2018], the *Quadtree* is the spatial partitioning technique selected to evaluate LocationSpark and compare it with other Spark-based DSDMSs. Due to the excellent performance results obtained for this DSDMS in the evaluation of DJQs, we choose *Quadtree* partitioning technique for LocatioSpark to use in the experiments of this section. Therefore, we will

[1]Available at `https://github.com/aseldawy/spatialhadoop2`
[2]Available at `https://github.com/merlintang/SpatialSpark`
[3]Available at `https://github.com/acgtic211/spatialhadoop2/tree/DJQ`
[4]Available at `https://github.com/acgtic211/SpatialSpark/tree/DJQ`

also select *Quadtree* partitioning method for SpatialHadoop, as it has also obtained great performance results, especially for kCPQ. Another interesting variation is that we need to partition and index the datasets before doing any query processing in SpatialHadoop. This *preprocessing* phase is carried out once per dataset and stored on HDFS for the subsequent execution of several spatial queries (this can be considered an advantage of SpatialHadoop). For instance, the times needed for the *Quadtree* partitioning technique are 94 sec for *LAKES*, 103 sec for *PARKS*, 150 sec for *ROADS*, 175 sec for *BUILDINGS* and 1053 sec for *ROAD_NETWORKS*. For LocationSpark (in-memory-based DSDMS), the partitions and indexes are generated for every spatial query and only cached in memory for the current operation. Therefore, the partitions/indexes are not stored on any persistent file system and cannot be reused in subsequent spatial operations.

The performance measure that we have adopted in our experiments was the total *execution time* (i.e., running time or response time); this measurement is reported in seconds (sec) and represents the time spent by the execution of each distributed DJQ algorithm in both DSDMSs.

Moreover, all experiments were conducted on the same 12 nodes OpenStack cluster outlined in Section 3.6. Additionally, for LocationSpark, we have used Spark 1.6.2 in cluster mode over YARN in the aforementioned cluster.

Finally, Table 5.1 summarizes the configuration parameters used in our experimental comparison. Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	**Values (default)**
# of nearest neighbors for kCPQ, k	1, 10, (10^2), 10^3, 10^4, 10^5
# of nearest neighbors for kNNJQ, k	1, (10), 25, 50, 75, 100
Distance threshold, ε ($\times 10^{-5}$)	7.5, 10, 25, 50, 75, (100)
Number of nodes, η	1, 2, 4, 6, 8, 10, (12)
Partitioning technique	Quadtree
Repartitioning technique	Quadtree
DSDMS	SpatialHadoop, LocationSpark

Table 5.1: Configuration parameters used in our experiments to compare SpatialHadoop and LocationSpark.

5.4.2 kCPQ and εDJQ experiments

Our first set of experiments aims to measure the behavior of the kCPQ and εDJQ algorithms on both DSDMSs (SpatialHadoop and LocationSpark) varying different parameters, as the dataset sizes, the type of spatial objects, and the values of k and ε.

In Figure 5.6, the chart on the left compares the $kCP(\mathbb{P}, \mathbb{Q}, k)$ for point datasets (where $\mathbb{P} \times \mathbb{Q} \equiv LAKES \times PARKS$ ($L \times P$), $PARKS \times ROADS$ ($P \times P$), $ROADS \times BUILDINGS$ ($R \times B$) and $BUILDINGS \times ROAD\_NETWORKS$ ($B \times RN$)) concerning the execution time, for a fixed $k = 100$. The main outcome of this experiment is that the larger the dataset, the higher the execution time for both DSDMSs. For the

© 2021 García-García, F.J.

combinations of $L \times P$ and $P \times R$, LocationSpark is faster than SpatialHadoop (e.g., for $P \times R$ LocationSpark is 48% (74 sec) faster), but the combinations of the biggest datasets ($R \times B$ and $B \times RN$) SpatialHadoop is the fastest, e.g., for $B \times RN$ SpatialHadoop is 38% (740 sec) faster than LocationSpark. That is, LocationSpark exhibits smaller runtime values for small-medium dataset sizes, even though neither pre-partitioning nor pre-indexing is done. However, SpatialHadoop is the fastest for the largest datasets, although it needs a pre-indexing time for each one, and that difference can be caused by memory constraints on the cluster. By increasing the size of the joined datasets, there is more data in each partition, and the *memory pressure* of the tasks increases. Therefore, we can conclude that LocationSpark is more affected by memory constraints than SpatialHadoop for the same cluster.

For real spatial object datasets ($L \times P$: *polygons* $\times$ *polygons*, $P \times R$: *polygons* $\times$ *line-strings*, $R \times B$: *line-strings* $\times$ *polygons* and $B \times RN$: *polygons* $\times$ *line-strings*), the right chart of Figure 5.6 shows the execution time for the kCPQ. The trend is similar than for points (left chart), where LocationSpark is the fastest for the combination of small-medium dataset sizes, although SpatialHadoop shows higher performance for the biggest dataset combinations. Moreover, if we compare both charts in Figure 5.6, we can see that when we execute a kCPQ between two datasets of spatial objects, the execution time increases for both DSDMSs. Remember that the running time was already slightly higher for spatial objects than for points when studying the kCPQ in SpatialHadoop (see Section 4.5.3.5). The reasons are that the distance between two spatial objects is computationally more expensive and that distance value is smaller than the distance between two points, so the effect of pruning is reduced and more pairs have to be considered. In this experiment, we can see that LocationSpark behaves similar to SpatialHadoop but with lower relative increase of the execution time (e.g., for $B \times RN$ the relative increment for LocationSpark is 17% (345 sec) while for SpatialHadoop is 24% (294 sec)).

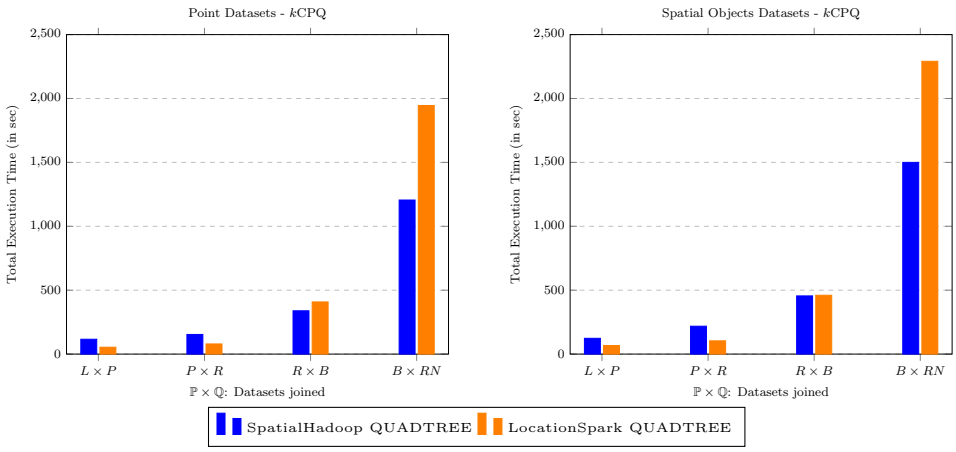


Figure 5.6: kCPQ cost, total execution time joining points (left) and non-points spatial objects (right).

Figure 5.7 shows the results for $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ execution times with the same configuration as that of Figure 5.6, for a fixed $\varepsilon = 0.001$ (100×10^{-5}). In the left chart, we can see similar results to the previous kCPQ experiment. For instance, SpatialHadoop exhibits the best performance for point datasets, being LocationSpark faster only for the smallest dataset combination $(L \times P)$. Especially for the combinations of the biggest datasets $(R \times B$ and $B \times RN)$, SpatialHadoop is the clear winner, e.g., for $B \times RN$ is 2.8 times (1938 sec) faster than LocationSpark. On the one hand, problems of *memory pressure* in LocationSpark appear again due to the dataset sizes, the number of elements computed in each partition, and the shuffling data and garbage collection processes performed on them. On the other hand, SpatialHadoop has better performance against this problem due to the use of *CombineFileSplits* [Karanth, 2014], which allows performing the join by partitioning at disk reading level and therefore, eliminating the *Reduce* shuffling cost.

In Figure 5.7, the right chart exposes the results of εDJQ for real spatial object datasets with respect to the execution time ($\varepsilon = 0.001$). The behavior is again comparable to the point datasets, with SpatialHadoop being the fastest for all combinations, except for the smallest one $(L \times P)$. Note that for the combination of the largest datasets $(B \times RN)$, LocationSpark shows *memory pressure* problems again. Another conclusion is that a εDJQ between two spatial objects datasets expends more execution time than for points for both DSDMSs. We have already seen in Section 4.5.4.4 that this behavior when joining spatial objects is similar to kCPQ. Therefore, because the distances between spatial objects are smaller, the same ε value returns more results, increasing the execution time. When comparing both DSDMSs, we can see that for εDJQ, the relative increase in execution time is similar for LocationSpark and SpatialHadoop (e.g., for $R \times B$, this relative increment is around 100% for both).

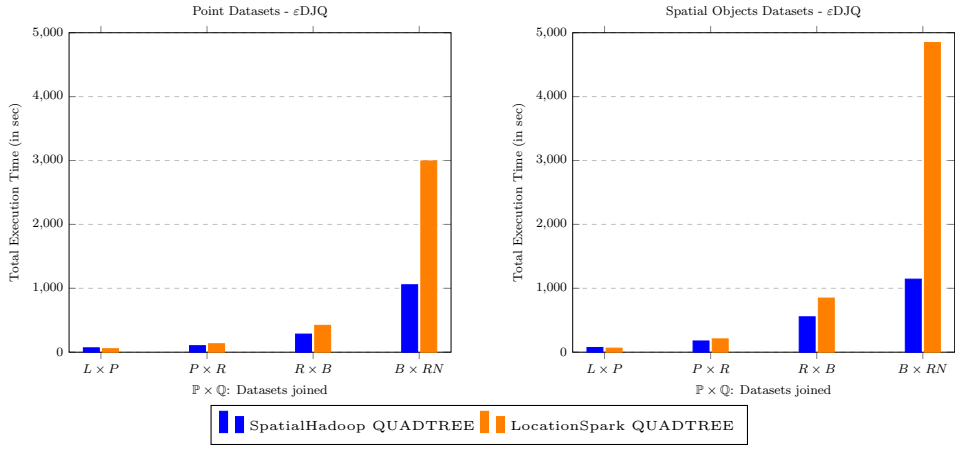


Figure 5.7: εDJQ cost, total execution time joining points (left) and non-points spatial objects (right).

© 2021 García-García, F.J.

Figure 5.8 shows the effect of increasing both k and ε values for the combination of the biggest datasets ($BUILDINGS \times ROAD\_NETWORKS$) for kCPQ and εDJQ. In Figure 5.8, the left chart shows that as the number of results to be obtained (k) increases, the total execution time grows slowly. The first conclusion is that SpatialHadoop shows the best performance when joining either points or spatial objects (*polygons* $\times$ *line-strings*), and even for large k values (e.g., $k = 10^5$). This behavior is thanks to the combination of Quadtree-based partitioning performed in the *preprocessing* step and the *filtering* function that reduces the number of candidates even before reading the data from the HDFS. LocationSpark, that also uses the *Quadtree*, is stable when k is small or medium ($k \leq 10^3$); however, for higher k values ($k = 10^4$ and $k = 10^5$), the execution time is very high due to memory constraints in the cluster. With the increment of k, the possibility of selecting more cells augment since the value of the distance of the k-th closest pair increases as well. Due to this, the resources needed by the kCPQ algorithm increment. Finally, note that the execution times of the algorithms show divergence with the highest values of k (see Figure 5.8), especially for LocationSpark.

