SpatialHadoop: A MapReduce Framework for Big Spatial Data

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ahmed Eldawy

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Mohamed F. Mokbel

June, 2016

Acknowledgements

First of all, I owe my deepest gratitude to my advisor Prof. Mohamed F. Mokbel for leading me all the way in my PhD from day one until my successful defense. Not only he gave me all the academic support I needed in my research, but he was also a model of how the good advisor should be. He was always there to help me making the best choices while put my own benefit as number one. I owe him not only this thesis, but all my future success in my career.

I also would like to thank all the committee members of my final oral exam, Ravi Janardan, Steven Manson, and Shashi Shekhar. Special thanks go to Prof. Shashi Shekhar who served in all my PhD committees and provided valuable feedback that helped me shaping my final work.

As a member of the data management lab, I am also grateful to all other lab members including Louai Alarabi, Rami Alghamdi, Saif Alharthi, Jie Bao, Harshada Chavan, Abdeltawab Hendawi, Christopher Jonathan, Mohamed E. Khalefa, Justin J. Levandoski, Amr Magdy, Mohamed Mirza, Ibrahim Sabek, and Mohamed Sarwat.

It is also an honor for me to thank all my collaborators and hosts in my internships and visits outside the University of Minnesota including, Saleh Basalamah, Michael J. Carey, Badrish Chandramouli, Jonathan Goldstein, Ihab Ilyas, Rohit Khandekar, Paul Larson, Chen Li, Andrea Ross, and Kun-Lung Wu.

Last but not least, I am in debt to my family members for their continuous support and endless patience, my parents, the first teachers in my life, my brother who led my way to success, my sisters with their sincere and endless love, my wife who always supports me through the struggles of life, and my kids, who added joy to my life.

Dedication

This work is dedicated to Nihal, Noura, and Laila.

Abstract

There has been a recent explosion in the amounts of spatial data produced by several devices such as smart phones, satellites, space telescopes, medical devices, among others. This variety of such spatial data makes it widely used across important applications such as brain simulations, identifying cancer clusters, tracking infectious disease, drug addiction, simulating climate changes, and event detection and analysis. While there are several distributed systems that are designed to handle Big Data in general, e.g., Hadoop, Hive, Spark, and Impala, they all fall short in supporting spatial data efficiently. As a result, there are great research efforts in either extending these systems or building new systems to efficiently support Big Spatial Data.

In this thesis, we describe SpatialHadoop, a full-fledged system for spatial data which extends Hadoop in its core to efficiently support spatial data. SpatialHadoop is available as an open source software and has been already downloaded around 80,000 times. SpatialHadoop consists of four main layers, namely, *language*, *indexing*, *query processing*, and *visualization*. In the *language* layer, SpatialHadoop provides a high level language, termed Pigeon, which provides standard spatial data types and query processing for easy access to non-technical users. The *indexing* layer provides efficient spatial indexes, such as grid, R-tree, R+-tree, and Quad tree, which organize the data nicely in the distributed file system. The indexes follow a two-level design of one *global* index that partitions the data across machines, and multiple *local* indexes that organize records in each machine. The *query processing* layer encapsulates a set of spatial operations that ship with SpatialHadoop including basic spatial operations, join operations and computational geometry operations. The *visualization* layer allows users to explore big spatial data by generating images that provide bird's-eye view on the data. SpatialHadoop is already used as a back bone in several real systems, including SHAHED, a web-based application for interactive exploration of satellite data.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Overview of SpatialHadoop 4

2 Pigeon: A Spatial MapReduce Language **7**

2.1 Overview 8

2.2 Spatial Datatypes 9

2.3 Spatial Functions 10

2.4 Case-study: OpenStreetMap 11

3 Spatial Indexing in MapReduce **16**

3.1 Overview of Indexing in SpatialHadoop 17

3.2 Index Building 18

3.3 Uniform Grid-based Index 20

3.4 R-tree-based Index 21

3.5 R+-tree-based Index 22

3.6 Quad-tree-based Index 24

3.7 K-d tree-based Index 24
3.8 Z-curve-based Index 25
3.9 Hilbert curve-based Index 25
3.10 Experiments 26

4 Spatial Query Processing in MapReduce 33
4.1 MapReduce Layer 33
4.2 Operations Layer 36
4.3 Experiments 41

5 HadoopViz: Extensible Visualization of Big Spatial Data 54
5.1 Single-Level Visualization 57
5.2 Multilevel Visualization 62
5.3 Visualization Abstraction 67
5.4 Case Studies 71
5.5 Experiments 77

6 SHAHED: Interactive Exploration of Satellite Data 85
6.1 Introduction 85
6.2 Background 89
6.3 Overview 92
6.4 Uncertainty 93
6.5 Spatio-temporal Indexing 95
6.6 Query Processing 102
6.7 Visualization 105
6.8 Experiments 111
6.9 Web Interface 113

7 Related Work 116
7.1 Implementation Approach 117
7.2 Architecture 118
7.3 Language 118
7.4 Spatial Indexes 118

7.5 Queries 119
7.6 Visualization 119

8 Conclusion 122

References 124

List of Tables

3.1 Experiments Parameters 26
3.2 Experiments datasets 27
7.1 Existing work in the area of big spatial data 121

List of Figures

1.1 Range query in Hadoop vs. SpatialHadoop 3
1.2 Overview of SpatialHadoop 4
2.1 Overview of Pigeon 8
3.1 Spatial indexing in SpatialHadoop 17
3.2 The Partitioning Phase 20
3.3 R+-tree index of a 400 GB road network dataset 22
3.4 A Quad-tree index for a 400GB road network dataset 23
3.5 Five Quality Measures for `Buildings` 28
3.6 Quality Measures 31
3.7 Performance of index construction 32
4.1 Map phase in Hadoop and SpatialHadoop 34
4.2 *kNN* query ($k = 3$) in SpatialHadoop 38
4.3 *Distributed* join between roads and rivers 40
4.4 Partitions. 41
4.5 Range query experiments with SYNTH dataset 43
4.6 Range query experiments with real datasets 44
4.7 Range Query 50
4.8 *kNN* algorithms with SYNTH dataset 51
4.9 Performance of *kNN* with TIGER dataset 51
4.10 Performance of spatial join algorithms 52
4.11 Spatial join performance (best viewed in color) 53
5.1 World heat map of temperature without HadoopViz 55
5.2 World heat map of temperature using HadoopViz 56
5.3 Single level visualization algorithm 58

5.4 Merging intermediate images 60
5.5 A sample three-level image 63
5.6 Merging phase 64
5.7 Coarse-grained pyramid partitioning 65
5.8 Smoothing of road segments 70
5.9 Six case studies implemented in HadoopViz 72
5.10 Road network visualization 73
5.11 Steps of frequency heat map (Best viewed in colors) 74
5.12 Single Level Image Performance 78
5.13 Single-level image tuning with case study II 81
5.14 Multilevel Image Performance 83
5.15 Multilevel image performance tuning with case study II 84
6.1 Structure of LP DAAC archive 90
6.2 Overview of SHAHED 92
6.3 An example of hole recovery with a heat map 94
6.4 Two-dimensional interpolation 95
6.5 Spatio-temporal aggregate index 97
6.6 Aggregate spatial index 98
6.7 A heat map of temperature in the whole world generated by SHAHED . 105
6.8 Heat map of temperature viewed on Google Earth 106
6.9 A multi-level heat map displayed on Google Maps 108
6.10 Pyramid of images generated for interactive heat map 110
6.11 Performance Experiments 111
6.12 Compare the temperature at two points 114
6.13 Web interface 115

Chapter 1

Introduction

Recently, there has been an explosion in the amounts of spatial data produced by several devices such as smart phones, space telescopes, medical devices, among others. For example, space telescopes generate up to 15 GB weekly spatial data [1], medical devices produce spatial images (X-rays) at a rate of 50 PB per year [2], a NASA archive of satellite earth images has more than 1 PB and is increased daily by 150 GB [3], while there are 10 million geotagged tweets issued from Twitter every day as 2% of the whole Twitter firehose [4, 5]. This variety in the data makes it widely used across different applications. For example, the Blue Brain Project [6] studies the brain's architectural and functional principles through modeling brain neurons as spatial data. Epidemiologists use spatial analysis techniques to identify cancer clusters [7], track infectious disease [8], and drug addiction [9]. Meteorologists study and simulate climate data through spatial analysis [10]. News reporters use geotagged tweets for event detection and analysis [11].

As a result of this surge in the size spatial data along with its practical applications, researchers and practitioners worldwide have started to take advantage of big data frameworks, e.g., Hadoop and Spark, in supporting large-scale spatial data. Most notably, in industry, ESRI has released 'GIS Tools on Hadoop' [12] that works with their flagship ArcGIS product. Meanwhile, in academia, several system prototypes were proposed including: (1) Parallel-Secondo [13] as a parallel spatial DBMS that uses Hadoop as a distributed task scheduler, (2) $\mathcal{MD}$-HBase [14] extends HBase [15], a non-relational database for Hadoop, to support multidimensional indexes, and (3) Hadoop-GIS [16] extends Hive [17], a data warehouse infrastructure built on top of Hadoop with a uniform grid index for range queries and self-join.

A main drawback in all these systems is that they still deal with Hadoop as a black box, and hence they remain limited by the limitations of existing Hadoop systems. For example, Hadoop-GIS [16], while the most advanced system prototype so far, suffers from the following limitations: (1) Hadoop itself is ill equipped in supporting spatial data as it deals with spatial data in the same way as non-spatial data. Relying on Hadoop as a black box inherits the same limitations and performance bottlenecks of Hadoop. Furthermore, Hadoop-GIS adapts Hive [17], a layer on top of Hadoop, which gives an extra overhead layer over Hadoop itself, (2) Hadoop-GIS can only support uniform grid index, which is applicable only in the rare case of uniform data distribution. (3) Being on-top of Hadoop, MapReduce programs defined through *map* and *reduce* cannot access the constructed spatial index. Hence, users cannot define new spatial operations beyond the already supported ones, range query and self-join. Parallel Secondo [13], $\mathcal{MD}$-HBase [14], and ESRI tools on Hadoop [12] suffer from similar drawbacks.

In this thesis, we introduce SpatialHadoop; a full-fledged MapReduce framework with native support for spatial data; available as open-source [18]. SpatialHadoop overcomes the limitations of Hadoop-GIS and all previous approaches as: (1) SpatialHadoop is *built-in* Hadoop base code (around 45,000 lines of code) that pushes spatial constructs and the awareness of spatial data inside the core functionality of Hadoop. This is a key point behind the power and efficiency of SpatialHadoop. (2) SpatialHadoop is able to support a set of spatial index structures including R-tree-based and Quad-tree-based indexes, which are built-in Hadoop Distributed File System (HDFS). This makes SpatialHadoop unique in terms of supporting various skewed data distributions in spatial data, and (3) SpatialHadoop users can interact with Hadoop directly to develop a myriad of spatial functions. For example, in this thesis, we show range queries, *kNN* queries, and spatial join operations, while in a separate work it has been used to build computational geometry operations [19]. This is in contrast to Hadoop-GIS and other systems that cannot support such kind of flexibility, and hence they are very limited in the functions they can support. SpatialHadoop is available as open source [18] and has been downloaded more than 80,000 times within one year of its release. It has been used by several research labs and industrial companies around the world.

SpatialHadoop to Hadoop is the same as spatial database management systems

```
Objects  =  LOAD 'points' AS (id:int, x:int, y:int);
Result   =  FILTER Objects BY   x < x2 AND x > x1
                                 AND y < y2 AND y > y1;
```

(a) Range query in Hadoop

```
Objects  =  LOAD 'points' AS (id:int, Location:POINT);
Result   =  FILTER Objects BY
              Overlaps (Location, Rectangle(x1, y1, x2, y2));
```

(b) Range query in SpatialHadoop

Figure 1.1: Range query in Hadoop vs. SpatialHadoop

(SDBMS) [20] to traditional DBMS, where the original functionality of Hadoop and DBMS is still preserved with the addition of a native support of spatial data. As relational DBMS was ill equipped for spatial data, SDBMS has stepped forward to provide orders of magnitude performance improvement for spatial data processing through spatial operations and spatial index structure. SpatialHadoop follows a similar approach where new spatial data types, spatial index structures, and spatial operators are provided as built-in functionality in Hadoop while preserving traditional functionality of Hadoop.

Figures 1.1(a) and 1.1(b) show how to express a spatial range query in Hadoop and SpatialHadoop, respectively. The query finds all points located within a rectangular area represented by two corner points ⟨*x1*, *y1*⟩ and ⟨*x2*, *y2*⟩. The first query statement loads an input file of *points*, while the second statement selects records that overlap with the given range. As Hadoop does not have any spatial indexes, it has to scan the whole dataset to answer the range query, which gives a very bad performance. In particular, it takes 200 seconds on a 20-node Hadoop cluster to process a workload of 60 GB (about 70 M spatial objects). On the other side, SpatialHadoop exploits its built-in spatial indexes to run the same query in about two seconds, which is two orders of magnitude improvement over Hadoop. In addition, the Hadoop program, written in Pig Latin language [21], is less readable due to the lack of spatial data support. SpatialHadoop uses Pigeon [22] language which makes the program simpler and more expressive as it

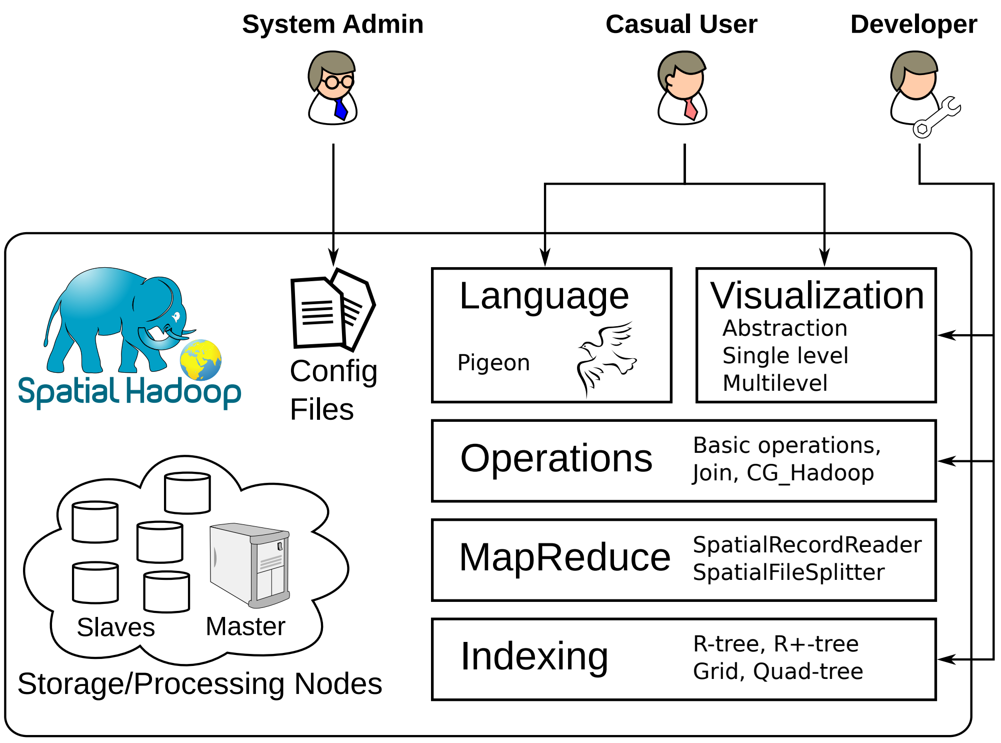


Figure 1.2: Overview of SpatialHadoop

uses spatial data types (`POINT` and `RECTANGLE`) and spatial functions (`Overlaps`).

1.1 Overview of SpatialHadoop

Figure 1.2 gives an overview of SpatialHadoop. SpatialHadoop runs on a cluster containing one master node, that breaks a MapReduce job into smaller tasks, and multiple slave nodes that carry out these tasks. Three types of users interact with SpatialHadoop: (1) *Casual users* who access SpatialHadoop through a spatial high-level language to process their datasets. (2) *Developers*, who have deeper understanding of the system and can add new spatial indexes, implement new operations, or define new visualization types. (3) *Administrators*, who can tune up the system by adjusting system parameters in the configuration files. The core of SpatialHadoop consists of five main layers, namely, *language*, *indexing*, *MapReduce*, *operations*, and *visualization*, described briefly below.

1. The **Language** layer contains **Pigeon** [22], a high level language with OGC-compliant spatial data types and functions. Pigeon provides a simple interface of SpatialHadoop for non-technical users which makes it easier to use without having to delve into the details of SpatialHadoop. The details of Pigeon are discussed in Chapter 2.

2. The **Indexing** layer provides standard spatial indexes, such as grid, R-tree, R+-tree, and Quad-tree, which are used to store the data in an efficient way in the Hadoop Distributed File System (HDFS). Indexes are organized in two-layers, one *global index* that partitions data across nodes, and multiple *local indexes* to organize records inside each node. The indexing layer is described in more details in Chapter 3.

3. The **MapReduce** layer introduces two new components that act as access methods to the underlying spatial indexes. The two new components, namely, *SpatialFileSplitter* and *SpatialRecordReader*, allow MapReduce programs to access the global and local indexes, respectively. The MapReduce layer is further discussed in Chapter 4.

4. The **Operations** layer encapsulates the spatial operations supported by SpatialHadoop. This includes *basic operations*, *join operations*, and *CG_Hadoop* [19] which is a suite of fundamental computational geometry operations. Developers and researchers can enrich this layer by implementing more advanced spatial operations. The spatial operations are further explained in Chapter 4.

5. The **Visualization** layer provides efficient algorithms to visualize big spatial data by generating images that give a bird's-eye view to the data. SpatialHadoop provides an extensible visualization framework, termed HadoopViz [23, 24], which exposes an extensible interface capable of visualizing a wide range of image types, e.g., scatter plot, road network, heat maps, and satellite data. Furthermore, HadoopViz supports both *single level* images, which are generated at a fixed resolution, and *multilevel* images, which are generated at multiple resolutions to allow users to zoom in. HadoopViz is described in details in Chapter 5.

The core of SpatialHadoop is designed to serve as a backbone for applications that deal with large scale data processing. There are many applications which uses Spatial-Hadoop as a core component to handle big spatial data including, SHAHED [25, 26, 27], a system for querying and visualizing spatio-temporal satellite data; TAREEG [28, 29], a web-based extractor for OpenStreetMap data; MNTG [30, 31], a web service for generating moving objects datasets; and GISQF [32, 33], an analyzer for world-wide events data. Chapter 6 will describe the implementation of SHAHED in details.

The goal of this thesis is to describe how Hadoop, as a big data framework, can be modified in its core to provide orders of magnitude better performance for spatial data handling. This document describes the full stack from the storage and indexing, at the bottom level, and up to the high level language and visualization. It is imperative to mention that, while this thesis uses Hadoop as a case study for big data frameworks, the concepts described here can be ported to other big data frameworks, with some necessary changes, to provide a better support for spatial data. For example, Sphinx [34], a recently introduced distributed spatial DBMS, extends Impala [35] to support SQL queries for spatial data. Sphinx uses the same spatial indexes built by SpatialHadoop and provides a different query processing layer that builds on the SQL query processing of Impala.

Chapter 2

Pigeon: A Spatial MapReduce Language

Dealing with Hadoop through MapReduce programs is cumbersome for non-technical users. Thus, several SQL-like high level languages were developed including Pig Latin[21], HiveQL[17] and Y-Smart[36] to simplify large scale data processing in Hadoop. These languages allow users to describe their programs in terms of standard operations such as filter, sort and join. These primitive operations are combined together to perform more complex operations the same way it is done in SQL. Unfortunately, existing languages do not support spatial data types or functions making it difficult to analyze large scale spatial datasets.

In this chapter, we present Pigeon [22], an extension to Pig Latin to support spatial data processing in SpatialHadoop. Pigeon is compliant with Open Geospatial Consortium (OGC) *Simple Feature Access* standard which is supported in both open source, e.g., PostGIS, and commercial, e.g., Oracle Spatial, spatial DBMS. This makes it easier for users who are familiar with existing SDBMS to migrate their existing code to Pigeon with the least effort. Pigeon supports OGC standard data types including point, linestring and polygon, as well as OGC standard functions for spatial data import/export, querying and manipulation. The spatial functionality is implemented as user defined functions (UDFs) which are seamless to integrate with existing non-spatial operations in Pig and also makes it compatible with all recent versions of Pig

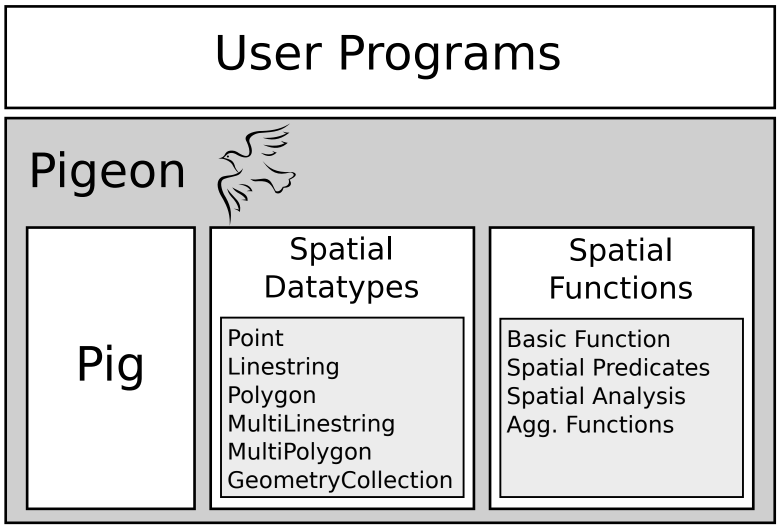


Figure 2.1: Overview of Pigeon

that support UDFs.

In this chapter, we describe how the spatial functionality is implemented in Pigeon and how it can be used to perform complex spatial queries efficiently on a cluster of machines. We also provide a showcase of extracting datasets from OpenStreetMap Planet.osm file which ships as one 500 GB semi-structured XML file. We show how a simple Pigeon script parses the file, extracts different datasets, and stores them back to HDFS in a structured format. After that, we provide more Pigeon scripts to query and analyze the extracted datasets efficiently on the cluster. We provide a range of scripts ranging from a simple query on one relation to more complex queries operating on multiple relations. In all cases, the scripts run efficiently by exploiting the underlying parallelism of the Hadoop framework.

2.1 Overview

Figure 2.1 gives an overview of Pigeon. The two main components of Pigeon are spatial datatypes and spatial functions. Pigeon provides support to the standard OGC datatypes including Point, Linestring, MultiLinestring, Polygon, MultiPolygon, and GeometryCollection. For compatibility with existing OGC-compliant systems (e.g., PostGIS), Pigeon can import spatial objects stored in the Well-Known Text (WKT) and Well-Known Binary (WKB) formats. Details of the datatypes are provided in Section 2.2.

Spatial functions are implemented as user-defined functions (UDFs) which mesh well with existing operations provided in Pig such as filter, join, and group by. There are four groups of spatial functions implemented in Pigeon: (1) Basic spatial methods, (2) Spatial predicates, (3) Spatial analysis, and (4) Aggregate functions. Details of how the spatial functions are implemented and used are given in Section 2.3.

On top of Pigeon, users can write their own MapReduce programs that deal with spatial data. In Section 2.4, we provide a case study where Pigeon is used to extract several spatial datasets out of OpenStreetMap Planet.osm file. It is also shown how Pigeon is used to perform simple and complex spatial analysis queries on the extracted datasets.

2.2 Spatial Datatypes

The first step in a Pig script is to load the input file (relation) and optionally specify its schema defined by a name and a data type for each column. Columns with unspecified types default to `bytearray` which is just the raw bytes stored in that field. Traditional Pig provides support for primitive datatypes such as numbers and strings. It falls short in supporting spatial datatypes such as point and polygon. It does not provide any means of supporting user-defined custom datatypes either. To overcome these limitations, Pigeon overloads the `bytearray` data type to work with spatial data types. The `bytearray` datatype represents the value of a field as a raw array of bytes. When a spatial function is called, the spatial object is retrieved from its binary format, the spatial function is executed on the spatial object, and finally, the result is converted back to binary. This has an advantage of being compatible with the standard Pig but imposes an overhead of converting between binary and geometry back and forth.

When a user loads a spatial file, the spatial column is initially loaded as a *bytearray*. When a spatial function is called, it automatically parses the value in that column to retrieve the spatial object stored in it. Pigeon supports the three common formats used by other tools, namely, Well-Known Text (WKT), Well-Known Binary (WKB) and hex-encoded WKB. To export the value of a spatial column, Pigeon provides three methods `AsBinary`, `AsText` and `AsHex` to convert a spatial object back to one of the three standard formats.

2.3 Spatial Functions

Pigeon utilizes the extensibility of Pig to implement the spatial functions as UDFs which makes it easy to setup and use in an existing Pig installation. Pigeon provides four categories of spatial functions: (1) Basic spatial functions, (2) Spatial predicates, (3) Spatial analysis, and (4) Aggregate functions. A full list of all supported functions can be found in the documents of OGC standards [37].

Basic Spatial Functions

A basic function retrieves some basic information of a single spatial object such as the perimeter length or area. Functions in this category are implemented as simple `Eval` functions where the input is one primitive value and the output is another primitive value. In methods where the input is expected to be a spatial object (e.g., `Area`), the function automatically converts the input from WKT or WKB to a spatial object. Similarly, if the output is a spatial object (e.g., `MakePoint`), the output is converted to WKB. This automatic conversion allows Pigeon to work without the need to define a new datatype for spatial data which is currently not supported by Pig.

Spatial Predicates

A spatial predicate function takes one or two spatial objects and returns a Boolean value based on the relationship of the input object(s) (e.g., IsClosed or Touches). These functions are implemented as simple UDF `Eval` functions which take one or two values and return a Boolean value. Similar to the basic functions, the input objects are automatically converted from WKB or WKT for easy integration with the standard Pig. No need to convert the output value as it is always of type Boolean which is natively supported by Pig.

Spatial Analysis

A spatial analysis function performs some spatial transformation on spatial objects. Some functions are unary (e.g., `Centroid`) while others are binary (e.g., `Intersection`). Functions in this category are implemented as `Eval` functions which take one or two objects as input and return one object as output. Similar to other functions, inputs and outputs are automatically converted to and from spatial objects as needed.

Aggregate Functions

A spatial aggregate function takes a set of spatial objects and returns a single value that summarizes all the input; e.g., the function `ConvexHull` returns one polygon that represents the minimal convex hull of all input spatial objects. These functions are implemented as `Algebraic` aggregate functions which are computed efficiently in Pig using incremental computations. First, the function is applied locally in each machine to compute partial results, then the same function is applied globally on the partial results to produce the final answer. For example, multiple local convex hulls are first computed in each machine, then the global convex hull is computed by combining all local hulls. This technique is very efficient as the local computation step runs in parallel and reduces the data size which speeds up the final step.

2.4 Case-study: OpenStreetMap

As an example of how Pigeon processes large scale spatial datasets, we take OpenStreetMap data as a case study. We start with the planet file which contains all the information about the globe in a semi-structured XML file. First, we show how a simple Pigeon script extracts all the data from the XML file and stores it in a structured format. Then, we show different examples of analyzing the extracted datasets using Pigeon.

Overview of OpenStreetMap Dataset

OpenStreetMap [38] is a project for collecting map information around the world by volunteers. This data is publicly available in the form of one big file (i.e., Planet.osm

Algorithm 1 Planet Extraction

```
nodes = LOAD 'planet.xml' USING XMLLoader('node');
points = FOREACH nodes GENERATE
        id, MakePoint(latitude, longitude) AS location, tags;
STORE points INTO 'points.csv';
nodes = LOAD 'planet.xml' USING XMLLoader('ways');
ways = FOREACH nodes GENERATE way_id, Flatten(nodes), tags;
way_nodes = JOIN points nodes BY id, ways BY node_id;
grouped_ways = GROUP way_nodes BY way_id;
ways_with_geom = FOREACH grouped_ways {
    GENERATE group AS way_id, MakeLine(grouped_ways) AS geom, tags;
}
STORE ways_with_geom INTO 'ways.csv';
```

file) stored in XML format. The planet file has three main sections, namely, *nodes*, *ways*, and *relations*. The case studies presented in this section only deal with the first two sections. The *nodes* section contains a set of points where each point is defined by an ID, latitude, longitude and textual tags. Some points have a physical meaning (e.g., a house or bus stop) while others are just logical points used later in the ways section. The *ways* sections contains a set of ways where each way is defined by an ID, list of points (referenced by their IDs) and textual tags. The list of points define the spatial information of this way while the tags are used to identify its type (e.g., park, lake or road) as well as meta information about this way (e.g., lake name or speed limit). Relations are used to combine several ways and points to represent more complex entities. For example, a university campus can be represented as a relation that contains buildings, roads and bus stops. For the sake of the case studies presented in this section, we only use *nodes* and *ways*.

Planet.osm Extraction

The first task we run using Pigeon is to extract all nodes and ways from OpenStreetMap planet file. To accomplish this task, we run the Pigeon script showed in Algorithm 1. The script runs in two phases, *point extraction* and *way extraction*. *Point extraction* (lines 1-4), starts by loading the XML nodes representing points from the XML file. In line 3, it selects the columns of interest from XML node, namely, ID, latitude, longitude and tags. Notice the use of the new method `MakePoint` to combine the latitude and longitude in one column of type `Point`. Finally, the extracted nodes are stored back to disk in line 4.

Way extraction 5-12 starts by loading the XML nodes representing ways. In line 6, it extracts way ID, tags and flatten the list of node IDs. This statement expands the list of nodes IDs by replicating the whole record for each node ID and setting the column node-id to a different value in the list. This is similar to normalization of 1NF in databases. Line 7 joins the `ways` relation with the `nodes` relation (extracted in phase 1) to substitute each node ID with its geo location. Then, line 8 groups ways by way_id to ensure that all points belonging to one way are grouped together. Lines 9-11, generate a linestring for each way by combining all nodes in this way in one linestring. Here we use the new method `MakeLine` which creates a linestring from a list of points. Finally, results are stored back to disk in line 12.

Filtered Extraction

In this task we extract a subset of the data by applying some spatial and non-spatial filters. First, we can apply a spatial filter to extract data in a specific area defined by a polygon using a `Filter` statement after line 3. For example, the statement:

```
Filter points BY Contains(
  Rectangle(west, south, east, north),
  location);
```

keeps only points contained in the defined rectangle. The inner join operation in line 7 will automatically remove ways that do not have any points in the area of interest. The final result will contain only data in the specified area.

We can also apply a non-spatial filter after line 5 which filters ways according to their tags and leaves only ways with specific tags. This can be used, for example, to extract ways that represent rivers by keeping only ways that has a tag of key ‘waterway’ and value ‘river’. For example, the statement:

```
Filter ways BY tags#'waterway' == 'river';
```

Algorithm 2 Largest city

```
1: cities = LOAD 'cities.csv' AS (city_id: int, city_geom);
2: city_area = FOREACH cities GENERATE city_id, Area(city_geom) AS area;
3: ordered_cities = ORDER city_area BY area DESC;
4: largest_city = LIMIT ordered_cities 1;
```

keeps only ways that represent rivers. We use non-spatial filters to extract datasets that represent roads, rivers, lakes, parks and cities. These datasets are used in analysis tasks that are described next.

Analysis

We use the datasets extracted above to run large scale spatial analysis queries using Pigeon. In the next part, we show three examples of spatial analysis queries. The first example shows a simple spatial function call on one relation. The second example is a traditional spatial join query which operates on two relations. The third example is a more comprehensive example which shows a non-trivial spatial query that operates on two relations. Notice that the three queries can be easily executed in SDBMS using SQL but the advantage of running them in Pigeon is that they automatically scale to hundreds of machines according to the cluster size.

Largest city

This simple example shows how to select the city with the largest area. Simply, it calculates the area of each city, orders them by area and selects top one. The Pigeon script is shown in Algorithm 2 where line 1 loads the cities relation and line 2 calculates the area of each city. After that, lines 3 and 4 order cities by their area in descending order and selects the first one (i.e., largest city).

Overlapping roads and rivers

In this example, we show a typical spatial join query which finds roads that overlap with rivers. As shown in Algorithm 3, this query can be represented easily using Pigeon by computing the cross product of the two relations (line 3) and applying the spatial

Algorithm 3 Overlapping roads and rivers
```
1: roads = LOAD 'roads.csv' AS (road_id: int, road_geom);
2: rivers = LOAD 'rivers.csv' AS (river_id: int, river_geom);
3: cross_prod = CROSS roads, rivers;
4: overlapping = FILTER cross_prod BY Overlap(road_geom, river_geom);
```

Algorithm 4 Lake area per country
```
1: lakes = LOAD 'lakes.csv' AS (lake_id: int, lake_geom);
2: countries = LOAD 'countries.csv AS (country_id: int, country_geom);
3: lake_country = CROSS lakes, countries;
4: overlap_lakes = FILTER lake_country BY Overlap(lakes_geom, country_geom);
5: grouped_lakes = GROUP overlap_lakes BY country_id;
6: total_area = FOREACH grouped_lakes
7:   GENERATE group AS country_id,
8:     Sum(Area(Intersection(overlap_lakes.lakes_geom, overlap_lakes.country_geom)))
   AS lakes_area;
```

predicate (i.e., overlap) as a post processing filter (line 4).

Lake area per country

This example calculates the total area occupied by lakes in each country. The script is shown in Algorithm 4. Lines 1, 2 load the relations of lakes and countries. Notice that lakes are not directly related to countries in OpenStreetMap data. Thus, we need to perform a spatial join step using the overlap predicate (lines 3, 4) to find the lakes which overlap with each country. After that, we group the join result by country ID in line 5. To calculate the total area for each country, we first compute the intersection between each pair of overlapping country and lake using the `Intersection` method to consider only the portion of the lake inside the country. Then, we calculate the area for this portion using the `Area` method. Finally, we sum all the areas using the standard `Sum` method. Notice how multiple spatial functions are nested in one line and how we mix spatial and non-spatial functions in one statement.

Chapter 3

Spatial Indexing in MapReduce

Since input files in traditional Hadoop are non-indexed heap files, the performance is limited as the input has to be scanned. To overcome this limitation, SpatialHadoop employs spatial index structures within Hadoop Distributed File System (HDFS) as a means of efficient retrieval of spatial data. Indexing in SpatialHadoop is the key point in its superior performance over Hadoop.

Traditional spatial indexes, e.g., Grid file, R-tree, and Quad-tree, are not directly applicable in Hadoop due to two challenges: (1) Index structures are optimized for *procedural* programming, where the program executes sequential statements, while SpatialHadoop employs *functional* (MapReduce) programming, where the program is composed of *map* and *reduce* functions executed by slave nodes. The parallel algorithms for R-trees [39] are still procedural where developers control execution of multiple threads. (2) A file in HDFS can be only written sequentially while traditional indexes are constructed incrementally. The bulk loading techniques for Grid file [40] and R-tree [41, 42] are still not applicable to the large datasets used in SpatialHadoop as they either require the whole dataset to fit in memory or incrementally insert records in batches which is not applicable in HDFS.

