High Performance Spatial Query Processing for Large Scale Spatial Data Warehousing

By
Abulimiti Aji

Doctor of Philosophy
Computer Science and Informatics

Fusheng Wang, Ph.D.
Advisor

Joel H. Saltz, M.D., Ph.D.
Committee Member

James J. Lu, Ph.D.
Committee Member

Xiaodong Zhang, Ph.D.
Committee Member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

High Performance Spatial Query Processing for Large Scale Spatial Data Warehousing

By

Abulimiti Aji
M.S., Emory University, 2010

Advisor : Fusheng Wang, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science and Informatics
2014

Abstract

High Performance Spatial Query Processing for Large Scale Spatial Data Warehousing
By Abulimiti Aji

Support of high performance queries on large volumes of spatial data have become important in many domains, including geo-spatial problems in numerous fields and emerging scientific applications that are increasingly data- and compute-intensive. There are two major challenges for managing and querying massive spatial data: the explosion of spatial data, and the high computational complexity of spatial queries due to the multi-dimensional nature of spatial analytics.

MapReduce provides a highly effective, efficient and reliable tool for large scale data analysis. While MapReduce model is amenable to a wide range of real-world tasks, spatial queries and analytics are intrinsically complex to fit into the MapReduce model. Meanwhile, hybrid systems combining CPUs and GPUs are becoming widely available, but the computing capacity of such systems is often underutilized. Providing new spatial querying methods on such architecture requires us to answer several fundamental research questions that have practical implications.

The goal of this dissertation is to create a framework with new systematic methods to support high performance spatial queries for massive spatial data on MapReduce and CPU-GPU hybrid platforms, driven by real-world use cases. We have researched multi-level parallelization approaches for spatial queries to run on hybrid cluster environment. Specifically, we have conducted following studies: 1) create new spatial data processing methods and pipelines with spatial partition level parallelism through MapReduce programming model, and multi-level indexing methods to accelerate spatial data processing; 2) develop two critical components to enable query parallelization: effective and scalable spatial partitioning in MapReduce (pre-processing), and query normalization methods for partition effect; 3) integrate GPU-based spatial operations into MapReduce query pipelines; 4) investigate optimization methods for data skew mitigation, and CPU/GPU resource coordination in MapReduce, and 5) support declarative spatial queries for workload composition, and create a query translator to automatically translate the queries into MapReduce programs.

Consequently, we have developed Hadoop-GIS — a MapReduce based high performance spatial querying system for spatial data warehousing. The system supports multiple types of spatial queries on MapReduce through spatial partitioning, implicit parallel spatial query execution on MapReduce, and effective methods for amending query results through handling boundary objects. Hadoop-GIS utilizes global partition indexing and customizable on demand local spatial indexing to achieve efficient query processing. We have integrated Hadoop-GIS into Hive to support declarative spatial queries, and released the system and developed approaches as an open source software package.

High Performance Spatial Query Processing for Large Scale Spatial Data Warehousing

By

Abulimiti Aji
M.S., Emory University, 2010

Advisor : Fusheng Wang, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science and Informatics
2014

To my family

Acknowledgments

My deepest gratitude is to my adviser Dr. Fusheng Wang for his advice and support. I am very fortunate to have an adviser who gave me the freedom to explore on my own, yet patiently help me navigate through uncertainty when my steps faltered. This thesis would not have been possible without him and encouragement he has given me over the course of my Ph.D. study.

I would like to thank the members of my dissertation committee. Thanks Dr. James Lu for encouraging me to pursue database research, and providing very detailed comments on my dissertation. Thanks Dr. Joel Saltz for the valuable suggestions which helped me identify important research problems. He showed me how interdisciplinary research can have lasting impact, and how computer science can help advance other fields. Thanks Dr. Xiaodong Zhang for his insightful comments on GPU related topic. His research style, that putting real applicability and reusable systems first, has strongly influenced me.