As shown in the right chart of Figure 5.8, the total execution time of εDJQ grows as the ε value is increased. Both DSDMSs have similar relative performance for all ε values, SpatialHadoop being faster in all cases, even when the datasets of spatial objects are combined. This difference is due to the way in which εDJQ is calculated in SpatialHadoop and its the *preprocessing* step that reduces time considerably even for the largest datasets. For higher ε values, both systems start to increase their execution times due to the increase of the number of elements that are part of the query results. A special case is for LocationSpark when $\varepsilon = 10^{-3}$, the execution time is very high because it has *memory pressure* problems again, more prominent than SpatialHadoop.

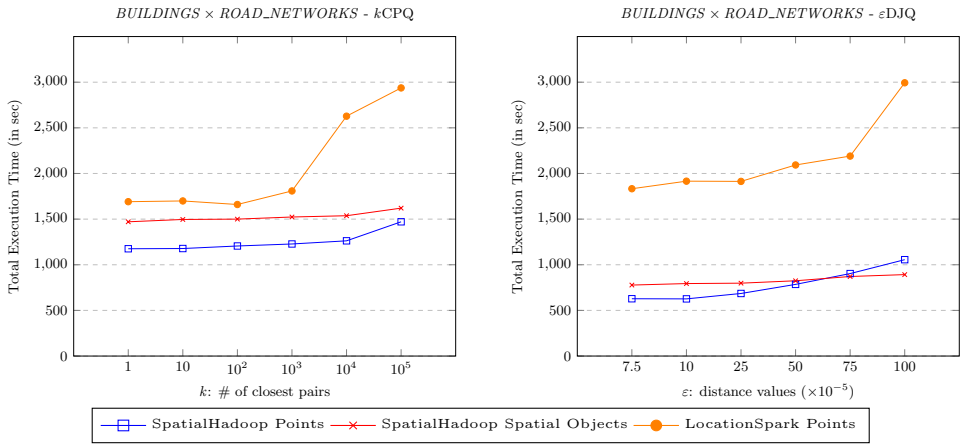


Figure 5.8: kCPQ cost, total execution time vs. k values (left). εDJQ cost, total execution time vs. ε values (right).

The main conclusions that we can draw from this set of experiments (kCPQ and εDJQ) are:

1. The higher k or ε values, the greater the possibility that pairs of candidates are not pruned, more tasks are needed, and more total execution time is consumed.

2. SpatialHadoop shows better performance, especially for higher values of k and ε, thanks to *Quadtree* partitioning technique and the reduction of the number of candidates by using the *preprocessing* step.

3. The trend of SpatialHadoop is quite stable for the execution time, even if the biggest datasets of spatial objects are combined, where the cost is more expensive than for points.

4. SpatialHadoop εDJQ shows the lowest and most stable execution times, demonstrating the benefits of its *Map*-based join implementation and the use of *CombineFileSplits* [Karanth, 2014].

5. LocationSpark is faster than SpatialHadoop for small and medium dataset sizes, but for the largest datasets, it needs more time to execute the kCPQ and εDJQ, even for small k and ε values; this is due to *memory pressure* problems resulting from the increase of the number of processed elements and the size of consumed memory, as well as the increase in the processing time needed for the shuffling data and garbage collection.

5.4.3 kNNJQ experiments

Similarly to the previous experiments, the following evaluation of the kNNJQ algorithms in LocationSpark and SpatialHadoop aims to measure the variation of different parameters as the dataset sizes to be joined (scalability) and k values. For LocationSpark, we have chosen the *Quadtree* as it got the best performance for kNNJQ in [Pandey et al., 2018]. For SpatialHadoop, we are using the *Quadtree* for both the *preprocessing* step and *repartitioning* phase to do a fair comparison with LocationSpark. Moreover, we have fixed the parameters $L = 100000$ (maximum number of elements in each partition) and $r = 2\%$ (sample ratio) for this *repartitioning* technique as they were the best setup for kNNJQ in SpatialHadoop (see Section 4.5.5).

In Figure 5.9, the left chart shows the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query for both DSDMSs concerning the execution time and a fixed $k = 10$, where the smallest dataset $\mathbb{P}$ is fixed to $LAKES$ and the other dataset $\mathbb{Q}$ is assigned to particular datasets ($PARKS$, $ROADS$, $BUILDINGS$ and $ROAD\_NETWORKS$), creating the following combinations of $\mathbb{P} \times \mathbb{Q}$: $L \times P$, $L \times R$, $L \times B$ and $L \times RN$. The first conclusion that we can extract is that SpatialHadoop is the fastest, with a difference of around 40%. This greater performance from SpatialHadoop with *Quadtree*-based repartitioning technique than LocationSpark is due to the excellent results when dealing with skewed data. While this technique in SpatialHadoop processes those partitions that exceed a certain number of elements, LocationSpark, in the current implementation, treats only the number of cells with the highest number of elements based on the input datasets. Note that for the same execution conditions in our cluster, LocationSpark could not execute the kNNJQ for the combination of $L \times RN$ because it consumes all available resources on

© 2021 García-García, F.J.

some workers, having to abort the execution. However, both DSDMSs exhibit quite stable execution times, and their increment is sub-linear as the size of the datasets ($\mathbb{Q}$) is augmented. Note that for SpatialHadoop, the relative increment of execution time from $L \times P$ to $L \times RN$ was around the 53.4% (1491 sec), while the increment in the number of points was massive from $PARKS$ (10M) to $ROAD\_NETWORKS$ (717M).

The right chart of Figure 5.9 shows the effect of varying the value of k for the combination of the datasets ($LAKES \times PARKS$). Comparing the DSDMSs, we can observe that SpatialHadoop is faster than LocationSpark, except for $k = 50$, where LocationSpark spends less execution time. At first sight, LocationSpark seems to scale better when increasing the value of k, but from $k = 75$, problems start to appear due to memory limitations in the cluster because of the increase in the number of elements and partitions. Consequently, it is demonstrated again that LocationSpark is more sensitive to this type of problem than SpatialHadoop since it is a memory-based DSDMS. Remember that for a kNNJQ, the higher the k value, the higher the possibility of selecting more partitions to refine the initial kNN lists (*kNNJ on Overlapping Partitions* for SpatialHadoop, and Stages 4 and 5 for LocationSpark). Therefore, the same point must be compared in more than one partition, increasing the size of shuffled data and having partial kNN lists for each partition that must be combined in the last phase of the algorithm.

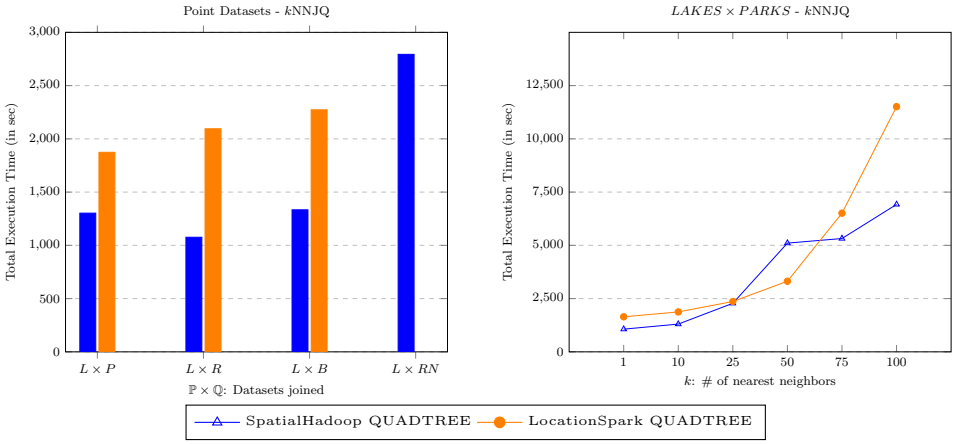


Figure 5.9: kNNJQ cost, total execution time considering different datasets (left) and varying the k values (right).

From this set of experiments, comparing the kNNJQ algorithm in SpatialHadoop and LocationSpark, we can conclude with the following:

1. SpatialHadoop is the fastest, especially for lower values of k, thanks to using the *Quadtree* for both the *preprocessing* step and *repartitioning* phase that greatly reduces the number of candidates.

2. LocationSpark presents low and stable execution times with small and medium

dataset sizes, and for small k values, but it shows poor results for larger datasets and k values since it is more sensitive to memory constraints problems.

3. Similarly to kCPQ, the higher the k value, the greater the possibility of more unpruned pairs of candidates, more needed tasks, and higher total execution time.

5.4.4 εDRJQ experiments

The next experiment compares the εDRJQ, in terms of execution time, in both SpatialHadoop and LocationSpark. The left chart of Figure 5.10 shows the results for εDRJQ execution times with several point dataset combinations ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) and $\varepsilon = 0.001$. Firstly, for the combinations of the smallest datasets ($L \times P$ and $P \times R$), SpatialHadoop exhibits slightly larger execution times. These execution times stem from being a *Reduce*-based join algorithm since there is extra time spent when performing the shuffling and sorting of the data between the *Map* and *Reduce* phases. However, for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) SpatialHadoop is faster than LocationSpark (e.g., for $B \times RN$ is 40% faster). Therefore, the problems of *memory pressure* in LocationSpark seem to influence more in the execution time than those caused by the shuffled data in SpatialHadoop. As shown in the right chart of Figure 5.10, the total execution time grows as the ε value increases. Both DSDMSs have similar relative performance for all ε values, with SpatialHadoop being faster in all cases.

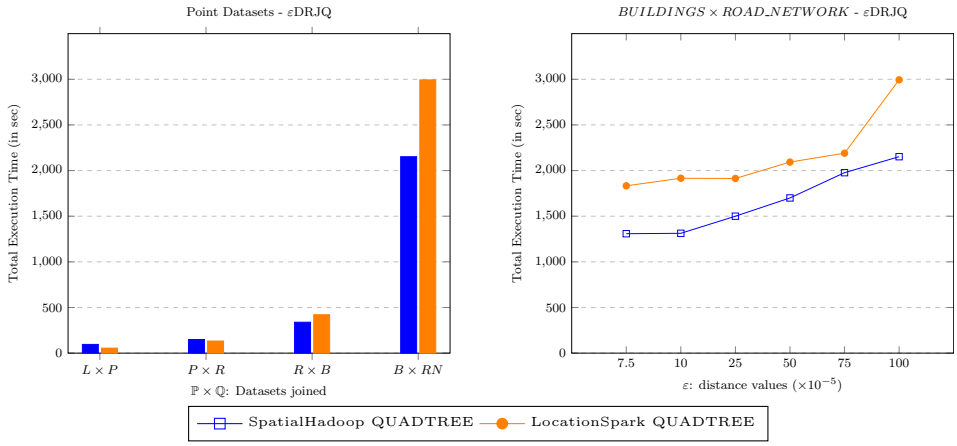


Figure 5.10: εDRJQ cost, total execution time considering different datasets (left) and varying the ε values (right).