As a result, existing techniques for spatial indexing in Hadoop fall in three broad categories: (1) *Build only*: A MapReduce approach is proposed to construct an R-tree [43, 44] but the R-tree has to be queried outside MapReduce using traditional techniques. (2) *Custom on-the-fly indexing*: With each query execution, a non-standard index is created and discarded after query completion, e.g., a Voronoi-based index for

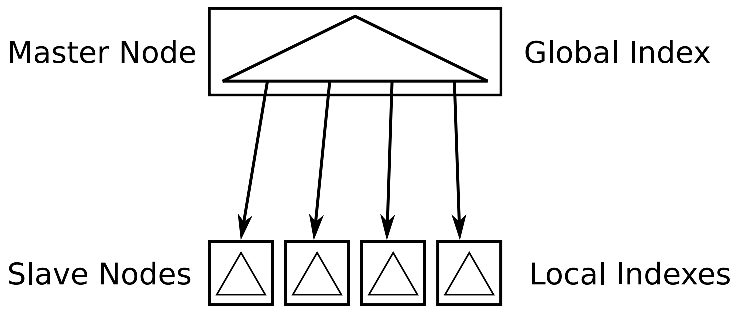


Figure 3.1: Spatial indexing in SpatialHadoop

kNN queries [45, 46] and space-filling-curve-based indexes for range queries [47, 48]. (3) *Indexing in HDFS*: Up to our knowledge, only [49] builds an HDFS-based index structure for a set of trajectories. Yet, the index structure can only support range queries on trajectory data, which is a very limited functionality for what we need in SpatialHadoop.

Spatial indexing in SpatialHadoop falls under the third category as it is built inside HDFS. Yet, unlike all existing approaches, indexing in SpatialHadoop adapts HDFS to accommodate general purpose standard spatial index structures, namely, uniform grid [50], R-tree [51], R+-tree [52], Quad-tree [53, 54], and K-d tree [55], and use them to support many spatial queries written in MapReduce. In the rest of this chapter, Section 3.1 gives an overview of spatial indexing in SpatialHadoop. Then, Section 3.2 describes a generic method for building any spatial index in SpatialHadoop. After that, we show how the generic method is applied to build indexes based on uniform grid, R-tree, R+-tree, Quad-tree, K-d tree, Z-curve, and Hilbert curve in Sections 3.3 to 3.9. Finally, Section 3.10 provides an experimental evaluation to both the quality and performance of the proposed indexes.

3.1 Overview of Indexing in SpatialHadoop

To overcome the challenges of building index structures in Hadoop, we employ a two-layers indexing approach of *global* and *local* indexes as shown in Figure 3.1. Each index contains one *global* index, stored in the Master node, that partitions data across a set of partitions, stored in slave nodes. Each partition has a local index that organizes its own records. Such organization overcomes the above two challenges because: (1) It lends itself to MapReduce programming, where the master node uses the *global* index for job planning, while slaves nodes process the *local* indexes in parallel during job execution, and (2) The small size of local indexes allows each one to be bulk loaded in memory and written to a file in an append-only manner. The following SpatialHadoop shell command can be executed by users to index an input file `src file` using an `index-type` and generate an output file `dst file`, where the index type can be `grid`, `rtree`, `r+tree`, `quadtree`, `kdtree`, `zcurve`, or `hilbert`:

```
index <src file> <dst file> sindex:<index-type>
```

3.2 Index Building

Regardless of the underlying spatial index structure, an index building in SpatialHadoop is composed of three main phases, namely, *partitioning*, *local indexing*, and *global indexing*. Details of each phase may depend on the type of the spatial index.

Phase I: Partitioning

This phase spatially partitions the input file into n partitions that satisfy three main goals: (1) *Block fit*; each partition should fit in one HDFS block of size 64 MB, (2) *Spatial locality*; spatially nearby objects are assigned to the same partition, and (3) *Load balancing*; all partitions should be roughly of the same size. To achieve these goals, we go through the following three steps:

Step 1: Number of partitions. Regardless of the spatial index type, we compute the number of partitions, n, per the equation $n = \left\lceil \frac{S(1+\alpha)}{B} \right\rceil$, where S is the input files size, B is the HDFS block capacity (64 MB), and α is an overhead ratio, set to 0.1 by default, which accounts for the overhead of replicating records and storing local indexes. Overall, this equation adjust the average partition size to be less than B.

Step 2: Partitions boundaries. In this step we decide on the spatial area covered by each single partition defined by a rectangle. To accommodate data with even or skewed distribution, partition boundaries are calculated differently according to the underlying index being constructed. The output of this step is a set of n rectangles representing boundaries of the n partitions, which collectively cover the whole space domain.

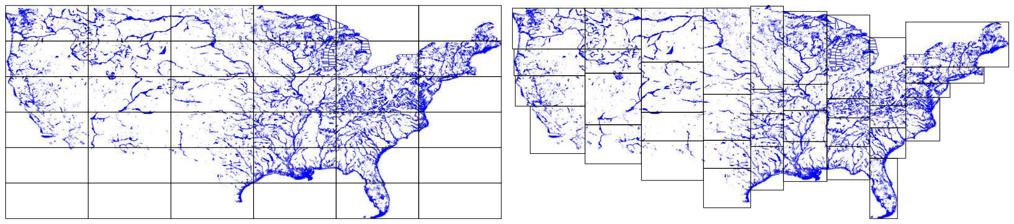
Step 3: Physical partitioning. Given the partition boundaries computed in Step 2, we initiate a MapReduce job that physically partitions the input file. The challenge here is to decide what to do with objects with spatial extents (e.g., polygons) that may overlap more than one partition. Some index structures assign a record to the *best* matching partition, while others replicate a record to all overlapping partitions. Replicated records are handled later by the query processor to ensure a correct answer. At the end, for each record r assigned to a partition p, the *map* function writes an intermediate pair $\langle p, r\rangle$. Such pairs are then grouped by p and sent to the *reduce* function for the next phase, i.e., *local indexing* phase.

Phase II: Local Indexing

The purpose of this phase is to build the requested index structure (e.g., Grid or R-tree) as a *local* index on the data contents of each physical partition. This is realized 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. Each local index has to fit in one HDFS block (i.e., 64 MB) for two reasons: (1) This allows spatial operations written as MapReduce programs to access local indexes where each local index is processed in one map task. (2) It ensures that the local index is treated by Hadoop *load balancer* as one unit when it relocates blocks across machines. According to the partitioning done in the first phase, it is expected that each partition fits in one HDFS block. In case a partition is too large to fit in one block, we break it into smaller chunks of 64 MB each, which can be written as single blocks. To ensure that local indexes remain aligned to blocks after concatenation, each file is appended with dummy data (zeros) to make it exactly 64 MB.

Phase III: Global Indexing

The purpose of this phase is to build the requested index structure (e.g., Grid or R-tree) as a *global* index that indexes all partitions. Once the MapReduce partition job is done, we initiate an HDFS `concat` command which concatenates all local index files into one file that represents the final indexed file. Then, the master node builds an in-memory global index which indexes all file blocks using their rectangular boundaries as the index



(a) Grid Partitioning (b) R-tree Partitioning

Figure 3.2: The Partitioning Phase

key. The global index is constructed using bulk loading and is kept in the main memory of the master node all the time. In case the master node fails and restarts, the global index is lazily reconstructed from the rectangular boundaries of the file blocks, only when required.

3.3 Uniform Grid-based Index

This section describes how the general index building algorithm outlined in Section 3.2 is used to build a grid index. The grid file [50] is a simple flat index that partitions the data according to a grid such that records overlapping each grid cell are stored in one partition which are then grouped in file blocks. For simplicity, we use a uniform grid assuming that data is uniformly distributed. In the *partitioning* phase, after the number of partitions n is calculated, partition boundaries are computed by creating a uniform grid of size $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ in the space domain and taking the boundaries of grid cells as partition boundaries as depicted in Figure 3.2(a). This might produce more than n partitions, but it ensures that the average partition size remains less than the HDFS block size. When physically partitioning the data, a record r with a spatial extent, is replicated to each overlapping grid cell. In the *local indexing* phase, 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. Finally, the *global indexing* phase concatenates all these files and builds the global index, which is a two dimensional directory table pointing to the corresponding blocks in the concatenated file.

3.4 R-tree-based Index

This section describes how the general index building algorithm outlined in Section 3.2 is used to partition spatial data over computing nodes based on R-tree indexing, as in Figure 3.2(b), followed by an R-tree local index in each partition. In the partitioning phase to compute partition boundaries, we bulk load a random sample from the input file to an in-memory R-tree using the Sort-Tile-Recursive (STR) algorithm [42]. The size of the random sample is set to a default ratio of 1% of the input file, with a maximum size of 100MB to ensure it fits in memory. Both the ratio and maximum limit can be set in configuration files. If the file contains shapes rather than points, the centroid of the shape's MBR is used in the bulk loading process. To read the sample efficiently when input file is very large, we run a MapReduce job that scans all records and outputs each one with a probability of 1%. This job also keeps track of the total size of sampled points, S, in bytes. If S is less than 100MB, the sample is used to construct the R-tree. Otherwise, a second sample operation is executed on the output of the first one with a ratio of $\frac{100^{MB}}{S}$, which produces a sub-sample with an expected size of 100MB.

Once the sample is read, the master node runs the STR algorithm with the parameter d (R-tree degree) set to $\lceil\sqrt{n}\rceil$ to ensure the second level of the tree contains at least n nodes. Once the tree is constructed, we take the boundaries of the nodes in the second level and use them in the physical partitioning step. We choose the STR algorithm as it creates a balanced tree with roughly the same number of points in each leaf node. Figure 3.2(b) shows an example of R-tree partitioning with 36 blocks ($d = 6$). Similar to a traditional R-tree, the physical partitioning step does not replicate records, but it assigns a record r to the partition that needs the least enlargement to cover r and resolves ties by selecting the partition with smallest area.

In the *local indexing* phase, records of each partition are bulk loaded into an R-tree using the STR algorithm [42], which is then dumped to a file. The block in a local index file is annotated with its minimum bounding rectangle (MBR) of its contents, which is calculated while building the local index. As records are overlapping, the partitions might end up being overlapped, similar to traditional R-tree nodes. The *global indexing* phase concatenates all local index files and creates the global index by bulk loading all blocks into an R-tree using their MBRs as the index key. Notice that the local indexing

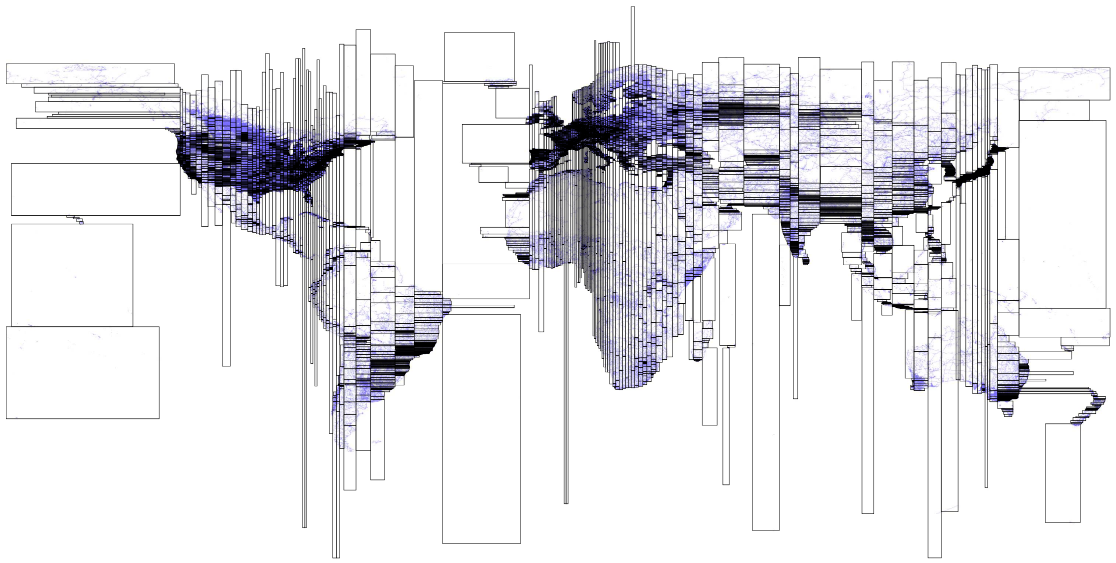


Figure 3.3: R+-tree index of a 400 GB road network dataset

step is optional and users are allowed to build a global-only STR index. Throughout this thesis, we refer to the global-only index as *STR* and the complete global plus local index as *R-tree*.

3.5 R+-tree-based Index

This section describes how the general index building algorithm outlined in Section 3.2 is used to partition spatial data over computing nodes based on R+-tree with an R+-tree local index in each partition. R+-tree [52] is a variation of the R-tree where nodes at each level are kept disjoint while records overlapping multiple nodes are replicated to each node to ensure efficient query answering. The algorithm for building an R+-tree in SpatialHadoop is very similar to that of the R-tree except for three changes. (1) In the R+-tree physical partitioning step, each record is replicated to each partition it overlaps with. (2) In the *local indexing* phase, the records of each partition are inserted into an R+-tree which is then dumped to a local index file. (3) Unlike the case of R-tree, the global index is constructed based on the partition boundaries computed in the partitioning phase rather than the MBR of its contents as boundaries should remain disjoint. These three changes ensure that the constructed index satisfies the properties

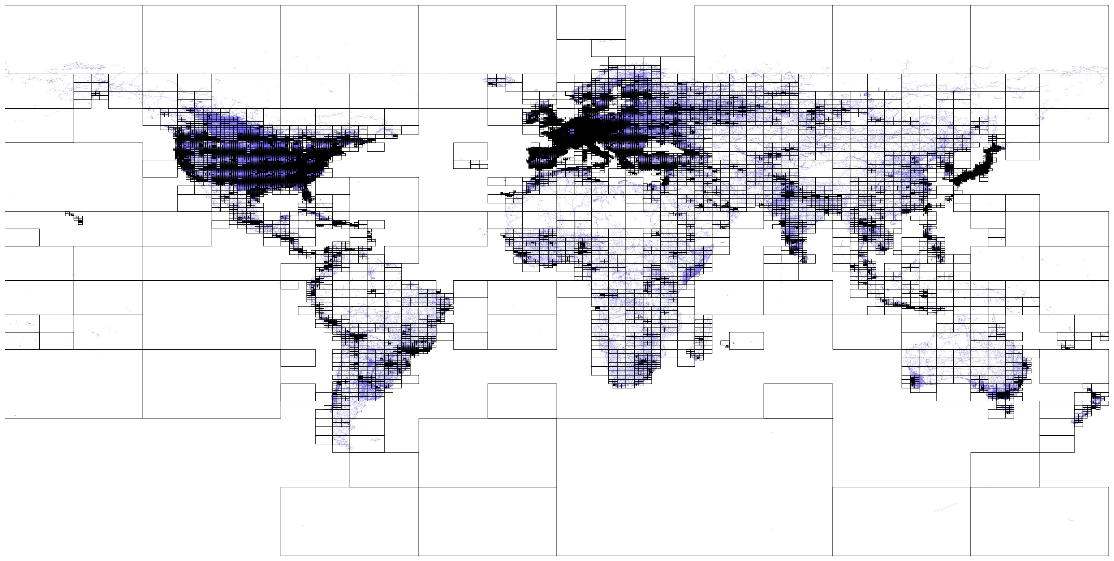


Figure 3.4: A Quad-tree index for a 400GB road network dataset

of the R+-tree. Similar to STR-based index, we refer to the global-only index as *STR+* and the global plus local index as *R+-tree.*

Figure 3.3 gives a prime example behind the need for R+-tree (and R-tree) indexing in SpatialHadoop, which also shows the key behind SpatialHadoop performance. The figure shows the partitioning of an R+-tree index constructed on a 400 GB file that includes all the road networks extracted from OpenStreetMap [38]. The black rectangles in the figure indicate partition boundaries of the global index. While some road segments cross multiple partitions, partition boundaries remain disjoint due to the properties of the R+-tree. As each partition is sufficient for only 64 MB worth of data, we can see that dense areas (e.g., Europe) are contained in very small partitions, while sparse areas (e.g., oceans) are contained in very large partitions. One way to look at this figure is that this is the way that SpatialHadoop divides a dataset of 400 GB into small chunks, each of 64 MB which is the key point behind the order of magnitude performance that SpatialHadoop got over traditional Hadoop.

3.6 Quad-tree-based Index

In this section, we describe how to use the general index building algorithm, outlined earlier, is used to build a Quad-tree-like index in SpatialHadoop as illustrated in Figure 3.4. In this algorithm, the space is partitioned following the Quad-tree [53] partitioning rule, where the space is partitioned into equi-sized quadrants. Each partition is further indexed using a Quad-tree. In the *partitioning phase*, a sample of points is drawn from the input file as described in the R-tree algorithm in Section 3.4. Then, partition boundaries are computed by bulk loading the sample points into an in-memory Quad tree with a leaf node capacity of $\lceil k/n \rceil$ points, where k is the sample size and n is the number of partitions. We use the PR-Quad-tree bulk loading algorithm [56], which works in a bottom up fashion. It starts by sorting all points based on a Z-curve. This ensures that points contained in each leaf node are consecutive in the sort order. After that, the points are scanned in order and each one is added to a list of candidate points for current leaf node. A node is created in one of two cases, (1) number of candidate points reaches the node capacity, or (2) the last candidate point causes the node boundary to intersect an existing leaf node. In these two cases, a new leaf node is created and the points in that node are removed from the candidate list. The boundaries of leaf nodes are used to partition the input dataset.

In the *local indexing* phase, each partition is bulk loaded into a Quad-tree which is flushed to disk in an HDFS block. Since the local Quad-tree might store arbitrary objects (e.g., lines or polygons), we use the PMR-Quad-tree bulk loading algorithm [57] which works in a top-down fashion starting at the root and splitting a node into four quadrants if the node size is beyond the threshold. The *global indexing* phase concatenates all local index files and creates the global index by bulk loading all blocks into a Quad-tree using their MBRs as the index key.

3.7 K-d tree-based Index

In this section, we describe how to build a K-d-tree-like [55] index using the generic algorithm outlined earlier. In this index, the space is recursively partitioned using alternative vertical and horizontal lines. This partitioning technique is also known as binary space partitioning (BSP) [58]. In the *partitioning* phase, a sample of points is drawn from the input file. Then, this sample is used to partition the space into n partitions by applying the K-d-tree partitioning scheme $n - 1$ times.

We start with one K-d tree node that covers the whole input space. At each step, a node is partitioned into two nodes using either a vertical or horizontal line such that number of sample points in the two nodes are balanced. After $n - 1$ step, the tree will contain n leaf nodes which represent the n partitions of the space. To ensure that number of sample points in leaf nodes are balanced, each step partitions a node such that the ratio of points in the two child nodes is equal to the ratio of the two leaf nodes under the two child nodes. This can be easily computed since the structure of the tree is only determined by the number of partition steps. After the boundaries of the n partitions are computed, we use the *replication* mechanism to partition the data records which replicates a boundary object to all overlapping partitions.

3.8 Z-curve-based Index

This section describes how the generic indexing algorithm is used to build a Z-curve-based index [59]. In the partitioning phase of the indexing algorithm, the sample points are sorted based on their order on the Z-curve. Then, the sorted list of points is split into n sublists of equal size where each sublist corresponds to a partition. The boundaries of each partition is set to the boundaries of the corresponding sublist defined in the domain of the Z-curve. While the actual records are physically partitioned, each record is matched with exactly on of the partitions by taking the centroid of its MBR and locating it into one of the partitions based on its order on the Z-curve. After all records are assigned to partitions, the MBR of each partition is computed based on all the records that were assigned to that partition. Notice that due to the partial loss of spatial locality in the Z-curve, many of the induced partitions might overlap or even contain each other.

3.9 Hilbert curve-based Index

Building an index based on the Hilbert curve [59] is very similar to the one with the Z-curve but it uses a Hilbert curve instead of the Z-curve for sorting the sample points.

Parameter	Values (default)
HDFS block capacity (B)	32, (64), 128, 256 MB
Cluster size (N)	5, 10, 15, (20), 25, 35, 100
Sample ratio (ρ)	(0.01), 0.02, 0.05, 0.1, 0.2, 0.5, 1.0

Table 3.1: Experiments Parameters

The Hilbert curve is a little bit more complex to compute as compared to the Z-curve, however, it is known to hold better spatial properties as it completely avoids the long jumps on the Z-curve.

3.10 Experiments

This section provides an extensive experimental study for the performance of the index creation in SpatialHadoop. The purpose of this part is to show the scalability of the index creation as well as the effect of the different system parameters, e.g., HDFS block size and sampling size, on the quality of the generated index. Chapter 4 will show how these indexes provide orders of magnitude better performance on the different spatial operations that run in SpatialHadoop.

Experimental Setup

Cluster Setup. All experiments run on Amazon EC2 ‘`m1.large`’ instances which have a dual core processor, 7.5 GB RAM and 840 GB disk storage. We use Hadoop 1.2.1 running on Java 1.6 and CentOS 6. Each machine is configured to run three mappers and two reducers. Table 3.1 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

Datasets. Table 3.2 summarizes the datasets we use in our experiments which are collected from four different sources: (1) TIGER: Real datasets which are collected and made available by the US Census Bureau [60] and they represent spatial features in the US, such as streets and rivers. (2) OSM: Real datasets which are extracted from OpenStreetMap [38] and they represent map data for the whole world. (3) NASA: Remote sensing data which represents vegetation indices for the whole world in a year

Source	Name	Size	Records	Average Record Size
OSM	`All Objects`	88GB	250M	378 bytes
	`Buildings`	25GB	109M	234 bytes
	`Roads`	23GB	70M	337 bytes
	`Lakes`	8.6GB	9M	1KB
	`Cities`	1.4GB	170K	8.4KB
TIGER	`Edges`	62GB	73M	916 bytes
	`Linear Water`	18.3GB	5.9M	3.2KB
NASA	`Vegetation`	4.6TB	120B	42 bytes
SYNTH	`Rectangles`	120GB	2B	64 bytes

Table 3.2: Experiments datasets

as one snapshot every 15 days. (4) SYNTH: A synthetic dataset generated in an area of 1M $\times$ 1M units, where each record is a rectangle of maximum size $d \times d$; d is set to 100 by default. The location and size of each record are both generated based on a uniform distribution. We generate up to 2 Billion rectangles of total size 128 GB. To allow researchers to repeat the experiments, we make the first two datasets available on SpatialHadoop website. The third dataset is already made available by NASA [61]. The generator is shipped as part of SpatialHadoop and can be used as described in its documentation.

Performance Metrics. To measure and compare the performance of the different spatial indexing techniques, we use five measures for the index quality and one measure for the performance of the indexing. The five **quality measures** are computed on the index data to assess the quality of the index. For simplicity, we only consider the global index while measuring these five measures. Four of them, Q1-Q4, are derived from the R*-tree optimization criteria, where they were shown to correlate with the performance of the range query [62]. Q1 is the total area occupied by all partitions and is used as an indicator of the *dead space* covered by partitions without containing any actual records. Q2 is the total overlap between pairs of partitions. Q3 is the total *margin* of all partitions where the margin of a rectangle is the sum of width and height. For a fixed area, minimizing the margin favors squares over rectangles. Q4 is the disk utilization measured as the ratio between actual data in partitions and the total capacity of all occupied file system blocks. Finally, Q5 is the standard deviation of partition sizes

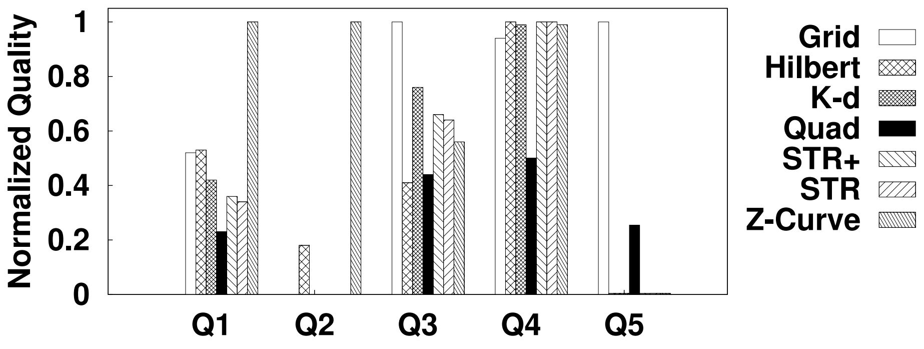


Figure 3.5: Five Quality Measures for `Buildings`

and is used to measure the load balance or skewness across partitions. To asses the scalability and performance of the spatial indexing process, we measure the end-to-end wall-clock time spent by the cluster to construct the index.

Quality Measures

Figure 3.5 shows the five quality measures for the `Buildings` dataset. All these measures are computed based on the global index only. All values are normalized to the range $[0, 1]$ to keep the figure concise. According to Q1, Quad tree partitioning gives the best performance as it keeps the overall area limited by regular space partitioning and pruning empty partitions. On the other hand, Z-curve partitioning is the worst as it generates a lot of overlap between partitions due to the huge jumps in the curve. Q2 does not seem to be very helpful as all *replication*-based techniques generate disjoint partitions, which always produce Q2=0. Unlike Q2, the variance in Q3 is high and, surprisingly, Hilbert curve provided the best value with the Quad tree being very close. This shows that there is a room for improvement if we apply more sophisticated partitioning techniques. It is interesting that both *space partitioning* techniques perform relatively poor in Q4 and Q5. The reason is that they employ regular space partitioning, which is susceptible to producing nearly-empty partitions (examples starred in Figure 3.2 a&b) which is adversary to both disk utilization (Q4) and skewness (Q5). In fact, the value of Q5 is too small to notice in other techniques as they have the freedom to accurately adjust partition boundaries to maximize utilization and minimize skewness. In a future work, we can try to combine several small partitions into one bigger partition to improve Q5, and study its effect on other quality measures.

In Figure 3.6(a), we measure Q1 while varying the sampling ratio ρ from 1% to 100% for the `Cities` dataset. Surprisingly, the quality measure is hardly affected by the sampling ratio. We observe a similar behavior across different datasets, partitioning techniques, and quality measures (except Q5). This finding might seem counter-intuitive because drawing a larger sample should reduce the error caused by sampling and increase the quality of the partitioning. However, there are two reasons that make this finding accountable. First, each partitioning technique has its own inherent limitations which impose some upper bound on its quality. For example, Z-curve partitioning would generally produce overlapping partitions due to the loss of locality in the Z-curve even if it operates on the whole file. From this experiment, it looks like that this upper bound is easily reached for the evaluated partitioning techniques even with a 1% sample. The second reason is that records are converted to points as they are sampled to make the in-memory bulk-loading step simpler and more efficient. This means that even when ρ =100%, there is still some approximation of shapes into points. We believe that this is a very important finding because it tells that the limitations of these partitioning techniques are not caused by the random sampling.

The only exception to the previous finding is with Q5 (skewness) which actually improves for most partitioning techniques as the sample ratio increases, as shown in Figure 3.6(b). There are two interesting findings in this figure. First, *space partitioning* techniques are hardly affected by the sample ratio. Grid partitioning does not use the sample at all while the Quad tree uses the sample only to decide which tree nodes to split, which is not much affected by the sample size. Second, as the sample size approaches 100%, the *distribution*-based techniques achieve a near-perfect load balancing, as opposed to *replication*-based techniques. The reason is that the effect of replication is not taken into account while subdividing the space using the sample of points.

Figures 3.6(c) and 3.6(d) show the tradeoff between partitioning quality and disk utilization as we increase the partition size for `Buildings`. This tradeoff is well known in the literature and this experiment confirms that it still holds in MapReduce environments.

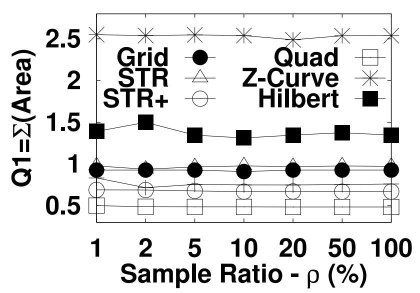
Finally, Figure 3.6(e) shows the value of the quality measure Q1 for all datasets. Although we cannot really compare the quality of different datasets as they have different characteristics, we can still observe that the relative quality between techniques is consistent across datasets. In addition, we see a huge drop for the *distribution*-based techniques with `All Objects` (largest dataset) as they have to greatly expand partitions to enclose very large objects (e.g., countries borders). This does not happen with *replication*-based techniques which do not expand partitions boundaries. This is an important finding which helps users in choosing a partitioning technique for a dataset depending on the shape of the objects.

Indexing Time

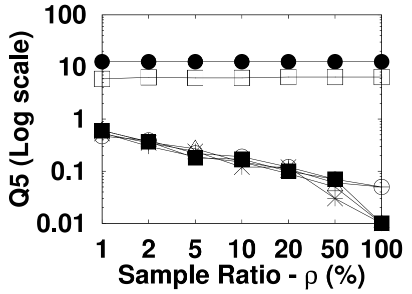
Figure 3.7 shows the performance of index construction. Figure 3.7(c) shows the overall partitioning time for different OSM datasets and various indexes. An interesting finding is that the performance of all techniques is very similar as the partitioning is dominated by the MapReduce job that scans the whole file. The main difference between all techniques is in the in-memory step which operates on a small sample. This opens the space to incorporate more complicated techniques. Similarly, Figure 3.7(a) shows the performance of index construction for the SYNTH dataset as the input size is increased from 1 to 128 GB.

To take SpatialHadoop to an extreme, we test it with NASA datasets of up to 4.6TB and 120 Billion records on a 100-node cluster of Amazon '`large`' instances. Figure 3.7(b) shows the indexing time for an R-tree index. As shown, it takes less than 15 hours do build a highly efficient R-tree index for a 4.6 TB dataset which renders SpatialHadoop very scalable in terms of data size and number of machines. Note that building the index is a one time process for the whole data, after which the index lives for long.

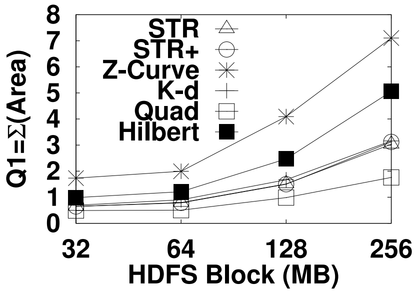
Finally, to test the scalability of the index construction algorithm with the cluster size, Figure 3.7(d) shows the indexing time, for `Buildings`, as the number of machines in the cluster increases from 5 to 20. In general, the index construction algorithm provides a near-perfect linear scale out with the cluster size. Although the sampling step takes less than two minutes, it does not scale as good as the partitioning step due to the fixed overhead associated with each map task in the MapReduce job. This calls for more efficient sampling techniques in terms of performance.



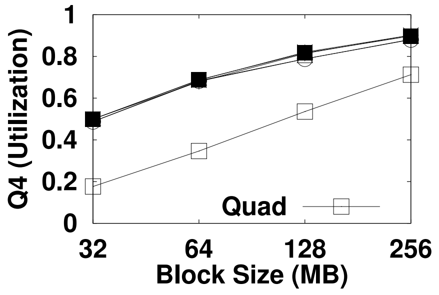
(a) Q1 for `Cities`



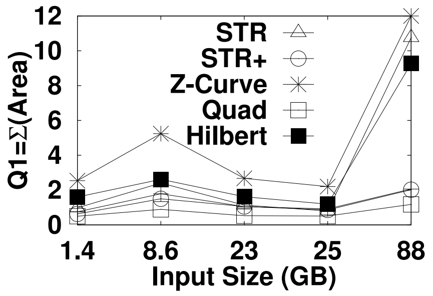
(b) Q5 for `Cities`



(c) Block Size - Q1

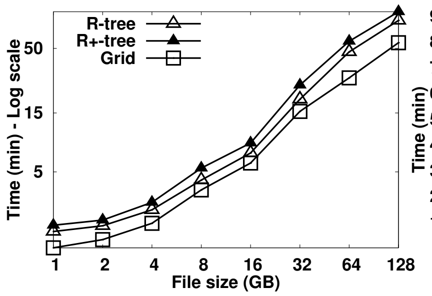


(d) Block Size - Q4

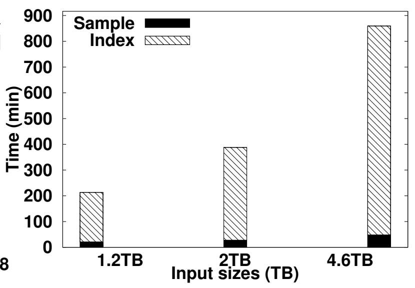


(e) Q1 - All Datasets

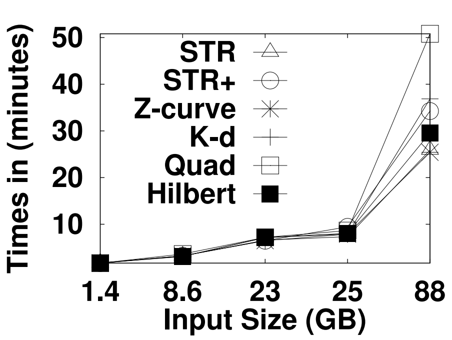
Figure 3.6: Quality Measures



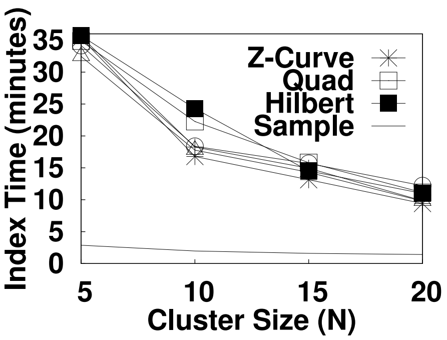
(a) SYNTH data



(b) NASA data



(c) OSM



(d) Scalability

Figure 3.7: Performance of index construction

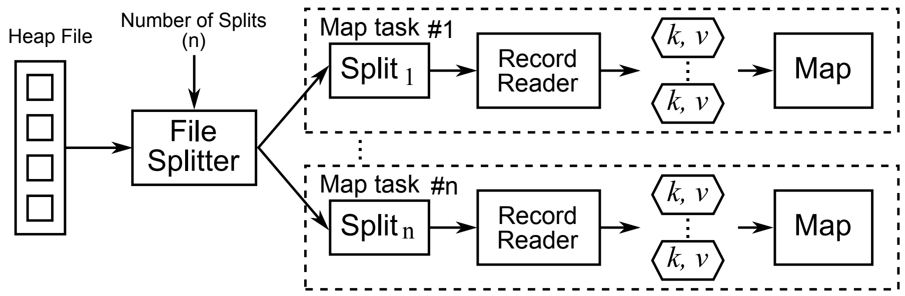
Chapter 4

Spatial Query Processing in MapReduce

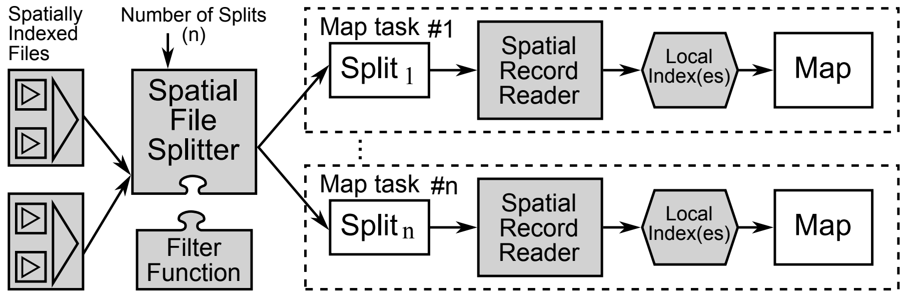
The spatial indexes described in Chapter 3 are used by SpatialHadoop to provide orders of magnitude performance speedup to several spatial operations. Rather than supporting a rigid set of spatial operations, SpatialHadoop provides an extensible core that can be used by developers to add new spatial operations as MapReduce programs. In this chapter, Section 4.1 describe two new components, namely, *SpatialFileSplitter* and *SpatialRecordReader*, which are added to the MapReduce layer and act as access methods to the *global* and *local* indexes, respectively. Section 4.2 gives three examples of spatial operations, namely *range query*, *k-nearest neighbor* (*kNN*), and *spatial join*, which are implemented using the new components. Section 4.3 provides an extensive experimental evaluation of the proposed operations on the different spatial indexes supported by SpatialHadoop.