Over the course of my study, a number of people have contributed to my career and personal growth in many ways. I am grateful to Dr. Eugene Agichtein who convinced me to come to Emory and took me as his student before Fusheng has arrived. I have spent amazing summers at Max-Planck Institute, Microsoft Research and Yahoo!. Thanks Dr. Martin Theobald for mentoring me while I was at MPII, and giving me career advice (Vielen Danke für deine Hilfe). Thanks Dr. Emre Kıcıman for mentoring me during my internship at MSR; thanks Dr. Hakan Ceylan for introducing me to the great engineers at Yahoo!, and being a good friend.

I would like to thank my friends and colleagues at Emory University, for making the Ph.D. study an unforgettable experience. Thanks to the IR folks Liu Yandong, Guo Qi and Wang Yu with whom I have shared many insightful discussions and fun times. Thanks to the faculty and students at BMI/CCI: Dr. Lee Cooper, Dr. Jun Kong, Dr. George Teodoro, Tony Pan, Vo Hoang, Ameen Kazerouni, Michael Nalisnik, Liang Yanhui, Zhang Wei, Chen Xin, Sanjay Agrawat and Stanly Xu.

I am eternally indebted to my parents and brothers for their endless love, support and encouragement throughout my life. Atam Haji Qadir manga chidam hem gheyretni, anam Bahargül Yüsüf turmushni qandaq söyüshni ügetti. Ularning terbiyesi hem muhebbitisiz men bu künlerge kilelmes idim. Akam Tursunjan, ukam Yasinjan, Abletjan we Ablexetler mining qanatlirim bolup keldi. Eger ular ata-anamgha hemra bolmighan bolsa, men bu xizmetnimu xatirjem qilalmas idim. Mihriban qirindashlirimdin cheksiz razimen. Ayalim Zahide mining dokturluq sepirimde yaxshi hemra bolup keldi. Kichilep ishligen chaghlarda yinimda bolghanliqidin tolimu xoshalmen. Dostlurum Batur, Nurmemetjan, Qadil hem bashqa sawaqdashliriming ziyaretlirimde hemra bolghanliqigha hem körsetken iltipatigha rehmet iytimen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The advancement in the computer technology and the rapid growth of Internet have brought many changes to society. In particular, the Big Data paradigm has disrupted (almost) all aspects of our lives. The rise of big data can be attributed to two main factors.

First, high volumes of data generated by the *machines*. The rapid improvement of high resolution data acquisition technologies and sensor networks have enabled us to capture large amounts of data at an unprecedented scale and rate. For example, the GeoEye-1 satellite has the highest resolution of any commercial imaging system and is able to collect images with a ground resolution of 0.41 meters in the panchromatic or black and white mode [1]; the Sloan Digital Sky Survey (SDSS), with a rate of about 200 GB per night, has amassed more than 140 terabytes of information [18]; the modern medial imaging scanners can capture the micro anatomic tissue details at the billion pixel resolution [51].

Second, traces of *human* activity and massive collaboration facilitated by the Internet. The proliferation of cost effective and ubiquitous positioning technologies, mobile devices, sensors have enabled us to collect massive amounts of spatial information of human activity and wildlife activity. For example, FourthSquare – a popular local search and discovery service – allow users to "check-in" at more than 60 million venues, and so far has more than 6 billion Check-ins [2]. Realising a huge business potential, more and more businesses and Internet services are making their service *location-aware*. At the same time, the Internet has made remote collaboration so easy that, now, crowd can even generate a free mapping of the world autonomously. OpenStreetMap [3] is a large collaborative mapping project that generated by users around the globe, and it has reached 1 million registered users as of this writing.

The volume and velocity of the data only increase significantly as we shift towards the Internet of Things paradigm in which devices have spatial awareness, and generate voluminous amounts of data.