The main conclusions extracted for this performance comparison are:

1. εDRJQ in SpatialHadoop is faster than LocationSpark for large datasets and any ε value.

© 2021 García-García, F.J.

2. LocationSpark is the fastest for small-medium datasets ($L \times P$ and $P \times R$), but when the dataset sizes grow, its performance degrades in terms of running time because of *memory pressure* problems.

5.4.5 Speedup varying the number of computing nodes

Our last experiment aims to measure and compare the speedup of all the DJQ MapReduce algorithms (kCPQ, εDJQ, kNNJQ, and εDRJQ) in both DSDMSs, for the number of computing nodes (η). The first chart of Figure 5.11 shows the impact of varying the number of computing nodes on the performance of distributed kCPQ algorithm, for $BUILDINGS \times PARKS$ with the default configuration values. From this chart, it could be concluded that the performance of our approach has a direct relationship with the number of computing nodes. It could also be deduced that better performance would be obtained if more computing nodes are added, but when the number of computing nodes exceeds the number of *map* tasks, no improvement is obtained. LocationSpark is still showing better behavior than SpatialHadoop for kCPQ.

In the second chart of Figure 5.11, we observe a similar trend for εDJQ MapReduce algorithm in SpatialHadoop, with less execution time when the number of available nodes is less than 8; However, in this case, LocationSpark shows the worse performance for a smaller number of nodes. The main reason is that LocationSpark and especially εDJQ depends tightly on the available memory. Thus, when the number of nodes decreases, this memory also decreases considerably.

The third chart of Figure 5.11 shows much higher execution times for kNNJQ than for previous DJQ algorithms, mainly since it is a much more complex algorithm and consists of several phases. However, both systems follow a similar behavior tendency to the one shown in kCPQ, exhibiting the lowest execution times in SpatialHadoop for kNNJQ.

Finally, the last chart of Figure 5.11 shows the execution times for εDRJQ, and as it happens between kCPQ and εDJQ, this algorithm has lower times than kNNJQ, which it is based on that. Furthermore, as seen for εDRJQ, SpatialHadoop shows a better behavior than LocationSpark when there are fewer computing nodes in use due to the sensitivity of the latter to memory constraints as the availability of this resource reduces by decreasing the number of nodes.

The main conclusions that we can extract for these experiments, varying the number of computing nodes (η), are:

1. All algorithms behave better when the number of computing nodes increases, however when not enough tasks available, no performance improvement can be obtained.

2. For LocationSpark, the value of the number of nodes is not as a determinant parameter for the speedup of the algorithms, as is the availability of memory resources.

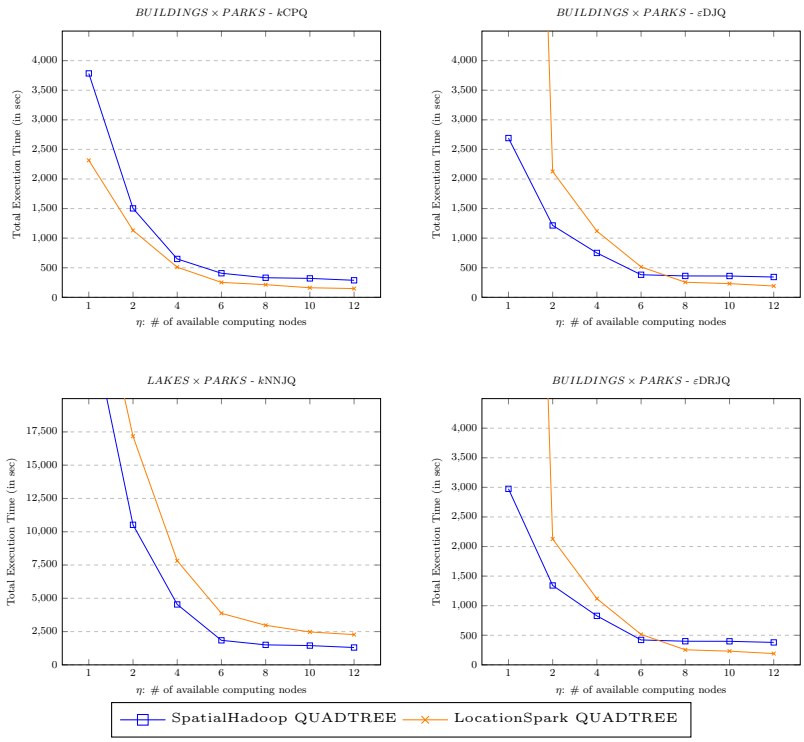


Figure 5.11: kCPQ, εDJQ, kNNJQ and εDRJQ cost with respect to the number of computing nodes η (Speedup).

5.4.6 Conclusions of the results

By analyzing all the previous experimental results, we can extract several important conclusions that are summarized below:

1. We have experimentally compared the *efficiency* (in terms of total execution time) and the *scalability* (in terms of k and ε values, sizes of datasets, and the number of computing nodes, η) of the proposed distributed algorithms for DJQs in SpatialHadoop and LocationSpark.

2. For kCPQ and εDJQ, the larger the k or ε values, the larger the probability of unpruned pairs of candidates, more needed tasks, and higher consumed total execution time for reporting the final result. On the one hand, SpatialHadoop shows excellent performance for large k and ε values thanks to the use of the *Quadtree* partitioning technique. On the other hand, LocationSpark is faster than SpatialHadoop for small and medium dataset sizes, but for the largest datasets, it needs more time to execute the queries, even for large k and ε values due to *memory pressure* problems. When real spatial objects are combined, the running time for both DSDMSs is a bit higher than for points, following a similar trend.

© 2021 García-García, F.J.

3. We have compared the built-in LocationSpark kNNJQ algorithm with our proposed MapReduce algorithm in SpatialHadoop (see Section 4.3.4) that we have improved by using an initial phase for repartitioning the dense partitions (see Section 4.4.3). Due to this, SpatialHadoop is the fastest, especially for lower k values, thanks to *Quadtree*-based repartitioning technique and the reduction of the number of candidates by using the *Repartitioning* phase. LocationSpark, using the currently available implementation, presents low and stable execution times with small and medium dataset sizes and for small k values. However, it shows poor results for larger datasets and k values due to being more sensitive to memory constraints problems.

4. For εDRJQ, SpatialHadoop is the fastest for the largest datasets and any ε value, except for small-medium dataset sizes where LocationSpark exhibits the best performance.

5. The larger the number of computing nodes (η), the faster the DJQ MapReduce algorithms.

6. The use of *CombineFileSplits* [Karanth, 2014] in SpatialHadoop [Eldawy and Mokbel, 2015] allows reducing the execution times considerably by avoiding the cost of shuffling and sorting of the *reduce* phase. Thus, it would be interesting to study its use to improve other algorithms such as kNNJQ, in which the size of shuffling data is a determinant factor.

7. LocationSpark is very sensitive to problems caused by memory restrictions, making it has worse performance than SpatialHadoop for the same cluster for heavy-sized datasets or high k and ε values.

8. Finally, as a general conclusion, both DSDMSs have similar performance trends in terms of execution time. However, LocationSpark shows better performance values when medium datasets are combined (if providing a suitable number of computing nodes with adequate memory resources), even though neither pre-partitioning nor pre-indexing is done. Therefore, LocationSpark, which is a recent DSDMS, needs further improvements, like the treatment of skewed data. On the other hand, SpatialHadoop is a more robust and mature DSDMS, since it has received numerous enhancements over the years (e.g., this thesis includes several improvements for kNNJQ in Section 4.4.3), so it exhibits better performance for the studied DJQ MapReduce algorithms when the sizes of the datasets increase.

5.5 Conclusions

This section highlights the main conclusions of this chapter. First, we have detailed the general spatial query processing scheme in LocationSpark and its numerous features and tools for handling spatial data in Spark. Especially, the *Spatial Index* and the *Query Scheduler* layers make the difference from other Spark-based spatial analytics systems, thanks to the analysis and mitigation of skew data in spatial queries. Next, we have

described the available spatial queries (i.e., spatial range search, kNN search (kNNQ), spatial range join, and kNNJQ) in the *Spatial Operators* layer of LocationSpark. Moreover, we have focused and explained the *Execution Plan* of the *indexed nested-loops* kNNJQ in LocationSpark to compare it with the one proposed for SpatialHadoop. Then we have exposed new spatial queries (i.e., kCPQ, εDJQ, and εDRJQ) for LocationSpark and some improvements over their original *Execution Plans*. Finally, the execution of an extensive set of experiments has compared these new DJQ distributed algorithms in LocationSpark with our proposed MapReduce DJQ algorithms in SpatialHadoop from Chapter 4. Besides, they have been tested in terms of efficiency, robustness, and scalability for parameters such as dataset sizes, k, ε, number of computing nodes (η), and others. On the one hand, LocationSpark is the clear winner for the execution time when up to medium-sized datasets are joined due to the efficiency of the in-memory processing provided by Spark and additional improvements (e.g., *Query Scheduler* layer). On the other hand, SpatialHadoop is faster when joining heavy-sized real-world datasets because it is a more mature and robust DSDMS due to the time invested in research and development (e.g., it provides more spatial partitioning techniques, computational geometry algorithms, repartitioning techniques for skewed data, etc.).

© 2021 García-García, F.J.

Chapter 6

Conclusions and Future Work

© 2021 García-García, F.J.

Chapter 6

Conclusions and Future Work

Contents

In this chapter, we conclude this dissertation. First, in Section 6.1, we provide the most important conclusions and results arising from the research in this thesis. Finally, in Section 6.2, we discuss related future work on open research lines.

6.1 Conclusions

Nowadays, a significant portion of Big Data is spatial data, and the size of such data is growing rapidly, at least by 20% every year. *Big Spatial Data* refers to spatial datasets exceeding the capacity of standard computing systems, and it also describes the process of capturing, storing, managing, analyzing, and visualizing huge amounts of spatial data, not using traditional tools and systems. *Distributed computing* has established itself as a successful paradigm to solve a big problem by dividing it into multiple tasks, where each of them is calculated on individual computers in a distributed system. Recent big spatial data developments have motivated the emergence of new technologies for distributed processing of large-scale spatial datasets in shared-nothing clusters of computers, leading to *Distributed Spatial Data Management Systems* (DSDMSs). These DSDMSs can be classified as disk-based or in-memory-based. The disk-based DSDMSs are characterized as Hadoop-based systems, and the most representative is *SpatialHadoop*. On the other hand, the in-memory-based DSDMSs are characterized as Spark-based systems, and one of the most complete and recognized is *LocationSpark*.

The most important architectural features of these DSDMSs are related to data partitioning methods, indexing schemes, and spatial queries that they support. Accordingly, an interesting challenge is to enrich these DSDMSs on these architectural functionalities to make them more competitive and complete. The overarching goal of this thesis is to research in detail existing DSDMSs (SpatialHadoop and LocationSpark), enhancing them by including new spatial data partitioning techniques, new local spatial indexing methods, and new DBQ algorithms for processing large real-world spatial datasets. In particular, we aim at making such DSDMSs richer by implementing Voronoi-Diagram based partitioning technique in SpatialHadoop, Quadtree as a local index in SpatialHadoop and new distance-based queries (εDRQ, kNNQ, kCPQ, kNNJQ, RkNNQ, εDJQ, εDRJQ, etc.) in SpatialHadoop and LocationSpark. Moreover, an extensive performance evaluation of multiple enhancements (spatial query algorithms, extensions, and improvements) in both DSDMSs is achieved. Finally, a comparative study between SpatialHadoop and LocationSpark is also carried out by executing an exhaustive set of experiments of several DBQs to identify which DSDMSs is the most appropriate for the distributed query processing of large volumes of spatial data.