4.1 MapReduce Layer

Similar to Hadoop, the MapReduce layer in SpatialHadoop is the query processing layer that runs MapReduce programs. However, contrary to Hadoop where the input files are non-indexed heap files, SpatialHadoop supports spatially indexed input files. Figure 4.1 depicts part of the MapReduce plan in both Hadoop and SpatialHadoop, where the modifications in SpatialHadoop are highlighted. In Hadoop (Figure 4.1(a)), the input



(a) Hadoop



(b) SpatialHadoop

Figure 4.1: Map phase in Hadoop and SpatialHadoop

file goes through a *FileSplitter* that divides it into n splits, where n is set by the the MapReduce program, based on the number of available slave nodes. Then, each split goes through a *RecordReader* that extracts records as key-value pairs which are passed to the *map* function. SpatialHadoop (Figure 4.1(b)) enriches traditional Hadoop systems by two main components: (1) *SpatialFileSplitter* (Section 4.1); an extended splitter that exploits the global index(es) on input file(s) to early prune file blocks not contributing to answer, and (2) *SpatialRecordReader* (Section 4.1), which reads a split originating from spatially indexed input file(s) and exploits the local indexes to efficiently process it.

SpatialFileSplitter

Unlike the traditional file splitter, which takes one input file, the *SpatialFileSplitter* can take one or two input files where the blocks in each file are globally indexed. In addition, the *SpatialFileSplitter* takes a filter function, provided by the developer to filter file blocks that do not contribute to answer. A single file input represents unary operations, e.g., range and k-nearest-neighbor queries, while two input files represent binary operations, e.g., spatial join.

In case of one input file, the *SpatialFileSplitter* applies the filter function on the global index of the input file to select file blocks, based on their MBRs, that should be processed by the job. For example, a range query job provides a filter function that prunes file blocks with MBRs completely outside the query range. For each selected file block in the query range, the *SpatialFileSplitter* creates a file split, to be processed later by the *SpatialRecordReader*. In case of two input files, e.g., a spatial join operation, the behavior of the *SpatialFileSplitter* is quite similar with two subtle differences: (1) The filter function is applied to *two global indexes*; each corresponds to one input file. For example, a spatial join operation selects pairs of blocks with overlapping MBRs. (2) The output of the *SpatialFileSplitter* is a *combined split* that contains a pair of file ranges (i.e., file offsets and lengths) corresponding to the two selected blocks from the filter function. This combined split is passed to the *SpatialRecordReader* for further processing.

SpatialRecordReader

The *SpatialRecordReader* takes either a split or combined split, produced from the *SpatialFileSplitter*, and parses it to generate key-value pairs to be passed to the map function. If the split is produced from a single file, the *SpatialRecordReader* parses the block to extract the local index that acts as an access method to all records in the block. Instead of passing records to the map function one-by-one as in traditional Hadoop record readers, the record reader sends all the records to the map function indexed by the local index. This has two main benefits: (1) it allows the map function to process all records together, which is shown to make it more powerful and flexible [45], and (2) the local index is harnessed when processing the block, making it more efficient than scanning over all records. To adhere with the key-value record format, we generate a key-value pair $\langle mbr, lindex \rangle$, where the key is the MBR of the assigned block and the value is a pointer to the loaded local index. The traditional record reader is still supported and can be used to iterate over records in case the local index is not needed.

In case the split is produced from two spatially indexed input files, the *SpatialRecordReader* parses the two blocks stored in the combined split and extracts the two local indexes in both blocks. It then builds one key-value pair $\langle\langle mbr_1, mbr_2\rangle, \langle lindex_1, lindex_2\rangle\rangle$, which is sent to the map function for processing. The key is a pair of MBRs, each corresponds to one block, while the value is a pair of pointers, each points to a local index extracted from a block.

4.2 Operations Layer

The combination of the spatial indexing with the new spatial functionality in the MapReduce layer (Section 4.1) gives the core of SpatialHadoop that enables the possibility of efficient realization of a myriad of spatial operations. In this section, we only focus on three fundamental spatial operations: range query (Section 4.2), *kNN* (Section 4.2), and spatial join (Section 4.2), as three case studies of how to use SpatialHadoop. The same efficient core of SpatialHadoop has already been used to build other spatial operations such as computational geometry [19]. Developers and researchers can add more operations, such as *kNN* join and RNN, by following similar approaches.

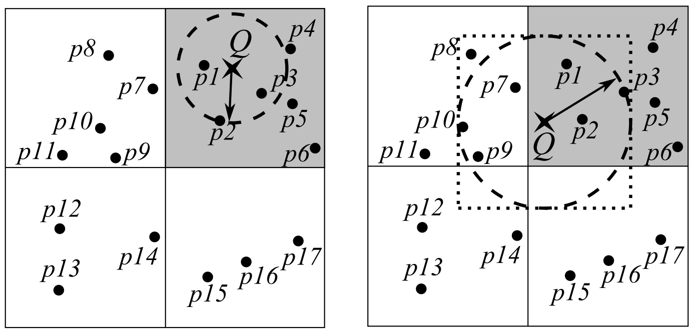
Range Query

A range query takes a set of spatial records R and a query area A as input, and returns the set of records in R that overlaps with A. In Hadoop, the input set is provided as a heap file, hence, all records have to be scanned to output matching records. SpatialHadoop achieves orders of magnitude speedup by exploiting the spatial index. There are two range query techniques implemented in SpatialHadoop depending on whether the index replicates entries or not.

No replication. In the cases of R-tree, Z-curve, and Hilbrt-curve indexes, where each record is stored in exactly one partition, the range query algorithm runs in two steps. (1) The *global filter* step, which selects file blocks that need to be processed. This step exploits the global index through feeding the *SpatialFileSplitter* with a range filter to select those blocks overlapping with the query area A. Blocks that are completely inside the query area are copied to the output without any further processing as all its records are within the query range. Blocks that are partially overlapping are sent for further processing in the second step. (2) The *local filter* step operates on the granularity of a file block and exploits the local index to return records overlapping with the query area. The *SpatialRecordReader* reads a block that needs to be processed, extracts its local index, and sends it to the map function, which exploits the local index with a traditional range query algorithm to return matching records.

Replication. In the cases of grid, R+-tree, Quad-tree, and K-d tree, where some records are replicated across partitions, the range query algorithm differs from the one described above in two main points: (1) In the global filter step, blocks that are completely contained in the query area A have to be further processed as they might have duplicate records that need to be removed. (2) The output of the local filter goes through an additional *duplicate avoidance* [63] step to ensure that duplicates are removed from the final answer. For each candidate record produced by the local filter step, we compute its intersection with the query area. A record is added to the final result only if the top-left corner of the intersection is inside the partition boundaries. Since partitions are disjoint, it is guaranteed that only one partition contains that point. The output of the duplicate avoidance step gives the final answer of the range query, hence, no reduce function is needed.

Single-machine Implementation. Hadoop imposes a significant overhead, up-to ten seconds, to any MapReduce job which makes it unsuitable for interactive jobs. Therefore, SpatialHadoop provides an alternative implementation that bypasses the MapReduce layer and processes the range query by directly accessing the index in HDFS. The algorithm runs in two or three steps, depending on whether the index is replicated or not, similar to the MapReduce implementations. The *global filter* step reads the global index from the master node and runs an in-memory range query to retrieve overlapping partitions. The *local filter* step processes the matching partitions, in parallel threads, where each thread processes the local index in a partition to retrieve matching records. Finally, if the index is replicated, the *duplicate avoidance* step runs, in parallel threads, to remove duplicate answers. SpatialHadoop employs a simple cost-model to choose between the single-machine and MapReduce implementations. Simply, it employs the single-machine algorithm if the number of matching partitions is no more than the number of cores available in the processor of the single machine. Otherwise, it launches the MapReduce algorithm.



(a) Correct initial answer (b) Refined Answer

Figure 4.2: *kNN* query ($k = 3$) in SpatialHadoop

k Nearest Neighbor (kNN)

A *kNN* query takes a set of spatial points P, a query point Q, and an integer k as input, and returns the k closest points in P to Q. In Hadoop, a *kNN* algorithm scans all points in the input file, calculates their distances to Q, and produces the top-k ones [64]. With SpatialHadoop, we exploit simple pruning techniques that achieve orders of magnitude better performance than that of Hadoop. A *kNN* query algorithm in SpatialHadoop is composed of the three steps: (1) *Initial answer*, where we come up with an initial answer of the k closest points to Q within the same file block (i.e., partition) as Q. We first locate the partition that includes Q by feeding the *SpatialFileSplitter* with a filter function that selects only the overlapping partition. Then, the selected partition goes through the *SpatialRecordReader* to exploit its local index with a traditional *kNN* algorithm to produce the initial k answers. (2) *Correctness check*, where we check if the initial answer can be considered final. We draw a test circle C centered at Q with a radius equal to the distance from Q to its k^{th} furthest neighbor, obtained from the initial answer. If C does not overlap any partition other than Q, we terminate and the initial answer is considered final. Otherwise, we proceed to the third step. (3) *Answer Refinement*, where we run a range query to get all points inside the MBR of the test circle C, obtained from previous step. Then, a scan over the range query result is executed to produce the closest k points as the final answer.

Figure 4.2 gives two examples of a *kNN* query for point Q (in a shaded partition) with k=3. In Figure 4.2a, the dotted test circle C, composed from the initial answer

$\{p_1, p_2, p_3\}$, overlaps only the shaded partition. Hence, the initial answer is considered final. In Figure 4.2b, the circle C intersects other blocks. Hence, a range query is issued with the MBR of C, and a refined answer is produced as $\{p_1, p_2, p_7\}$, where p_7 is closer to Q than p_3.

Spatial Join

A spatial join takes two sets of spatial records R and S and a spatial join predicate θ (e.g., `overlaps`) as input, and returns the set of all pairs $\langle r, s\rangle$ where $r \in R$, $s \in S$, and θ is true for $\langle r, s\rangle$. In Hadoop, the SJMR algorithm [65] is proposed as the MapReduce version of the partition-based spatial-merge join (PBSM) [66]; a classic spatial join algorithm for distributed systems. SJMR employs a map function that partitions input records according to a uniform grid, and then a reduce function that joins records in each partition. Though SJMR is designed for Hadoop, it can still run, as is, on SpatialHadoop, yet with a better performance since the input files are already partitioned. To better utilize the spatial indexes, we equip SpatialHadoop with a novel spatial join algorithm, termed *distributed join* which is composed of three main steps, namely *global join*, *local join*, and *duplicate avoidance*. In some cases, an additional preprocessing step can be added to speed up the distributed join algorithm.

Step 1: Global join. Given two input files of spatial records R and S, this step produces all pairs of file blocks with overlapping MBRs. Apparently, only an overlapping pair of blocks can contribute to the final answer of the spatial join since records in two *non-overlapping* blocks are definitely disjoint. To produce the overlapping pairs, the *SpatialFileSplitter* module is fed with the overlapping filter function to exploit two spatially indexed input files. Then, a traditional spatial join algorithm is applied over the two global indexes to produce the overlapping pairs of partitions. The *SpatialFileSplitter* will finally create a combined split for each pair of overlapping blocks.

Step 2: Local join. Given a combined split produced from the previous step, this step joins the records in the two blocks in this split to produce pairs of overlapping records. To do so, the *SpatialRecordReader* reads the combined split, extracts the records and local indexes from its two blocks, and sends all of them to the map function for processing. The map function exploits the two local indexes to speed up the process of

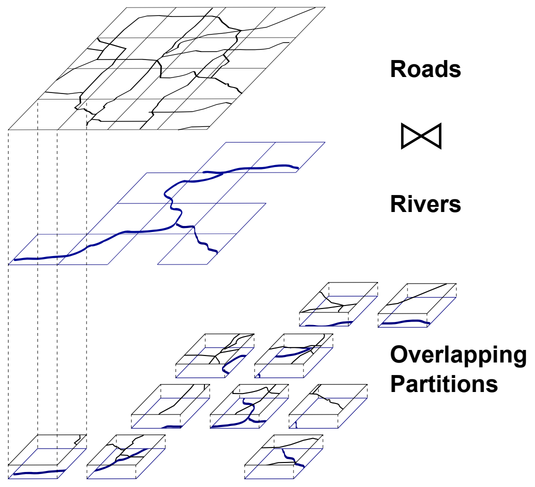


Figure 4.3: *Distributed* join between roads and rivers

joining the two sets of records in the combined split. The result of the local join may contain duplicate results due to having records overlapping with multiple blocks.

Step 3: Duplicate avoidance. Similar to the case of range queries, this step runs only for indexes with replication (i.e., Grid and R+-tree) and employs the reference-point duplicate avoidance technique [63]. For each detected overlapping pair of records, the *intersection* of their MBRs is first computed. Then, the overlapping pair is reported as a final answer only if the top-left corner (i.e., *reference point*) of the intersection falls in the overlap of the MBRs with the two processed blocks.

Example. Figure 4.3 gives an example of a spatial join between a file of `Roads` and a file of `Rivers`. As both files are partitioned using the same 4 × 4 grid structure, there is no need for a preprocessing step. The global join step is responsible on matching the overlapped partitions together. The local join step joins the contents of each matched partitions. Finally, the duplicate avoidance step ensures that each matched record is produced only once.

Preprocessing step. The two input files to the spatial join could be partitioned independently upon their loading into SpatialHadoop. For example, Figure 4.4 gives an example of joining two grid files with 3 × 3 (solid lines) and 4 × 4 (dotted lines) grids. In this case, our *distributed* spatial join algorithm has two options to proceed:

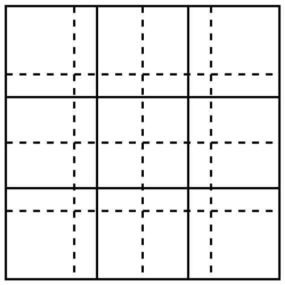


Figure 4.4: Partitions.

(1) Work exactly as described above without any preprocessing, where joining the two grid files produces 36 overlapping pairs of grid cells that are processed in 36 map tasks, or (2) Repartitioning the smaller file (the one with 9 cells) into 16 partitions to match the same partitioning of the larger one. Hence, the number of overlapping pairs of grid cells is decreased from 36 to 16. There is a clear trade-off between these two options. The repartitioning step is costly, yet it reduces the time required for joining as there are less overlapping grid cells. To decide whether to run the preprocessing step or not, SpatialHadoop estimates the cost in both cases and chooses the one with least estimated cost. For simplicity, we use the number of map tasks as an estimator for the cost. When the two files are joined directly, the number of map tasks m_j is the total number of overlapping blocks in the two files. When adding the preprocessing step, the number of map tasks m_p is the sum of the number of blocks in both files. This is because the preprocessing step reads and partitions every block in the smaller file, then joins with every block in the larger file. Only if $m_p < m_j$, the preprocessing step is carried out, otherwise, the files are joined directly.

4.3 Experiments

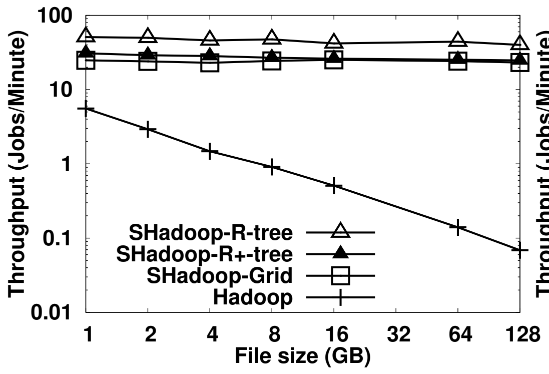
This section provides an extensive experimental study for the performance of the spatial operations implemented in SpatialHadoop as compared to the standard Hadoop. We decided to compare with the standard Hadoop and not other parallel spatial DBMSs for two reasons. First, as our contributions are all about spatial data support in Hadoop, the experiments are designed to show the effect of these additions or the overhead imposed by the new features compared to a traditional Hadoop. Second, the different architectures of spatial DBMSs have great influence on their respective performance, which are out of the scope of this experimental evaluation. Interested readers can refer to a previous study [67] which has been established to compare different large scale data analysis architectures. Meanwhile, we could not compare with $\mathcal{MD}$-HBase [14] or Hadoop-GIS [16] as they support much limited functionality than what we have in SpatialHadoop. Also, they rely on the existence of HBase or Hive layers, respectively, which are outside the scope of SpatialHadoop.

In our experiments, we compare the performance of the range query, *kNN*, and distributed join algorithms in SpatialHadoop proposed in Section 4.2 to their traditional implementation in Hadoop [65, 64]. For range query and *kNN*, we use system throughput as the performance metric, which indicates the number of MapReduce jobs finished per minute. To calculate the throughput, a batch of 20 queries is submitted to the system to ensure full utilization and the throughput is calculated by dividing 20 over the total time to answer all the queries. For spatial join, we use the processing time of one query as the performance metric as one query is usually enough to keep all machines busy. We use the same experimental setup and datasets that are previously described in Section 4.3. The experimental results for range queries, *kNN* queries, and spatial join are reported in Sections 4.3, 4.3, and 4.3, respectively.

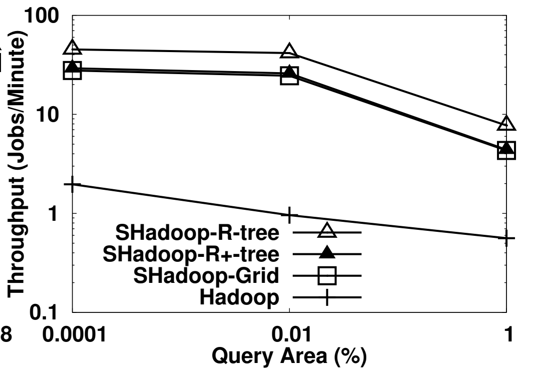
Range Query

Figures 4.5 and 4.6 give the performance of range query processing on Hadoop [64] and SpatialHadoop for both SYNTH and real datasets, respectively. Queries are centered at random points sampled from the input file. The generated query workload has a natural skew where dense areas are queried with higher probability to simulate realistic workloads. Unless mentioned otherwise, we set the file size to 16 GB, query area size to 0.01% of the space, block size to 64 MB, and edge length of generated rectangles to 100 units.

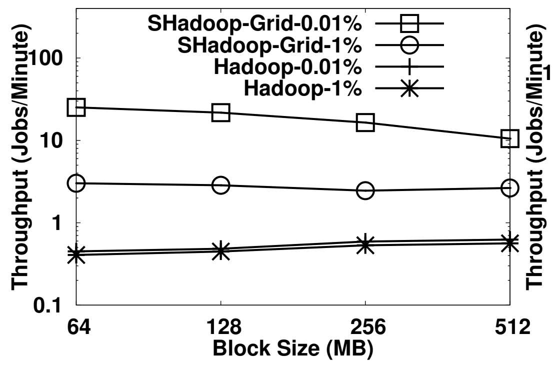
In Figure 4.5(a), we increase the file size from 1 GB to 128 GB, while measuring the throughput of Hadoop, SpatialHadoop with Grid, R-tree and R+-tree indexes. For all file sizes, SpatialHadoop has consistently one or two orders of magnitude higher throughput due to pruning employed by the *SpatialFileSplitter* and the global index.



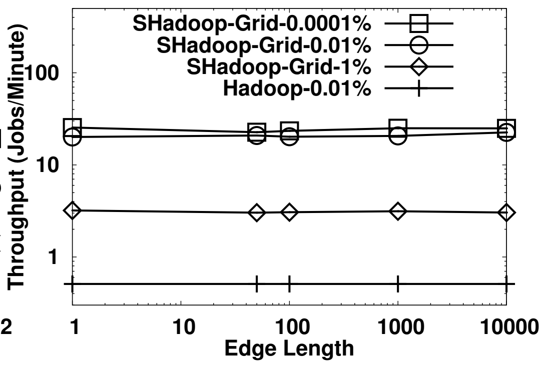
(a) File size



(b) Query window size



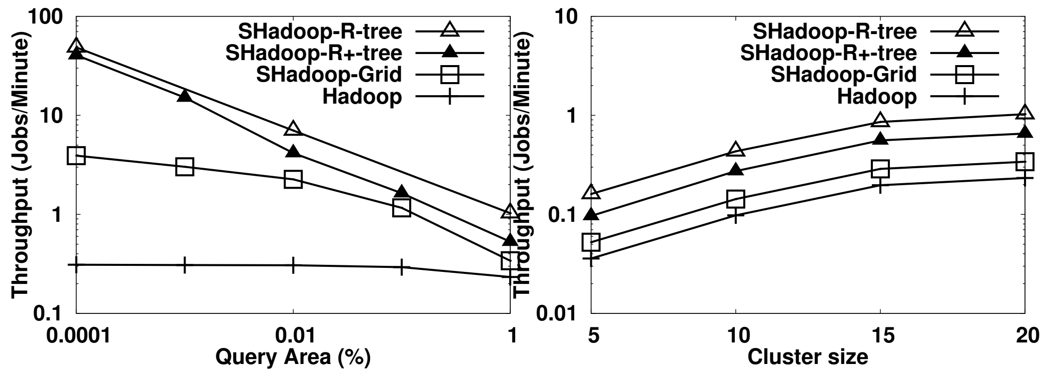
(c) Block size



(d) Edge length

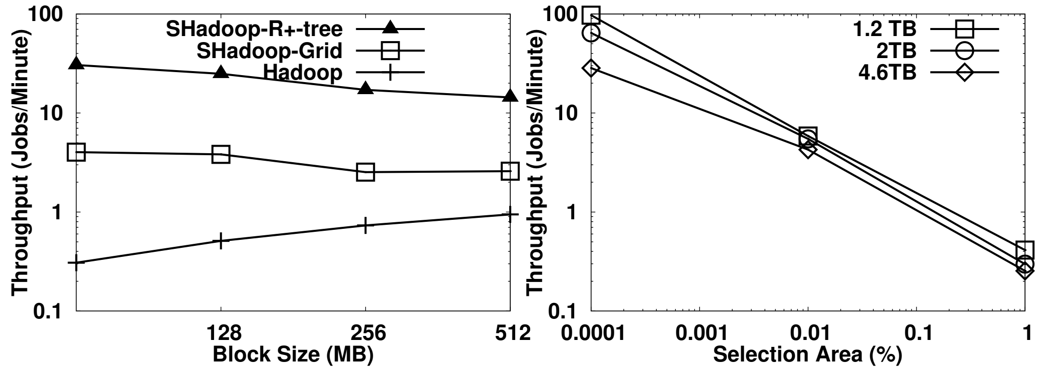
Figure 4.5: Range query experiments with SYNTH dataset

As Hadoop needs to scan the whole file, its throughput decreases with the increase in file size. On the other hand, the throughput of SpatialHadoop remains stable as it processes only a fixed area of the input file. As data is uniformly distributed, R+-tree becomes similar to the grid file with the addition of a local index in each block. R-tree is significantly better as it skips processing of partitions completely contained in the query range while R+-tree suffers from the overhead of replication and duplicate avoidance technique. In Figure 4.5(b), the query area increases from 0.0001% to 1% of the total area. In all cases, SpatialHadoop gives more than an order of magnitude better throughput than Hadoop. The throughput of both systems decreases with the increase of the query area, where: (a) we need to process more file blocks, and (b) The size of the output file becomes larger. R-tree is more resilient to increased query areas as it skips processing of blocks totally contained in query area as well as duplicate avoidance.



(a) TIGER-Query area

(b) TIGER-Cluster size



(c) TIGER-Block size

(d) NASA-File size

Figure 4.6: Range query experiments with real datasets

Figure 4.5(c) gives the effect of increasing the block size from 64 MB to 512 MB, while measuring the throughput of Hadoop and SpatialHadoop for two sizes of the query area, 1% and 0.01%. For clarity, we show only the grid index as other indexes produce similar trends. When increasing block size, Hadoop performance slightly increases as it requires less number of blocks to process while SpatialHadoop performance decreases as the number of processed blocks remain the same while block sizes increase. Figure 4.5(d) gives the overhead of the duplicate avoidance technique used in grid and R+-tree indexing. The edges length of spatial data is increased from 1 to 10K within a space area of 1M×1M, which increases replication in the indexed file. As shown in figure, the overhead of the duplicate avoidance technique ends up to be minimal and SpatialHadoop manages to keep its performance within orders of magnitude higher throughput.

Figure 4.6(a) gives the performance of range query on the TIGER dataset when increasing the query area. SpatialHadoop shows two orders of magnitude throughput increase over traditional Hadoop. Unlike the SYNTH dataset where grid and R-tree indexes behave similarly, the TIGER dataset is more suited with an R-tree index due to the natural skewness in the data. Figure 4.6(b) shows how SpatialHadoop scales out with cluster size changing from 5 to 20 nodes when executing range queries with a selection area of 1%. Both Hadoop and SpatialHadoop scale smoothly with cluster size, while SpatialHadoop is consistently more efficient. Figure 4.6(c) shows how block size affects the performance of range queries on the TIGER real dataset. The results here conforms with those of synthetic dataset in Figure 4.5(c) where Hadoop performance is enhanced while SpatialHadoop degrades a little bit. The difference here is higher due to the high skewness of the TIGER dataset. Figure 4.6(d) shows the running time for range queries on subsets of NASA dataset of sizes 1.2TB, 2TB and the whole dataset of size 4.6TB. The datasets are indexed using R-tree on an EC2 cluster of 100 *large* nodes, each with a quad core processor and 8GB of memory. This experiment shows the high scalability of SpatialHadoop in terms of data size and number of machines where it takes only a couple of minutes with the largest selection area on the 4.6TB dataset.

Figure 4.7 shows the performance of range queries over the constructed indexes. In each experiment, we submit a batch of queries, and measure the throughput of the cluster in terms of queries/minute. In general, we found that all partitioning techniques that have been experimented behave roughly the same due to the simplicity of the query in the MapReduce environment. It has been shown in [62] that the performance of the range query reflects the four quality measures Q1-Q4. Therefore, we will focus in measuring the effect of query selectivity and number of machines in the MapReduce environment. In Figure 4.7(a), the input size is increased and the throughput is measured for each partitioning technique. As expected, the performance degrades as the input size increases as more partitions need to be processed with larger files.

In Figure 4.7(b), the block size of the partitioned data is increased from 32MB to 256MB and the performance of range query is measured for `All Objects`. Despite the slight variance in times across the different techniques, we can notice a common *bitonic* trend of the performance where it rises up and then goes down again. This interesting behavior happens due to the trade-off between number of matched partitions and amount of processing per partition. A smaller block size reduces the amount of work per partition but increases number of matched partitions per query, while a larger block size has an opposite effect. As shown in figure, the sweet spot that balances this trade-off varies from one partitioning technique to another but most techniques are balanced at the 128MB block which is the default value in recent Hadoop releases.

To further study the effect of block size, Figure 4.7(c) shows the range query performance on `Buildings` when it is indexed using a Quad tree of different block sizes. We run three sets of range queries with $\sigma \in \{0.0001\%, 0.01\%, 1\%\}$. We see the same bitonic trend in all cases but the peak is different where it is 64MB when $\sigma = 0.0001\%$ and 128MB for the other two values. The peak at 128MB when $\sigma = 1\%$ does exist but is hard to notice due to the scale of the figure. This experiments indicates that the workload should be accounted while tuning the index.

In Figure 4.7(d), we increase the selection ratio (σ) and evaluate the range query performance for all partitioning techniques. As the selection ratio increases, the performance degrades as more partitions are processed. Eventually, the performance of all of them converge as they end up scanning large portions (or all) of the input file. Unlike centralized systems, Quad tree takes more time scanning the file as the MapReduce job needs to create one task per partition while our Quad tree technique produces much more partitions than other techniques.

In Figure 4.7(e) we show the speedup of running range query using MapReduce compared to a centralized techniques. In this figure, values below one, illustrated by a horizontal line, indicate that the centralized query is faster. This experiment shows that a centralized system outperforms MapReduce when the query area is small as it avoids the overhead of MapReduce when only a little work needs to be done. The results of this experiment can be incorporated into a query optimizer to choose the best approach based on the size of the query area.

Figure 4.7(f) shows how the range query scales out when the cluster size changes from 5 to 35 nodes. On each cluster, we submit five batches of sizes 1, 5, 10, 15, and 20 range queries, and measure the speedup of MapReduce over centralized processing. Unlike centralized systems, which better utilize all its processing capabilities, MapReduce has an upper bound on the speedup regardless of number of machines in the cluster. The reason is that MapReduce breaks a job into coarse-grained tasks, as one per partition, which limits the level of parallelism it can achieve, while centralized systems break jobs into finer-grained partitions to achieve a higher level of parallelism as its resources allow. As the batch size increases, MapReduce can achieve a higher speedup but it again stabilizes as soon as the cluster becomes underutilized.

K-Nearest-Neighbor Queries (*kNN*)

Figures 4.8 and 4.9 give the performance of *kNN* query processing on Hadoop [64] and SpatialHadoop for both SYNTH and TIGER datasets, respectively. In both experiments, query locations are set at random points sampled from the input file. Unless otherwise mentioned, we set the file size to 16 GB, k to 1000, and block size to 64 MB. We omit the results of the R+-tree as it becomes similar to R-tree when indexing points because there is no replication.

Figure 4.8(a) measures system throughput when increasing the input size from 1 GB to 16 GB. SpatialHadoop has one to two orders of magnitude higher throughput. Hadoop performance decreases dramatically as it needs to process the whole file while SpatialHadoop maintains its performance as it processes one block regardless of the file size. Unlike the case of range queries, the R-tree with local index shows a significant speedup as it allows the *kNN* to be calculated efficiently within each block, while the grid index has to scan each block. As k is varied from 1 to 1000 in Figure 4.8(b), SpatialHadoop keeps its speedup at two orders of magnitude as k is small compared to number of records per block.

In Figure 4.8(c), as the block size increases from 64 MB to 256 MB, the performance of SpatialHadoop stays at two orders of magnitude higher than Hadoop. Since Hadoop scans the whole file, it becomes a little bit faster with larger block sizes as the number of blocks gets lower. Figure 4.8(d) shows how the throughput is affected by the location of the query point Q relative to the boundary lines of the global index partitions. Rather than generated totally at random, the query points are placed on the diagonal of a random partition where its distance to the center of the partition is controlled by a *closeness factor* $0 \leq c \leq 1$; 0 means that Q is in the partition center while 1 means that Q is a corner point. When c is close to zero, the query answer is likely to be found in one partition. When c is close to 1, it is likely that we need to refine the initial answer, which significantly decreases throughput, yet it is still of two orders of magnitude higher than Hadoop, which is not affected by the value of c.

Figure 4.9(a) gives the effect of increasing k from 1 to 1000 on TIGER dataset. While all algorithms seem to be unaffected by k as discussed earlier, SpatialHadoop gives an order of magnitude performance with grid index and two orders of magnitude performance with an R-tree index. Figure 4.9(b) gives the effect of increasing the block size from 64 MB to 512 MB. While the performance with grid index tends to decrease with increased block sizes, the R-tree remains stable for higher block sizes. The performance of Hadoop increases with higher block sizes due to the decrease in total number of map tasks.

Spatial Join

Figure 4.10 gives the results of the spatial join experiments, where we compare our *distributed* join algorithm for SpatialHadoop with two implementations of the SMJR algorithm [65] on Hadoop and SpatialHadoop. Figure 4.10(a) gives the total processing time for joining `edges` and `linearwater` files from TIGER dataset of sizes 60GB and 20GB, respectively. Both R-tree and R+-tree give the best results as they deal well with skewed data with the R+-tree significantly better due to the non-overlapping partitions. Both Grid index and SJMR give poor performance as they use a uniform grid.

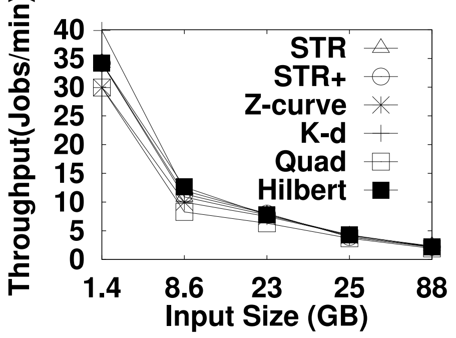
Figure 4.10(b) gives the response time of joining two generated files of the same size (1 to 16GB). To keep the figures concise, we show only the performance of the distributed join algorithm operating on R-tree indexed files as other indexes give similar trends. In all cases, our *distributed* join algorithm shows a significant speedup over SJMR in Hadoop. Moreover, SJMR runs faster on SpatialHadoop compared to Hadoop as the partition step becomes more efficient when the input file is already partitioned. In Figure 4.10(c), the response times of the different spatial join algorithms are depicted when the two input files are of different sizes. In this case, a preprocessing step may be needed which is indicated by a black bar. For small file sizes, the distributed join carries out the join step directly as the repartition step is costly compared to the join step. In all cases, distributed join significantly outperforms other algorithms with double to triple performance, while SJMR on Hadoop gives the worst performance as it needs to partition both input files.

Figure 4.10(d) highlights the tradeoff in the preprocessing step. We rerun the same join experiments of two different file sizes *with and without* a preprocessing step. We also run a third instance (DJ-Smart) that decides whether to run a preprocessing step or not based on the number of map tasks in each case as discussed in Section 4.2. DJ-Smart manages to take the right decision in most cases. It only misses the right decision in two cases where it performs the preprocessing step when the direct join is faster. Even for these two cases, the difference in processing time is very small and does not cause major degradation in performance. The figure also shows that for some cases, such as 1×8, the preprocessing step manages to speedup the join step but it incurs a big overhead rendering it to be unuseful for this case.

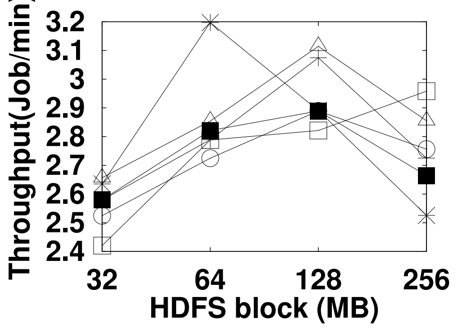
Figure 4.10(e) shows the performance of joining `Roads` and `Buildings` when both are partitioned using the same technique. In general, all the techniques scale well with the cluster size. Unlike range query, one spatial join query is usually enough to utilize all cluster resources and achieve a high degree of parallelism. The two SFC-based techniques perform relatively worse than other techniques as they produce a lot of overlapping partitions (e.g., see Figure 3.2 e&f).

Figure 4.11 further studies the performance of spatial join when the two input files are partitioned using different techniques. In this experiment, we run two spatial join queries, `Roads`$\times$`Lakes` and `Roads`$\times$`Buildings`, for every possible combination of partitioning techniques on inputs. The figure shows the values of Q1 for input files, and the total running times which are color-coded from red (slowest) to green (fastest). The figure shows a direct correspondence between the values of Q1 and the overall performance. Statistically, we found a strong linear correlation with a coefficient of 89% and 92%, for the two experimented queries.

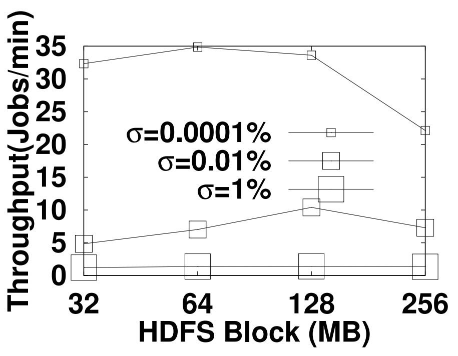
In addition to its good quality measures, the Quad tree outperforms all other techniques as it minimizes the number of overlapping partitions between the two files by employing a regular space partitioning which is perfectly aligned across different files. This finding conforms with earlier work in the literature which showed that Quad tree partitioning outperforms both R-tree and R+-tree partitioning when running a spatial join query in a traditional DBMS environment [68]. Furthermore, the state-of-the-art work in spatial join in main memory is based on Quad-tree-like partitioning [69]. This work is the first to extend those earlier findings to MapReduce as well.