In many applications and scientific studies, there is a growing need to manage spatial entities and their topological, geometric, or geographic properties. Analyzing such large amounts of spatial data to derive values and guide decision making have become essential to business success and scientific progress. For example, Location Based Social Networks (LBSNs) are utilizing large amounts of user location information to provide geo-marketing and recommendation services. Social scientists are relying on such data to study dynamics of social systems and understand human behavior. Epidemiologists are combining such spatial data with public health data to study the patterns of disease outbreak and spread. In all those domains, the availability of big spatial data analytics is a key enabler.

In the past, Geographic Information System (GIS) has been the primary software tool that designed to capture, store, manipulate, analyze, manage, and present various spatial or geographical data. However, as we are increasingly dealing with large amounts of data, the conventional GIS and spatial data management systems, that designed in an era of desktop servers and workstations, have been limited by their scalability and performance. Modern data intensive spatial applications require a different approach to face the big data challenges.

1.2 Data Intensive Spatial Applications

The rapid growth of spatial data is driven by not only industrial applications, but also emerging scientific applications that have become data-intensive and compute-intensive.

1.2.1 Analysis of Derived Scientific Spatial Data

With the rapid improvement of data acquisition technologies such as high-resolution tissue slide scanners and remote sensing instruments, it has become more efficient to capture extremely large spatial data to support scientific research. For example, digital pathology imaging has become an emerging field in the past decade, where examination of high resolution images of tissue specimens enables novel, more effective ways of screening for disease, classifying disease states, understanding its progression and evaluating the efficacy of therapeutic strategies. In clinical environment, medical professionals have been relying on the manual judgement from pathologists – a process inherently subject to human bias – to diagnose, and understand the disease.

Today, *in silico* pathology image analysis offers a means of rapidly carrying out quantitative, reproducible measurements of micro-anatomical features in high-resolution pathology images and large image datasets. Medical professionals and researchers can use computer algorithms to calculate the distribution of certain cell types, and perform associative analysis with other data such as patient genetic composition and clinical treatment.

Figure 1.1 shows a protocol for in silico pathology image analysis pipeline. From left to the right, the subfigures represent: glass slides, high resolution image scanning,

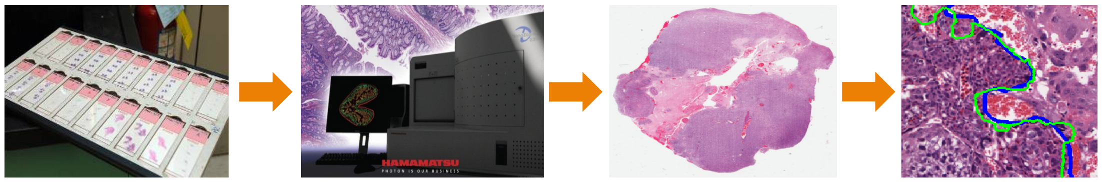


Figure 1.1: Derived spatial data in pathology image analysis

whole slide images, and automated image analysis. The first 3 steps are data acquisition process that mostly done in a pathology laboratory environment, and the final step is where the computerized analysis is performed. In the image analysis step, regions of micro-anatomic objects (millions per image) such as nuclei and cells are computed through image segmentation algorithms, represented with their boundaries, and image features are extracted from these objects. Exploring the results of such analysis involves complex queries such as spatial cross-matching, overlay of multiple sets of spatial objects, spatial proximity computations between objects, and queries for global spatial pattern discovery. These queries often involve billions of spatial objects and heavy geometric computations.

Besides the observational data, scientific simulation also generates large amounts spatial data. Scientists often use models to simulate natural phenomena, and analyze the simulation process and data. For example, earth science uses simulation models to help predict the ground motion during earthquakes. Ground motion is modeled with an octree based hexahedral mesh, using soil density as input. Simulation tools calculates the propagation of seismic waves through the Earth by approximating the solution to the wave equation at each mesh node. During each time step, for each node in the mesh, simulator calculates node velocity in the spatial directions, and records those information to the primary storage. The simulation result is a spatio-temporal earthquake data set describing the ground velocity response [30]. As the scale of the experiment increases, the resulting dataset also increases, and scientists have been struggling to query and manage such large amounts of spatio-temporal data in an efficient and cost-effective manner.