First, we have developed a survey of the state-of-the-art Hadoop-based and Spark-based DSDMSs. For this aim, we have examined the most representative research prototypes of DSDMSs that appear in the literature and compare them based on structural features like spatial index and spatial queries they support. Moreover, we have also reviewed the most common spatial data partitioning techniques and classified them by how they use data or space properties to split the spatial datasets. Finally, an overview

© 2021 García-García, F.J.

of the most studied and known distance-based queries in the spatial context is thoroughly analyzed. In particular, we emphasize this study on distance-based join queries (kCPQ, kNNJQ, εDJQ, εDRJQ, and others), where two datasets are combined. We have presented all these surveys in Chapter 2.

The next objective of this dissertation was to propose a new spatial data partitioning technique based on Voronoi-Diagrams for SpatialHadoop. This data partitioning scheme is especially suitable for DBQs like kNNQ and kNNJQ since it is a distance-based partitioning method to split the spatial dataset into smaller units, enabling the processing of DBQ in parallel and reducing the I/O activity by only visiting the partitions that contain the relevant data to the query constraint. An extensive experimental evaluation of the spatial partitioning methods that are already implemented in SpatialHadoop and a comparison with the Voronoi-Diagram based technique for kCPQ and kNNJQ is also accomplished. We have also included the Quadtree as a local index in SpatialHadoop since this spatial access method is widely used in commercial spatial database systems. A comparative study between R-tree and Quadtree as local indexes for kNNQ and kCPQ has been carried out and has demonstrated the excellent performance of the Quadtree for these top-k queries in SpatialHadoop. All this material has been proposed in Chapter 3, along with interesting experimental conclusions as shown below:

1. To implement the Voronoi-Diagram based partitioning technique in SpatialHadoop, the best sampling technique to find a small but representative profile of the large spatial dataset for DJQ processing is k-means++ (partition-based sampling method).

2. The use of k-means++, as a clustering method for pivot selection, obtained the best results for the generation of the partitions based on Voronoi-Diagrams for the kNNJQ MapReduce algorithm in SpatialHadoop. V_{kkI} represents the use of k-means++ both in the sampling technique and the pivot selection in the Voronoi-Diagram based partitioning technique in SpatialHadoop, with additional improvements using new *pruning rules* relied on Voronoi-Diagram based distances and *less data* technique.

3. For kNNJQ (it follows a multiple nearest neighbor queries processing schema), V_{kkI} is faster than Quadtree partitioning because it deals better with skewed data and it gets more results earlier. For kCPQ (it follows a global query processing schema), Quadtree outperforms V_{kkI} because Quadtree generates less *map* tasks when large spatial datasets are combined.

4. For kCPQ, Quadtree outperforms R-tree, as a local index, due to the morphology of the nodes and the number of generated partitions. The nodes of the Quadtree show a more regular shape that causes a smaller number of overlaps between nodes, which implies a reduction in the number of pairs to process.

5. For kNNQ and kCPQ, when the value of k or the size of the datasets varies, Quadtree, as a local index, is the clear winner, and its execution times are more stable than those of R-tree when dealing with higher values of k or larger spatial datasets. Moreover, both R-tree and Quadtree show better performance when the

number of computing nodes (η) is increased. But for kCPQ, if there are not enough tasks available for a specific number of nodes, no performance improvements are obtained. Besides, Quadtree takes much less time than R-tree when there is only one computing node available.

Related to the enhancement of SpatialHadoop with new spatial queries using the MapReduce programming model, we have proposed the design and implementation of new distance-based queries (εDRQ, kNNQ, kCPQ, kNNJQ, εDJQ, εDRJQ, RkNNQ, etc.). In the first stage, we have programmed MapReduce algorithms for εDRQ and kNNQ, based on range and k nearest neighbor queries included in SpatialHadoop. Next, we have designed and implemented new MapReduce algorithms to perform DJQs like kCPQ and εDJQ efficiently. For this aim, we have utilized plane-sweep-based kCPQ algorithms and improved them to compute an upper bound of the distance of the k-th closest pair (β) and make the original version of the kCPQ MapReduce algorithm more efficient and faster. We have also implemented a new version of the distributed kCPQ algorithm using R-trees as a local index(es) provided by SpatialHadoop, and we compared this approach to the plane-sweep-based version (without indexes). Finally, we have evaluated the performance of the proposed distributed algorithms in several situations with large real-world as well as synthetic datasets. The experimental results have demonstrated the efficiency and scalability of our proposals. Part of Chapter 4 shows all these DJQ MapReduce algorithms together with interesting experimental conclusions listened below:

1. The use of local sampling to compute β makes the kCPQ MapReduce algorithm faster than the other two alternatives (global sampling and approximate technique) because the time required to perform the local sampling is very small. The use of β improves the execution time of the individual *map* tasks. The use of local sampling generates low β values, and the power of pruning increases when it is passed to the *map* tasks.

2. The improved version of the plane-sweep-based algorithm (Reverse Run) in the MapReduce implementation of kCPQ in SpatialHadoop obtains the best performances in terms of execution time. Moreover, it is faster than using local indices (R-trees) in each *map* task.

3. For kCPQ and εDJQ, the Quadtree spatial partitioning technique, included in SpatialHadoop, significantly reduces the total execution time. That is, Quadtree outperforms STR, Hilbert, and STR. This is due to Quadtree partitions space according to the data distribution (the concentration of the cells depends on the concentration of points).

4. The larger the k or ε values, the larger the probability that pairs of partitions are not pruned. It means more *map* tasks will be needed for the query processing, and more total execution time is spent for reporting the final query result.

5. The larger the number of computing nodes (η), the faster the DJQ MapReduce algorithms are, but when η exceeds the number of *map* tasks, no improvement for the whole job is obtained.

© 2021 García-García, F.J.

Continuing with the upgrade of SpatialHadoop, we have designed and implemented new MapReduce algorithms to perform DJQs like kNNJQ and εDRJQ efficiently. These DJQs follow a multiple query processing schema (kNNQ and εDRQ, respectively). We have also extended these DJQ MapReduce algorithms for managing non-points spatial objects like rectangles and line-segments. Improved versions of kNNJQ and εDRJQ MapReduce algorithms have been designed and implemented by using *repartitioning* techniques to handling dense areas (i.e., to repartition the densest generated partitions). For these DJQs, we have also incorporated into SpatialHadoop the *less data* technique to try to move as little data as possible between computing nodes and reduce the size of the output data. As a last enhancement of SpatialHadoop, we have proposed the design and implementation of two new RkNNQ MapReduce algorithms: MRSFT is the baseline MapReduce algorithm based on the SFT algorithm, and MRSLICE is the MapReduce version of SLICE algorithm, which is the state-of-the-art in RkNNQ. Finally, we have evaluated the performance of the proposed MapReduce algorithms in several situations with large real-world datasets, showing the efficiency and scalability of our proposals. The other part of Chapter 4 shows all these distributed DBQ algorithms beside interesting experimental conclusions like the following:

1. For kNNJQ and εDRJQ, the Quadtree-based partitioning technique, included in SpatialHadoop, significantly reduces the total execution time for large points datasets. When more complex spatial objects (rectangles and line-segments) are combined, the running time is a bit more costly than for points, following a similar trend.

2. The use of *repartitioning* and *less data* techniques in SpatialHadoop considerably reduces the total execution time and the shuffled data, mainly when large spatial datasets are combined in kNNJQ and εDRJQ. This reduction indicates that *repartitioning* of densest partitions is a good policy for MapReduce algorithms based on phases. And the *less data* technique transfers the necessary data for the query between computing nodes.

3. The best method to make the kNNJQ and εDRJQ MapReduce algorithms faster is to use the Quadtree-based repartitioning technique since the total time in *Merge Results* phase is considerably reduced. Moreover, this repartitioning based on Quadtree is the best choice to reduce the shuffled data between nodes.

4. For kNNJQ and εDRJQ, the larger k or ε, the higher the possibility that pairs of candidates will not be pruned, more tasks will be needed, and longer total execution time will be consumed for reporting the query result.

5. MRSLICE outperforms MRSFT several orders of magnitude, thanks to its pruning capabilities and the limited number of MapReduce jobs used in the executions.

6. The larger the k values, the greater the number of candidates to be verified, but for MRSLICE, the number of jobs and partitions involved is quite restricted, and the total execution time increases substantially less than for MRSFT.

7. The use of computing nodes (η) by MRSLICE is small, allowing the execution of several queries in parallel, unlike MRSFT that can leave the cluster busy.

Comparing a Hadoop-based DSDMS (SpatialHadoop) with Spark-based DSDMS (LocationSpark) is a demanding challenge. To carry out this comparison, we had to thoroughly study LocationSpark and design and implement new distributed DBQ algorithms (εDRQ, kCPQ, εDJQ, and εDRJQ). We have also extended the distributed DBQ algorithms for managing non-points spatial objects and improved them (for example, we have improved the kNNJQ included in LocationSpark) with similar techniques used in SpatialHadoop. Lastly, we have accomplished the performance evaluation of the proposed distributed DBQ algorithms in LocationSpark (using the Quadtree-based partitioning technique) with large real-world datasets and the experimental comparison with SpatialHadoop. All this information has been shown in Chapter 5, and next, we synthesize the most relevant experimental conclusions.

1. LocationSpark for kCPQ and εDJQ, the larger the k or ε values, the larger the possibility that pairs of candidates are not pruned, and more total execution time is needed. However, it is very fast for small and medium dataset sizes.

2. For the native implementation of kNNJQ, LocationSpark's execution times are fast and stable with small and medium dataset sizes and small k values, but its results are slower for larger datasets and higher k values since it is more sensitive to memory constraints.

3. For εDRJQ, LocationSpark is fast for small and medium datasets, but when the dataset sizes grow, its performance in terms of running time declines because of *memory pressure* problems.

4. For LocationSpark, the number of computing nodes (η) is not a determinant parameter for speedup of the algorithms as the availability of enough memory resources is.

5. For kCPQ and εDJQ, LocationSpark was faster than SpatialHadoop for small and medium dataset sizes. But for larger datasets, it required more time to execute the queries, even for large k and ε values due to *memory pressure* problems.

6. For kNNJQ and εDRJQ, SpatialHadoop is the fastest for large datasets and, for any k or ε value, except for small-to-medium dataset sizes where LocationSpark performance is the best.

7. As a general conclusion, performance trends of both DSDMSs are similar in terms of total execution time although, LocationSpark is the clear winner for execution time when small-medium datasets are combined, due to the efficiency of in-memory processing provided by Spark and additional improvements, like the *Query Plan Scheduler*. However, SpatialHadoop is faster when joining large real-world datasets because it is a more mature and robust DSDMS due to the time invested in research and development (e.g., it provides more spatial partitioning techniques, computational geometry algorithms, repartitioning techniques for skewed data, etc.).

© 2021 García-García, F.J.