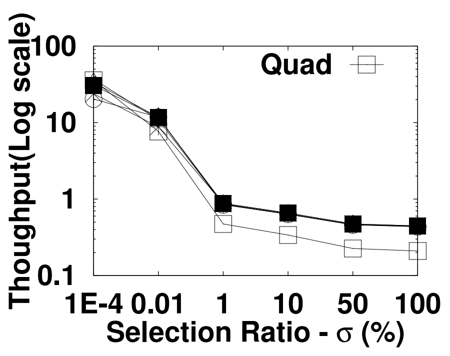
(a) Input Size (I)



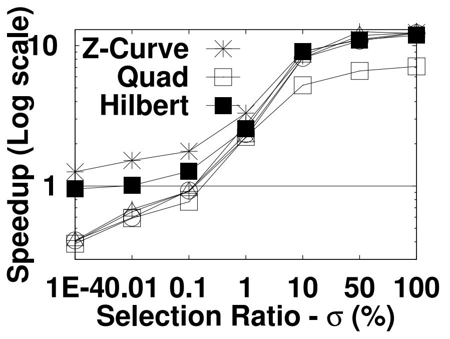
(b) Block Size (B)



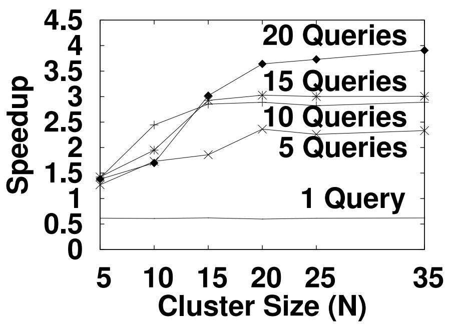
(c) Block Size (B)



(d) Selection Ratio (σ)

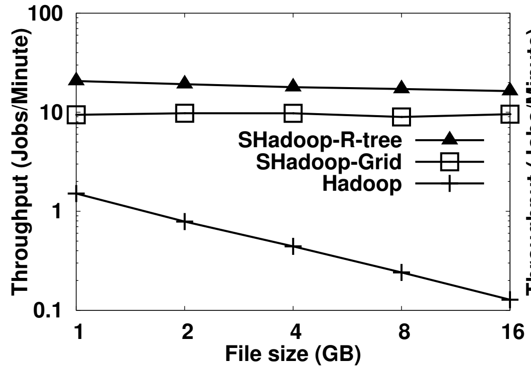


(e) Speedup

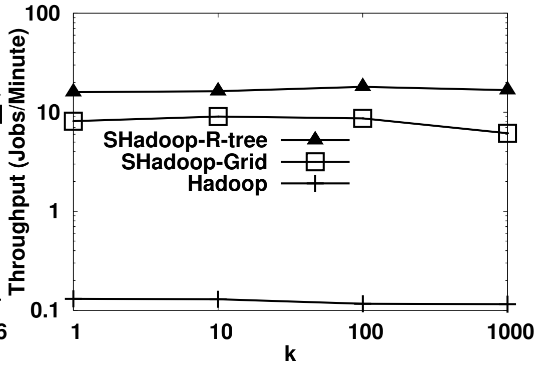


(f) Scale out

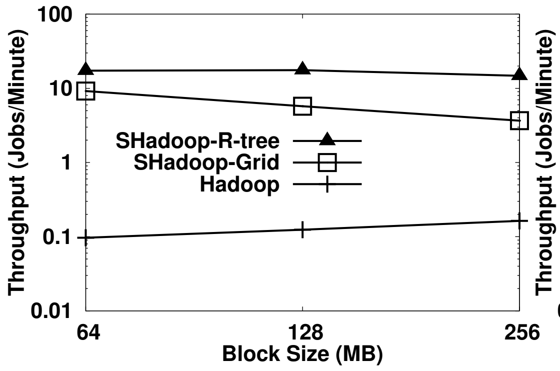
Figure 4.7: Range Query



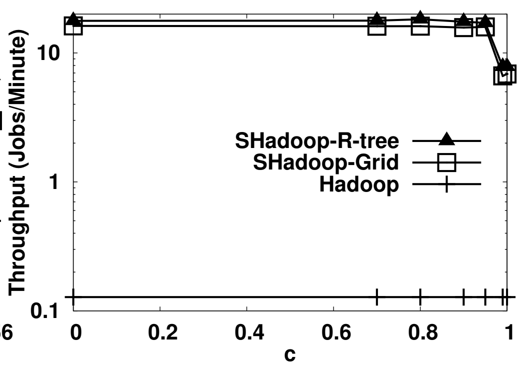
(a) File size



(b) *k*

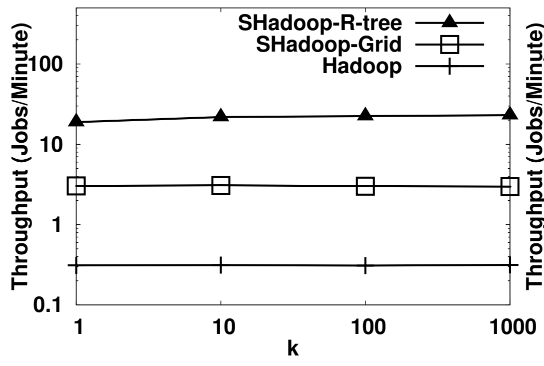


(c) Block size

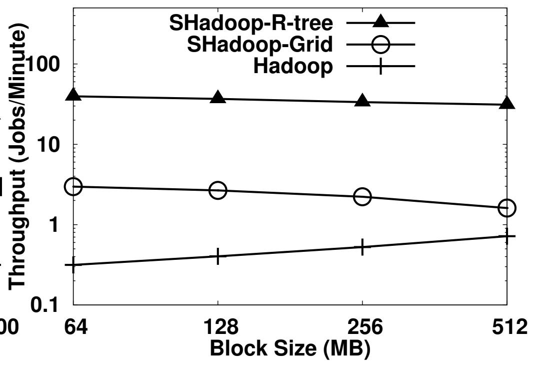


(d) Closeness to grid

Figure 4.8: *kNN* algorithms with SYNTH dataset

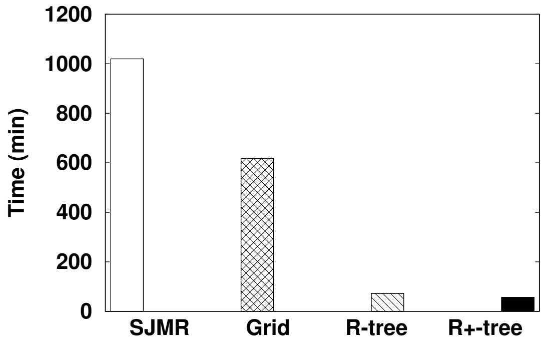


(a) *k*

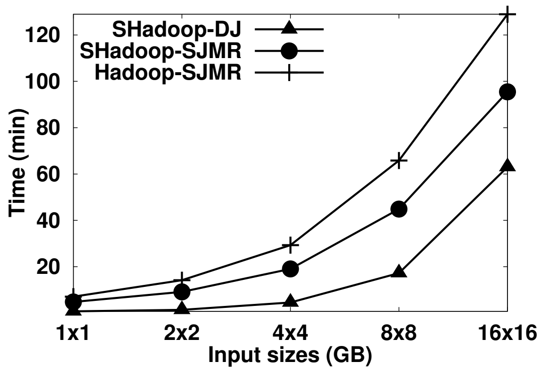


(b) Block size

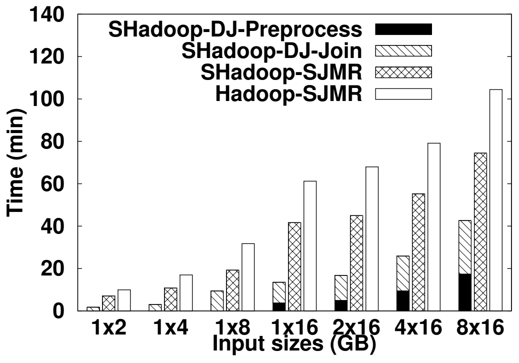
Figure 4.9: Performance of *kNN* with TIGER dataset



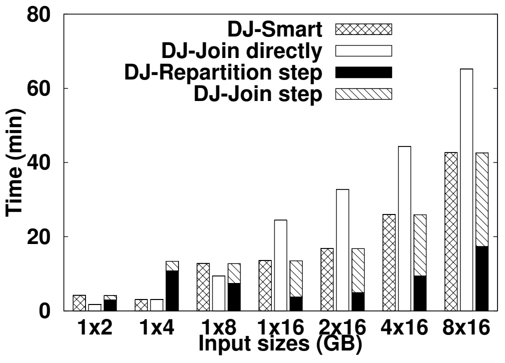
(a) TIGER dataset



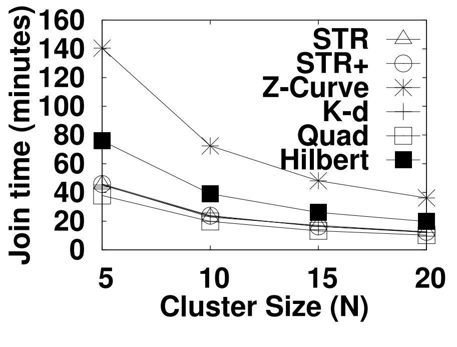
(b) Equal sizes



(c) Different sizes



(d) Tradeoff in preprocessing



(e) Scale out

Figure 4.10: Performance of spatial join algorithms

		Lakes (8.6 GB)						Buildings (25 GB)					
	Q1	0.75	0.89	0.89	1.47	1.47	2.71	0.50	0.71	0.78	0.71	1.07	1.81
Q1		Quad	STR+	K-d	STR	Hilbert	Zcurve	Quad	STR+	K-d	STR	Hilbert	Zcurve
Roads (23GB) 0.52	Quad	276	440	463	675	749	1446	628	919	888	895	1054	1633
0.66	STR+	419	381	439	577	613	1048	801	740	834	763	960	1346
0.71	K-d	465	404	416	560	664	1058	787	836	735	782	935	1321
0.71	STR	409	401	437	570	599	1031	834	765	795	733	928	1321
1.03	Hilbert	512	464	491	612	630	1100	1248	1085	1026	1030	1184	1646
1.88	Zcurve	740	589	608	837	780	1246	1979	1587	1519	1519	1644	2223

Figure 4.11: Spatial join performance (best viewed in color)

Chapter 5

HadoopViz: Extensible Visualization of Big Spatial Data

A major need for all the applications of big spatial data is the ability to visualize this data by generating an image that provides a bird's-eye data view. Visualization is a very common tool that allows users to quickly spot interesting patterns which are very hard to detect otherwise. Examples of spatial data visualization include visualizing a world temperature heat map of NASA satellite data, a scatter plot of billions of tweets worldwide, a frequency heat map for Twitter data showing the hot spots of generated tweets, a road network for the whole world, or a network of brain neurons. In all these visualization examples, users should be able to zoom in and out in the generated image to get different resolutions of the whole data set.

Figure 5.1 portrays an example of visualizing the heat map of world temperature in one month, with a total of 14 billion points. Traditional single-machine visualization techniques [70, 71, 72, 73] have limited performance, thus they take around 1 hour to visualize this data on a machine with 1TB of memory. GPUs can significantly speed up the processing [74, 75], yet they are still hindered by the limited memory resources of a single machine. Meanwhile, there exist three distributed algorithms for visualizing spatial data [76, 25, 77]. Two of them [76, 77] rely on a *pixel-level-partitioning* phase, which partitions and groups records by the pixel they affect, and then combines these records to calculate the color of that pixel, which takes 25 minutes on a 40-core cluster.

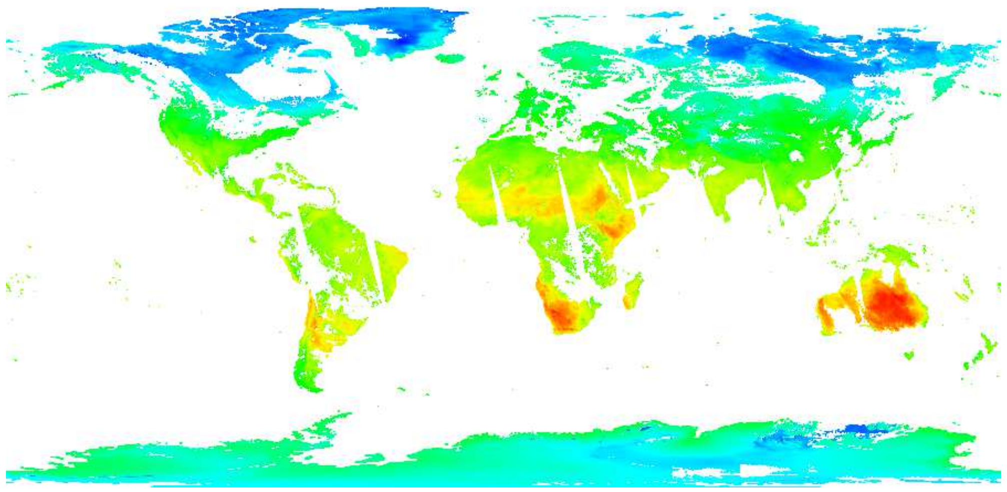


Figure 5.1: World heat map of temperature without HadoopViz

The third technique [25], achieves better performance but it relies on an expensive preprocessing phase which limits its application. In general, these techniques suffer from three limitations: (a) They do not support a *smoothing* function to fuse nearby records together, which limits the image types they can generate. For example, Figure 5.1 contains white spots and strips due to missing values that need to be *smoothed* out. (b) The performance degrades with giga-pixel images due to the excessive number of pixels. (c) Each algorithm is tailored to a specific image type, e.g., satellite images [76, 25] or 3D triangles [77], and it cannot be used to visualize other kinds of big spatial data, e.g., scattered points or road networks.

This chapter presents HadoopViz, an extensible MapReduce-based framework for visualizing big spatial data. HadoopViz overcomes the limitations of existing systems as: (a) It applies a *smoothing* technique which allows it to produce more image types that require fusing nearby records together. For example, it produces the image in Figure 5.2 where missing values are smoothed out by interpolating nearby points. (b) It employs a three-phase approach, *partition-plot-merge*, where it automatically chooses an appropriate *partitioning* scheme to scale up to generate giga-pixel images. For example, it takes only 90 seconds to visualize the image in Figure 5.2. HadoopViz uses this approach to generate both *single-level* images with a fixed resolution, and *multi-level images* where users can zoom in and out. (c) It proposes a novel *visualization abstraction*

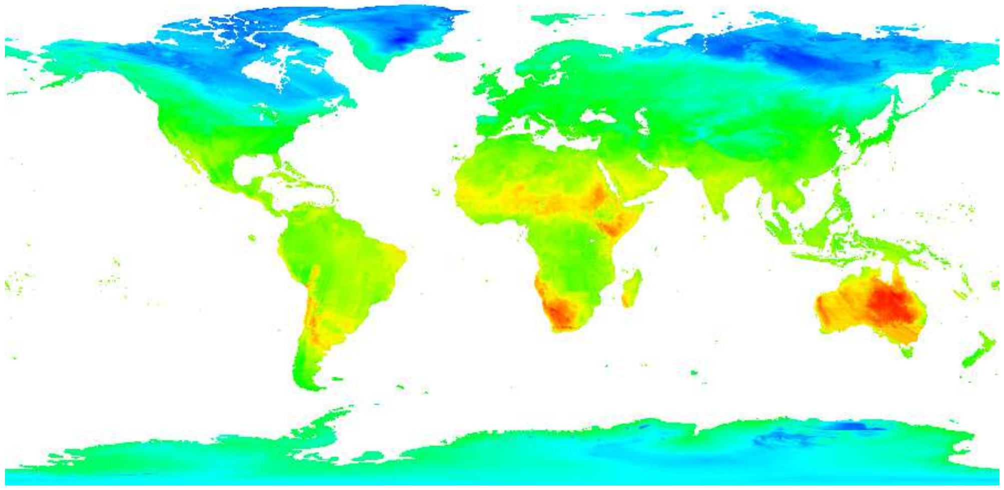


Figure 5.2: World heat map of temperature using HadoopViz

which allows the same efficient core algorithms to be used with dozens of image types, such as scatter plot, road networks, or brain neurons. This allows users to focus only on designing how the desired image should look like, while HadoopViz is responsible of scaling it up to thousands of nodes.

Without HadoopViz, to equip a system with data visualization, one needs to implement an algorithm for visualizing satellite data [76], another algorithm for visualizing tweets [74], a third algorithm for heatmap visualization [25], and so on. This is not practical as each technique has its own data structure and storage requirements. From a system point of view, the holistic and extensible approach of HadoopViz is very appealing and industry-friendly. One needs to realize it once in the system, then, a wide range of various forms of visualization types are immediately supported efficiently.

HadoopViz realizes this extensibility through five *abstract* functions where all visualization algorithms in HadoopViz are implemented using these functions, namely, `smooth`, `create-canvas`, `plot`, `merge`, and `write`. Once a user defines them, HadoopViz plugs these functions into its generic visualization algorithms to scale image generation to thousands of nodes and terabytes of data. (1) The optional `smooth` function can be used to fuse nearby records together, e.g., merge intersecting road segments or recover missing values in satellite data. (2) The `create-canvas` function initializes an empty drawing on which records are plotted, e.g., it initializes an in-memory image for drawing road network or a 2D histogram to create a heat map. (3) The `plot` function does the actual drawing of input records, e.g., it draws a line segment in a road network or updates the histogram according to a point location. (4) The `merge` function combines multiple canvases to form the final picture, e.g., it merges two images by blending pixel colors or merges two histograms by adding up the values in each entry. (5) The `write` function generates the final picture out of a canvas, e.g., it generates a colorful histogram out of a frequency map, or compresses a vector image into a standard vector image format.

This chapter shows the extensibility of HadoopViz using six examples, *scatter plot*, *road network*, *frequency heat map*, *satellite heat maps*, *vectorized map*, and *countries borders*, all implemented using the five abstract functions. As HadoopViz is open source [78], users can refer to these case studies while providing their own image types, e.g., brain neurons or traffic data. In addition, users can reuse existing code or third party libraries in these abstract functions to scale them out. For example, we scale out the single-machine ImageMagick [79] package using HadoopViz which gives it a 48X performance boost. HadoopViz is extensively experimented with several real datasets including the world road network (165 million polylines), and NASA satellite data (14 billion points). HadoopViz efficient design allows it to visualize NASA dataset in 90 seconds. It also generates a video (composed of 72 frames) out of 1 trillion points in three hours on a 10-node cluster [1] .

The rest of this chapter is organized as follows: Sections 5.1 and 5.2 describe HadoopViz algorithms for generating single and multilevel images, respectively. Section 5.3 describes HadoopViz visualization abstraction. Section 5.4 shows six visualization case studies using that abstraction. Finally, Section 5.5 provides an experimental evaluation.

5.1 Single-Level Visualization

This section explains how HadoopViz visualizes satellite data as a single-level image, i.e., an image which shows all the details in one level. Sections 5.3 and 5.4 generalize

[1] Please refer to the generated video at http://youtu.be/-mRRBMBtDa0

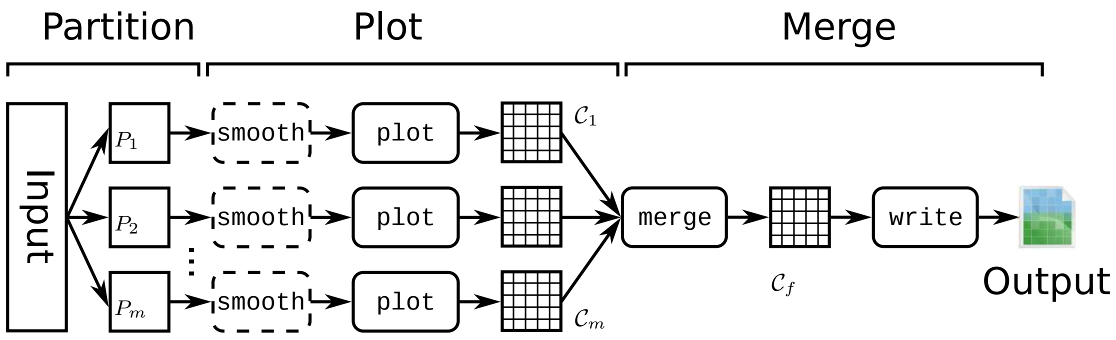


Figure 5.3: Single level visualization algorithm

the described algorithms to other data types using the *visualization abstraction*. The inputs to this algorithm are an input file to visualize, the MBR of its contents, and the desired *ImageSize* in pixels, while the output is an image of the desired size which contains a temperature heat map, as in Figure 5.2.

Figure 5.3 gives an architectural view of the single-level visualization process in HadoopViz which follows a three phase approach where (1) the *partitioning* phase splits the input into m partitions, (2) the *plotting* phase plots a partial image for each partition, and (3) the *merging* phase combines the partial images into one final image. This section describes two algorithms that use this approach, *default-Hadoop partitioning* (Section 5.1), and *spatial partitioning* (Section 5.1). Section 5.1 describes how HadoopViz automatically chooses an appropriate algorithm for a given visualization problem.

Default Hadoop Partitioning

This section describes the single-level visualization algorithm which uses the *default-Hadoop* partitioning. As shown in Algorithm 5, the algorithm assumes that the input is already loaded in HDFS, which splits the input into equi-sized blocks of 128MB each. This means that the *partitioning* phase has nothing to do.

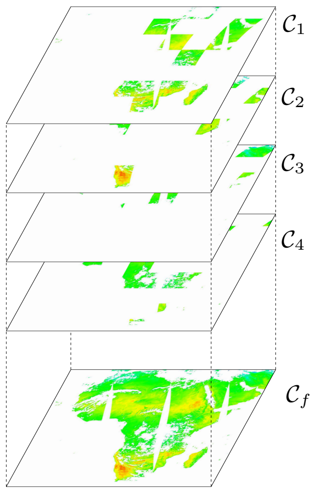
The *plotting* phase in Lines 3-7 runs as a part of the *map* function where each mapper generates a partial image $\mathcal{C}_i$ for each partition P_i, $1 \leq i \leq m$. In Line 5, it initializes a matrix $\mathcal{C}_i$ with the same size as the desired *ImageSize* in pixels ($width \times height$), which acts as a *canvas* to plot records in P_i. Each entry contains two numbers, *sum*

Algorithm 5 Visualization using default Hadoop partitioning

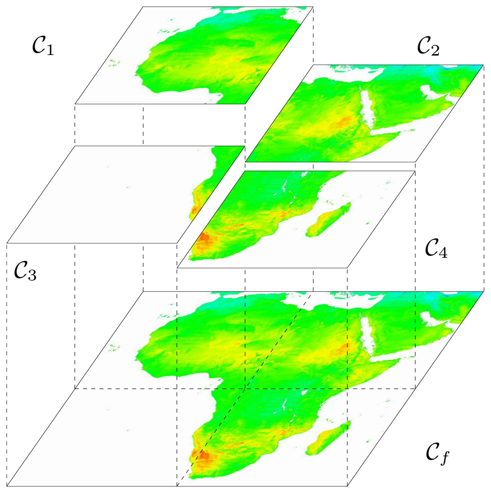
```
1: function SINGLELEVELPLOT(InFile, InMBR, ImageSize)
2: // The input is already partitioned into $m$ partitions $P_1$ to $P_m$
3: // The Plotting Phase
4: for each partition $\langle P_i, BR_i \rangle$ do
5:     Create a 2D matrix $\mathcal{C}_i$ of size $ImageSize$
6:     Update $\mathcal{C}_i$ according to each point $p \in P_i$
7: end for
8: // The Merging Phase
9: Create a final matrix $\mathcal{C}_f$ with the desired ImageSize
10: For each reducer $j$, calculate $\mathcal{C}_j$ as the sum of all assigned matrices
11: One machine computes $\mathcal{C}_f$ as the sum of all $\mathcal{C}_j$ matrices
12: Generate an image by mapping each entry in $\mathcal{C}_f$ to a color
13: Write the generated image as the final output image
```

and *count*, which, respectively, contain the summation and count of all temperature values in the area covered by one pixel, both initialized to zeros. These two values are used together to *incrementally* compute the average temperature in each pixel. Line 6 scans the point in the assigned partition P_i and updates the matrix $\mathcal{C}_i$ according to its location and temperature. The point location determines the matrix entry to update, the temperature value is added to the *sum* entry, and the *count* entry is incremented by one. Finally, the mapper writes the matrix contents as an intermediate record to be processed by the next *merging* phase. These intermediate matrices are shuffled across the R reducers so that each reducer machine receives an average of m/R matrices.

In the *merging* phase, Lines 8-13 merge all intermediate matrices $\mathcal{C}_i$, in parallel, into one final matrix $\mathcal{C}_f$ and writes it as an output image. This phase runs in three steps, *partial merge*, *final merge*, and *write image*. The *partial merge* step runs locally in each machine where each reducer sums up all its assigned matrices into one final matrix $\mathcal{C}_j$ where $1 \leq j \leq R$. In the *final merge* step, a single machine reads back the R matrices and adds them up to a single final matrix $\mathcal{C}_f$. Figure 5.4(a) shows how this step *overlays* the intermediate matrices into one final matrix. Finally, the *write image* step in Line 13 computes the average temperature for each array position and generates the final image



(a) Overlay



(b) Stitch

Figure 5.4: Merging intermediate images

by coloring each pixel according to the average temperature of the corresponding array position. For example, the pixels can be colored with shades that range from blue to red between the lowest and highest temperatures, respectively. The image is finally written to disk as one file in a standard image format such as PNG.

Spatial Partitioning

This section describes the single-level visualization algorithm using *spatial partitioning*. It differs from the previous algorithm in two parts, the *partitioning* phase uses a spatial partitioning, and the *merging* phase *stitches* intermediate images rather than *overlaying* them as shown in Figure 5.4(b). If the user requests a *smooth* image, as in Figure 5.2, then only this algorithm is applicable. This algorithm is implemented as a single MapReduce job as described below.

The *partitioning* phase runs in the map function and uses the SpatialHadoop partitioner [80, 81] to break down the input space into *disjoint* cells and assign each record to all overlapping cells. Notice that the *pixel-level partitioning* technique, employed in previous work [77, 76], is a special case of this phase where it applies a uniform grid equal to the desired *ImageSize*.

The *plotting* phase runs in the *reduce* function where all records in each partition are grouped together and visualized to generate one *partial* image. First, it applies the 2D interpolation techniques employed by SHAHED [25] to recover missing values. Unlike SHAHED, this function is applied on-the-fly which gives more flexibility to apply different smoothing functions. Then, the *plotting* phase initializes a matrix $\mathcal{C}_i$, similar to the one described in the *default-Hadoop* algorithm. However, the size of this matrix is calculated as $width = ImageSize.width\frac{BR_i.width}{InMBR.width}$ and $height = ImageSize.Height\frac{BR_i.height}{InMBR.height}$, where $ImageSize$ is the desired image size in pixels and $InMBR$ is the minimum bounding rectangle (MBR) of the input space. Finally, the reduce function scans all records in the given partition and updates the *sum* and *count* of the array entries as described in the *default-Hadoop* algorithm.

The *merging* phase merges the intermediate matrices $\mathcal{C}_i$ into one big matrix by stitching them together according to their locations in the final image, as shown in Figure 5.4(b). Similar to the *default-Hadoop* algorithm, this phase runs in three steps, *partial merge*, *final merge*, and *write image*. The *partial merge* step runs in the *reduce-commit* function, where each reducer $j \in [1, R]$ creates a matrix $\mathcal{C}_j$ equal to the desired image size and adds all intermediate matrices to it. Each matrix $\mathcal{C}_i$ is added to a position in $\mathcal{C}_j$ according to the bounding rectangle BR_i of its corresponding partition P_i. The *final merge* step runs on a single machine in the *job-commit* function, where it reads back the R matrices written the previous step and adds them up into one matrix $\mathcal{C}_f$. Finally the *write image* step converts the final array to an image as described earlier and writes the resulting image to the output as the final answer.

Discussion

If the user needs to generate a *smooth* image, then only the *spatial partitioning* algorithm is applicable. However, if no smoothing is required, both techniques are applicable and HadoopViz has to decide which one to use. By contrasting the two techniques, we find that there is a tradeoff between the *partitioning* and *merging* phases, where the first algorithm uses a *zero-overhead* partitioning phase, and an expensive *overlay merging* phase, while the second one pays an overhead in *spatial partitioning* but saves with the more efficient *stitching* technique in the *merging* phase. By intuition, the *default-Hadoop* algorithm is useful as long as the *plotting* phase can reduce the size of the input partitions by generating *smaller* partial images. This condition holds if the desired *ImageSize* in bytes is smaller than the size of one input partition, which is equal to HDFS block capacity. Otherwise, if the *ImageSize* is larger than an HDFS block, the *spatial partitioning* would be more efficient as it partitions the input, without significantly increasing its size, while ensuring that each partition is visualized into a much smaller partial image.

To prove the above condition analytically, we measure the overhead of the *partitioning* and *merging* phases. We ignore the overhead of the *plotting* phase and writing the output since they are the same for both algorithms. In the *default-Hadoop* algorithm, the *partitioning* phase has a zero overhead, and the *merging* phase processes m partial images, each with a size equal to the desired *ImageSize*, which makes its total cost equal to $m \times ImageSize$. In the *spatial* partitioning algorithm, the cost of *partitioning* phase is equal to the input size, as it scans the input once, and the cost of the *merging* phase is equal to the size of the desired image because partial images are disjoint and they collectively cover the desired image size. HadoopViz decides to use the spatial partitioning when it produces a less estimated cost, i.e., $InputSize + ImageSize < m \times ImageSize$. By rearranging terms and substituting $InputSize = m \times BlockSize$, the inequality becomes $(m-1)ImageSize > m.BlockSize$. Given that $m \gg 1$ for large inputs, the condition becomes $ImageSize > BlockSize$ as mentioned above.

5.2 Multilevel Visualization

This section presents HadoopViz algorithm for generating gigapixel multilevel images where users can zoom in/out to see more/less details in the generated image. Similar to Section 5.1, we focus on the case study of visualizing temperature data as a heat map while the next two sections show how HadoopViz is generalized to a wider range of images. Figure 5.5 gives an example of a three-level image, also called a *pyramid*, containing 1, 4, and 16 *image tiles* in three zoom levels, each of a fixed default size 256× 256 pixels. Google and Bing Maps use this type of images where the web interface allows users to navigate through offline-generated image tiles. This section shows how HadoopViz generates those tiles efficiently for custom datasets, while the same Google Maps APIs are used to view them. In addition to the input dataset and its MBR

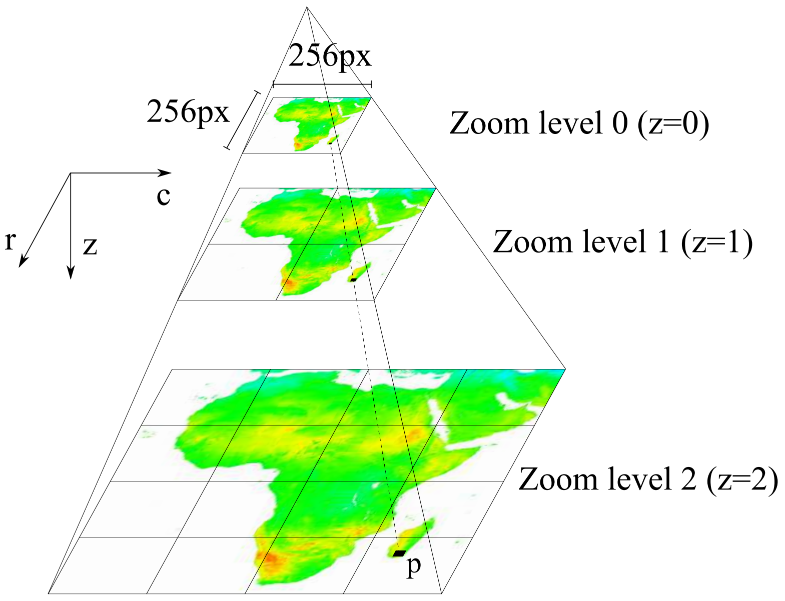


Figure 5.5: A sample three-level image

(*InMBR*), the user specifies a range of zoom levels to generate $[z_{\min}, z_{\max}]$, which actually decides the image size, by knowing the number of 256× 256 pixels tiles in each level. The output is *a set* of images, one for each tile in the desired range of zoom levels along with an HTML file that uses Google Maps APIs to navigate these tiles.

One way to generate such multilevel images is to use our single-level visualization algorithm, described in Section 5.1, to generate each tile of 256× 256 pixels. However, this solution is not practical even for a 10-level image as it contains millions of tiles. Another way is to generate one big image at the highest resolution and chop it down into tiles but this is not practical either as the size of that single big image could be hundreds of gigabytes which does not fit on most commodity machines. HadoopViz uses a smart algorithm which generates all the tiles efficiently in one MapReduce job by taking into account the structure of the pyramid.

The main idea of the multilevel visualization algorithm is to partition the data cross machines and plot each record to *all* overlapping tiles in the pyramid. The tiles that are partially generated by multiple partitions are merged to produce one final image for that tile. Similar to the single-level visualization algorithms, the choice of a *partitioning* technique plays an important role with multilevel visualization and affects

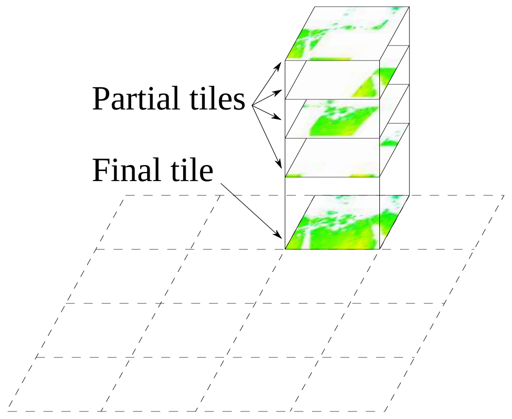


Figure 5.6: Merging phase

the overall performance. This section describes two algorithms which use *default-Hadoop* partitioning (Section 5.2) and a novel *coarse-grained pyramid* partitioning (Section 5.2). After that, we show how HadoopViz combines these two algorithms to generate a user-requested image efficiently (Section 5.2).

Default-Hadoop Partitioning

In this section, we describe how to generate a multilevel image with the default partitioning scheme of Hadoop. The main idea is to allow each machine to plot a record to *all* overlapping pyramid tiles, and then to apply a *merging* phase which combines them to produce the final image for each tile. Since this algorithm uses the *default* partitioning scheme, the *partitioning* phase has a zero overhead.

The *plotting* phase runs in the *map* function and plots each record in the assigned partition P_i to all overlapping tiles in the pyramid. As shown in Figure 5.5, for a record $p \in P_i$, it finds all overlapping tiles in the pyramid in all zoom levels $[z_{min}, z_{max}]$, as at least one tile per zoom level. For each overlapping tile, it initializes a two-dimensional array of the fixed tile size, e.g., 256×256, updates the corresponding entries in the array, and caches that tile in memory in case other records in P_i overlap the same tile again. Once all records are plotted, all tiles, which are cached in memory, are written as intermediate key-value pairs, where the key is the tile ID and the value is the 2D array.

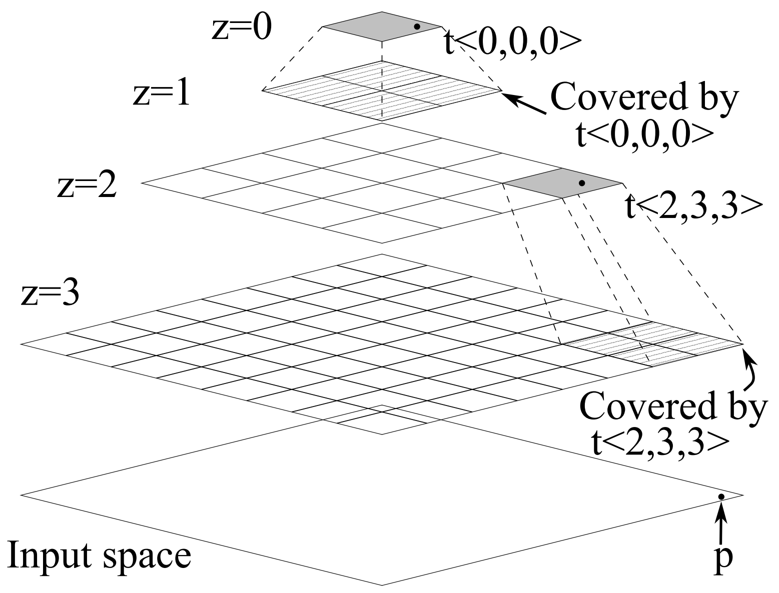


Figure 5.7: Coarse-grained pyramid partitioning

The *merging* phase runs in the *reduce* function and merges intermediate tiles to produce the final image for each tile. This step is necessary because the default Hadoop partitioner might assign overlapping records to different machines and each machine will generate a partial image for the same tile ID, as shown in Figure 5.6. The *merging* phase combines all those partial images by summing the corresponding entries into one final image, which is finally written to disk, as done in the single-level visualization algorithm. The image is written with a standard naming convention '`tile-z-c-v.png`', where z is the zoom level of the tile and (c, r) is the position of the tile in that zoom level.

Coarse-grained Pyramid Partitioning

The algorithm described above suffers from two main drawbacks when the pyramid is very large. First, the *plotting* phase incurs a huge memory footprint as it needs to keep all tiles of the whole pyramid in main-memory until all records are plotted. Second, the *merging* phase adds a huge processing overhead for merging all these tiles. This section describes how HadoopViz uses a novel *coarse-grained pyramid* partitioning technique to avoid the drawbacks of the *default* partitioning algorithm. In this technique, the input is partitioned according to pyramid tiles [82], which ensures that each machine generates a fixed number of tiles while totally avoiding the *merging* phase. While this can be done using a traditional *fine-grained* pyramid partitioning, which creates one partition per-tile, HadoopViz uses the *coarse-grained* partitioning which reduces the partitioning overhead by creating a fewer number of partitions while ensuring the correct generation of all image tiles. This algorithm runs in two phases only, *partition* and *plot*, which are implemented in one MapReduce job.

The *partitioning* phase runs in the *map* function and it uses the *coarse-grained pyramid* partitioning which assigns each record p to *select* tiles. This technique reduces the partitioning overhead by creating partitions for tiles only in levels that are multiples of a system parameter k, which controls the *partitioning granularity*. At one extreme, setting $k = 1$ is similar to using a *fine-grained* partitioning where it creates one partition per tile. On the other extreme, setting $k > z_{\max}$ generates only one partition at the top of the pyramid which contains all input records. Figure 5.7 shows an example with $k = 2$ where it assigns a record p to only two partitions at levels $z = 0$ and $z = 2$. The machine that processes each partition will be responsible of generating the tile images in up-to k levels rooted at the assigned partition tile.

The *plotting* phase runs in the *reduce* function and it takes all the records in one partition, which corresponds to a tile t_i, and plots these records to all pyramid tiles under the tile t with at most k levels. For example, in Figure 5.7, the partition at tile $t = \langle 0, 0, 0 \rangle$ generates the five tiles at zoom levels 0 and 1. Once all records are plotted, an image is generated for each tile and all images are written to the output. No *merging* phase is required for this algorithm since each tile is generated by at most one machine.

Discussion

The default-Hadoop partitioning and pyramid-partitioning algorithms complement each other in generating a multilevel image of an arbitrary range of zoom levels, where the default-Hadoop partitioning is used to generate the top levels, while the pyramid partitioning is used to generate the remaining deeper levels. For the top levels, the *default-Hadoop* algorithm avoids the overhead of partitioning while the overheads of the *plot* and *merge* phases are minimal due to the small pyramid size. On the other hand, the *pyramid partitioning* algorithm would perform poorly in top levels as each tile will contain a huge number of records, e.g., the top tile overlaps all input records as it covers the whole input space. In deeper levels, the algorithms change roles as the *default-Hadoop* algorithm suffers from the overhead of the *plot* and *merge* phases, while the *pyramid* partitioning algorithm overcomes those limitations. Therefore, HadoopViz defines a threshold level z_θ, where levels $z < z_\theta$ are generated using default-partitioning, while other levels $z \geq z_\theta$ are generated using pyramid-partitioning.

To find the value of z_θ analytically, we compare the estimated cost of the two algorithms for a specific zoom level z in the pyramid and find the threshold level at which *pyramid* partitioning starts to give a lower cost. For the *default-Hadoop* partitioning algorithm, the cost of the *partitioning* phase is zero, while the cost of the *merging* phase is $m.4^z.TileSize$, where m is the number of partitions, 4^z is the number of tiles at level z, and *TileSize* is the fixed size of one tile. For the *pyramid* partitioning algorithm, the amortized cost of the *partitioning* phase for one level is $InputSize/k$, because the whole input is replicated once for each consecutive k levels, while there is a zero overhead of the *merging* phase. To find z_θ, we find the range of zoom levels where pyramid partitioning gives a less estimated cost, that is $InputSize/k < m.4^z.TileSize$. By rearranging the terms and separating z, it becomes $z \geq \frac{1}{2}\lg(\frac{B}{k.TileSize})$, i.e., $z_\theta = \lceil \frac{1}{2}\lg(\frac{B}{k.TileSize}) \rceil$.

5.3 Visualization Abstraction

HadoopViz is an extensible framework that supports a wide range of visualization procedures for various image types. In this section, we show how the single-level and multilevel visualization algorithms described in Sections 5.1 and 5.2 are generalized to handle a wide range of image types, e.g., scatter plot, road network, frequency heat map, and vectorized map. To support one more image type within HadoopViz framework, the user needs to define five abstract functions, namely, `smooth`, `create-canvas`, `plot`, `merge`, and `write`. The goal is to make the designers of visualization algorithms worry free from the scalability and detailed implementation of their algorithms. So, algorithm designers only think about the visualization logic, while HadoopViz is responsible on scaling up that logic by employing thousands of computing nodes within a MapReduce environment. For example, ScatterDice [83] is a well known visualization system that is used to visualize multidimensional data using scatter plot. As HadoopViz supports scatter plot, among others, it can complement ScatterDice by scaling out its techniques to generate giga-pixel images of petabytes of data. HadoopViz can similarly scale out other visualization packages such as VisIt [84] or ImageMagick [79].

Algorithm 6 gives the pseudo-code of the *abstract* spatial-partitioning single-level visualization algorithm where the five abstract functions are used as building blocks. Any user-defined implementations for these functions can be directly plugged into this algorithm to generate a single-level image using HadoopViz. In the *partitioning* phase, Line 2 partitions the input spatially into m partitions, each partition i is defined by a bounding rectangle BR_i and a set of records P_i. In the *plotting* phase, Line 5 applies the `smooth` abstract function on each partition i to smooth its records. Line 6 in Algorithm 6 calls the `create-canvas` function to initialize a partial image $\mathcal{C}_i$, for each partition i. Line 7 calls the `plot` function to plot each record $p \in P_i$ on that partial image $\mathcal{C}_i$. In the *merging* phase, Line 10 calls `create-canvas` to initialize the final image canvas $\mathcal{C}_f$. After that, Line 11 calls `merge` successively on partial canvases to merge them into the final canvas. At the end of Algorithm 6, Line 12 uses the `write` function to write the final canvas $\mathcal{C}_f$ to the output as an image. We omit the abstract pseudo code of other algorithms due to limited the space, while interested readers can refer to the source code of HadoopViz [78]. In the rest of this section, we describe the five abstract functions and show how they can differ according to the visualization type. The next section gives six case studies of how these functions are implemented in real scenarios.

The Smooth abstract function

This is an optional preparatory function, where the input is the set of records that need to be visualized. The output is another set of records that represent a smoothed (or cleaned) version of the input by fusing nearby records together to produce a better looking image. HadoopViz tests for the existence of this function to decide whether to go for *spatial* or *default* partitioning. In addition, the plotting phase calls this function to smooth records in each partition before they are visualized. In the example of satellite data described earlier, the `smooth` function applies an interpolation technique to estimate missing temperatures as shown in Figure 5.2. Figure 5.8 gives another example of smoothing the visualization of a road network, where the `smooth` function merges intersecting road segments. If no smoothing is applied, road segments will be crossed,

Algorithm 6 The abstract single-level algorithm

```
1: // The Partitioning Phase
2: Use spatial partitioning to create $m$ partitions
3: // The Plotting Phase
4: for each partition $\langle P_i, BR_i \rangle$ do
5:     Apply smooth($P_i$)
6:     $\mathcal{C}_i \leftarrow$ create-canvas($ImageSize \frac{BR_i}{InMBR}$)
7:     for each $p \in P_i$, plot($p$, $\mathcal{C}_i$)
8: end for
9: // The Merging Phase
10: $\mathcal{C}_f \leftarrow$ create-canvas(ImageSize)
11: for each intermediate canvas $\mathcal{C}_i$, merge($\mathcal{C}_f$, $\mathcal{C}_i$)
12: write($\mathcal{C}_f$, outFile)
```

giving a not so accurate visualization (Figure 5.8(a)). Having this logic in a user-defined abstract function allows users to easily inject a more sophisticated logic. For example, if more attributes are available, the `smooth` function can correctly overlap (not merge) road segments at different levels such as roads and bridges. It is important to note that the `smooth` function has to be applied on the input data rather than the generated image because it can process all input attributes that probably disappear in the final image. One limitation in the current design is that it cannot smooth records across different partitions which we can support in the future.

The Create-Canvas abstract function

This function creates and initializes an appropriate in-memory data structure that will be used to create the requested image. The input to the `create-canvas` abstract function is the required image resolution in terms of *width* and *height* (e.g., in pixels). The output is an initialized canvas of the given resolution. If the desired image is a *raster* image, the canvas typically consists of a two-dimensional matrix as one per pixel. If the desired image is a *vector* image, it contains a vectorized representation of objects shapes. The `create-canvas` function is used in both the *plotting* and *merging* phases. In the *plotting* phase, HadoopViz calls this function to initialize the partial images used

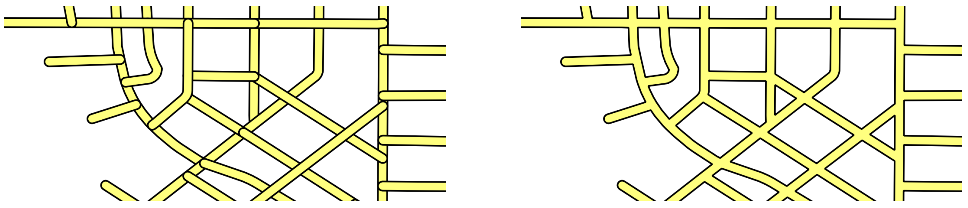

(a) Not smoothed (b) Smoothed

Figure 5.8: Smoothing of road segments

to plot each partition. In the *merging* phase, it is used to prepare the final image on which all partial images will be merged. For example, when visualizing a road network, it returns an in-memory *blank* image of the given size, while for a heat map, it returns a frequency map represented as a 2D array of numbers initialized to zeros.

The Plot abstract function

The `plot` function is called for each record in the input data set. It takes as input a canvas, previously created using `create-canvas`, and a record. Then, it plots the input record on the canvas. The *plotting* phase calls this function for each record in the partition to draw the partial images. The `plot` function uses a suitable algorithm to draw the record on the canvas. For example, when visualizing a road network, the Bresenham mid-point algorithm [85] is used to draw a line on the image. When visualizing a heat map, this function updates the two-dimensional histogram (created by the `create-canvas` function) based on the point location. To generate a vector image, it simplifies the record shape and represents its geometry as a vector. In general, this function can call any third party visualization package, e.g., VisIt [84] or ImageMagick [79], which allows HadoopViz to easily reuse and scale out existing visualization packages.

The Merge abstract function

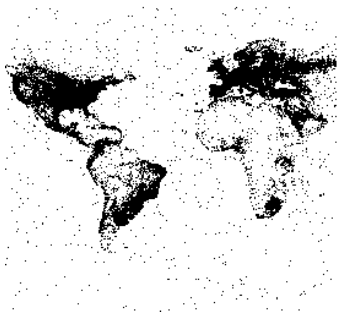
The input to the `merge` function is two partial canvases, while the output is one final canvas that is composed by combining the two input partial layers. The *merging* phase calls this function successively on a set of layers to merge them into one. If the partial layers are disjoint, i.e., each one covers a different part of the image, merging them is straightforward as each pixel in the final image has only one value in one canvas. In case the partial layers cover overlapping areas, the same pixel in the final image may have more than one value. In this case, the `merge` function needs to decide how these values are combined together to determine the final value of that pixel. For example, if the canvases are raster images, two pixels are merged by taking the average of each color component in the two pixels. In case of generating a heat map, two entries in the histogram are merged together by adding up the corresponding values as each one represents a partial count.

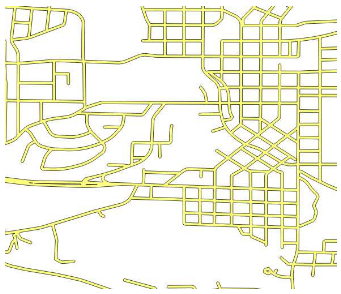
The Write abstract function

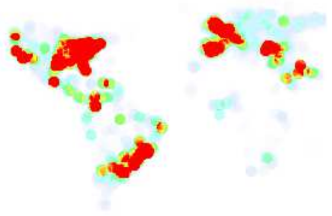
The `write` function writes the final canvas (i.e., image), computed by the `merge` function, to the output in a standard image format (e.g., PNG or SVG). This abstract function allows developers to use any custom representation for canvases which might contain additional metadata, and generate the image as a final step using the `write` function. For example, while generating a heat map, the canvas stores the frequencies as integers while the `write` function transforms them into colors and writes a PNG image. In the case of generating vector images, the canvas contains geometric representation of shapes and the `write` function encodes them in a standard Scalable Vector Graphics (SVG) format.

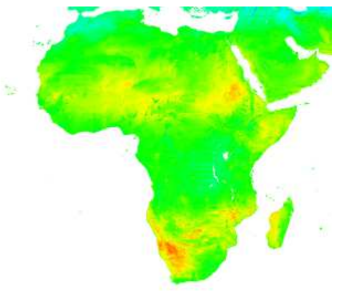
5.4 Case Studies

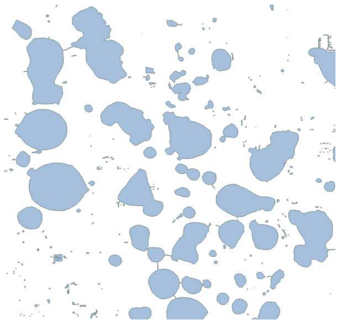
This section describes six case studies of how to define a visualization type by implementing the five abstract functions described in Section 5.3. These case studies are carefully selected to cover different aspects of the visualization process. Notice that all case studies described using the abstract functions can be used to generate both single and multilevel images. Case studies I and II give an example of non-aggregate visualization, where records are directly plotted, with and without a smoothing function. Case studies III and IV give examples of aggregate-based visualization, where records are aggregated before plotted, with and without a smoothing function. Case study V gives an example of generating a vector image with a smoothing function. Finally, case study VI shows how to reuse and scale out an existing visualization package which is used as a black box.


(a) I. Scatter plot


(b) II. Road network


(c) III. Heat map


(d) IV. Satellite data


(e) V. Vectorized map

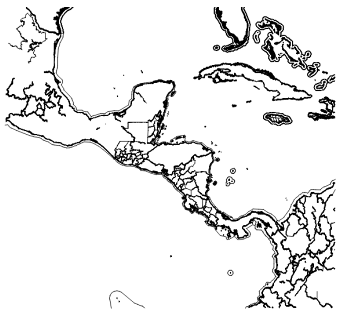
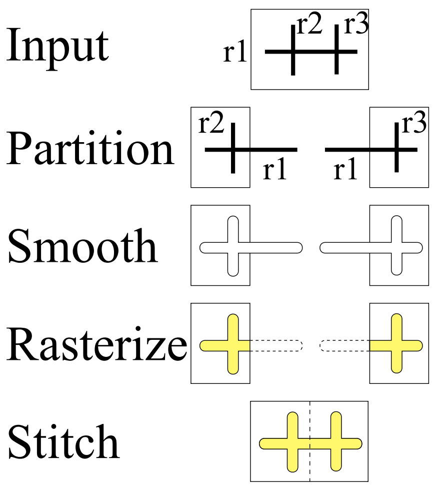

(f) VI. Countries borders

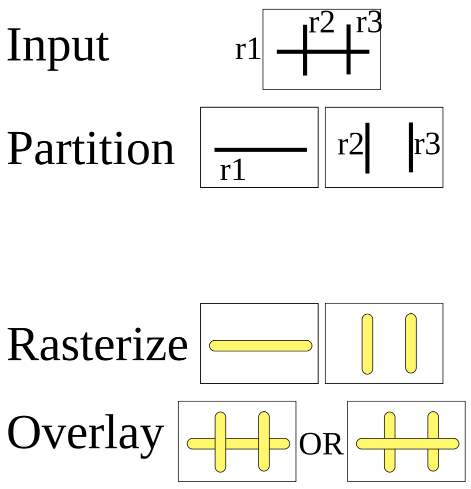
Figure 5.9: Six case studies implemented in HadoopViz

Case Study I: Scatter Plot

In *scatter* plot, the input is a set of points, e.g., geotagged tweets, and each point is plotted as a pixel in the final image as shown in Figure 5.9(a). To make HadoopViz support such images for both single and multi-level images, we need to define its five abstract functions as follows: (1) The `smooth` function is not required for scatter plot, as there are no records that need to be smoothed together. (2) The `create-canvas` function takes an image size in pixels and returns a blank in-memory image with the given size initialized with a transparent background. (3) The `plot` function takes an in-memory image and an input record r, and it projects the input point on the image and colors the corresponding pixel in black. (4) The `merge` function takes two in-memory images and merges them together. To merge two images, the top-left corner of the first image is projected to a pixel in the other one as done in the `plot` function. Then, the first image is painted on the second one at the projected location. Since each image is initialized



(a) With smooth



(b) Without smooth

Figure 5.10: Road network visualization

with a transparent background, empty spaces in one image will reveal the contents of the other image. (5) The **`write`** function encodes the in-memory image in a standard image format (e.g., PNG) and writes it to disk.

Case Study II: Road Network

In road network visualization, the input is a set of road segments, each represented as a line, and the desired output is an image similar to the one illustrated in Figure 5.9(b). To support such images in HadoopViz, we need to define its five abstract functions as follows: (1) The **`smooth`** function takes a set of road segments, applies a *buffer* operation to each line segment to give it some thickness, and then applies a *union* operation to all resulting polygons to merge intersecting road segments. Specifying a **`smooth`** function enforces HadoopViz to use a spatial partitioning as shown in Figure 5.10(a). (2) The **`create-canvas`** function is exactly the same as in *scatter plot*, which returns an in-memory image of the provided size. (3) The **`plot`** function reads the result of the union operation returned by the **`smooth`** function, and draws each polygon in the dataset onto the image. The polygon is first projected from input space to image space, then, the interior of the polygon is filled with yellow while the boundaries are stroked in black. As spatial partitioning is used, some records might be replicated to two overlapping

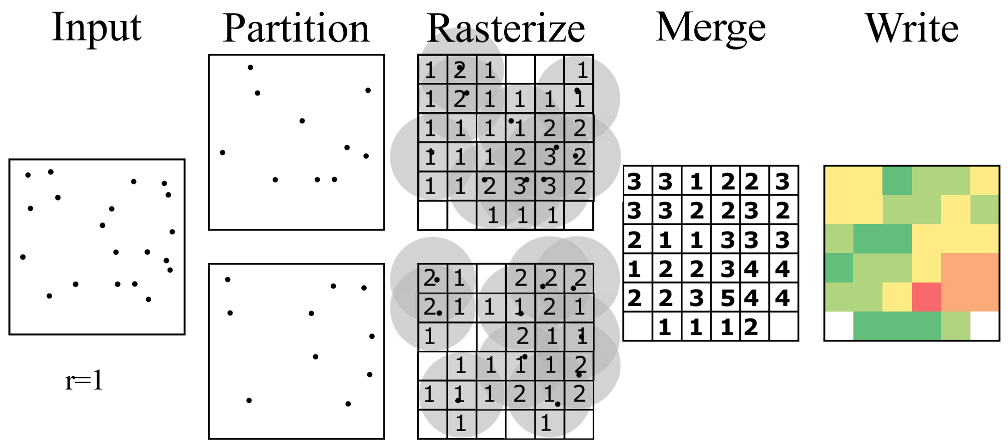


Figure 5.11: Steps of frequency heat map (Best viewed in colors)

partitions such as r_1 in Figure 5.10(a). Each replica is merged with a different set of road segments and is plotted to two different partial images. However, it is automatically clipped at partition boundaries when plotted (denoted by dotted lines). (4) The **`merge`** function is exactly the same as in the scatter plot where one image is painted on the other according to its location. Notice that in this case, it will always *stitch* partial images together because they are all disjoint. As in Figure 5.10(a), the two images are clipped at the stitch line, hence, putting them side-by-side generates the correct picture without worrying about any overlaps. (5) The **`write`** function is exactly the same as in the *scatter plot*.

To justify the use of a smoothing function, Figure 5.10(b) shows how the visualization process would work if no **`smooth`** function is provided. First, HadoopViz would use the default Hadoop partitioning which causes overlapping records to be in two different partitions. The **`plot`** function can still apply the *buffer* operation to each segment but not the *merge* operation because overlapping records are in two different machines. The **`merge`** function would overlay partitions instead of stitching them. Depending on which image goes on top, there are two possible final images which are both incorrect.

Case Study III: Frequency Heat Map

In this case study, the input is a set of points, e.g., geotagged tweets, and the output is a colored map, e.g., Figure 5.9(c), where dense areas are colored in red and sparse areas are colored in blue. To support such kind of visualization in HadoopViz, we need to define the five abstract functions as follows: (1) The `smooth` function is not needed for such image type. (2) The `create-canvas` function creates a two-dimensional array of integers (called, *frequency map*), where the value in each entry represents the number of records around it; all initialized with zeros. (3) The `plot` function, takes one point, projects it to the frequency map, and increments all entries within a predefined radius ρ to denote that the point is within the range of those pixels. As more points are plotted, entries in the frequency map will denote the density of points in each pixel, as shown in Figure 5.11. (4) The `merge` function takes two frequency maps, and merges them together by adding up corresponding entries in both. (5) The `write` function takes one frequency map and converts it to an image by coloring corresponding pixels in the image according to the density in the frequency map. First, it normalizes densities to the range $[0, 1]$, and then it calculates the color of each pixel by making a linear combination between the two colors, blue and red. Finally, the created image is written to the output in the standard PNG format.

Case Study IV: Satellite Data

In this case study, the input is a set of temperature readings, each associated with a geolocation, and the output is a temperature heat map, like the one in Figure 5.9(d), where the color represents the average temeprature of the underlying region. In addition, some regions do not contain any values due to clouds or satellite mis-alignment leaving some blind spots [25]. The values in those uncovered areas need to be estimated using a two-dimensional interpolation technique. To support such image type in HadoopViz, we need to define the five abstract functions as follows: (1) The `smooth` function takes a set of points in a region and *recovers* missing points using a two-dimensional interpolation function in a way similar to SHAHED [25]. Although HadoopViz uses the same technique as in SHAHED, it applies the `smooth` function on-the-fly allowing users to easily inject a better smoothing function. (2) The `create-canvas` function initializes a two-dimensional array of the input size where each entry contains two numbers, *sum* and *count*, used together to compute the average temperature incrementally. (3) The `plot` function projects a point onto the 2D array, and updates both *sum* and *count* in the array according to the temperature value of the point. (4) The `merge` function is very similar to that of the frequency heat map but it adds up both *sum* and *count* in the merged layers. (5) The **`write`** function starts by calculating the average temperature in each entry as $avg = sum/count$. Then, we use the same **`write`** function of the frequency heat maps.

Case Study V: Vectorized Map

This case study shows how to create a vector image that represents a map, such as Google Maps or Bing Maps. Many recent applications on smart phones and on the web prefer vector images over raster images due to their smaller size and nice rendering. For simplicity, this case study explains how to plot a map of lakes as shown in Figure 5.9(e), however, the technique can be easily expanded to plot more map objects through more sophisticated implementations of the **`plot`** function. To generate a vector image for lakes in HadoopViz, we define the abstract functions as follows: (1) In the **`smooth`** abstraction function, we use a *map simplification* algorithm which reduces the amounts of details in the polygons according to the resolution of the final image. The goal of this function is to reduce the generated image size by removing very fine details that will be hardly noticed by users according to the image size. If a multilevel image is generated, the **`smooth`** function will be called once for each zoom level so that it keeps more details in deeper levels. This function also removes very small lakes which are too small to plot in the image. (2) The **`create-canvas`** abstract function initializes an empty list of polygons. (3) The **`plot`** function adds the polygon representation of the lake geometry to the list of polygons in the canvas. (4) The **`merge`** function, simply, merges the two lists of polygons in the two canvases into one list in the output canvas. (5) The **`write`** function encodes the canvas contents as a standard SVG image and writes it to the output.

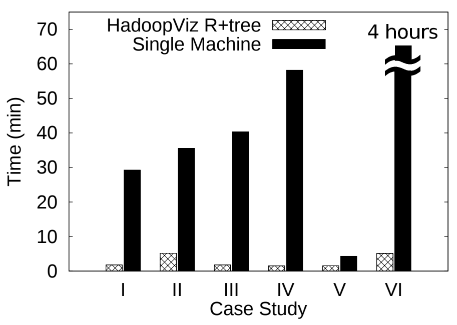
Case Study VI: Parallel ImageMagick

ImageMagick [79] is a popular open source package that can produce and edit images. However, it is single-machine and does not support any multi-node parallelization functionality. This case study shows how to use the binaries of ImageMagick as a blackbox and utilize the extensibility of HadoopViz to seamlessly parallelize it. This allows users to visualize extremely large datasets using ImageMagick that it cannot handle otherwise. For example, this technique speeds up the visualization of a 130GB file from four hours on a single machine, to five minutes using HadoopViz. In this case study, the input is a set of straight line segments that represent the administrative borders in the whole world (e.g., countries and cities) which need to be visualized as shown in Figure 5.9(f). To visualize the input as lines in the final image, we define the five functions as follows: (1) No `smooth` function is needed. (2) The `create-canvas` function spawns an ImageMagick process in the background and sends it a '`viewbox`' command to initialize an image with the desired size. (3) The `plot` function projects a line from the input to the image space, and send the ImageMagick process a '`draw line`' command with the projected boundaries. As HadoopViz needs to transfer canvases from *mappers* to *reducers*, it cannot simply move a running process. So, to transfer a canvas, we close the ImageMagick instance and transfer the generated image across network. (4) The `merge` function uses the '`draw image`' ImageMagick command to draw one image onto the other image. (5) The `write` function closes the ImageMagick process of the final canvas, retrieves the image created by that instance, and writes it to the output as a file.

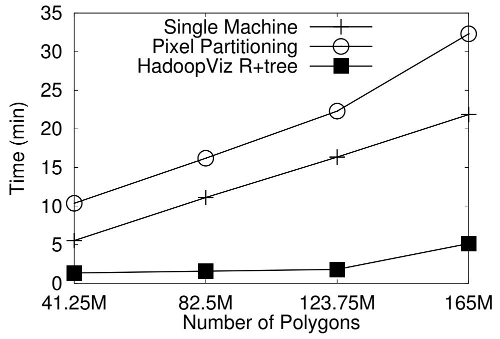
5.5 Experiments

This section provides an experimental evaluation for HadoopViz to show its extensibility and scalability. All HadoopViz experiments are conducted on a cluster of 20 nodes of Apache Hadoop running on Ubuntu 10.04.4 machines with Java 1.7. Each machine has an Intel(R) Xeon E5472 processor with 4 cores @3 GHz, 8GB of memory and a 250GB hard disk. The HDFS block size is 128 MB. All single machine experiments run on a machine with 12 cores of Intel(R) Haswell E5-2680v3 CPU @ 2.50GHz and 1TB memory. In all experiments, we use total execution time as the main performance metric of our experiments.

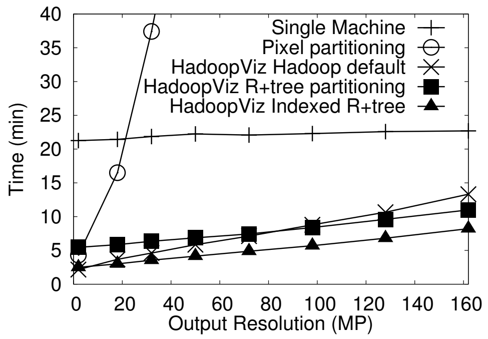
For the input data, we use three real datasets extracted from OpenStreetMap [38], namely, `nodes`, `ways`, and `lakes`, in addition to one satellite dataset from NASA called `nasa`. The `nodes` dataset (1.7 billion points) is used for case studies I and III, the `ways` dataset (165 million polylines) for case studies II and VI, the `lakes` dataset (8.4 million



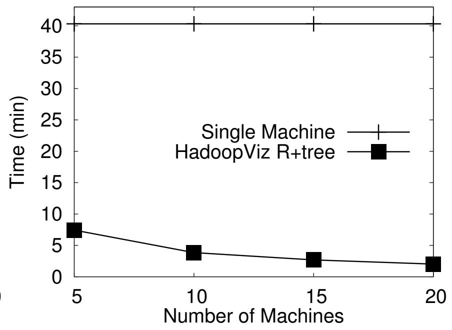
(a) Case studies



(b) Input size - case study II



(c) Image size - case study I



(d) Cluster size - case study III

Figure 5.12: Single Level Image Performance

polygons) for case study V, and the `nasa` dataset (14 billion points) for case study IV. For experiments repeatability, OpenStreetMap and NASA datasets are made available at the two following links, respectively. http://spatialhadoop.cs.umn.edu/datasets.html#osm - http://e4ftl01.cr.usgs.gov/MOLA/MYD11A1.005/.

Single-Level Visualization

For single-level image visualization, we consider seven different algorithms where each is applied in suitable experiments. (1) A *single-machine* algorithm which loads the whole dataset into main-memory, smooths records in memory, scans records and plots them to an in-memory image, and finally writes that image to the output. (2) Distributed *pixel-level-partitioning* [76, 77] which is implemented in HadoopViz using a uniform grid partitioning with a grid size equal to image size. (3, 4, 5) HadoopViz with default Hadoop partitioning, grid, and R+-tree partitioning. (6, 7) HadoopViz with grid-based and R+-tree-based indexes which utilize an existing index to avoid the spatial partitioning phase.

Figure 5.12(a) compares the performance of HadoopViz with R+-tree partitioning to the single-machine algorithm, as they are both applied to the six case studies described in Section 5.4 to visualize a 32 mega-pixels image. This experiment shows an order of magnitude speedup of HadoopViz as compared to traditional single machine algorithms. It also shows the flexibility and efficiency of HadoopViz when used with different image types using the single level algorithm. For example, it visualizes 14 Billion points of NASA data in a 90 seconds. Case study V runs relatively faster as it operates on the much smaller `lakes` dataset. The figure also shows the power of HadoopViz as it provides a 48X speedup of the single-machine ImageMagick visualization package.

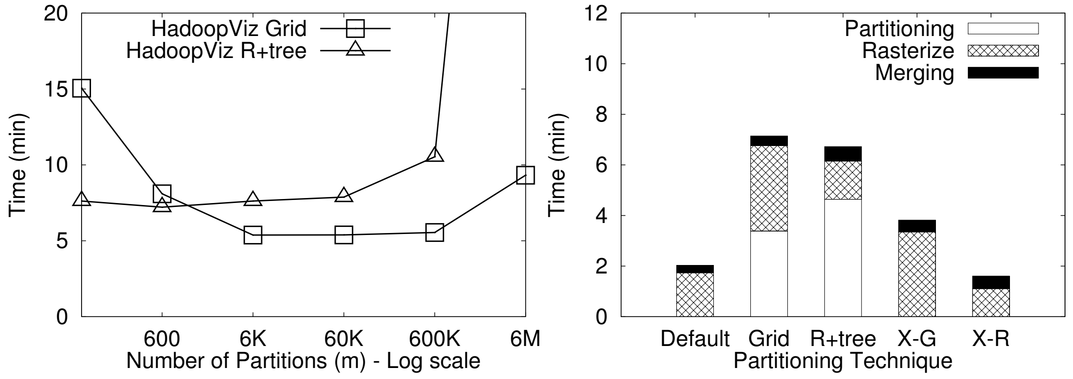
In Figure 5.12(b), we change the input size of the road network, by sampling at 25%, 50% and 75%, while fixing the image size at 32 mega-pixels. HadoopViz outperforms both single machine and pixel-level-partitioning algorithms for all input sizes. The performance of pixel-level-partitioning is very poor as it has to process 32 Million partitions, as one per pixel. At this image size, pixel-level-partitioning is even slower than a single machine which relies on a huge main memory of 1TB.

Figure 5.12(c) gives the effect of changing the desired image size from 2 to 160 mega-pixels. This experiment runs on the scatter plot case study as it does not contain a `smooth` function which allows for the use of default Hadoop partitioning. This figure shows clearly that pixel-level partitioning is only useful with a small image sizes of less than 20 mega-pixels. The single-machine algorithm is slightly affected by the image size as it performs all processing in main-memory after the file is loaded from disk. On the other hand, all techniques in HadoopViz scale very well with the generated image size making it useful for generating both small and big images. As described in Section 5.1, the default Hadoop partitioning performs better with small images while spatial R+-tree partitioning performs better with big images. This experimentally justifies the decision made in the partitioning phase where it switches from default Hadoop partitioning to spatial partitioning when image size grows larger than a block size. Although this condition holds at the points at 50 and 72 mega-pixels, while the performance crossover happens at 80 mega-pixels, the difference is very small and both techniques perform very similar. This experiment also employs the indexed R+-tree technique which skips the spatial partitioning phase by utilizing an existing R+-tree index constructed using SpatialHadoop [80]. As shown, this technique outperforms all other techniques as it gets the good performance of R+-tree partitioning without having to pay the overhead of partitioning.

Figure 5.12(d) shows how HadoopViz scales out with cluster size when visualizing a heat map as compared to a single machine algorithm. This figure also shows that HadoopViz scales out very nicely with cluster size as it parallelizes all of the three phases of the algorithm. Even with five machine, it outperforms the single-machine algorithm.