In summary, in this thesis, we have studied and enhanced SpatialHadoop with the design and implementation of a new spatial partitioning technique based on Voronoi-Diagrams, new local indexing (Quadtree) and new DBQs (εDRQ, kNNQ, kCPQ, kNNJQ, RkNNQ, εDJQ, εDRJQ, etc.) MapReduce algorithms. LocationSpark has been also studied and enriched with new distributed DBQs (kCPQ, εDJQ, and εDRJQ) algorithms. We have thoroughly evaluated and compared both DSDMSs using various benchmark queries and large real-world datasets. This dissertation makes fundamental contributions in the two thrust areas of distributed spatial query processing and benchmarking DSDMSs. These advances are critical to the design of DSDMSs and significantly advances the state-of-the-art in that field.

6.2 Future Work

The continuing growth of spatial data sources, the advance of novel spatial data applications with new spatial queries, and the evolution of the infrastructure of DSDMSs result in new and interesting research challenges. While some of these future research directions are direct extensions of the contributions presented in this thesis, others look more innovative and challenging.

Spatial joins are traditionally defined and studied concerning geometries or locations while ignoring other types of information attached to the input objects like text or social information. An example of these extended join operations is *spatio-textual similarity join* [Bouros et al., 2012] which comes as a hybrid of a spatial ε-distance join and a set similarity join. One interesting research direction could be to design and implement top-k spatio-textual similarity join (e.g., to identify the top-k similar pairs by considering both textual relevancy and spatial proximity) [Hu et al., 2016] in SpatialHadoop and/or LocationSpark. Another related top-k join that is worth studying to include in such DSDMSs would be top-k spatial distance join [Qi et al., 2020].

A multi-way spatial join is a specific spatial query that addresses spatial join issues for multiple inputs. This extension of spatial join has been recently studied in MapReduce frameworks [Gupta et al., 2013], providing improvements in the context of communication costs [Bhattu et al., 2020], treatment of skewed data distribution [Kadari et al., 2019], etc. Considering this problem and all these enhancements, another interesting challenge could be the design and implementation of multi-way spatial join in DSDMSs based on Spark (e.g., LocationSpark). In the same research direction, it would be exciting to extend multi-way distance join [Corral et al., 2004b] to Spark-based DSDMSs.

A recent survey [Alam et al., 2021] reviews the existing ecosystems of spatial and spatio-temporal data analytics. They can be classified into three categories: (1) spatial databases (SQL and NoSQL), (2) big spatio-temporal data processing infrastructures, and (3) programming languages and software tools for processing spatio-temporal data. In the area of big spatio-temporal data processing infrastructures, a major challenge could be to enhance Spatio-Temporal-Hadoop [Alarabi et al., 2018] or STARK [Hagedorn and Räth, 2017] with more complex spatio-temporal operations like spatio-temporal joins [Whitman et al., 2019] and spatio-temporal distance-based joins.

Two of the most recent and actively maintained spatial and spatio-temporal data an-

alytics systems are *Sedona*[1] (formerly GeoSpark) and *Beast*[2] (formerly SpatialHadoop). *Sedona* [Yu et al., 2019], as a full-edged cluster computing framework that can process vector data at scale, extends the core engine of Apache Spark and SparkSQL to support spatial data types, indexes, and geometrical operations; with the aim of being able to load, process, and analyze large-scale spatial data. *Beast* [Zhang and Eldawy, 2020] is another Spark-based system for Big Exploratory Analytics on Spatio-Temporal data that supports both vector and raster data with multidimensional data types and index structures. It extends the Spark RDD API by adding geometry data types, spatio-temporal input formats, multidimensional indexes, query processing, and visualization. The base systems of *Beast* are Spark and Hadoop since it provides the several improvements over SpatialHadoop: (1) support of multidimensional data; (2) inclusion of new spatial partitioning techniques like R*-Grove [Vu and Eldawy, 2020]; (3) support of processing raster and vector data concurrently, etc. Therefore, another interesting research line could be to design and implement new spatial, spatio-temporal, and spatio-textual queries in these Big Spatial and Spatio-Temporal Data Analytics Systems.

Due to the fact that *Beast* is a recent system for Big Exploratory Analytics on Spatio-Temporal data that supports both vector and raster data, we can study the possibility to incorporate new indexes for points like xBR$^+$-tree [Roumelis et al., 2015] and raster like k^2-tree [Brisaboa et al., 2017] or k^2-raster [Silva-Coira et al., 2020] to perform efficient spatial and/or spatio-temporal queries between points [Roumelis et al., 2017], and points with raster data [Silva-Coira et al., 2020].

Big spatial data processing can be mainly classified into batch-only, streaming-only, and hybrid processing. Some applications are gaining much prominence today, such as IoT platforms or other sensor-based systems, in which data is constantly changing. For these systems, it is more appropriate to treat them as a continuous flow of data in which spatial continuous queries are performed better than through batch processing on a dataset created with a snapshot. *Apache Flink*[3] [Carbone et al., 2015] has emerged as an open-source system that provides a unified and scalable model for processing streaming and batch (hybrid) data under the premise that both types can be expressed and executed as pipelined fault-tolerant dataflows. Because *Flink* does not natively support spatial data processing, there have been efforts to provide its support. *GeoFlink* [Shaikh et al., 2020] is a system that provides efficient processing of spatial continuous queries (i.e., spatial range, spatial kNN, and spatial join queries) on point data streams and through the use of a Grid-based index. Consequently, one interesting challenge could be to design and implement spatial continuous queries over *Apache Flink*, either by extending it or by improving *GeoFlink* with new indexes and queries.

Finally, the number of frameworks with which to process Big Data grows at a similar speed to this one. We have multiple options to perform both batch and stream processing of data, and when a better framework emerges, it takes time to have to rewrite and adapt algorithms and queries. *Apache Beam*[4] appears as an open-source unified framework that allows the implementation of batch and streaming data processing jobs that run

[1] Available at `https://sedona.apache.org/`
[2] Available at `https://bitbucket.org/eldawy/beast/src/master/`
[3] Available at `https://flink.apache.org/`
[4] Available at `https://beam.apache.org/`

© 2021 García-García, F.J.

on any execution engine. Once a data pipeline is created, it can be processed by any of the supported runner back-ends like *Apache Flink*, *Apache Samza*, *Apache Spark*, and *Google Cloud Dataflow*. Also, in [Jacobs and Surdy, 2016] where *Apache Flink* vs. *Spark* is compared (concluding that the former provides better performance, lower latency, and better batch processing than the latter), it is described that *Apache Beam* comes to fix the shortcomings of both data processing frameworks. However, there is no native support for managing spatial data, although runners may use some of the previously discussed frameworks. *GeoBeam* [He et al., 2019] extends the *Apache Beam Model* for the spatial domain with a spatial pipeline, collection, and several transforms to support efficient spatial query processing. Moreover, the authors show the results of different experiments to test the efficiency of a range query operation on top of *Spark* and *Flink* clusters. Therefore, another open research line could be to design and implement new spatial queries, either batch or continuous, on top of this unified model.

Acronyms

© 2021 García-García, F.J.

Acronyms

API	Application Programming Interface
BEAST	Big Exploratory Analytics for Spatio-Temporal data
BF	Best-First
BSD	Big Spatial Data
BSDAS	Big Spatial Data Analytics System
BSP	Binary Space Partitioning
CHQ	Convex Hull Query
CP	Closest Pair
CPQ	Closest Pair Query
CPU	Central Processing Unit
DBQ	Distance-Based Query
DF	Depth-First
DJQ	Distance-based Join Query
εDJQ	εDistance Join Query
εDRJQ	εDistance Range Join Query
εDRQ	εDistance Range Query
DSDMS	Distributed Spatial Data Management System
HDFS	Hadoop Distributed File System
FINCH	Fast rknn processing using INtersections' Convex Hull
FPQ	Farthest Pair Query
GB	GigaBytes
GFS	Grid File Spatial

GIS	Geographic Information Systems
GNN	Group kNearest Neighbor Query
GNN	Group Nearest Neighbor
GPS	Global Positioning System
GS	Global Sampling
HP	Half-Plane
kCPQ	kClosest Pairs Query
kNN	kNearest Neighbor
kNNJQ	kNearest Neighbor Join Query
kNNQ	kNearest Neighbor Query
LAN	Local Area Network
LS	Local Sampling
MB	MegaBytes
MBR	Minimum Bounding Rectangle
MR	MapReduce
NE	North East
NN	Nearest Neighbor
NW	North West
PBSM	Partition-Based Spatial-Merge
PDDAI	Pair Data Density Area Intersection
PGBJ	Partitioning-Grouping Block-based Join
PS	Plane-Sweep
PSKCPQ	Plane-Sweep kClosest Pairs Query
PSKNNQ	Plane-Sweep kNearest Neighbor Query
PUQ	Polygon Union Query
RBF	Recursive Best-First search
RDD	Resilient Distributed Dataset
RkNNQ	Reverse kNearest Neighbor Query
RQ	Range Query
SATO	Sample, Analyze, Tear and Optimize

SBD	Spatial Big Data
SCM	Spatial Coding Matrix
SE	South East
SIS	Sensing Information Set
SJQ	Spatial Join Query
SMS	Short Message Service
SPM	Spatial Partitioning Matrix
SQ	Skyline Query
SQL	Structured Query Language
SRDD	Spatial Resilient Distributed Dataset
ST	Spatio-Temporal
STR	Sort-Tile-Recursive
SW	South West
VC	Voronoi-Cell
VD	Voronoi-Diagram
VDQ	Voronoi-Diagram Query
WAN	Wide Area Network

© 2021 García-García, F.J.

Bibliography

© 2021 García-García, F.J.

Bibliography

[Aji et al., 2015] Aji, A., Vo, H., and Wang, F. (2015). Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910:1–12.

[Aji et al., 2013] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. H. (2013). Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *PVLDB*, 6(11):1009–1020.

[Akdogan et al., 2010] Akdogan, A., Demiryurek, U., Kashani, F. B., and Shahabi, C. (2010). Voronoi-based geospatial query processing with MapReduce. In *CloudCom Conference*, pages 9–16.

[Alam et al., 2021] Alam, M. M., Torgo, L., and Bifet, A. (2021). A survey on spatio-temporal data analytics systems. *CoRR*, abs/2103.09883:1–44.

[Alarabi et al., 2018] Alarabi, L., Mokbel, M. F., and Musleh, M. (2018). St-hadoop: a mapreduce framework for spatio-temporal data. *GeoInformatica*, 22(4):785–813.

[Andrew, 1979] Andrew, A. M. (1979). Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219.

[Ankerst et al., 1999] Ankerst, M., Breunig, M. M., Kriegel, H., and Sander, J. (1999). OPTICS: ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60.

[Arge et al., 2008] Arge, L., de Berg, M., Haverkort, H. J., and Yi, K. (2008). The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:30.

[Arthur and Vassilvitskii, 2007] Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. In *SODA Conference*, pages 1027–1035.

[Aurenhammer, 1991] Aurenhammer, F. (1991). Voronoi diagrams - A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405.

[Baig et al., 2017] Baig, F., Vo, H., Kurç, T. M., Saltz, J. H., and Wang, F. (2017). Sparkgis: Resource aware efficient in-memory spatial query processing. In *SIGSPATIAL Conference*, pages 28:1–28:10.

[Bechini et al., 2016] Bechini, A., Marcelloni, F., and Segatori, A. (2016). A mapreduce solution for associative classification of big data. *Information Sciences*, 332:33–55.

[Beckmann et al., 1990] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990). The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331.

[Belussi et al., 2020a] Belussi, A., Migliorini, S., and Eldawy, A. (2020a). Cost estimation of spatial join in spatialhadoop. *GeoInformatica*, 24(4):1021–1059.

[Belussi et al., 2020b] Belussi, A., Migliorini, S., and Eldawy, A. (2020b). Skewness-based partitioning in spatialhadoop. *ISPRS International Journal of Geo-Information*, 9(4):201.

[Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

[Bhattu et al., 2020] Bhattu, S. N., Potluri, A., Kadari, P., and Subramanyam, R. B. V. (2020). Generalized communication cost efficient multi-way spatial join: revisiting the curse of the last reducer. *GeoInformatica*, 24(3):557–589.

[Blömer et al., 2016] Blömer, J., Lammersen, C., Schmidt, M., and Sohler, C. (2016). Theoretical analysis of the k-means algorithm - A survey. *CoRR*, abs/1602.08254:1–35.

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

[Bouros et al., 2012] Bouros, P., Ge, S., and Mamoulis, N. (2012). Spatio-textual similarity joins. *PVLDB*, 6(1):1–12.

[Brisaboa et al., 2017] Brisaboa, N. R., de Bernardo, G., Gutiérrez, G., Luaces, M. R., and Paramá, J. R. (2017). Efficiently querying vector and raster data. *Computer Journal*, 60(9):1395–1413.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 38(4):28–38.

[Chaudhuri et al., 1999] Chaudhuri, S., Motwani, R., and Narasayya, V. R. (1999). On random sampling over joins. In *SIGMOD Conference*, pages 263–274.

[Cheema et al., 2011] Cheema, M. A., Lin, X., Zhang, W., and Zhang, Y. (2011). Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE Conference*, pages 577–588.

[Chen and Zhang, 2014] Chen, C. L. P. and Zhang, C. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347.

[Chen and Patel, 2007] Chen, Y. and Patel, J. M. (2007). Efficient evaluation of all-nearest-neighbor queries. In *ICDE Conference*, pages 1056–1065.

[Comer, 1979] Comer, D. (1979). The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms (3. ed.)*. MIT Press.

[Corral, 2002] Corral, A. (2002). *Algorithms for Processing of Spatial Queries using R-trees. The Closest Pairs Query and its Application in Spatial Databases*. PhD thesis, University of Almeria.

[Corral and Almendros-Jimenez, 2007] Corral, A. and Almendros-Jimenez, J. M. (2007). A performance comparison of distance-based query algorithms using r-trees in spatial databases. *Information Sciences*, 177(11):2207–2237.

[Corral et al., 2000] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2000). Closest pair queries in spatial databases. In *SIGMOD Conference*, pages 189–200.

[Corral et al., 2004a] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2004a). Algorithms for processing k-closest-pair queries in spatial databases. *Data & Knowledge Engineering*, 49(1):67–104.

[Corral et al., 2004b] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2004b). Multi-way distance join queries in spatial databases. *GeoInformatica*, 8(4):373–402.

[Corral et al., 2006] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2006). Cost models for distance joins queries using r-trees. *Data & Knowledge Engineering*, 57(1):1–36.

[Corral and Vassilakopoulos, 2005] Corral, A. and Vassilakopoulos, M. (2005). On approximate algorithms for distance-based queries using r-trees. *Computer Journal*, 48(2):220–238.

[de Berg et al., 2008] de Berg, M., Cheong, O., van Kreveld, M. J., and Overmars, M. H. (2008). *Computational geometry: algorithms and applications*. Springer.

[de Carvalho Castro et al., 2020] de Carvalho Castro, J. P., Carniel, A. C., and de Aguiar Ciferri, C. D. (2020). Analyzing spatial analytics systems based on hadoop and spark: A user perspective. *Software - Practice and Experience*, 50(12):2121–2144.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI Conference*, pages 137–150.

[Doulkeridis and Nørvåg, 2014] Doulkeridis, C. and Nørvåg, K. (2014). A survey of large-scale analytical query processing in mapreduce. *VLDB Journal*, 23(3):355–380.

© 2021 García-García, F.J.

[Elashry et al., 2018] Elashry, A., Shehab, A., Riad, A. M., and Aboul-Fotouh, A. (2018). 2dpr-tree: Two-dimensional priority r-tree algorithm for spatial partitioning in spatialhadoop. *ISPRS International Journal of Geo-Information*, 7(5):179.

[Eldawy et al., 2015] Eldawy, A., Alarabi, L., and Mokbel, M. F. (2015). Spatial partitioning techniques in spatialhadoop. *PVLDB*, 8(12):1602–1613.

[Eldawy et al., 2013] Eldawy, A., Li, Y., Mokbel, M. F., and Janardan, R. (2013). Cg_hadoop: computational geometry in mapreduce. In *SIGSPATIAL Conference*, pages 284–293.

[Eldawy and Mokbel, 2015] Eldawy, A. and Mokbel, M. F. (2015). Spatialhadoop: A mapreduce framework for spatial data. In *ICDE Conference*, pages 1352–1363.

[Eldawy and Mokbel, 2017] Eldawy, A. and Mokbel, M. F. (2017). The era of big spatial data. *PVLDB*, 10(12):1992–1995.

[Emrich et al., 2010] Emrich, T., Graf, F., Kriegel, H., Schubert, M., and Thoma, M. (2010). Optimizing all-nearest-neighbor queries with trigonometric pruning. In *SSDBM Conference*, pages 501–518.

[Emrich et al., 2013a] Emrich, T., Kriegel, H., Kröger, P., Niedermayer, J., Renz, M., and Züfle, A. (2013a). Reverse-k-nearest-neighbor join processing. In *SSTD Conference*, pages 277–294.

[Emrich et al., 2015] Emrich, T., Kriegel, H., Kröger, P., Niedermayer, J., Renz, M., and Züfle, A. (2015). On reverse-k-nearest-neighbor joins. *GeoInformatica*, 19(2):299–330.

[Emrich et al., 2013b] Emrich, T., Kröger, P., Niedermayer, J., Renz, M., and Züfle, A. (2013b). A mutual pruning approach for rknn join processing. In *BTW Conference*, pages 21–35.

[Ester et al., 1996] Ester, M., Kriegel, H., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD Conference*, pages 226–231.

[Evans et al., 2014] Evans, M. R., Oliver, D., Zhou, X., and Shekhar, S. (2014). Spatial big data: Case studies on volume, velocity, and variety. In Karimi, H. A., editor, *Big Data: Techniques and Technologies in Geoinformatics*, chapter 8, pages 149–176. CRC Press.

[Faloutsos and Roseman, 1989] Faloutsos, C. and Roseman, S. (1989). Fractals for secondary key retrieval. In *PODS Conference*, pages 247–252.

[Finkel and Bentley, 1974] Finkel, R. A. and Bentley, J. L. (1974). Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9.

[Fuchs et al., 1980] Fuchs, H., Kedem, Z. M., and Naylor, B. F. (1980). On visible surface generation by a priori tree structures. In *SIGGRAPH Conference*, pages 124–133.

[Gaede and Günther, 1998] Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231.

[Gao et al., 2015] Gao, Y., Chen, L., Li, X., Yao, B., and Chen, G. (2015). Efficient k-closest pair queries in general metric spaces. *VLDB Journal*, 24(3):415–439.

[García-García et al., 2020a] García-García, F., Corral, A., and Iribarne, L. (2020a). Including the quadtree index in spatialhadoop. In *Panhellenic Conference on Informatics*, pages 376–379.

[García-García et al., 2017a] García-García, F., Corral, A., Iribarne, L., Mavrommatis, G., and Vassilakopoulos, M. (2017a). A comparison of distributed spatial data management systems for processing distance join queries. In *ADBIS Conference*, pages 214–228.

[García-García et al., 2016a] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2016a). Distance range queries in spatialhadoop. In *JISBD Conference*, pages 1–14.

[García-García et al., 2017b] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2017b). Rknn query processing in distributed spatial infrastructures: A performance study. In *MEDI Conference*, pages 200–207.

[García-García et al., 2018a] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2018a). Voronoi-diagram based partitioning for distance join query processing in spatialhadoop. In *MEDI Conference*, pages 251–267.

[García-García et al., 2019] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2019). MRSLICE: efficient rknn query processing in spatialhadoop. In *MEDI Conference*, pages 235–250.

[García-García et al., 2020b] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2020b). Improving distance-join query processing with voronoi-diagram based partitioning in spatialhadoop. *Future Generation Computer Systems*, 111:723–740.

[García-García et al., 2021] García-García, F., Corral, A., Iribarne, L., and Vassilakopoulos, M. (2021). Enhancing sedona (formerly geospark) with efficient k nearest neighbor join processing. In *MEDI Conference*, page Submitted.

[García-García et al., 2016b] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2016b). Enhancing spatialhadoop with closest pair queries. In *ADBIS Conference*, pages 212–225.

[García-García et al., 2018b] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2018b). Efficient large-scale distance-based join queries in spatialhadoop. *GeoInformatica*, 22(2):171–209.

© 2021 García-García, F.J.

[García-García et al., 2020c] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. (2020c). Efficient distance join query processing in distributed spatial data management systems. *Information Sciences*, 512:985–1008.

[Gounaris and Torres, 2018] Gounaris, A. and Torres, J. (2018). A methodology for spark parameter tuning. *Big Data Research*, 11:22–32.

[Gupta et al., 2013] Gupta, H., Chawda, B., Negi, S., Faruquie, T. A., Subramaniam, L. V., and Mohania, M. K. (2013). Processing multi-way spatial joins on map-reduce. In *EDBT Conference*, pages 113–124.

[Gutierrez and Sáez, 2013] Gutierrez, G. and Sáez, P. (2013). The k closest pairs in spatial databases - when only one set is indexed. *GeoInformatica*, 17(4):543–565.

[Güting et al., 2010] Güting, R. H., Behr, T., and Düntgen, C. (2010). SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Engineering Bulletin*, 33(2):56–63.

[Guttman, 1984] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57.

[Hagedorn and Räth, 2017] Hagedorn, S. and Räth, T. (2017). Efficient spatio-temporal event processing with STARK. In *EDBT Conference*, pages 570–573.

[Harada et al., 1990] Harada, L., Nakano, M., Kitsuregawa, M., and Takagi, M. (1990). Query processing for multi-attribute clustered records. In *VLDB Conference*, pages 59–70.

[Hassani, 2017] Hassani, H. (2017). Research methods in computer science: The challenges and issues. *CoRR*, abs/1703.04080:1–16.

[He et al., 2019] He, Z., Liu, G., Ma, X., and Chen, Q. (2019). Geobeam: A distributed computing framework for spatial data. *Computers & Geosciences*, 131:15–22.

[Hilbert, 1891] Hilbert, D. (1891). Ueber die reellen züge algebraischer curven. *Mathematische Annalen*, 38(1):115–138.

[Hjaltason and Samet, 1998] Hjaltason, G. R. and Samet, H. (1998). Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248.

[Hjaltason and Samet, 1999] Hjaltason, G. R. and Samet, H. (1999). Improved bulk-loading algorithms for quadtrees. In *ACM-GIS Conference*, pages 110–115.

[Hjaltason and Samet, 2003] Hjaltason, G. R. and Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580.