Figure 5.13(a) gives the effect of tuning number of partitions m on HadoopViz running with grid and R+-tree partitioning. We cannot compare with default Hadoop partitioning as number of partitions is automatically calculated by HDFS according to input size and HDFS block capacity. We change number of partitions from 60 to 6 Million to cover the spectrum of all values which are all shown on a log scale. Grid partitioning performs poorly on both extremes where it suffers from load imbalance at $m = 60$ and 600, and huge processing overhead at $m = 6$ Million. On the other hand, R+-tree-based partitioning is very stable when m changes from 60 to 600K, then the performance suddenly drops at $m = 6$ Million. The reason of this huge drop is that the partitioning phase has to search 6M rectangles for each input record to find overlapping partitions. This incurs a huge overhead even with an optimized in-memory R+-tree index with $\log n$ search time, as opposed to constant time in uniform grid partitioning.

Figure 5.13(b) breaks down the time of single-level plot in HadoopViz into the three phases. In this experiment, we skip the `smooth` function to be able to apply default Hadoop partitioning. The indexed R+-tree and indexed grid techniques are denoted X-G and X-R, respectively. At the small image size generated in this experiment (4 MegaPixels), the default Hadoop partitioning performs very well where the plotting phase accounts for most of the time. On the other hand, both spatial partitioning techniques, grid and R+-tree, are much slower due to the overhead of the partitioning phase. Although the grid partitions very quickly, it performs poorly in the plotting phase due to load imbalance. Existing indexes save the partitioning time of both techniques making R+-tree much better than grid. To conclude, we recommend the use of R+-tree if the `smooth` and `plot` functions are complex to achieve a better load balance.



(a) Number of Partitions (m) (b) Performance breakdown

Figure 5.13: Single-level image tuning with case study II

Otherwise, a grid partitioning can be used when these functions are simple as it saves in the partitioning time. If no smooth function is required, default partitioning is good enough.

Multilevel Visualization

In multilevel visualization, we compare the performance of three techniques. (1) A single-level machine algorithm which builds the whole pyramid in main-memory and then dumps it to disk as one image per tile. (2) HadoopViz with default-Hadoop partitioning. (3) HadoopViz with pyramid partitioning.

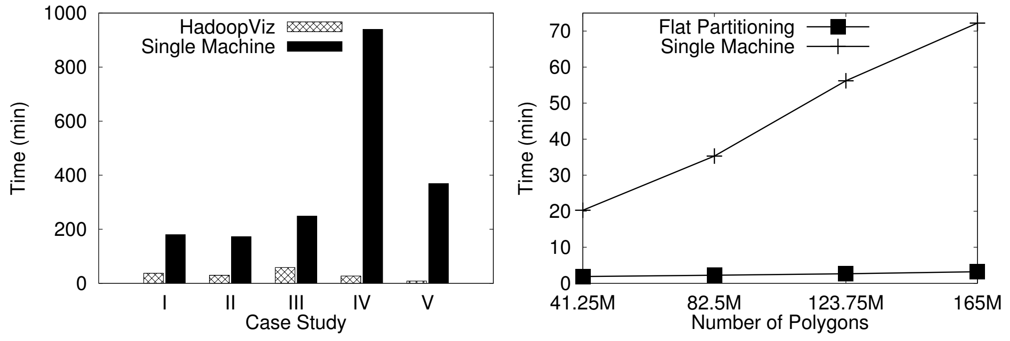
Figure 5.14(a) compares the performance of a traditional single-machine algorithm to HadoopViz for visualizing a multilevel image. In this experiment, we generate a pyramid of 11 levels, which resembles a 70 giga-pixel image at the highest level. The experiment shows up-to two orders of magnitude speedup of HadoopViz as compared to a single machine algorithm. The speedup is much higher compared to single-level experiments, in Figure 5.12(a), due to the huge output size which is written to a single disk in single-machine experiment, as opposed to the HDFS in HadoopViz. This experiment also shows the great power and flexibility of HadoopViz as these multilevel images are created using the same five abstract functions that were used to implement the single-level images. In other words, users do not need to do any additional effort, other than defining the five functions, to generate multilevel images.

Figure 5.14(b) gives the performance of generating a pyramid of six levels using default partitioning and single-machine algorithms. As the input size increases from 41M polygons to 165M polygons, the performance of the single machine drops as it needs to read and parse the whole input file. On the other hand, HadoopViz scales very well as the scanning of the input is done in parallel using the MapReduce framework.

Figure 5.15(a) gives the performance of both *default* and *pyramid* partitioning algorithms in HadoopViz while generating each level in the pyramid. This experiment confirms our earlier discussion that default-partitioning performs better at top levels while pyramid-partitioning performs better at deeper levels. At top-levels, default partitioning performs better as it avoids the overhead of partitioning while the cost of merging is still low. On the other hand, pyramid partitioning performs poorly due to the load imbalance as the top levels contain only a few tiles. At deeper levels, the performance of default partitioning drops due to the huge number of tiles that need to be shuffled over network and then merged. In this experiment, it failed at levels six and higher with an out-of-memory exception due to the huge pyramid size that has to be stored in each machine. On the other hand, pyramid partitioning performs better as it partitions the input into thousands of tiles improving the load balance. The running time increases again at deeper levels due to the exponential increase in the size of the output which cannot be avoided. According to the system configuration and the analysis made in Section 5.2, HadoopViz should use pyramid-partitioning for levels five and higher. Although default-Hadoop partitioning algorithm performs better at level five, the difference is slight and both algorithms perform well.

Figure 5.15(b) shows the scalability of HadoopViz with cluster size where we increase the cluster size from 5 to 20 and measure the performance of generating an image with 11 levels with $z_\theta = 4$. This experiment shows a near perfect scale out for both algorithms due to the parallelization of all of the phases in both algorithms. It also shows a huge speedup over the baseline of single-machine performance. Deeper levels (5 to 10) take much more time due to the exponential increase in number of tiles.

In Figure 5.15(c), we change the grouping granularity (k) to verify its effect on the performance of the pyramid partitioning algorithm. This parameter was introduced in Section 5.2 to control the trade-off between load balance and partitioning overhead by grouping multiple pyramid levels in one partition. As expected, a smaller value of k provides a poor performance as it produces too many partitions. On the other extreme,

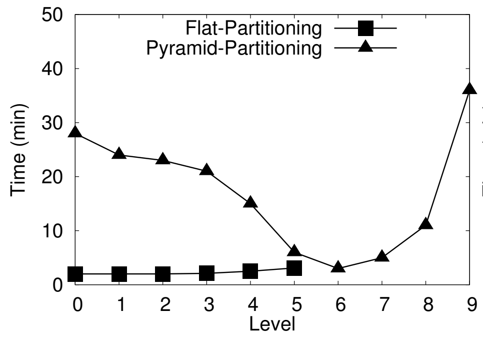


(a) Case studies (b) Input size - case study II

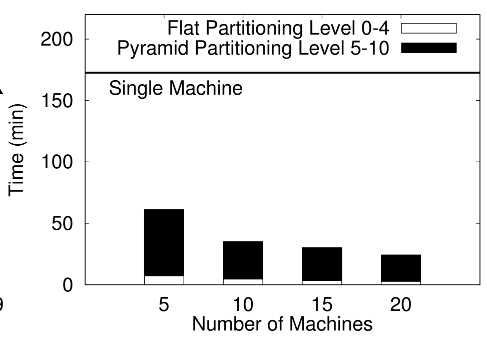
Figure 5.14: Multilevel Image Performance

using a large value of k produces a few partitions hurting the load balance. Both values of 3 and 4 provide good performance as they achieve a good balance between load balance and partitioning overhead.

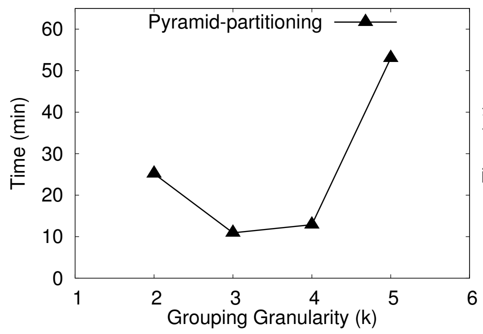
Figure 5.15(d) shows the percentage breakdown of the processing time of HadoopViz into the three phases. Default-partitioning algorithm spends most of its time in the plotting phase while the merging phase takes less time. The reason is that the plotting phase processes the whole input and produces partial pyramids of small sizes while the merging phase processes these partial pyramids in parallel to produce the final answer. On the other hand, the pyramid-partitioning algorithm finishes the partitioning phase quickly while most of the time is spent in the plotting phase which draws the final tiles directly and writes them to the output.



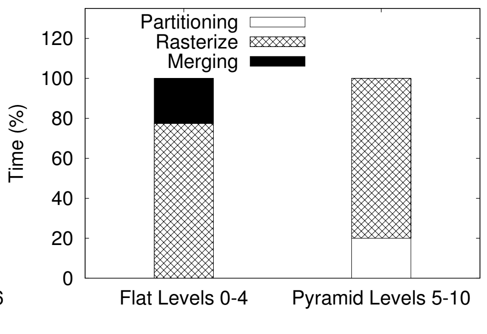
(a) Number of levels



(b) Cluster size



(c) Grouping granularity (k)



(d) Phases

Figure 5.15: Multilevel image performance tuning with case study II

Chapter 6

SHAHED: Interactive Exploration of Satellite Data

This chapter introduces SHAHED, an application which uses SpatialHadoop to query and visualize spatio-temporal satellite data. This application provides an example of how SpatialHadoop is used in a real application to support efficient query processing on big spatial data. It also shows how SpatialHadoop provides seamless scalability for non-technical end-users who are just interested in obtaining query results efficiently.

6.1 Introduction

Several space agencies such as National Aeronautics and Space Administration (NASA) [86] and European Space Agency (ESA) [87] collect enormous amounts of remote sensing data using satellites that continuously orbit the earth. For example, the Land Process Distributed Active Archive Center (LP DAAC) provided by NASA contains more than 1PB of data and is increasing on a daily basis [3]. This archive contains historical satellite data of a dozen of natural phenomena including temperature, vegetation, surface reflectance, and thermal anomalies, for every 250 square meter[2] for the whole world over the last 15 years. The archive is updated on a daily basis with newly collected data. Such archive is very useful and is actually being used by meteorologists

[2] Some datasets have a lower resolution with data for every one KM^2

and other scientists in several important applications, including detection of desertification [88], studies of land cover change [89], understanding ocean dynamics [90], and more generally in climate informatics [91] as well as in various governmental initiatives about climate change, e.g., see [92, 93, 94].

Although they are very useful, the huge size of such archives makes it hard for scientists and researchers to use. As a result, all prior research analysis attempts are either done offline (i.e., query response time may take hours or days) or based on a very small sample of the whole archive. A standard way to use the LP DAAC archive is to go through a web interface (e.g., Reverb [95]) in which the user provides some selection criteria including spatio-temporal predicates. Then, a list of files that match this selection criteria is returned. Such web interface methods are not efficient for many queries. For example, a simple selection query such as "What is the daily temperature of a specific location in the year of 2013?" would select 365 files which contain around 500 million points with a total download size of 2GB. All this data will need to be processed to select only 365 points that provide the query answer. What is even more challenging here is that the data returned from these files may be missing important information either because of the satellites misalignment or because the reading area was covered by clouds at the time the image is taken. Hence to answer such query, data has to be cleaned first on-the-fly before responding to the query. Such challenges and overhead make it very hard to use such available rich satellite data.

In this chapter, we present SHAHED; a system that lays out the necessary infrastructure to query, mine, and visualize big spatio-temporal satellite data. In particular, SHAHED downloads its data on a daily basis from the LP DAAC archive, resolves its data uncertainty, and locally indexes the downloaded data, making it ready for querying, mining, and visualization. SHAHED had to face two contradicting challenges; the need to process large-scale satellite data (in order of tens of tera bytes) and the need to provide real-time query response time. Large-scale data processing calls for relying on a MapReduce-based environment to allow an elastic computing environment that employs a large number of computational nodes. Meanwhile, real-time query response calls for *not* using such MapReduce environments due to their overhead in initiating job requests. Generally, MapReduce environments are made for batch queries rather than online queries. As a result, SHAHED deploys SpatialHadoop [96], a MapReduce framework for spatial data, for all of its offline functionality, which is needed to build rich and powerful index structures. Then, a separate query engine is used to retrieve the answer from the already built index structures without going through any MapReduce environment.

SHAHED is divided into two main sets of components; *data interface* and *user interface* components. The *data interface* is basically a background process that wakes up once everyday (e.g., at midnight) to download newly added data from NASA LP DAAC archive. Then, it triggers the execution of the following two consecutive modules: (1) The *uncertainty handling* module, which goes through the downloaded data to fill in the missing information. This is done by developing a two-dimensional smoothing technique over missing information. For efficient uncertainty processing, we use SpatialHadoop to scale up this module. (2) The *indexing* module, which takes the cleaned data from the *uncertainty handling* module, builds a spatial index for this data (using SpatialHadoop), and appends this new index to the current list of available spatial index structures in a way that forms a global spatio-temporal index over all available satellite data. In addition to being triggered daily, the *indexing* module is also triggered monthly and yearly, to combine the set of daily and monthly indexes into one bigger spatial index structure that covers data for a whole month and year, respectively. This is still done within the main spatio-temporal index structures maintained by SHAHED *indexing* module.

The *user interface* of SHAHED receives three kinds of requests from its users, namely, querying, mining, and visualization requests. Each request goes to a corresponding module. Hence, the user interface of SHAHED is composed of three main module, *querying*, *mining*, and *visualization*. In this chapter, we focus and discuss only the *querying* and *visualization* modules, while the *mining* module is out of the scope of our work. The *querying* module supports two kinds of queries: (1) spatio-temporal *selection* queries, where users can request a set of values (e.g., temperature or vegetation) for a certain spatial region over a certain temporal interval, and (2) spatio-temporal *aggregate* queries, which is similar to selection queries, yet we report the aggregate (e.g., average or maximum) of all the values within the specified spatio-temporal range. For higher efficiency, the *querying* module does not go through SpatialHadoop. Instead, it has its own separate query engine that returns the query answer efficiently in an interactive real-time response time; something that cannot be provided should we go with a MapReduce environment. This is achieved by exploiting the spatio-temporal index structure, built and maintained by the *indexing* module.

The *visualization* module supports two main functionality: (1) *spatio-temporal heat maps*, where users can request to generate a sequence of heat maps of certain values (e.g., temperature or vegetation) for a certain spatial region and over a certain temporal period. The sequence of heat maps are returned as a set of images as well as an animated video. This is a very important and needed functionality by meteorologists as they need to visualize the change of behavior over a certain temporal period, (2) *multi-level spatial heat maps*, where users can request a single multi-level heat map for a certain time instance over a certain spatial region. Such multi-level heat map allows the user to zoom in and out within the picture to get either lower or higher resolution heat maps in an interactive way, which is another important functionality requested by meteorologists. Unlike the *querying* module, the *visualization* module is not interactive. Instead, it works as a web service, where users submit their requests through a nicely designed web interface. Once a request is submitted, SHAHED exploits its index structure to retrieve the required data while generating the requested heat maps. Once this process is done, an email is sent to the user as a notification that the request is finished with a link to download the requested data. The time to satisfy a request heavily depends on the size of the area covered by the request and the length of the temporal period. Requests with large areas (e.g., the whole world) over a long time period (e.g., a whole year) may take an hour or so to satisfy. That is still acceptable as usually such requests do not need real-time response. This also allows us to comfortably use SpatialHadoop to scale up the heat map generation.

Our reported experience and use of SHAHED show that it is a very efficient system with a wide use. In terms of execution time, the *uncertainty* and *indexing* modules run as background processes triggered periodically. The process usually takes up to few minutes, and it does not affect the query performance. In the mean time, spatio-temporal selection and aggregate queries within the *querying* module are all supported with a real-time response time. This is mainly due to the fact that they have their own path to exploit the already built spatio-temporal index structure. Finally, generating a heat map within the *visualization* module may take up to few minutes. For example, we have generated a single heat map for the whole world (using 450 Million points) in less than three minutes using a cluster of only four quad-core machines. Such numbers show the scalability, efficiency, and usability of SHAHED system.

The rest of this chapter is organized as follows. Section 6.2 gives a background overview of NASA satellite data. The system overview of SHAHED is presented in Section 6.3. The *uncertainty*, *indexing*, *querying*, and *visualization* modules are described in Sections 6.4, 6.5, 6.6, and 6.7, respectively. In Section 6.8, we report experimental numbers from our use of SHAHED. Finally, the web interface of SHAHED is highlighted in Section 6.9.

6.2 Background

The Land Process Distributed Active Archive Center (LP DAAC) [3] stores historical satellite data of a dozen of natural phenomena including temperature, vegetation, surface reflectance, and thermal anomalies. This section gives a necessary background of such data archive.

Structure and Format of the LP DAAC Archive

The LP DAAC archive is organized in a hierarchical structure that makes it easy to locate files by dataset, time, and location. Figure 6.1 illustrates the structure of files in the LP DAAC archive organized in four levels. In the first level, files are organized by their data sets, where each data set is stored in a separate directory (e.g., temperature or vegetation). In the second level, each data set is temporally partitioned in daily partitions, each stored in a separate directory named by the day of this snapshot. In the third level, data in each snapshot is partitioned using a uniform grid over the whole globe. Each grid tile is identified by its two-dimensional coordinate in the grid, e.g., h21v06 represents the cell in column 21 and row 6. In the fourth level, each file contains a two-dimensional array of numbers, which represent the values (e.g., temperatures) for each point in the given region and time. Files are of the Hierarchical Data Format (HDF), which is a binary format where readings are arranged in a two-dimensional array that covers the associated tile. The size of the array is either 1200×1200, 2400×2400,

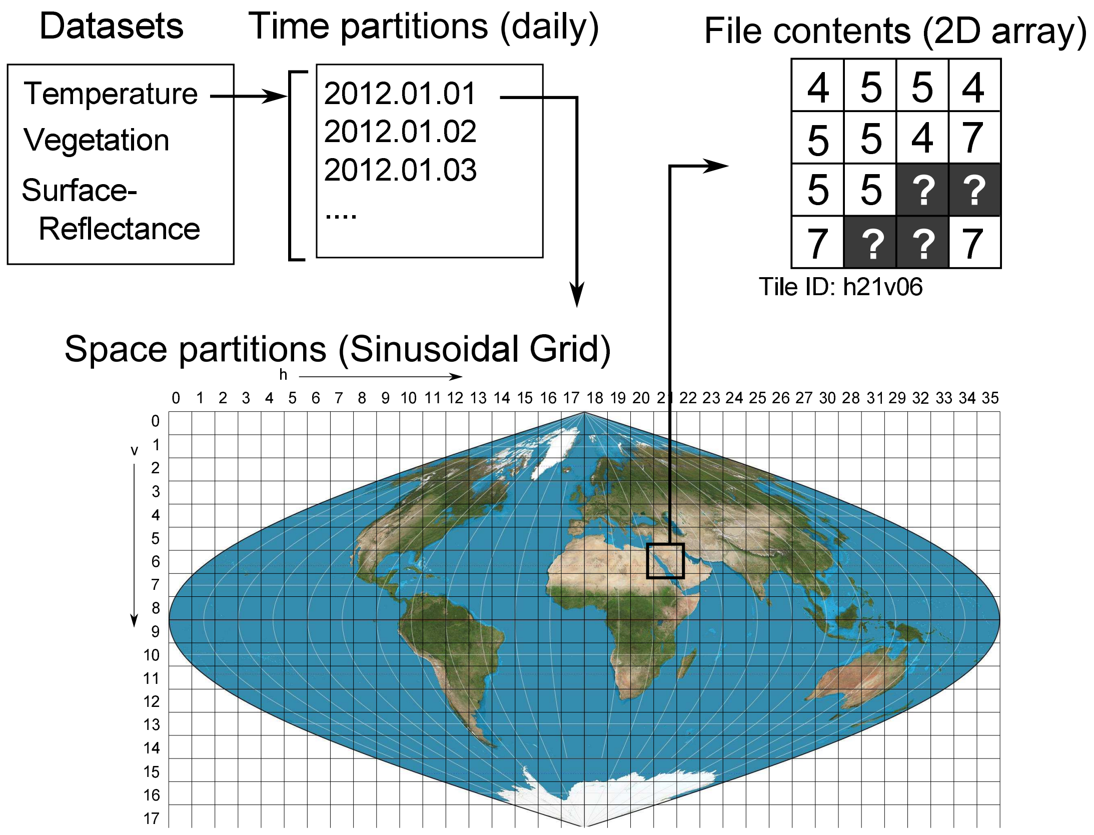


Figure 6.1: Structure of LP DAAC archive

or 4800×4800 depending on the resolution of the data set, where each value in the array represents an area of size $1000 \times 1000m$, $500 \times 500m$, and $250 \times 250m$, respectively.

The coordinates of each value in the array are not explicitly stored, but it can be computed using the sinusoidal projection as follows: Given the temperature data set where each tile is of size 1200×1200 and a point in tile h21v06 at the position (100, 100) in the two-dimensional array. To compute its latitude and longitude coordinates, we first calculate the point location in the sinusoidal space as: $x = 21 + 100/1200$ and $y = 6 + 100/1200$. Then, the latitude and longitude are computed as: $lat = (9 - y) \times 10$ and $lon = (x - 18) \times 10 \times \cos(lat)$. The same equations can be reversed to compute the position of a point in a file given a latitude and longitude offsets.

Data Retrieval from LP DAAC Archive

With its current hierarchical organization, LP DAAC provides a simple way to retrieve a certain value given the type of the data set, a temporal range, and a spatial range. First, the list of directories is scanned to locate the directory of the requested dataset type. Then, the directories are kept sorted by time and the given time range is translated to a range of directories to access. Finally, in the third level of the hierarchy, a two-dimensional grid index is constructed *on the fly*, which makes it easy and efficient to select the files that match the user specified spatial range. To build the spatial grid index, each tile has to be assigned a spatial range according to the tile identifier (e.g., h21v06). The spatial range is calculated using the sinusoidal projection, described in Section 6.2.

Unfortunately, such straightforward method is extremely inefficient when selecting a large set of data as this requires reading large number of files. It is also inefficient when reading a single value as an on-the-fly index will be built while a large file is retrieved just to get a single value. Such inefficiency makes it hard for scientists to access such valuable archive. This becomes our main motivation to develop SHAHED to make such valuable archive easily accessible to scientists.

Data Uncertainty in LP DAAC Archive

A main challenge in processing NASA LP DAAC files is the data uncertainty imposed by *missing* values. Missing values are mainly a result of one of the following four reasons: (1) Data in regions outside the earth. Depending on the angle in which the image was taken by the satellite, part of this image might be outside earth. This type of missing values can be detected when a point is converted from sinusoidal space to latitude-longitude space as it produces an invalid longitude value, i.e., less than -180 or larger than 180. (2) The specified data set is available only for land (e.g., land temperature) while the missing point is located in a water area (e.g., ocean). This type of missing data is detected by imposing a water mask and detecting points that fall within water areas. The water mask is provided by NASA as a set of HDF files at the highest available resolution, i.e., 250×250 meters. (3) Mis-alignment of satellites results in uncovered sharp strip areas of the earth. The strips may cover different areas of the globe based

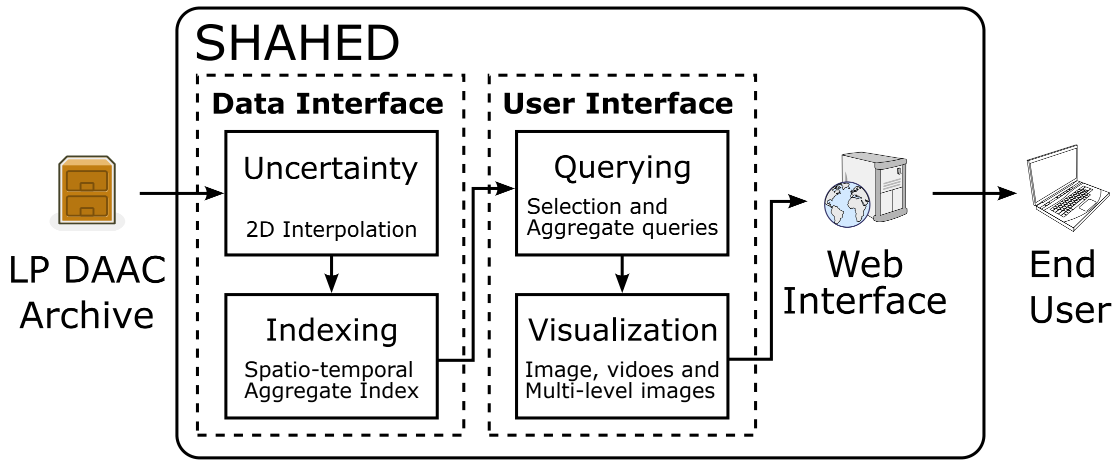


Figure 6.2: Overview of SHAHED

on earth and satellite movements. (4) The satellites were not able to read values of certain areas as it was covered by the clouds at the time of taking the snapshot. All missing values of the four above types are marked in the LP DAAC files by a special (fill) value. These missing values should be handled and cleaned to avoid any incorrect computations. In SHAHED, we do so by skipping the values of the first two types as they are truly irrelevant and should be missing, while we clean the data from the last two types using SHAHED *uncertainty* module.

6.3 Overview

Figure 6.2 gives an overview of SHAHED system architecture, which consists of four main modules, namely, *uncertainty*, *indexing*, *querying*, and *visualization*, described below:

The uncertainty module. This module is triggered on a daily basis to clean the newly downloaded daily data from NASA LP DAAC archive. The objective is to estimate and recover the missing values of the satellite data that emerge either from satellite mis-alignment or cloud coverage. To do so, the *uncertainty* module employs a two-dimensional interpolation function that smooths out the missing values within the two-dimensional space. More details are described in Section 6.4.

The indexing module. This module employs a novel multi-resolution spatio-temporal index for efficient data indexing and retrieval. It is triggered by two events: (1) The end of the *uncertainty* module, where the new cleaned data is added to the current spatio-temporal index structure, and (2) Periodic monthly and yearly execution to compact the index structure. More details are described in Section 6.5.

The querying module. This module receives spatio-temporal selection and aggregate queries from SHAHED users. Then, it exploits the spatio-temporal index structure with early pruning techniques to return the requested answer. More details are described in Section 6.6.

The visualization module. This module runs on top of the *querying* module to generate snapshot, multi-resolution, or animated heat maps based on a user query request. It employs novel visualization techniques that scale up the image generation process using the underlying MapReduce environment. More details are described in Section 6.7.

SHAHED also has one more module, namely, *mining* module which is *not* depicted in Figure 6.2, though it should lie between the *querying* and *visualization* modules. The *mining* module supports more complex and analysis queries, e.g., "find any outliers in a specific area over a certain range of time", or "Given a set of dates and areas of past earthquakes, find out if there is a certain pattern that appears before earthquakes". We omit the details of this module here as it is outside the scope of this thesis.

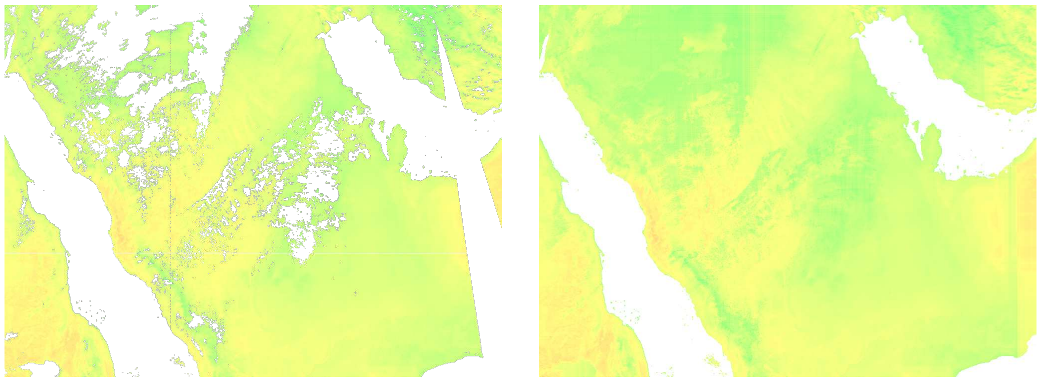
SHAHED is equipped with an easy-to-use map-based *web interface* layer that hides the complexity of the system through a simple and elegant web interface that accesses all SHAHED functionality. Details of the web interface are described in Section 6.9.

6.4 Uncertainty

This section describes the *uncertainty* module in SHAHED. We start by showing the effect of uncertainty on satellite data, then, we present our algorithm to recover such uncertain data.

Data Uncertainty

Figure 6.3 depicts the plotting of two heat maps for the same area of Saudi Arabia on the same day. The first plotting (Figure 6.3(a)) relies on the raw LP DAAC data that includes a lot of missing values, while the second plotting (Figure 6.3(b)) relies on the



(a) With missing data (b) Missing data recovered

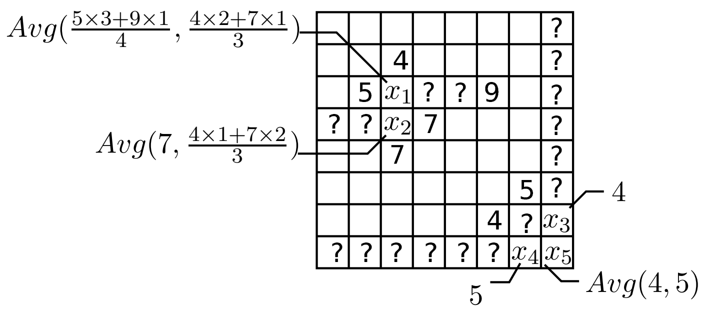
Figure 6.3: An example of hole recovery with a heat map

same set of data, yet, after applying our uncertainty recovering technique. Focusing on the figure with uncertain data (Figure 6.3(a)), we can easily distinguish the effect of two different factors: (1) blank curvy polygon areas in the middle and top left of the figure, which is a result of areas covered by clouds at the time of taking the satellite snapshot, and (2) a blank sharp rectangle coming from the bottom right corner of the figure and going close to the top of the figure, which is a result of satellite misalignment.

Recovering Uncertain Data

Per Figure 6.3(a), it is clear that having blank areas and missing information in satellite data significantly reduces its usage. As a result, we have developed a simple data recovery technique that aims to predict the missing values using a two-dimensional interpolation function. The basic idea is to calculate two estimates for each missing point, namely, x-estimate and y-estimate, which are calculated using a traditional linear interpolation function based on the nearest two valid points on the same horizontal and vertical lines, respectively, as the missing point. Then the estimated value is computed by taking the average of the two estimates.

Figure 6.4 gives an example of how the two-dimensional interpolation technique works. All cells marked with a question mark or x represent a missing value. Empty cells are non-relevant to this example and are omitted for clarity. The missing value x_1 is estimated by taking the average of the x-estimate $= \frac{5\times3+9\times1}{4} = 6$ and the y-estimate



$Avg(\frac{5\times3+9\times1}{4}, \frac{4\times2+7\times1}{3})$
$Avg(7, \frac{4\times1+7\times2}{3})$
4
5
$Avg(4,5)$

Figure 6.4: Two-dimensional interpolation

$= \frac{4\times2+7\times1}{3} = 5$. The x-estimate is computed using a traditional interpolation function of the two values 5 and 9 with distances 1 and 3, respectively. Similarly, the y-estimate is calculated from the values 4 and 7 with distances 1 and 2, respectively. For x_2, there is no valid value on the same row left to it. Thus, we compute the x-estimate as 7, as the nearest value on the same row, which is then averaged with the y-estimate as before. x_3 does not have a y-estimate as there are no other values on the same column which means it is estimated using only the x-estimate. Similarly, x_4 is estimated using only the y-estimate as there are no other values on the same row. Finally, for x_5, there are no valid values either on the same row or column. For this special case, we compute its estimate after x_3 and x_4 are estimated by taking the average of x_3 and x_4. Figure 6.3(b) depicts the heat map after filling all missing values using our recovery technique. This makes querying, mining, and visualizing satellite date much beneficial.

6.5 Spatio-temporal Indexing

As discussed in Section 6.2, the only available way to access LP DAAC data is through using on-the-fly indexes. However, this may end up to be very inefficient, even for simple queries. For example, retrieving all temperature values in a specific point over a period of 100 days would retrieve 100 files just to read one value from each file, which is extremely inefficient. Furthermore, aggregate queries such as computing the average temperature over a given area will retrieve a lot of points in this area before computing their average. To overcome such inefficiency, we equip SHAHED with a spatio-temporal index structure that is designed mainly to support the main SHAHED functionality of supporting spatio-temporal selection and aggregate queries as well as the visualization functionality.

Figure 6.5 gives the layout of our spatio-temporal index structure. The index has two orthogonal hierarchies as follows:

Temporal hierarchy. The index is organized in three temporal layers, each representing the whole dataset using a different temporal resolution. The lowest resolution layer contains yearly index structures, i.e., the whole data of one year is included in one index, while the highest resolution layer contains daily index structures. A monthly data structure is built only after the whole month is concluded. Similarly, the yearly index structure is built only at the end of the year. Hence, in Figure 6.5, we can see that there are 80 daily index structures from 2014, yet, only two monthly index structures for January and February, as the March index is not built yet. Similarly, the 2014 index is not built yet. It is important to note that indexes in one level are independent of indexes in other levels, which means that data is replicated three times. This replication is the storage overhead we choose to pay to provide efficient query processing, as will be described later.

Spatial hierarchy. Figure 6.6 gives the details of each yearly, monthly, or daily index structure. Each index by itself is an aggregate spatial index in the form of an aggregate quad-tree [53]. The lowest level of the aggregate spatial index partitions the data spatially using a uniform grid (Figure 6.6(a)). The reason we use a grid partitioning is that the data in NASA LP DAAC archive is uniformly distributed. Then, in each partition, the points are sorted using their respective Z-order values within the partition (Figure 6.6(b)). Finally, on top of the sorted points, we build an aggregate quad tree [53], which can be efficiently constructed when the points are already sorted [97]. The quad tree is prepended to the data file to ensure they are both stored in the same machine in HDFS. Each node in the aggregate quad tree includes a list of aggregates, namely, sum, minimum, maximum, and count, of all entries in its children nodes. Other aggregate values can be derived from the cached values such as average ($sum/count$) or range ($maximum - minimum$), and more functions can be added such as *variance* or *standard deviation*.

When the system is deployed for the first time, the index is bulk loaded with all data that is already available in the LP DAAC archive. This bulk loading process starts

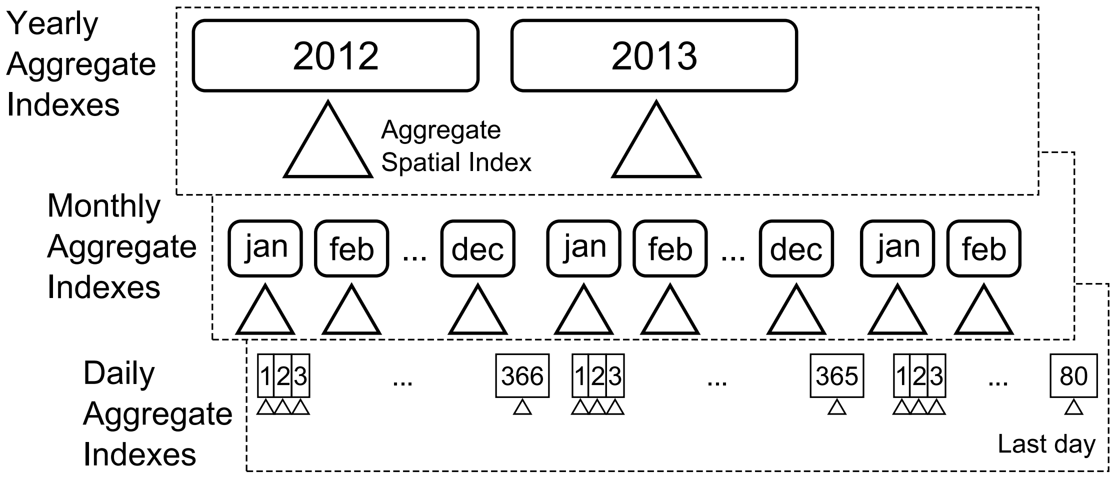


Figure 6.5: Spatio-temporal aggregate index

by building all the indexes in the daily temporal layer. Then, it builds indexes for the larger time interval (i.e., monthly) by merging daily indexes for all previous (i.e., completed) months. After that, it builds yearly spatial indexes for all past years by merging monthly spatial indexes. Once the index is bulk loaded with all existing data, the indexing module is then called at regular time intervals to add new daily snapshots of data as they are added to the LP DAAC archive.