[Hu et al., 2016] Hu, H., Li, G., Bao, Z., Feng, J., Wu, Y., Gong, Z., and Xu, Y. (2016). Top-k spatio-textual similarity join. *IEEE Transactions on Knowledge and Data Engineering*, 28(2):551–565.

[Hu et al., 2020] Hu, Y., Peng, G., Wang, Z., Cui, Y., and Qin, H. (2020). Partition selection for large-scale data management using knn join processing. *Mathematical Problems in Engineering*, 2020(7898230):1–14.

[Hughes et al., 2015] Hughes, J. N., Annex, A., Eichelberger, C. N., Fox, A., Hulbert, A., and Ronquest, M. (2015). Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, volume 9473, page 94730F.

[Jacobs and Surdy, 2016] Jacobs, K. and Surdy, K. (2016). Apache flink: Distributed stream data processing. Technical report, CERN.

[Jacox and Samet, 2007] Jacox, E. H. and Samet, H. (2007). Spatial join techniques. *ACM Transactions on Database Systems*, 32(1):7:1–44.

[Jacox and Samet, 2008] Jacox, E. H. and Samet, H. (2008). Metric space similarity joins. *ACM Transactions on Database Systems*, 33(2):1–38.

[Jain et al., 1999] Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323.

[Ji et al., 2013] Ji, C., Hu, H., Xu, Y., Li, Y., and Qu, W. (2013). Efficient multi-dimensional spatial rknn query processing with MapReduce. In *ChinaGrid Conference*, pages 63–68.

[Ji et al., 2015] Ji, C., Qu, W., Li, Z., Xu, Y., Li, Y., and Wu, J. (2015). Scalable multi-dimensional RNN query processing. *Concurrency and Computation: Practice and Experience*, 27(16):4156–4171.

[Kadari et al., 2019] Kadari, P., Potluri, A., Sristy, N. B., Subramanyam, R. B. V., and Kumar, N. V. N. (2019). Skew aware partitioning techniques for multi-way spatial join. In *MIKE Conference*, pages 52–61.

[Kalyvas and Maragoudakis, 2019] Kalyvas, C. and Maragoudakis, M. (2019). Skyline and reverse skyline query processing in spatialhadoop. *Data & Knowledge Engineering*, 122:55–80.

[Karanth, 2014] Karanth, S. (2014). *Mastering Hadoop*. Packt Publishing.

[Karim et al., 2018] Karim, M. R., Cochez, M., Beyan, O. D., Ahmed, C. F., and Decker, S. (2018). Mining maximal frequent patterns in transactional databases and dynamic data streams: A spark-based approach. *Information Sciences*, 432:278–300.

[Kim et al., 2016] Kim, W., Kim, Y., and Shim, K. (2016). Parallel computation of k-nearest neighbor joins using mapreduce. In *Big Data Conference*, pages 696–705.

[Kim and Patel, 2010] Kim, Y. J. and Patel, J. M. (2010). Performance comparison of the r*-tree and the quadtree for knn and distance join queries. *IEEE Transactions on Knowledge and Data Engineering*, 22(7):1014–1027.

© 2021 García-García, F.J.

[Kini and Emanuele, 2014] Kini, A. and Emanuele, R. (2014). Geotrellis: Adding geospatial capabilities to spark. *Spark Summit*.

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

[Kuhlman et al., 2017] Kuhlman, C., Yan, Y., Cao, L., and Rundensteiner, E. A. (2017). Pivot-based distributed k-nearest neighbor mining. In *ECML/PKDD Conference*, pages 843–860.

[Leutenegger et al., 1997] Leutenegger, S. T., Edgington, J. M., and López, M. A. (1997). STR: A simple and efficient algorithm for r-tree packing. In *ICDE Conference*, pages 497–506.

[Li et al., 2014] Li, F., Ooi, B. C., Özsu, M. T., and Wu, S. (2014). Distributed data management using mapreduce. *ACM Computing Surveys*, 46(3):31:1–31:42.

[Li and Taniar, 2017] Li, L. and Taniar, D. (2017). A taxonomy for distance-based spatial join queries. *International Journal of Data Warehousing and Mining*, 13(3):1–24.

[Li et al., 2019] Li, Y., Eldawy, A., Xue, J., Knorozova, N., Mokbel, M. F., and Janardan, R. (2019). Scalable computational geometry in mapreduce. *VLDB Journal*, 28(4):523–548.

[Li et al., 2011] Li, Z., Lee, K. C. K., Zheng, B., Lee, W., Lee, D. L., and Wang, X. (2011). Ir-tree: An efficient index for geographic document search. *IEEE Transactions on Knowledge and Data Engineering*, 23(4):585–599.

[Lu and Güting, 2012] Lu, J. and Güting, R. H. (2012). Parallel secondo: Boosting database engines with Hadoop. In *ICPADS Conference*, pages 738–743.

[Lu et al., 2012] Lu, W., Shen, Y., Chen, S., and Ooi, B. C. (2012). Efficient processing of k nearest neighbor joins using MapReduce. *PVLDB*, 5(10):1016–1027.

[Ma et al., 2009] Ma, Q., Yang, B., Qian, W., and Zhou, A. (2009). Query processing of massive trajectory data based on MapReduce. In *CloudDb Conference*, pages 9–16.

[MacQueen, 1967] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297.

[Mavrommatis et al., 2017] Mavrommatis, G., Moutafis, P., Vassilakopoulos, M., García-García, F., and Corral, A. (2017). Slicenbound: Solving closest pairs and distance join queries in apache spark. In *ADBIS Conference*, pages 199–213.

[Mokbel et al., 2003] Mokbel, M. F., Aref, W. G., and Kamel, I. (2003). Analysis of multi-dimensional space-filling curves. *GeoInformatica*, 7(3):179–209.

[Moutafis et al., 2019a] Moutafis, P., García-García, F., Mavrommatis, G., Vassilakopoulos, M., Corral, A., and Iribarne, L. (2019a). Mapreduce algorithms for the K group nearest-neighbor query. In *ACM SAC Conference*, pages 448–455.

[Moutafis et al., 2021] Moutafis, P., García-García, F., Mavrommatis, G., Vassilakopoulos, M., Corral, A., and Iribarne, L. (2021). Algorithms for processing the group k nearest-neighbor query on distributed frameworks. *Distributed and Parallel Databases*, In Press.

[Moutafis et al., 2019b] Moutafis, P., Mavrommatis, G., Vassilakopoulos, M., and Sioutas, S. (2019b). Efficient processing of all-k-nearest-neighbor queries in the mapreduce programming framework. *Data & Knowledge Engineering*, 121:42–70.

[Nievergelt et al., 1984] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71.

[Nodarakis et al., 2016a] Nodarakis, N., Pitoura, E., Sioutas, S., Tsakalidis, A. K., Tsoumakos, D., and Tzimas, G. (2016a). kdann+: A rapid aknn classifier for big data. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 24:139–168.

[Nodarakis et al., 2016b] Nodarakis, N., Rapti, A., Sioutas, S., Tsakalidis, A. K., Tsolis, D., Tzimas, G., and Panagis, Y. (2016b). (a)knn query processing on the cloud: A survey. In *ALGOCLOUD Conference, Revised Selected Papers*, pages 26–40.

[Okabe et al., 2000] Okabe, A., Boots, B., Sugihara, K., and Chiu, S. N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley.

[Orenstein and Merrett, 1984] Orenstein, J. A. and Merrett, T. H. (1984). A class of data structures for associative searching. In *PODS Conference*, pages 181–190.

[Pandey et al., 2018] Pandey, V., Kipf, A., Neumann, T., and Kemper, A. (2018). How good are modern spatial analytics systems? *PVLDB*, 11(11):1661–1673.

[Papadopoulos et al., 2006] Papadopoulos, A. N., Nanopoulos, A., and Manolopoulos, Y. (2006). Processing distance join queries with constraints. *Computer Journal*, 49(3):281–296.

[Patel and DeWitt, 1996] Patel, J. M. and DeWitt, D. J. (1996). Partition based spatial-merge join. In *SIGMOD Conference*, pages 259–270.

[Peano, 1890] Peano, G. (1890). Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160.

[Pertesis and Doulkeridis, 2015] Pertesis, D. and Doulkeridis, C. (2015). Efficient skyline query processing in spatialhadoop. *Information Systems*, 54:325–335.

[Phillips, 2002] Phillips, S. J. (2002). Acceleration of k-means and related clustering algorithms. In *ALENEX Conference*, pages 166–177.

© 2021 García-García, F.J.

[Preparata and Shamos, 1985] Preparata, F. P. and Shamos, M. I. (1985). *Computational Geometry - An Introduction*. Springer.

[Qi et al., 2020] Qi, S., Bouros, P., and Mamoulis, N. (2020). Top-k spatial distance joins. *GeoInformatica*, 24(3):591–631.

[Ros and Guillaume, 2016] Ros, F. and Guillaume, S. (2016). DENDIS: a new density-based sampling for clustering algorithm. *Expert Systems with Applications*, 56:349–359.

[Ros and Guillaume, 2017] Ros, F. and Guillaume, S. (2017). DIDES: a fast and effective sampling for clustering algorithm. *Knowledge and Information Systems*, 50(2):543–568.

[Roumelis et al., 2014] Roumelis, G., Corral, A., Vassilakopoulos, M., and Manolopoulos, Y. (2014). A new plane-sweep algorithm for the k-closest-pairs query. In *SOFSEM Conference*, pages 478–490.

[Roumelis et al., 2016] Roumelis, G., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. (2016). New plane-sweep algorithms for distance-based join queries in spatial databases. *GeoInformatica*, 20(4):571–628.

[Roumelis et al., 2017] Roumelis, G., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. (2017). Efficient query processing on large spatial databases: A performance study. *Journal of Systems and Software*, 132:165–185.

[Roumelis et al., 2015] Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., and Manolopoulos, Y. (2015). The xbr$^+$-tree: An efficient access method for points. In *DEXA Conference*, pages 43–58.

[Roussopoulos et al., 1995] Roussopoulos, N., Kelley, S., and Vincent, F. (1995). Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79.

[Sagan, 2012] Sagan, H. (2012). *Space-filling curves*. Springer Science & Business Media.

[Samet, 1984] Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260.

[Schneider and Vlachos, 2013] Schneider, J. and Vlachos, M. (2013). Fast parameterless density-based clustering via random projections. In *CIKM Conference*, pages 861–866.

[Schneider, 2009] Schneider, M. (2009). Spatial data types. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 2698–2702. Springer US.

[Schoier and Gregorio, 2017] Schoier, G. and Gregorio, C. (2017). Clustering algorithms for spatial big data. In *ICCSA Conference*, pages 571–583.

[Schubert and Gertz, 2018] Schubert, E. and Gertz, M. (2018). Improving the cluster structure extracted from OPTICS plots. In *LWDA Conference*, pages 318–329.

[Schubert and Zimek, 2019] Schubert, E. and Zimek, A. (2019). ELKI: a large open-source library for data analysis. *CoRR*, abs/1902.03616:1–134.

[Sellis et al., 1987] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. (1987). The r^+-tree: A dynamic index for multi-dimensional objects. In *VLDB Conference*, pages 507–518.

[Shaikh et al., 2020] Shaikh, S. A., Mariam, K., Kitagawa, H., and Kim, K. (2020). Geoflink: A distributed and scalable framework for the real-time processing of spatial streams. In *CIKM Conference*, pages 3149–3156.

[Sharker and Karimi, 2014] Sharker, M. H. and Karimi, H. A. (2014). Distributed and parallel computing. In Karimi, H. A., editor, *Big Data: Techniques and Technologies in Geoinformatics*, chapter 1, pages 1–29. CRC Press.

[Shekhar and Chawla, 2003] Shekhar, S. and Chawla, S. (2003). *Spatial databases - a tour*. Prentice Hall.

[Shin et al., 2003] Shin, H., Moon, B., and Lee, S. (2003). Adaptive and incremental processing for distance join queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1561–1578.

[Shou et al., 2003] Shou, Y., Mamoulis, N., Cao, H., Papadias, D., and Cheung, D. W. (2003). Evaluation of iceberg distance joins. In *SSTD Conference*, pages 270–288.

[Silva and Reed, 2012] Silva, Y. N. and Reed, J. M. (2012). Exploiting mapreduce-based similarity joins. In *SIGMOD Conference*, pages 693–696.

[Silva-Coira et al., 2020] Silva-Coira, F., Paramá, J. R., Ladra, S., López, J., and Gutiérrez, G. (2020). Efficient processing of raster and vector data. *PLoS ONE*, 15(1):1–35.

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

[Song et al., 2016] Song, G., Rochas, J., Beze, L. E., Huet, F., and Magoulès, F. (2016). K nearest neighbour joins for big data on mapreduce: A theoretical and experimental analysis. *IEEE Transactions on Knowledge and Data Engineering*, 28(9):2376–2392.

[Sriharsha, 2021] Sriharsha, R. (2018 (accessed March 3, 2021)). *Magellan: Geospatial Analytics Using Spark*.

[Stanoi et al., 2000] Stanoi, I., Agrawal, D., and El Abbadi, A. (2000). Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53.

[Tan et al., 2012] Tan, H., Luo, W., and Ni, L. M. (2012). Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *CIKM Conference*, pages 2139–2143.

© 2021 García-García, F.J.

[Tang et al., 2020] Tang, M., Yu, Y., Mahmood, A. R., Malluhi, Q. M., Ouzzani, M., and Aref, W. G. (2020). Locationspark: In-memory distributed spatial query processing and optimization. *Frontiers Big Data*, 3:30.

[Tang et al., 2016] Tang, M., Yu, Y., Malluhi, Q. M., Ouzzani, M., and Aref, W. G. (2016). Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB*, 9(13):1565–1568.

[Tao et al., 2004] Tao, Y., Papadias, D., and Lian, X. (2004). Reverse knn search in arbitrary dimensionality. In *VLDB Conference*, pages 744–755.

[Thusoo et al., 2009] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive - A warehousing solution over a MapReduce framework. *PVLDB*, 2(2):1626–1629.

[Velentzas et al., 2021] Velentzas, P., Corral, A., and Vassilakopoulos, M. (2021). Big spatial and spatio-temporal data analytics systems. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 47:155–180.

[Vo et al., 2014] Vo, H., Aji, A., and Wang, F. (2014). SATO: a spatial data partitioning framework for scalable query processing. In *SIGSPATIAL Conference*, pages 545–548.

[Vu et al., 2020] Vu, T., Belussi, A., Migliorini, S., and Eldawy, A. (2020). Using deep learning for big spatial data partitioning. *ACM Transactions on Spatial Algorithms and Systems*, 7(1):3:1–3:37.

[Vu and Eldawy, 2020] Vu, T. and Eldawy, A. (2020). R*-grove: Balanced spatial partitioning for large-scale datasets. *Frontiers Big Data*, 3:28.

[Weber et al., 1998] Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB Conference*, pages 194–205.

[Whitman et al., 2019] Whitman, R. T., Marsh, B. G., Park, M. B., and Hoel, E. G. (2019). Distributed spatial and spatio-temporal join on apache spark. *ACM Transactions on Spatial Algorithms and Systems*, 5(1):6:1–6:28.

[Whitman et al., 2017] Whitman, R. T., Park, M. B., Marsh, B. G., and Hoel, E. G. (2017). Spatio-temporal join on apache spark. In *SIGSPATIAL Conference*, pages 20:1–20:10.

[Wilson et al., 2016] Wilson, B., Palamuttam, R., Whitehall, K., Mattmann, C., Goodman, A., Boustani, M., Shah, S., Zimdars, P., and Ramirez, P. M. (2016). Scispark: Highly interactive in-memory science data analytics. In *BigData Conference*, pages 2964–2973.

[Wu et al., 2019] Wu, J., Zhang, Y., Wang, J., Lin, C., Fu, Y., and Xing, C. (2019). Improving distributed similarity join in metric space with error-bounded sampling. *CoRR*, abs/1905.05981:1–17.

[Wu et al., 2008] Wu, W., Yang, F., Chan, C. Y., and Tan, K. (2008). FINCH: evaluating reverse k-nearest-neighbor queries on location data. *PVLDB*, 1(1):1056–1067.

[Xia et al., 2004] Xia, C., Lu, H., Ooi, B. C., and Hu, J. (2004). Gorder: An efficient method for KNN join processing. In *VLDB Conference*, pages 756–767.

[Xie et al., 2016] Xie, D., Li, F., Yao, B., Li, G., Zhou, L., and Guo, M. (2016). Simba: Efficient in-memory spatial analytics. In *SIGMOD Conference*, pages 1071–1085.

[Xu and Tian, 2015] Xu, D. and Tian, Y. (2015). A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193.

[Yang and Lin, 2001] Yang, C. and Lin, K. (2001). An index structure for efficient reverse nearest neighbor queries. In *ICDE Conference*, pages 485–492.

[Yang and Lin, 2002] Yang, C. and Lin, K. (2002). An index structure for improving closest pairs and related join queries in spatial databases. In *IDEAS Conference*, pages 140–149.

[Yang et al., 2015] Yang, S., Cheema, M. A., Lin, X., and Wang, W. (2015). Reverse k nearest neighbors query processing: Experiments and analysis. *PVLDB*, 8(5):605–616.

[Yang et al., 2014] Yang, S., Cheema, M. A., Lin, X., and Zhang, Y. (2014). SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In *ICDE Conference*, pages 760–771.

[Yang et al., 2017] Yang, S., Cheema, M. A., Lin, X., Zhang, Y., and Zhang, W. (2017). Reverse k nearest neighbors queries and spatial reverse top-k queries. *VLDB Journal*, 26(2):151–176.

[Yao and Li, 2018] Yao, X. and Li, G. (2018). Big spatial vector data management: a review. *Big Earth Data*, 2(1):108–129.

[Yao et al., 2017] Yao, X., Mokbel, M. F., Alarabi, L., Eldawy, A., Yang, J., Yun, W., Li, L., Ye, S., and Zhu, D. (2017). Spatial coding-based approach for partitioning big spatial data in hadoop. *Computers & Geosciences*, 106:60–67.

[Yokoyama et al., 2012] Yokoyama, T., Ishikawa, Y., and Suzuki, Y. (2012). Processing all k-nearest neighbor queries in hadoop. In *WAIM Conference*, pages 346–351.

[You et al., 2015] You, S., Zhang, J., and Gruenwald, L. (2015). Large-scale spatial join query processing in cloud. In *ICDE Workshops*, pages 34–41.

[Yu et al., 2015] Yu, J., Wu, J., and Sarwat, M. (2015). Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL Conference*, pages 70:1–70:4.

[Yu et al., 2019] Yu, J., Zhang, Z., and Sarwat, M. (2019). Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–78.

© 2021 García-García, F.J.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI Conference*, pages 15–28.

[Zeidan et al., 2018] Zeidan, A., Lagerspetz, E., Zhao, K., Nurmi, P., Tarkoma, S., and Vo, H. T. (2018). Geomatch: Efficient large-scale map matching on apache spark. In *Big Data Conference*, pages 384–391.

[Zhang et al., 2012] Zhang, C., Li, F., and Jestes, J. (2012). Efficient parallel kNN joins for large data in MapReduce. In *EDBT Conference*, pages 38–49.

[Zhang et al., 2015] Zhang, H., Chen, G., Ooi, B. C., Tan, K., and Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.

[Zhang et al., 2004] Zhang, J., Mamoulis, N., Papadias, D., and Tao, Y. (2004). All-nearest-neighbors queries in spatial databases. In *SSDBM Conference*, pages 297–306.

[Zhang et al., 2009a] Zhang, S., Han, J., Liu, Z., Wang, K., and Feng, S. (2009a). Spatial queries evaluation with MapReduce. In *GCC Conference*, pages 287–292.

[Zhang et al., 2009b] Zhang, S., Han, J., Liu, Z., Wang, K., and Xu, Z. (2009b). SJMR: parallelizing spatial join with MapReduce on clusters. In *CLUSTER Conference*, pages 1–8.

[Zhang and Eldawy, 2020] Zhang, Y. and Eldawy, A. (2020). Evaluating computational geometry libraries for big spatial data exploration. In *ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data*, pages 3:1–3:6.

[Zhao et al., 2018] Zhao, X., Zhang, J., and Qin, X. (2018). *k*nn-dp: Handling data skewness in *kNN* joins using mapreduce. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):600–613.

This file has been generated using LaTeX.

All Figures and Tables in this file are originals

Efficient Query Processing in
Distributed Spatial Data Management Systems

Francisco José García García
Departament of Informatics
Grupo de Investigación de Informática Aplicada (TIC-211)
University of Almería
Almería, June, 2021

http://acg.ual.es

Francisco José García García

Efficient Query Processing in Distributed Spatial Data Management Systems

Big Spatial Data (BSD) is now one of the most active research fields in spatial computing, mainly motivated by the rapid development of smart, sensor, and mobile technologies. Recent BSD developments have motivated the emergence of novel technologies for distributed processing of large-scale spatial data in shared-nothing clusters of computers, leading to Distributed Spatial Data Management Systems (DSDMSs). Two of the most leading DSDMSs are SpatialHadoop (disk-based DSDMS) and LocationSpark (in-memory-based DSDMS), and they support several characteristics like spatial data partitioning, indexing methods, and spatial query processing.

In this dissertation, we intend to study and enrich different DSDMSs to develop new efficient spatial data algorithms that take advantage of the features they provide. To achieve this goal, we propose new partitioning, indexing, and pruning techniques that allow optimal processing of spatial data. The execution of an extensive set of experiments on data sets, both synthetic and real, will enable us to demonstrate the efficiency and scalability of our proposals.

Through this thesis, we review the most relevant DSDMSs (research prototypes), the state-of-the-art spatial partitioning techniques, and the most representative Distance-Based Queries (DBQs). Then, we focus our study on the structure and operations of spatial data partition methods and indexing structures in SpatialHadoop, by proposing a spatial partitioning technique based on Voronoi-Diagrams and including the Quadtree as a local index in such a DSDMS. Driven by an exhaustive study on spatial query processing in SpatialHadoop, we identify and implement new DBQs with different extensions and improvements. Next, the general spatial query processing scheme of LocationSpark is studied to extend it with new distributed Distance-based Join Query (DJQ) algorithms and improvements. Finally, an extensive performance evaluation and comparison of such enhancements in SpatialHadoop and LocationSpark are achieved to identify which DSDMS is the most appropriate for the distributed query processing on large volumes of spatial data.