To be able to realize such indexing layout, there are three main components that are used as building blocks to realize the spatio-temporal indexing in SHAHED: (1) Building a *stock* quad tree, which is basically, a template that will be used for all quad trees of the same resolution. This one-time process is done at the system start up and used throughout the system afterward, (2) Building one daily spatial index for a new snapshot added to the LP DAAC archive by NASA. This component is called on a daily basis to add any newly added snapshots, and (3) Building spatial indexes for larger time intervals (e.g., monthly or yearly), which is done by merging smaller indexes (e.g., daily or monthly). This component is called regularly by the end of each month or year according to the temporal resolution. In the rest of this section, we describe each of these three components in details.

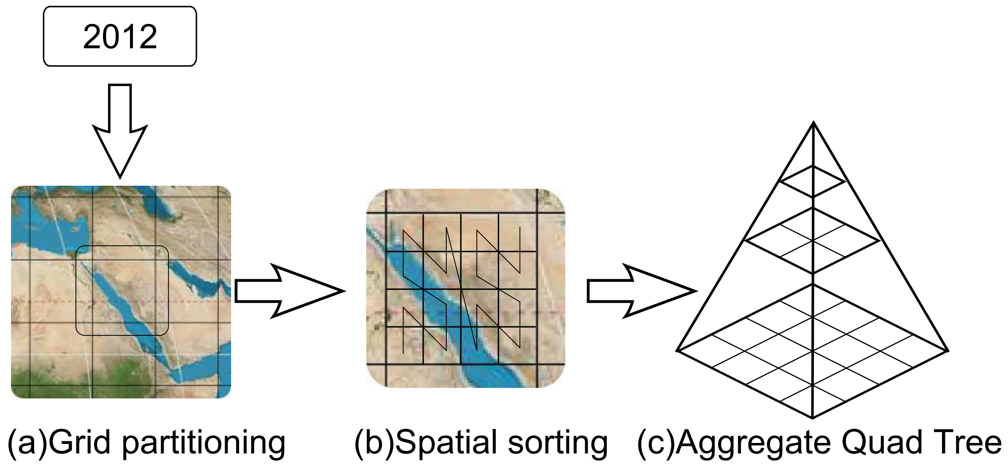


Figure 6.6: Aggregate spatial index

Building a Stock Quad Tree

One way to build the desired spatio-temporal index structure over existing and newly arriving data is to scan the whole data set; for each daily data snapshot, we (1) partition the data in a grid structure, (2) compute the Z-order of each point, (3) sort the points according to their Z-order value, and (4) construct the aggregate quad tree. The main overhead here is in constructing the aggregate quad tree, as partitioning is already done within the data archive itself as described in Section 6.2, while computing the Z-order of a point is straightforward. However, we aim to make the index construction process more space and time efficient by making use of the following two properties of the data stored in tile files: (1) In each tile, the points are uniformly distributed in the space covered by the tile, and (2) There are only three available tile sizes, 1200, 2400, and 4800. These two properties make the quad trees constructed on these files share a lot of similarities, which allows us to construct *stock quad trees* and reuse them to save space and time while indexing. In a one-time offline phase, three stock quad trees are constructed for tiles of sizes 1200, 2400, and 4800. Later on, these stock trees are used while constructing and querying tiles. Below, we describe the structure of the stock trees, their construction process, and how they are used to index tiles.

SHAHED constructs one quad tree for each of the three resolutions supported by NASA, 1200, 2400 and 4800. These quad trees are built at system start up and are kept in main memory to use while building a quad tree index for a tile or while querying one of the constructed indexes. To build a stock quad tree of a specified resolution (e.g., $res = 1200$), we start with a two-dimensional array of size $res \times res$. The values in this array are not relevant to indexing as they represent non-spatial values such as temperature. Each value is assigned a coordinate (x, y) equal to its position in the two-dimensional array. This means that both x and y are integers in the range $[0, res)$. Using array position as coordinates has two advantages. First, it makes it easier and straight forward to compute Z-order values by interleaving bits from the two integer values. Second, it generates exactly the same structure for all tiles with the same resolution regardless of its position on the map which allows us to reuse the stock quad tree for all those tiles. Once coordinates are assigned to points, their respective Z-order values are computed and the points are sorted. While sorting the points, we keep track of the final position in the sorted list for each entry in the original array. This mapping is kept in a *lookup table*, which helps us later while indexing actual data by mapping each value directly to its position in the sorted array without repeating the computation of Z-order values or the sort algorithm.

Once the points are sorted, the quad tree is constructed based on the sorted order as described in [97]. Each node in the quad tree is assigned a unique ID, $start$ and end positions. The $start$ and end positions specify the range of values in the sorted order covered by this node. By the properties of the Z-curve, points covered by any node in the quad tree are contiguous in the sorted order. We start by creating the root node of the tree with $(ID = 1, start = 0, end = res \times res)$. Under the root node, four children nodes are created by partitioning the space into four quarters. The range of values covered by the root node is divided into four partitions, one for each child node. These partitions are found by detecting where the two high-order bits of Z-order values change from $00 \rightarrow 01 \rightarrow 10 \rightarrow 11$. The split process is repeated for each node as long as number of records in the node is larger than the capacity of a leaf node (e.g., 100). When four child nodes are created, they are assigned the IDs $PID \times 4 + i$, where PID is the ID of the parent node and $i \in \{0, 1, 2, 3\}$ is the child number. More details of the quad tree construction from Z-curve ordered values are in [97]. Notice that the stock tree does not hold any actual values of the input array as these values are stored in the actual quad tree for each indexed file. The stock quad tree contains mainly the structure of the tree as well as the *lookup table* created while sorting the points by Z-order values.

Building a Daily Spatial Index Structure

By the end of each day (at midnight), a background process is triggered by SHAHED to collect all newly inserted snapshots from the LP DAAC archive and build a spatial index for each one. Since the data in the archive is already partitioned using the sinusoidal grid, the process is simplified to building an aggregate quad tree for each tile. In this step, a tile of a standard resolution needs to be indexed. As these indexes are independent, this step can be done in parallel, where each tile is processed by a different machine.

The input is a two-dimensional array V of size $res \times res$, where res is the resolution of the tile (e.g., 1200) and the output is an aggregate quad tree that indexes the input values V. The quad tree is initialized with a header that contains two values res and c, where res is the resolution of the tree and c is the cardinality of the tree, which indicates the number of values stored at each location. For a newly constructed quad tree from a daily snapshot, c is set to *one*. When multiple quad trees are merged, c is updated to reflect the number of values at each point as described later in the merge process. To build the quad tree, we first sort these points in V according to their Z-order values. Instead of recomputing the Z-order values and sorting them, the *lookup table* of the *stock quad tree* with the same resolution is fetched and used to map each point to its position in the sorted values. Sorted values are stored in a one-dimensional array V' of size res^2. The sorted array is filled in linear time by mapping each value from V to its position in V' directly using the lookup table. Once the sorted array is ready, the tree can be queried using the structure stored in the stock quad tree, which is shared among all quad trees of the same resolution. Since the stock quad tree is kept in main memory, and the size of the sorted array is the same as the original array, this quad tree index (without aggregate values) is considered a zero-overhead index in terms of storage. It is also very efficient because the quad tree structure is kept in main memory.

After the values are sorted, the next step is to compute the aggregate values in each node of the quad tree. Aggregate values are stored in a hash table in the quad tree using node ID as the key. To fill in this hash table, we traverse the tree in a bottom-up manner starting with leaf nodes. For each leaf node, the set of values stored under this node are scanned by obtaining the *start* and *end* positions from the stock quad tree and iterating over them in the sorted list V'. All supported aggregate functions are calculated in a linear time and stored in the hash table. For non-leaf nodes, the aggregate values of the four child nodes are obtained from the hash table and are further aggregated to compute the aggregate values of the parent node. This procedure is repeated until the root node is reached. With the addition of the aggregate values, the overhead of the index is no longer zero but is still minimal compared to a fully structured quad tree.

Temporally Merging Spatial Index Structures

The constructed daily indexes are efficient for answering selection and aggregate queries on a specific day or a small range of few days. However, answering a selection query over a period of one year would still be inefficient as it requires searching 365 quad trees. To overcome this challenge, SHAHED regularly combines these daily trees into larger trees, where each tree covers a whole month or a year. This process is triggered at regular intervals (i.e., monthly or yearly) and it merges smaller trees to build a larger tree. For example, at the end of each month, this process is triggered to combine all daily indexes constructed for that month to form one quad tree that covers the whole month.

The input of the merge step is a list of quad trees of the same space (e.g., 1200×1200) and time (e.g., daily) resolution, while the output is one quad tree of the same space resolution, but lower time resolution (i.e., larger time interval) that includes all values in all input trees. The structure of the merged tree is the same as the input trees. This makes it easier when further merging the output tree (e.g., monthly) into even larger trees (e.g., yearly) using the same merge algorithm. The header of the merged tree is initialized with a space resolution res equals to the resolution of the input trees and a cardinality c equal to the sum of the cardinality of all input trees. The sorted values V' of the output tree is formed by merging the values of all input trees while keeping them in time order. This step is simple because all input lists V' are already sorted and they are all of the same size. Simply, we go over all trees in a round-robin fashion and grab one value out of the sorted values V' of each tree and store it in the output. This is repeated until all values from all trees are consumed. Notice that the memory footprint of this algorithm is minimal as the merge step can be done directly within the external storage. In case an input tree has a cardinality $c > 1$, each iteration reads c values from this tree instead of one value to keep the resulting values sorted temporally. For example, if an input tree represents a 30-day month, the 30 values are read as one block and written to output in this order.

After the sorted list V' is calculated, the final step is to compute the aggregate values in the output tree nodes. Notice that although the output tree contains more values, it has the same number of nodes with the same structure as all input trees. Think of it as another tree with the same number of records, where each record contains a list of values instead of one value. The query processor uses the cardinality c to determine the number of values in each record. Having the same number of nodes with the same structure simplifies the calculation of the aggregate values in the output tree. The aggregate values of a node in the output tree is calculated by aggregating all values in the corresponding nodes with the same ID in all input nodes.

6.6 Query Processing

The spatio-temporal index introduced in SHAHED supports two types of queries, *selection* and *aggregate* queries. In selection queries, a set of values are returned in a given spatio-temporal range, while in aggregate queries, only aggregate values (e.g., average) are returned for the selected range. To provide an interactive query answer and avoid MapReduce overhead, both queries run on a single machine without MapReduce.

Selection Queries

In spatio-temporal selection queries, the input is a spatial rectangular range and a temporal range of dates; the answer is all readings in the specified range. For example, *find all temperature values in Minneapolis area from Feb., 10, to March 15, 2013.* The query processing runs in three steps, *temporal filter*, *spatial filter*, and *spatial refine.* (1) In the *temporal filter* step, the temporal index with the lowest granularity (i.e., year) is visited first, and if a partition in that level is completely contained in the specified temporal range, this partition is added to the selection list and the temporal range is updated to exclude the selected partitions. This process is then repeated on levels with higher granularity until the level with the highest granularity is visited (i.e., daily) which is guaranteed to cover any remaining parts in the temporal range. (2) In the *spatial filter* step, the grid in each temporal partition is used to select grid tiles that overlap the spatial range. Tiles that are completely contained in the query range are directly copied to output without further processing as all values in them are in the answer, while partially overlapping tiles are further processed in the next step. Notice that the same grid is used in all temporal partitions which allows us to run this step once on one grid and reuse the answer with all other temporal partitions selected by the first step. (3) The *spatial refine* step processes tiles that partially overlap query range to select values that are inside the query range. Since each tile is indexed using a quad tree, the quad tree is processed to select points that satisfy the spatial range. Notice that no temporal filtering is required because we only match temporal partitions that are completely covered by the query range. No partially overlapping partitions are ever selected.

To process the range query on the quad tree, we first transform the query range from the latitude-longitude space to the sinusoidal space in which quad trees are created, and then run the range query on them. First, we apply the sinusoidal projection to each dimension in the query range to transform it to the sinusoidal space. Then, for each tile matched by the spatial filter, the query range is clipped to the range covered by this tile so that the clipped range is completely contained in the tile. The clipped query range is then normalized to the resolution of the tile such that the coordinates of the query range are integers in the range $[0, res]$ where res is the resolution of the tile, $res \in \{1200, 2400, 4800\}$. This normalization is done to transform the query range to the space of the quad tree as all points in the quad tree have coordinates in the range $[0, res]$. Finally, the stock quad tree of the matching resolution is processed with a traditional range query starting at the root and going deeper in the tree as needed. At each node, if the minimum bounding rectangle (MBR) of this node is completely contained in the query range, all values under this node are returned. If the MBR of the node partially overlaps query range and it is an internal node, the four child nodes under this node are visited and their MBRs are tested in the same way. Otherwise, if it partially overlaps the query range and it is a leaf node, all points under this node are scanned and only those in the query range are returned.

To retrieve all values in a node, the *start* and *end* positions for this node are retrieved from the stock quad tree and the values in the range $[c \times start, c \times end)$ in the sorted values V' in the tree are retrieved, where c is the cardinality of the tree. Notice that all spatial attributes are kept in the stock quad tree which is completely stored in memory while only the non-spatial values (e.g., temperature) are stored in the aggregate quad tree on disk. This means that the range query is entirely executed in the main memory and only matching values are retrieved from disk which makes this range query algorithm optimal in terms of the amount of data read from disk.

Since all temporal partitions selected by the *temporal filter* step are indexed using the same grid and quad trees, the result of a range query search can be reused in all temporal partitions. In other words, the spatial range query is executed only once on one temporal partition, and when matching values are to be retrieved from disk on a particular tile, they are retrieved from all quad trees built on the same spatial tile on all selected temporal partitions.

Aggregate Queries

Similar to selection queries, in aggregate queries, the user specifies a spatial and temporal range; the answer is all aggregate values supported by the index for data points satisfying the spatio-temporal range. A straightforward implementation for this query is to run it as a post processing step after the selection query. However, we apply a more efficient query processing technique that makes use of the aggregate values stored in the quad tree nodes. The query runs in three steps, namely, *temporal filtering*, *spatial filtering* and *aggregate calculation*. The first two steps are the same as the selection query except for one difference. In *spatial filtering* step, all tiles overlapping the query range are sent for further processing in the aggregate calculation step. In other words, tiles that are completely contained in query range are treated the same as partially overlapping tiles.

Then, in the *aggregate calculation* step, the aggregate quad tree in each selected tile is processed to compute part of the aggregate value. For each quad tree, the query range is first normalized as described in selection queries where range query dimensions are in the range $[0, res]$. The processing starts from the root of the corresponding stock quad tree. If a node is completely contained in the query range, the aggregate values of its contents are retrieved from the corresponding node in the matching tree and accumulated to the result. Otherwise, if a node partially overlaps the query range, its

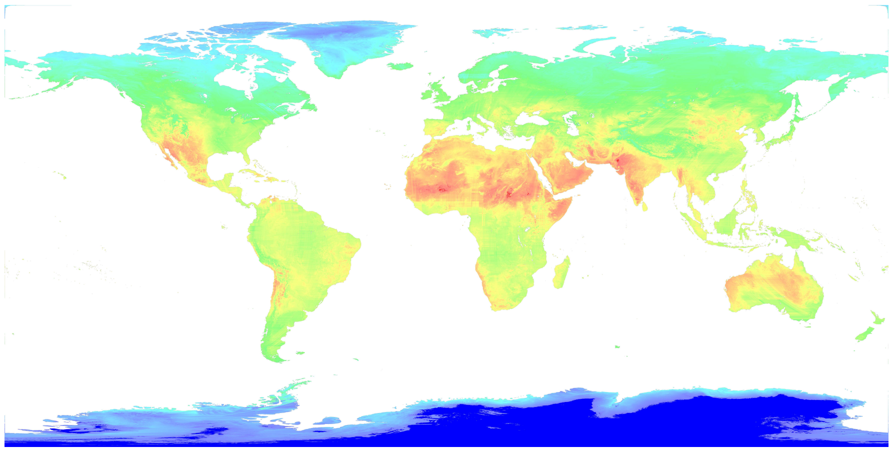


Figure 6.7: A heat map of temperature in the whole world generated by SHAHED

four children nodes are checked. This process is repeated until leaf nodes are reached. The points under a matching leaf node are scanned and the values of points contained in the query range are accumulated. This algorithm is much faster than retrieving all points in the range as the aggregate values of trees or nodes completely contained in the query range are directly retrieved without scanning the points stored in it.

6.7 Visualization

The values returned by spatio-temporal selection and aggregate queries need to be visualized for overview, analysis, and comparisons. This section describes how the results of the queries are visualized as heat maps. SHAHED supports three output formats, *static images* that represent a heat map of a selected dataset on a user-specified date, *videos* that visualize the changes of a dataset over a date range specified by the user, and *multi-level images* which represent a heat map for a specified date at different zoom levels allowing the user to navigate using pan, zoom, and fly-to interactions. The generation of both static images and videos is described in Section 6.7 while multi-level images are described in Section 6.7.

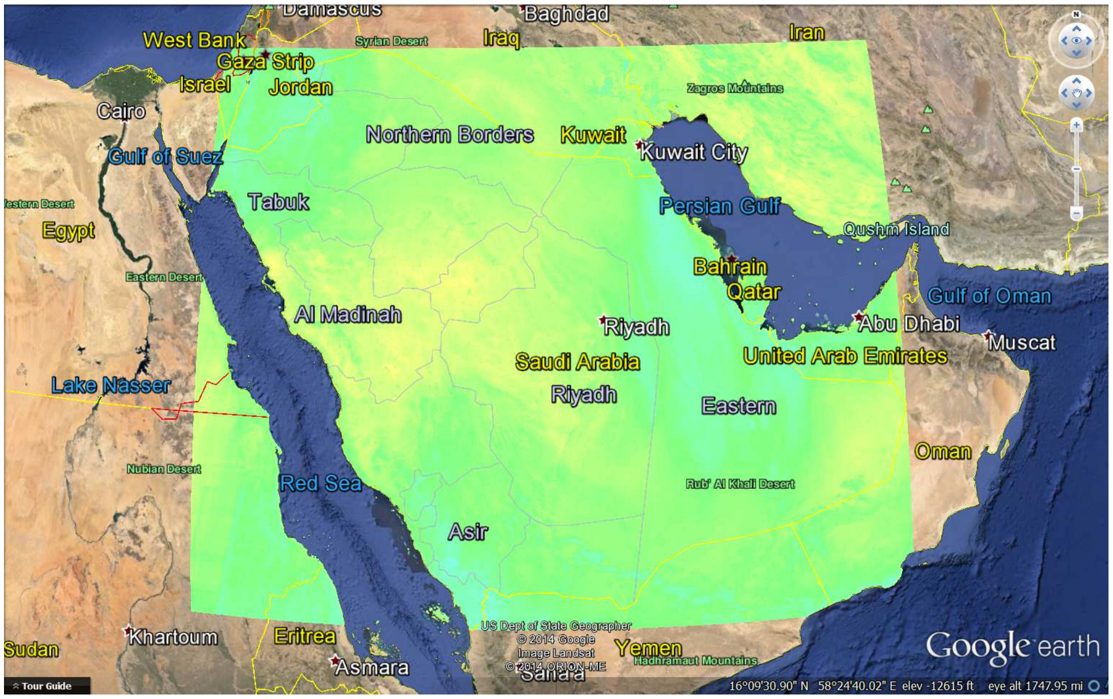


Figure 6.8: Heat map of temperature viewed on Google Earth

Heat Map Images and Videos

Figure 6.8 shows an example of a heat map for temperature on a selected date generated by SHAHED and visualized on Google Earth. SHAHED can also generate a sequence of images where each image represents a heat map of a day in a selected date range. This sequence can be combined in a video to show the change of values over time [1] . The generated heat maps give an overall picture of value distribution and can be included in a report or a presentation. A heat map is generated as a static image using a MapReduce program described below. A video is generated as a sequence of images each representing a heat map on each day and then these images are combined to make a video.

The heat map visualization operation takes as input a dataset, a specified date, a spatial range as a rectangle, and a size of generated image as width and height in pixels. The output is an image of the specified size, which represents the heatmap

[1] Please refer to an example at http://youtu.be/hHrOSVAaak8

for the specified area. The visualization operation works in three steps, *selection*, *tile draw*, and *overlay* steps. In the *selection* step, the archive is accessed to select grid cells that overlap with the query area. Each tile is assigned to a machine which becomes responsible of drawing this part of the heat map in the second step. In the *tile draw* step, each machine takes a tile and generates a heat map for the data in this tile. It starts by creating an image of size $width \times height$ pixels initialized with a transparent background. Then, points are read one-by-one and each point is plotted as a rectangle that represents the area it covers according to the resolution of the data (e.g., $1km \times 1km$). The color of the rectangle is selected from the spectrum of all colors according to the value of the associated point where the minimum possible value is colored blue and the maximum possible value is colored red. If multiple points map to the same pixel in the generated image, the average of their values is taken to smooth the image.

If the *recover* option is enabled in the uncertainty module (Section 6.4), missing values are automatically recovered as the input files are read so that the image becomes complete. The output of the *tile draw* step is a set of images all of the same size and each one representing part of the heat map for one tile. Finally, the *overlay* step overlays all the generated images on top of each other to generate the final picture. Since each machine plots part of the image and leaves other parts transparent, overlaying images on top of each other will generate the correct final picture. Along with the generated image, SHAHED also produces a KML file which allows the heat map to be displayed in a GIS software such as Google Earth as depicted in Figure 6.8. For video generation, multiple MapReduce jobs are executed by SHAHED each corresponding to one day in the range. Upon completion of all these jobs, a final call to a video generator is made to combine all images together in one video.

Figure 6.7 shows an example of a heat map of the temperature on April 8th, 2014 for the whole world generated from more than 300 files containing around 450 Million points. The resolution of this image is about 8000×4000 pixels and it took around five minutes to generate on a cluster of four nodes. All missing data is recovered in this image to give a smooth image that covers all land areas.

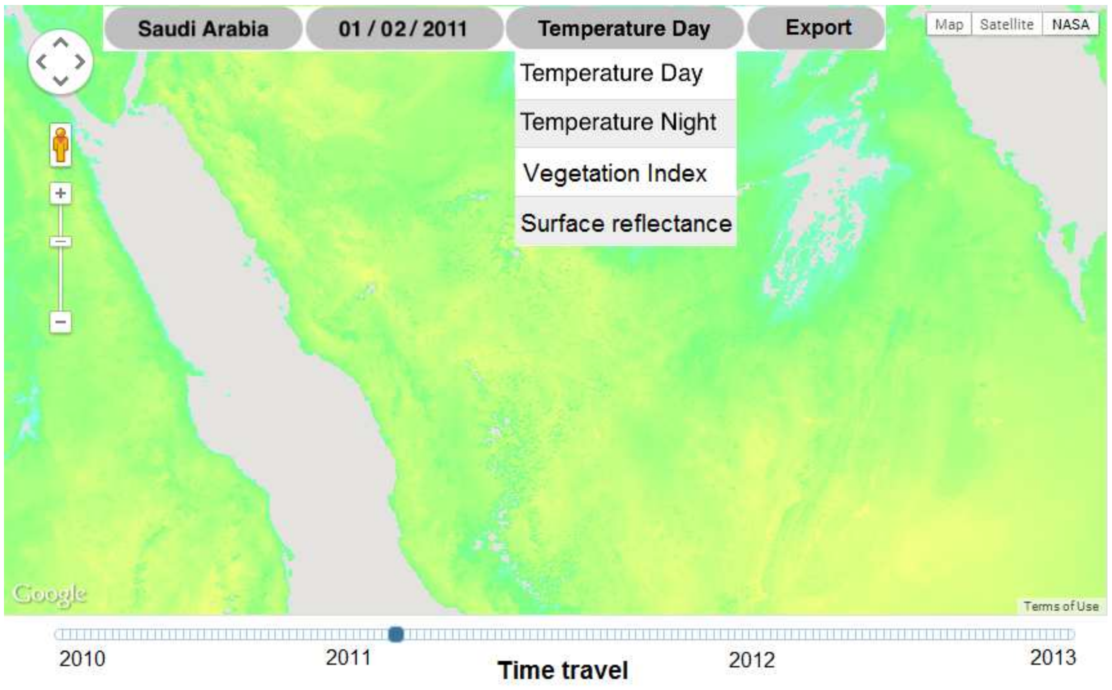


Figure 6.9: A multi-level heat map displayed on Google Maps

Multi-level Heat Maps

In addition to generating heat maps as images or videos, SHAHED is also capable of generating multi-level interactive heat maps where users can navigate through a map of the world and visualize the heat map of the visible area interactively using the standard navigation options, pan, zoom, and fly-to. Figure 6.9 shows an example of a multi-level heat map displayed as a layer on top of Google Maps. The main idea behind interactive heat maps is to precompute the heat maps for all regions and zoom levels. As the user navigates through the map, the system just picks from these precomputed heat map images and display them on the map. The challenge here is to generate all these images efficiently using MapReduce.

The multi-level heat map operation takes as input a dataset, a specific date, a spatial range, and number of zoom levels as integer value. The output is a set of images which represent heat maps at all supported zoom levels and regions in the specified spatial range. Figure 6.10 gives an example of a multi-level heat map generated at three zoom levels. Each image is of size 256×256 pixels and covers a different region based on its position in the pyramid. For example, the image at the top of the pyramid represents the whole range at the lowest zoom level. In level 1, the same area is represented by four images each of size 256×256 and so on. The operation runs in three steps, *selection*, *partition*, and *plot*. In the *selection* step, a spatio-temporal selection query is executed against the spatio-temporal index to retrieve all points in the user-specified range. Selected points are sent to the second step for further processing. In the *partition* step, a map function running in parallel on all machines scans the selected points and replicates each one to all overlapping pyramid tiles. Figure 6.10 illustrates an example where a point p is replicated to three tiles, one in each zoom level. Finally, in the *plot* step, each reducer takes a pyramid tile ID and all points in this tile and it plots an image of size 256×256 pixels which represents the heat map in the corresponding area. The heat map is generated exactly the same as described earlier in Section 6.7. The generated image is directly stored in the output folder with the naming convention `tile-i-x-y.png` where i is the zoom level, (x, y) is the position of this tile in the grid at zoom level i.

The above algorithm works fine, but it has a major drawback at higher levels of the pyramid. Since the tiles at higher levels in the pyramid cover larger regions, there will be more points replicated to these tiles. As an extreme case, the tile at the top level covers the whole space (e.g., the whole world), which means all points in the whole world will be replicated to this tile. For some queries where the selection area is very large, the set of selected points might contain several billions of points that should be plotted as a heat map by the machine that is assigned to the top of the pyramid. The processing of that tile might be prohibitively large. At the same time, this processing is unnecessary for two reasons. (1) Usually, images at higher levels of the pyramid do not have to be accurate as they just give a general picture. (2) The amount of details that a single image can contain is limited by number of pixels in it which is around $256 \times 256 \approx 64K$ pixels.

To overcome such unnecessary overhead, we introduce an optimization to the above algorithm to be more efficient without losing much of the output quality. In the partition step, instead of blindly replicating each point to all overlapping tiles, we adopt an *adaptive sampling* technique that only writes a random sample of points to higher levels

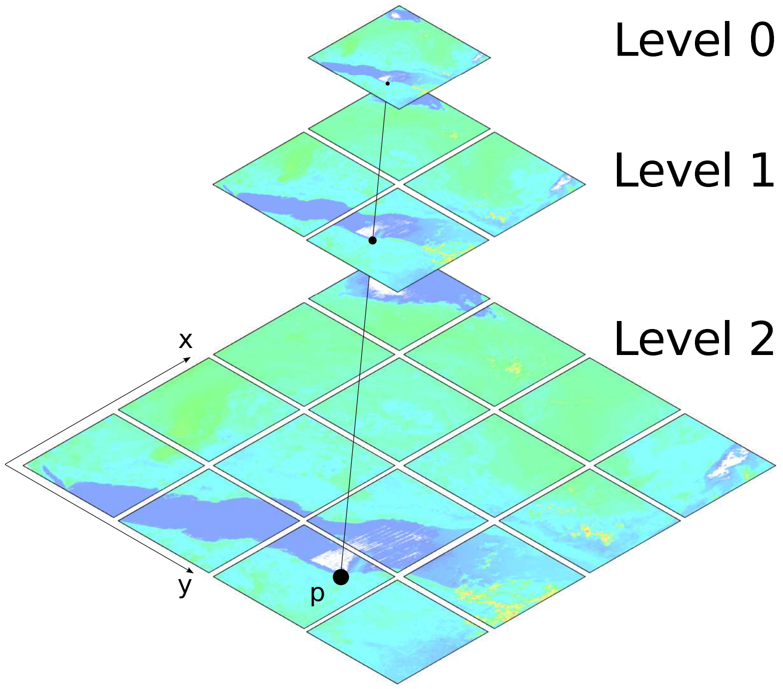
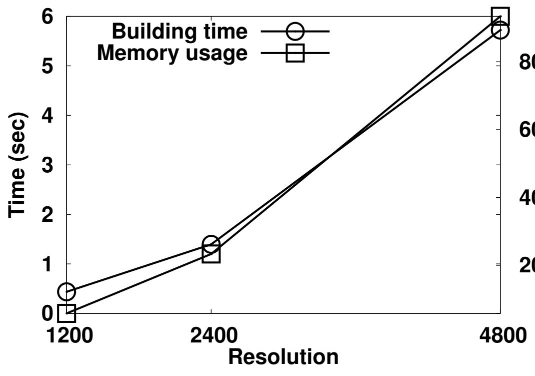
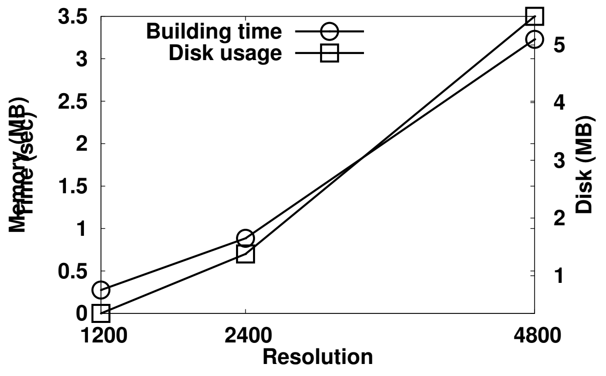


Figure 6.10: Pyramid of images generated for interactive heat map

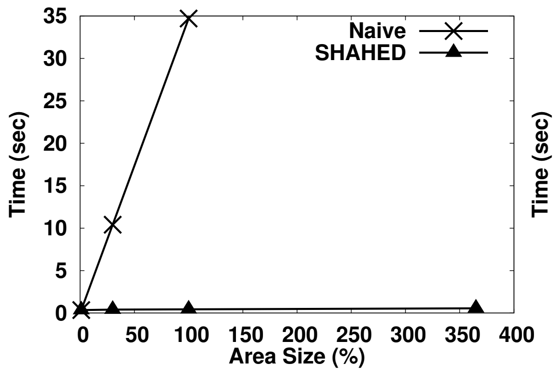
of the pyramid. Each point is replicated to each tile with a probability that is calculated adaptively based on the zoom level. The goal is to make the expected number of points in each tile equal to number of pixels in the generated image. This makes the load more balanced as each tile is expected to contain the same number of points regardless of its zoom level. To accomplish this goal, a point is replicated to a tile at level i with a probability $\alpha_i = \alpha_0 . \min\{1, \frac{T^2}{|P|/4^i}\}$, where α_0 is the base sampling factor described below, T is the edge size of a tile in pixels (i.e., 256) and $|P|$ is the total number of points in the user specified spatial range which can be easily calculated since data is uniformly distributed and the spatial density is known beforehand. The term $|P|/4^i$ gives the number of points covered by one tile at level i, assuming data in uniformly distributed. The adaptive sampling factor α_0 is a system parameter that can be adjusted to increase the quality of generated heat maps for this algorithm. The default value of α_0 is one and it can be increased to increase number of sampled points. If more than one point are sampled and they map to the sample image pixel, their values are averaged to produce smoother looking images with higher quality.



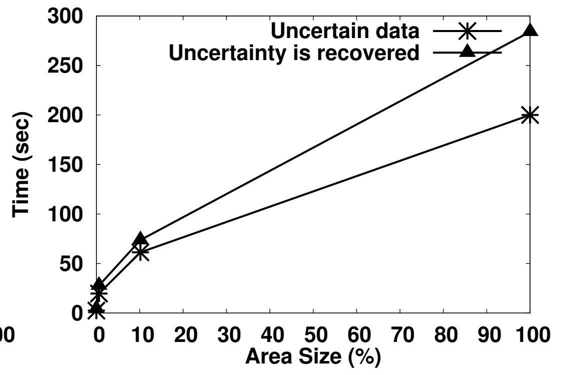
(a) Stock Quad Tree



(b) Aggregate Quad Tree



(c) Selection Query



(d) Visualization

Figure 6.11: Performance Experiments

6.8 Experiments

In this section, we report the performance of SHAHED as it is running live on a small cluster of five machines (one master and four slaves). This cluster has Hadoop 1.2.1 and SpatialHadoop 2.2 deployed on it and running on Ubuntu 12.04. All machines have 16GB of memory, 2TB hard disk and a quad core processor which gives a total of 16 processing cores. Unless mentioned otherwise, we use the temperature dataset (MYD11A1 V005), which is collected daily at 1200×1200 resolution.

Figure 6.11(a) reports both building time and memory usage when building the stock quad trees. The system contains only three quad trees of resolutions 1200, 2400, and 4800. As shown in figure, it takes only a few seconds to build the stock quad tree and the memory consumption is at most 100MB for the largest one. Most of this processing and memory overhead are directly resulting from the computation of Z-order values and keeping the *lookup table* which maps each item to its sorted index. This overhead is paid once when the system starts up and is used to save computation and disk overhead when building aggregate quad trees.

Figure 6.11(b) gives the time and storage overhead for building the aggregate quad tree. The disk and time overhead shown in the figure result mainly from the computation of the aggregate functions as sorting is done in linear time and does not involve any comparisons. By contrasting the two figures, we could get rough estimates of the processing and disk savings, which result from the use of stock quad trees. For example, for building a single quad tree, we save about six seconds for computing Z-order values and sorting them, which is shown in Figure 6.11(a). In addition, for an aggregate quad tree of resolution 4800, storing the tree structure of the tree would cost roughly the same as storing aggregate values as it consists of a few numbers attached to each node. Given that the LP DAAC archive contains millions of tiles, the stock quad tree would save tera bytes of storage and thousands of hours of indexing time.

Figure 6.11(c) gives the performance of a selection query for a single point over time intervals of 1, 30, 100, and 365 days. This figure compares the performance of the naive implementation which runs directly on non-indexed HDF files, with the performance of SHAHED which uses our spatio-temporal index. It is clear that the naive solution does not scale at all because it needs to open a different file to obtain each point in the query range. The performance of SHAHED is almost constant as all the points in the answer are contained in only few files due to the spatio-temporal index which packs data in monthly and yearly indexes.

Figure 6.11(d) gives the performance of the visualization component when generating static images of heat maps at different sizes. In this figure, we vary the area size by choosing four areas that cover a city, a country, a continent, and the whole world with areas of size 0.0002%, 0.6%, 10%, and 100%, respectively. For each one, a MapReduce job is run to generate the image and the end-to-end time is measured. This figure shows the great scalability of the visualization algorithm where it generates a heat map for the whole world in about 200 seconds without using the uncertainty module. It also shows the efficiency of the uncertainty module where the overhead is less than 50% when compared to visualization. Generating a video involves generating a list of static images and combining the result with a video converter.

6.9 Web Interface

SHAHED has a simple and interactive interface (depicted in Figure 6.13) that is easily accessible to end users from any web browser and provides access to all its functionality. The main area of the interface is occupied by a map which allows the user to easily navigate to any place either through pan and zoom or typing the place name to fly there directly. There is also a dataset selector that allows users to choose any dataset from the ones available in the NASA archives, e.g., temperature or vegetation. A temporal range can be set either by explicitly typing the start and end date or by using a slide control to set the requested time interval. Clicking the 'Overlay Data' check box adds an interactive heat map layer for the selected dataset on top of the current view, which can be navigated in a way similar to Google Maps. Finally, the user can click on one of the two 'generate' buttons to generate either an image or a video for the heat map of the selected region and time interval. In addition, the user can also choose an option to specify a spatio-temporal selection along with a dataset and the system will visualize the results as graphs for easy analysis and comparison. The details of each of these functionality is described below.

Spatio-temporal Queries

The user interface accepts spatio-temporal queries from the user and uses the spatio-temporal index described in Section 6.5 to answer these queries. In Figure 6.12, two points are selected. Then, for each selected point, a spatio-temporal selection query retrieves all values of the selected dataset (temperature) in the selected time interval and results are visualized in form of graphs. The interface also allows the user to select a range rather than a single point and an aggregate query on the selected area is executed.

Image/Video Generation

SHAHED provide the functionality to export a static image or a video that which represent the heat map in the specified region and time interval. For images, the user selects a dataset, specifies a region on the map, a date on the calendar, and an email address. SHAHED accepts these parameters and issues a MapReduce job in the back end which generates an image according to the specified user request. Upon completion

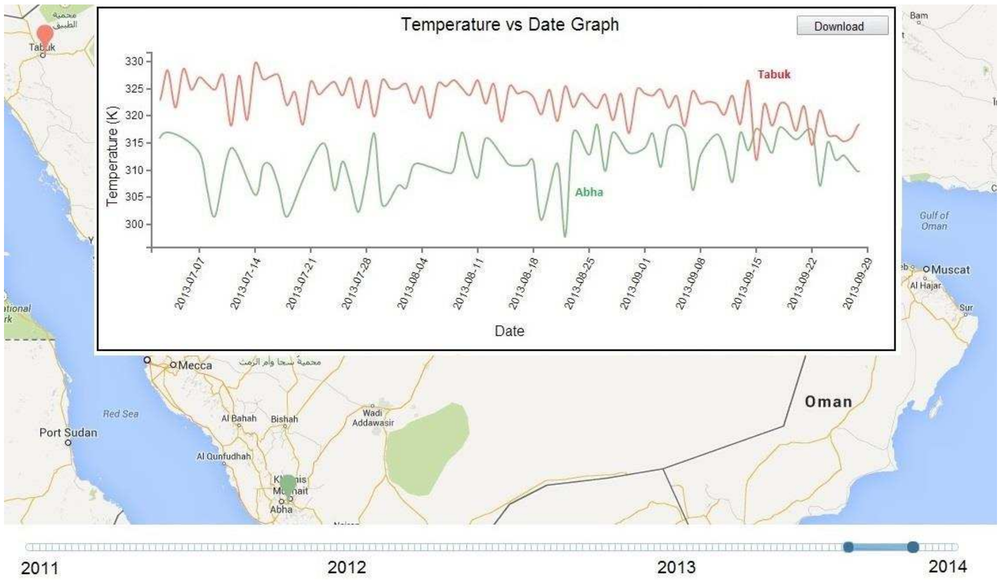
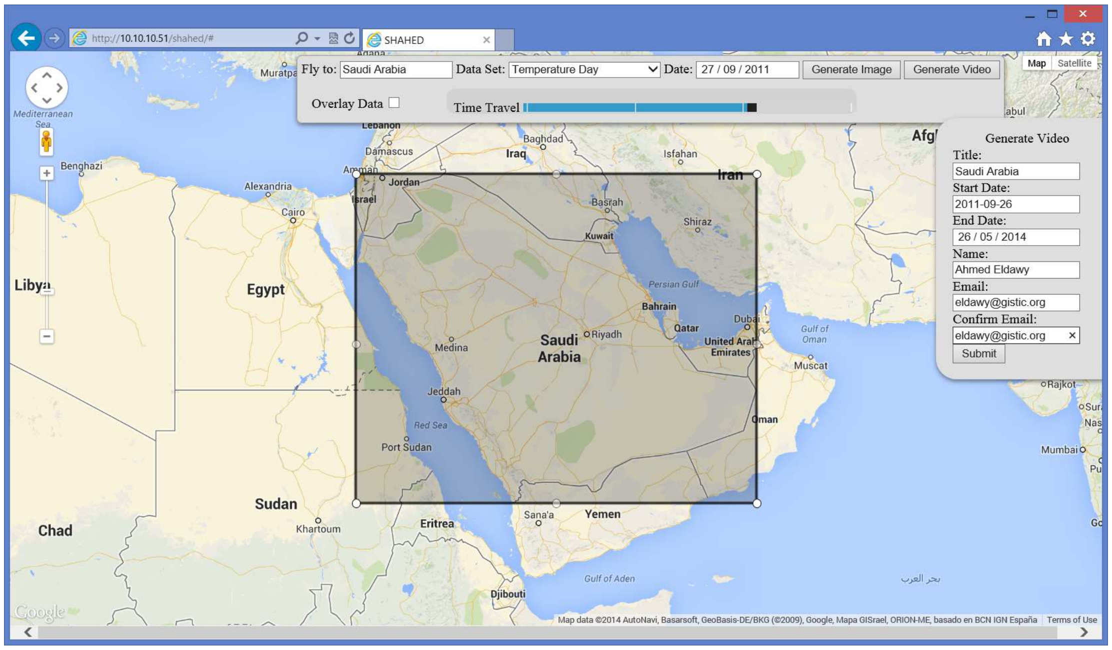


Figure 6.12: Compare the temperature at two points

of the job, the output is sent to the user as an email containing a link to download the requested image as both a static image and a KML file, which allows this image to be displayed on a GIS software such as Google Earth. For video generation, the user specifies all the previous information but a time interval is provided rather than one specific date. SHAHED runs a batch of MapReduce jobs, where each job generates a heat map for one day in the specified time interval. Once all jobs are completed, a call is made to a video generator to combine all generated images in a video that is finally sent to the user as a link to download on the specified email address.

Multi-level Heat Maps

Other than generating images and videos, SHAHED also allows users to browse the heat maps directly from the browser. Once the ‘Overlay Data’ check box is checked, a new layer is added to the map, which shows the interactive heat map of the current selected dataset and date. For the interactive heat map to be displayed, we precompute all heat maps of all regions and times of interest which allows the web interface to provide a smooth and interactive browsing experience for users. As the map view is changed, the browser automatically loads the set of images that cover the current view according to



Fly to: Saudi Arabia
Data Set: Temperature Day
Date: 27 / 09 / 2011

Generate Video
Title: Saudi Arabia
Start Date: 2011-09-26
End Date: 26 / 05 / 2014
Name: Ahmed Eldawy
Email: eldawy@gistic.org
Confirm Email: eldawy@gistic.org

Figure 6.13: Web interface

the zoom level and displayed region. For areas or times where the system does not have precomputed heat maps, users can still issue an export command which works with the raw data and generates the required heat map accordingly.

Chapter 7

Related Work

The recent explosion in the amounts of spatial data urged many researchers to propose new research attempts for handling them using big data frameworks. This chapter classifies existing work by considering six aspects of big spatial data systems. (1) The implementation *approach* which defines whether is implemented *on-top* of an existing system, *built inside* its core, or developed completely *from scratch.* (2) The underlying *architecture* which tells the primary processing model of the systems, such as, parallel DBMS, MPI, MapReduce, key-value store, array DB, RDD, or Hyracks. (3) The high-level *language* of the system, if any exists. (4) The existence of *spatial indexes* in the system and the types of these indexes. (5) The types of *queries* supported by the system, such as, range query, spatial join, computational geometry, or spatial data mining. (6) The support of big spatial data *visualization* in the system.

Table 7.1 (on Page 121) outlines the surveyed work in the area of big spatial data. Each row represents a system or a body of work related to big spatial data, while each column represents one of the six aspects we will discuss about big spatial data, namely, *approach*, *architecture*, *language*, *indexing*, *querying*, and *visualization.* The rest of this chapter goes over each one of these aspects.

7.1 Implementation Approach

As shown in the second column of Table 7.1, the surveyed work can be categorized according to the implementation approach into three main categories, *on-top*, *from-scratch*, and *built-in*.

1. In the *on-top* approach, an existing system is used as a black box while the logic of spatial data is provided as user-defined functions (UDFs). The simplicity of this approach attracted researchers to use it in implementing many operations including R-tree construction [43], range query [64], spatial join [65], kNN join [47, 46], and Voronoi diagram construction [45]. However, this implementation approach provides a limited performance as the underlying system is still unaware of the properties of spatial data.

2. The *from-scratch* approach is the other extreme where a new system is built from scratch to support big spatial data processing. This gives the full flexibility to provide the best performance but its complexity limited its application to trajectory querying [104] and behavioral simulation [108], in addition to a few big systems such as SciDB [98] and RasDaMan [102] array databases, and AsterixDB [114] big data management system.

3. The *built-in* approach balances efficiency with simplicity as it injects the spatial data awareness inside an existing system. This makes it efficient as the internal system becomes aware of spatial data and still it is not as complicated as building a new system from scratch. Besides, it is more practical for users who wish to mix spatial and non-spatial workload as it maintains the efficiency of the system with non-spatial data. This approach has been used in several systems including Parallel Secondo [13], Sphinx [34], $\mathcal{MD}$-HBase [14], GeoMesa [103], PRADASE [49], Hadoop GIS [16], and GeoSpark [113].

SpatialHadoop follows the built-in approach which makes it both efficient and simple. The main difference between SpatialHadoop and other built-in systems, is that it modifies all the layers of Hadoop starting from the storage layer and up-to the language and visualization layers.

7.2 Architecture

The systems discussed in this survey typically follow one of the standard approaches used in other big data systems, such as parallel DBMS [13, 34], key-value stores [14, 103], array databases [98, 102], message passing interface (MPI) [104], MapReduce [43, 64, 105, 106, 45, 77, 46, 47, 107, 108, 49, 16], resilient distributed datasets (RDD) [113], and BDMS [114], as described in the third column of Table 7.1. Some of these systems modify the underlying system to better support spatial data but they still keep the main architecture. SpatialHadoop follows the MapReduce architecture. However, it is the only system that modifies the MapReduce query processing engine, as described in Section 4.1, to assist other MapReduce developers and researchers in adding their own operations.

7.3 Language

The fourth column of Table 7.1 shows examples of high level languages supported in big spatial data systems. A high level language is extremely important as it allows non-technical users to easily interact with the system. There are some industry standards for spatial data types and operations, OGC [37] is a prime example, which are supported by existing systems for spatial data including PostGIS [115], Oracle Spatial [116], and ESRI ArcGIS [117]. Only a few systems provide a high-level programming language for spatial data, and only three of them follow the OGC standard, namely, GeoMesa [103], ESRI Tools for Hadoop [118, 110] and SpatialHadoop.

7.4 Spatial Indexes

Spatial indexes define an efficient way of storing data such that some queries run more efficiently. The fifth column of Table 7.1 shows the different types of indexes supported in the related work. While there are many in-memory and on-disk index structures used in traditional systems, they cannot be used as-is on distributed systems due to the different storage and processing models used in such systems. Most distributed systems, including SpatialHadoop, follow the two-layer index design of one *global index* and multiple *local indexes*. While most of these system focus on providing specific spatial indexes, ScalaGiST [109] provides a generic GiST-like interface for spatial indexing. However, the generic index is only used for local indexing while the global index is always based on K-d tree. This makes SpatialHadoop unique in that it is the only system that provides generic indexes in both the global and local levels. This is considered a major difference as our experiments showed that the global index is the one that can provide orders of magnitude speedup while the gain of local indexes is only marginal.

7.5 Queries

The main functionality of big spatial data systems is the query processing which performs spatial operations on the data. As shown in the sixth column of Table 7.1, we categorize these queries into five categories as follows. (1) Basic queries such as point query, range query, and nearest neighbor queries [98, 13, 34, 14, 103, 104, 64, 45, 49, 16, 109, 110, 113, 114]. (2) Spatial join queries such as self-join [16], binary join [113, 111, 108, 65, 34, 13], multi-way join [107], and kNN join [46, 47]. (3) Computational geometry queries such as Voronoi diagram construction [45]. (4) Spatial data mining such as k-means [105] and DBSCAN [106] clustering algorithms. (5) Raster operations which deal with raster data represented as two-dimensional arrays of values [43, 98, 102].

Rather than providing a rigid set of operations, SpatialHadoop is the only system that provides an extensible core where researchers can add their own operations to SpatialHadoop. This flexibility makes SpatialHadoop, by far, the most rich system in terms of spatial operations.

7.6 Visualization

A highly desirable feature of data management in general, and for spatial data in particular, is visualization. Some systems support efficient single-level image visualization algorithms which produce a single image with a fixed resolution [98, 102, 77]. GeoMesa [103] follows a different approach where it provides a standard API that allows it to connect to GeoServer [119], a standalone visualization system. This approach provides the flexibility of supporting a wide range of visualization types but the performance is limited to the capability of the single machine that runs GeoServer.

SpatialHadoop has two main features that distinguish it from related work. First, it provides an extensible interface which allows users to add new visualization types, and still generate the image efficiently using the efficient core of HadoopViz. Second, it is the only system that generates both single-level and multilevel images making it more suitable for exploring big spatial datasets where users need to zoom in and out to see all the details.

	Approach	Architecture	Language	Indexes	Queries	Visualization
Parallel Secondo [13]	Built-in	Parallel DB	SQL-Like	Local only	RQ, SJ	-
Sphinx [34]	Built-in	Parallel DB	SQL	R-tree, Quad tree	RQ, SJ	-
SciDB [98, 76]	From-scratch	Array DB	AQL, AFL	K-d tree	RQ, KNN	Single level
RasDaMan [99, 100, 101, 102]	From-scratch	Array DB	RasQL	-	Raster	Single level
$\mathcal{MD}$-HBase [14]	Built-in	KV store	-	Quad Tree, K-d tree	RQ, KNN	-
GeoMesa [103]	Built-in	KV store	CQL*	Geohash	RQ	Through GeoServer
EMINC [104]	From-scratch	MPI	-	K-d tree, R-tree	RQ	-
R-tree construction [43]	On-top	MapReduce	-	R-tree	Image quality	-
SJMR [64, 65, 48, 44]	On-top	MapReduce	-	R-tree	RQ, KNN, SJ, ANN	-
K-Means [105]	On-top	MapReduce	-	-	K-means	-
MR-DBSCAN [106]	On-top	MapReduce	-	-	DBSCAN	-
Voronoi Diagram [45]	On-top	MapReduce	-	-	VD, NN Queries	-
3D Visualization [77]	On-top	MapReduce	-	-	-	Single level
KNN Join [46, 47]	On-top	MapReduce	-	-	KNN Join	-
Multiway SJ [107]	On-top	MapReduce	-	-	Multiway SJ	-
BRACE [108]	From-scratch	MapReduce	BRASIL	Grid	SJ	-
PRADASE [49]	Built-in	MapReduce	-	Quad-tree	RQ	
Hadoop GIS [16]	Built-in	MapReduce	QL^{SP}	Grid	RQ, KNN, SJ	-
ScalaGiST [109]	Built-in	MapReduce	-	GiST	RQ, KNN	-
ESRI API for Hadoop [110]	Built-in	MapReduce	HiveQL*	PMR Quad Tree	RQ, KNN	-
ISP-MC[111]	On-top	RDD	Scala-based	On-the-fly	SJ	-
GeoTrellis [112]	On-top	RDD	Scala-based	-	Map Algebra	-
GeoSpark [113]	Built-in	RDD	Scala-based	R-tree, Quad-tree	RQ, KNN, SJ	-
Asterix-DB [114]	From-scratch	BDMS	AQL	R-tree local index	RQ	-
SpatialHadoop	Built-in	MapReduce	Pigeon*	R tree,Quad tree,others	RQ, KNN, SJ, CG	Single-, multi-level

$*$ OGC-compliant

Table 7.1: Existing work in the area of big spatial data

Chapter 8

Conclusion

This thesis describes SpatialHadoop, a full-fledged MapReduce framework with native support for big spatial data available as free open-source. SpatialHadoop is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the *language*, *indexing*, *MapReduce*, *operations*, and *visualization* layers.

Chapter 2

Chapter 3 describes the *indexing* layer which provides a two-level spatial index of one global index that partitions records across nodes, and multiple local indexes that organize records in each partition. SpatialHadoop provides a generic indexing algorithm and uses it to build a wide range of spatial indexes based on uniform grid, R-tree, R+-tree, Quad tree, K-d tree, Z-curve, and Hilbert curve.

Chapter 4 discusses the query processing engine of SpatialHadoop. First, it showed the modifications to the MapReduce layer where two new components are added as access methods to the global and local indexes. Then, it showed how these new components are used to implement three fundamental spatial operations, namely, range query, k nearest neighbors, and spatial join.

Chapter 5 describes HadoopViz, an extensible visualization framework for big spatial data. HadoopViz has three main features which make it unique. First, it exposes a visualization abstraction which allows users to define new visualization types without having to delve in to the details of the image generation. Second, it provides a simple query optimizer that allows it to generate both small and large images efficiently by automatically selecting the right algorithm. Third, it is the only system that efficiently generates mulit-level images which are more suitable for exploration of large data.

Chapter 6 shows how SpatialHadoop is used to build SHAHED, an end-user application for interactive exploration and visualization of satellite data. SHAHED uses the spatial indexes, query processing, and visualization and it provides users with a simple web interface for exploring a peta byte archive of satellite data.

Chapter 7 discusses the related work in the area of big spatial data. It goes through six aspects of related work, namely, implementation approach, architecture, language, indexing, operations, and visualization. It reviews the state-of-the-art in the literature and shows where SpatialHadoop stands along each of these six dimensions.

References

[1] Telescope Hubbel Site: Hubble Essentials: Quick Facts, 2015. `http://hubblesite.org/the_telescope/hubble_essentials/quick_facts.php`.

[2] European XFEL: The Data Challenge, September 2012. `http://www.eiroforum.org/activities/scientific_highlights/201209_XFEL/index.html`.

[3] Land Process Distributed Active Archive Center (LP DAAC), 2016. `https://lpdaac.usgs.gov/about`.

[4] GnipBlog, 2014. `https://blog.gnip.com/tag/geotagged-tweets/`.

[5] Twitter. The About webpage., 2015. `https://about.twitter.com/company`.

[6] Henry Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[7] L. Pickle, M. Szczur, D. Lewis, , and D. Stinchcomb. The Crossroads of GIS and Health Information: A Workshop on Developing a Research Agenda to Improve Cancer Control. *International Journal of Health Geographics*, 5(1):51, 2006.

[8] A. Auchincloss, S. Gebreab, C. Mair, and A. Diez Roux. A Review of Spatial Methods in Epidemiology: 2000-2010. *Annual Review of Public Health*, 33:107–22, April 2012.

[9] Y. Thomas, D. Richardson, and I. Cheung. Geography and Drug Addiction. *Springer Verlag*, 2009.

[10] James Faghmous and Vipin Kumar. *Spatio-Temporal Data Mining for Climate Data: Advances, Challenges, and Opportunities.* Advances in Data Mining, Springer, 2013.

[11] Jagan Sankaranarayanan, Hanan Samet, Benjamin E. Teitler, and Michael D. Lieand Jon Sperling. TwitterStand: News in Tweets. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM SIGSPATIAL*, pages 42–51, 2009.

[12] `http://esri.github.io/gis-tools-for-hadoop/`.

[13] Jiamin Lu and Ralf Hartmut Guting. Parallel Secondo: Boosting Database Engines with Hadoop. In *International Conference on Parallel and Distributed Systems*, pages 738 –743, Nanyang Executive Center, Singapore, December 2012.

[14] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. $\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. *DAPD*, 31(2):289–319, 2013.

[15] HBase, 2015. `http://hbase.apache.org/`.

[16] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *Proceedings of the VLDB Endowment, PVLDB*, 6(11):1009–1020, 2013.

[17] Ashish Thusoo, Joydeep Sarma Sen, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proceedings of the VLDB Endowment, PVLDB*, pages 1626–1629, 2009.

[18] `http://spatialhadoop.cs.umn.edu/`.

[19] Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel, and Ravi Janardan. CG_Hadoop: Computational Geometry in MapReduce. In *SIGSPATIAL*, pages 284–293, Orlando, FL, November 2013.

[20] Shekhar Shekhar and S. Chawla. *Spatial Databases: A Tour.* Prentice Hall Upper Saddle River, NJ, 2003.

[21] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 1099–1110, 2008.

[22] Ahmed Eldawy and Mohamed F. Mokbel. Pigeon: A Spatial MapReduce Language. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1242–1245, 2014.

[23] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. A Demonstration of HadoopViz: An Extensible MapReduce System for Visualizing Big Spatial Data. *Proceedings of the VLDB Endowment, PVLDB*, 8(12):1896–1907, 2015.

[24] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data. To Appear. In *Proceedings of the International Conference on Data Engineering, ICDE*, May 2015.

[25] Ahmed Eldawy, Mohamed F. Mokbel, Saif Alharthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data. In *ICDE*, pages 1585–1596, Seoul, Korea, April 2015.

[26] Ahmed Eldawy, Saif Al-Harthi, Abdulhadi Alzaidy, Anas Daghistani, Sohaib Ghani, Saleh Basalamah, and Mohamed F. Mokbel. A Demonstration of Shahed: A MapReduce-based System for Querying and Visualizing Satellite Data. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1444–1447, Seoul, Korea, April 2015.

[27] SHAHED web interface, 2015. `http://shahed.cs.umn.edu/`.

[28] Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. TAREEG: a MapReduce-based Web Service for Extracting Spatial Data from

OpenStreetMap. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 897–900, June 2014.

[29] Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. TAREEG: A MapReduce-Based System for Extracting Spatial Data from OpenStreetMap. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM SIGSPATIAL*, Dallas, TX, November 2014.

[30] Mohamed F. Mokbel, Louai Alarabi, Jie Bao, Ahmed Eldawy, Amr Magdy, Mohamed Sarwat, Ethan Waytas, and Steven Yackel. MNTG: An Extensible Web-Based Traffic Generator. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, pages 38–55, August 2013.

[31] Mohamed F. Mokbel, Louai Alarabi, Jie Bao, Ahmed Eldawy, Amr Magdy, Mohamed Sarwat, Ethan Waytas, and Steven Yackel. A Demonstration of MNTG - A Web-based Road Network Traffic Generator. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1246–1249, March 2014.

[32] Khaled Mohammed Al-Naami, Sadi Evren Seker, and Latifur Khan. GISQF: An Efficient Spatial Query Processing System. In *International Conference on Cloud Computing Technology and Science*, pages 681–688, Anchorage, AK, 2014.

[33] Khaled Mohammed Al-Naami, Sadi Evren Seker, and Latifur Khan. GISQAF: MapReduce Guided Spatial Query Processing and Analytics System. *Software: Practice and Experience*, 2015.

[34] Ahmed Eldawy, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. Sphinx: Distributed Execution of Interactive SQL Queries on Big Spatial Data (Poster). In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM SIGSPATIAL*, November 2015.

[35] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source

SQL Engine for Hadoop. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2015.

[36] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 25–36, Washington, DC, June 2011.

[37] Open Geospatial Consortium, 2015. `http://www.opengeospatial.org/`.

[38] OpenStreetMap, 2015. `http://www.openstreetmap.org/`.

[39] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 195–204, San Diego, CA, June 1992.

[40] Scott Leutenegger and David Nicol. Efficient Bulk-Loading of Gridfiles. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 9(3):410–420, 1997.

[41] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 500–509, Santiago, Chile, September 1994.

[42] Scott Leutenegger, Mario Lopez, and Jeffrey Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 497–506, Birmingham U.K, April 1997.

[43] Ariel Cary, Zhengguo Sun, Vagelis Hristidis, and Naphtali Rishe. Experiences on Processing Spatial Data with MapReduce. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, pages 302–319, New Orleans, Louisiana, June 2009.

[44] Haojun Liao, Jizhong Han, and Jinyun Fang. Multi-dimensional Index on Hadoop Distributed File System. *International Conference on Networking, Architecture, and Storage*, 0:240–249, 2010.

[45] Afsin Akdogan, Ugur Demiryurek, Farmoush Banaei-Kashani, and Cyrus Shahabi. Voronoi-based Geospatial Query Processing with MapReduce. In *International Conference on Cloud Computing Technology and Science*, pages 9–16, Indianapolis, IN, November 2010.

[46] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. *Proceedings of the VLDB Endowment, PVLDB*, pages 1016–1027, 2012.

[47] Chi Zhang, Feifi Li, and Jeffrey Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 38–49, Berlin, Germany, March 2012.

[48] Kai Wang, Jizhong Han, Bibo Tu, Jiao Dai amd Wei Zhou, and Xuan Song. Accelerating Spatial Data Processing with MapReduce. In *International Conference on Parallel and Distributed Systems*, pages 229–236, Shanghai, China, December 2010.

[49] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query Processing of Massive Trajectory Data Based on MapReduce. In *International Workshop on Cloud Data Management, CloudDB*, pages 9–16, HongKong, China, October 2009.

[50] Jürg Nievergelt, Hans Hinterberger, and Kenneth Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems, TODS*, 9(1):38–71, 1984.

[51] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 47–57, Boston, MA, June 1984.

[52] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, pages 507–518, Brighton, England, September 1987.

[53] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACMCS*, 16(2):187–260, 1984.

[54] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[55] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[56] Gísli R. Hjaltason and Hanan Samet. Improved Bulk-Loading Algorithms for Quadtrees. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS*, pages 110–115, November 1999.

[57] Gísli R. Hjaltason, Hanan Samet, and Yoram J. Sussmann. Speeding up bulk-loading of quadtrees. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS*, pages 50–53, November 1997.

[58] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH*, pages 124–133, July 1980.

[59] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Analysis of multi-dimensional space-filling curves. *GeoInformatica*, 7(3):179–209, 2003.

[60] `http://www.census.gov/geo/www/tiger/`.

[61] `http://aws.amazon.com/blogs/aws/process-earth-science-data-on-aws-with-nasa-nex/`.

[62] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 322–331, 1990.

[63] Jens-Peter Dittrich and Bernhard Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

[64] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial Queries Evaluation with MapReduce. In *Proceedings of the International Conference on Grid and Cooperative Computing*, pages 287–292, Washington, DC, August 2009.

[65] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, New Orleans, LA, August 2009.

[66] Jignesh Patel and David DeWitt. Partition Based Spatial-Merge Join. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 259–270, Montreal, Quebec, Canada, June 1996.

[67] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel Abadi, David DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 165–178, Providence, RI, 2009.

[68] Erik G. Hoel and Hanan Samet. Performance of Data-Parallel Spatial Operations. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 156–167, 1994.

[69] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. Touch: In-memory Spatial Join by Hierarchical Data-oriented Partitioning. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 701–712, 2013.

[70] Ariane Middel. A Framework for Visualizing Multivariate Geodata. In *Visualization of Large and Unstructured Data Sets*, pages 13–22, 2007.

[71] Isabel F. Cruz, Venkat R. Ganesh, Claudio Caletti, and Pavan Reddy. GIVA: a semantic framework for geospatial and temporal data integration, visualization, and analytics. In *SIGSPATIAL*, pages 534–537, 2013.

[72] Justin Song, Richard Frank, Patricia L. Brantingham, and Jim LeBeau. Visualizing the spatial movement patterns of offenders. In *SIGSPATIAL*, pages 554–557, 2012.

[73] Ross Maciejewski, Stephen Rudolph, Ryan Hafen, Ahmad Abusalah, Mohamed Yakout, Mourad Ouzzani, William S Cleveland, Shaun J Grannis, and David S

Ebert. A Visual Analytics Approach to Understanding Spatiotemporal Hotspots. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):205–220, March 2010.

[74] MapD Twitter Demo, 2015. `http://mapd.csail.mit.edu/tweetmap-desktop/`.

[75] Todd Mostak. An Overview of MapD (Massively Parallel Database). Technical report, Harvard University, Cambridge, MA, 2015. `http://geops.cga.harvard.edu/docs/mapd_overview.pdf`.

[76] Gary Planthaber, Michael Stonebraker, and James Frew. EarthDB: Scalable Analysis of MODIS Data using SciDB. In *Proceedings of the ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 11–19, Redondo Beach, CA, November 2012.

[77] Huy T. Vo, Jonathan Bronson, Brian Summa, João Luiz Dihl Comba, Juliana Freire, Bill Howe, Valerio Pascucci, and Cláudio T. Silva. Parallel Visualization on Large Clusters using MapReduce. In *IEEE Symposium on Large Data Analysis and Visualization, LDAV*, pages 81–88, Providence, Rhode Island, October 2011.

[78] HadoopViz Source Code, 2015. `https://github.com/aseldawy/spatialhadoop2/tree/master/sr`

[79] ImageMagick, 2015. `http://www.imagemagick.org/`.

[80] Ahmed Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1352–1363, Seoul, South Korea, April 2015.

[81] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial Partitioning Techniques in SpatialHadoop. In *Proceedings of the VLDB Endowment, PVLDB*, pages 1602–1605, Kohala Coast, HI, sep 2015.

[82] Walid G. Aref and Hanan Samet. Efficient Processing of Window Queries in The Pyramid Data Structure. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, pages 265–272, Nashville, TN, April 1990.

[83] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Rolling the Dice: Multidimensional Visual Exploration using Scatterplot Matrix Navigation. *IEEE*

Transactions on Visualization and Computer Graphics, 14(6):1539–1148, November 2008.

[84] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, Kathleen Bonnell, Mark Miller, Gunther H. Weber, Cyrus Harrison, David Pugmire, Thomas Fogal, Christoph Garth, Allen Sanderson, E. Wes Bethel, Marc Durant, David Camp, Jean M. Favre, Oliver Rübel, Paul Navrátil, Matthew Wheeler, Paul Selby, and Fabien Vivodtzev. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *The International Conference on Scientific Discovery through Advanced Computing*, pages 1539–1148, Denver, CO, July 2011.

[85] Jack E Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[86] The National Aeronautics and Space Administration, 2016. `http://www.nasa.gov/`.

[87] European Space Agency, 2016. `http://www.esa.int/`.

[88] Xun Zhou, Shashi Shekhar, and Dev Oliver. Discovering Persistent Change Windows in Spatiotemporal Datasets: A Summary of Results. In *Proceedings of the ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 37–46, 2013.

[89] Shyam Boriah, Ankush Khandelwal, Vipin Kumar, Varun Mithal, and Karsten Steinhaeuser. Change Detection from Temporal Sequences of Class Labels: Application to Land Cover Change Mapping. In *SDM*, pages 650–658, 2013.

[90] James H. Faghmous, Matthew Le, Muhammed Uluyol, Vipin Kumar, and Snigdhansu Chatterjee. A Parameter-Free Spatio-Temporal Pattern Mining Model to Catalog Global Ocean Dynamics. In *Proceedings of the IEEE International Conference on Data Mining, ICDM*, pages 151–160, 2013.

[91] C. Monteleoni *et al.* Climate Informatics. In T. Yu, S. Simoff, and N. Chawla, editors, *Computational Intelligent Data Analysis for Sustainable Development*, chapter 4, pages 81–126. CRC Press, April 2013.

[92] NASA. ”Global Climate Change: Vital Signs of the Planet”, 2016. `http://climate.nasa.gov/`.

[93] National Science Foundation (NSF) Expeditions in Computing program. ”Understanding Climate Change: A Data Driven Appraoach”, 2016. `http://climatechange.cs.umn.edu/`.

[94] United States Environmental Protection Agency (EPA). ”Climate Change Research”, 2016. `http://www.epa.gov/research/climatescience/`.

[95] Reverb - The Next Generation Earth Science Discovery Tool, 2016. `http://reverb.echo.nasa.gov/reverb/`.

[96] Ahmed Eldawy and Mohamed F. Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *Proceedings of the VLDB Endowment, PVLDB*, 6(12):1230–1233, 2013.

[97] Marshall Bern, David Eppstein, and Shang-Hua Teng. Parallel Construction of Quadtrees and Quality Triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[98] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[99] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system rasdaman. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 575–577, June 1998.

[100] Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. Geo/Environmental and Medical Data Management in the RasDaMan System. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 548–552, August 1997.

[101] Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. The RasDaMan Approach to Multidimensional Database Management. In *Proceedings*

of the ACM Symposium on Applied Computing (SAC), pages 166–173, March 1997.

[102] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. Spatio-Temporal Retrieval with RasDaMan. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 746–749, September 1999.

[103] Anthony Fox, Chris Eichelberger, James Hughes, and Skylar Lyon. Spatio-temporal Indexing in Non-relational Distributed Databases. In *International Conference on Big Data*, pages 291–299, Santa Clara, CA, 2013.

[104] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An Efficient Multi-Dimensional Index for Cloud Data Management. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, pages 17–24, Hong Kong, China, 2009.

[105] Weizhong Zhao, Huifang Ma, and Qing He. Parallel K-Means Clustering Based on MapReduce. In *CloudCom 2009*, pages 674–679, 2009.

[106] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: A Scalable MapReduce-based DBSCAN Algorithm for Heavily Skewed Data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[107] Himanshu Gupta, Bhupesh Chawda, Sumit Negi, Tanveer A. Faruquie, L. V. Subramaniam, and Mukesh Mohania. Processing multi-way spatial joins on map-reduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT, pages 113–124, New York, NY, USA, 2013.

[108] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral Simulations in MapReduce. *Proceedings of the VLDB Endowment, PVLDB*, 3(1):952–963, 2010.

[109] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. *PVLDB*, 7(14):1797–1808, 2014.

[110] Randall T. Whitman, Michael B. Park, Sarah A. Ambrose, and Erik G. Hoel. Spatial Indexing and Analytics on Hadoop. In *SIGSPATIAL*, 2014.

[111] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in Cloud. In *International Workshop on Cloud Data Management, CloudDM, in Conjunction with the IEEE International Conference on Data Engineering, ICDE*, pages 34–41, April 2015.

[112] Ameet Kini and Rob Emanuele. Geotrellis: Adding Geospatial Capabilities to Spark, 2014. `http://spark-summit.org/2014/talk/geotrellis-adding-geospatial-capabilities-to-spa`

[113] Jia Yu, Mohamed Sarwat, and Jinxuan Wu. GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM SIGSPATIAL*, Seattle, WA, nov 2015.

[114] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment, PVLDB*, 7(14):1905–1916, 2014.

[115] PostGIS, 2015. `http://postgis.net/`.

[116] Ravi Kothuri and Siva Ravada. Oracle spatial, geometries. In *Encyclopedia of GIS.*, pages 821–826. Springer, 2008.

[117] ESRI ArcGIS, 2015. `http://www.esri.com/software/arcgis/`.

[118] ESRI Tools for Hadoop, 2015. `http://esri.github.io/gis-tools-for-hadoop/`.

[119] GeoServer, 2015. `http://geoserver.org/`.