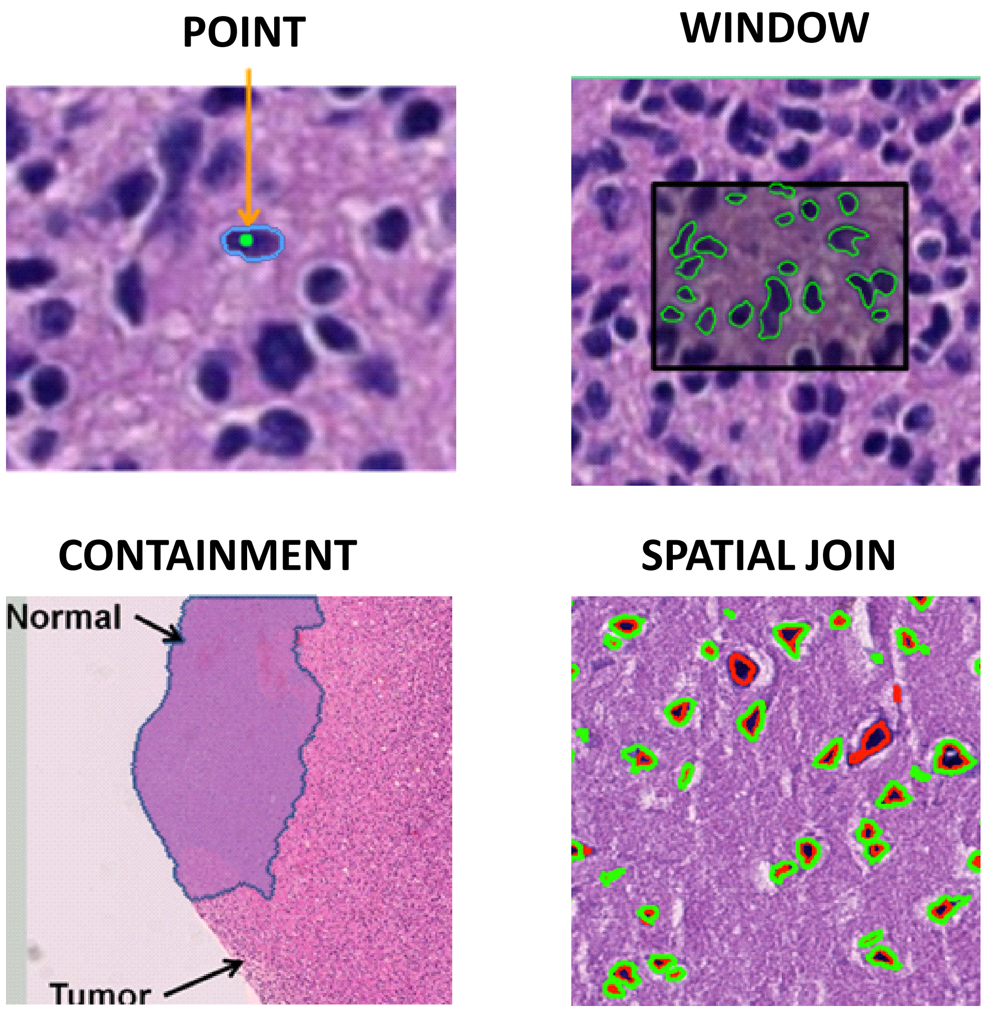
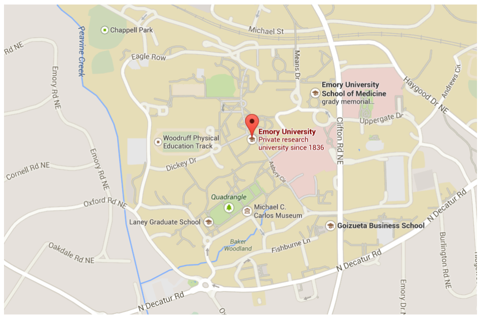
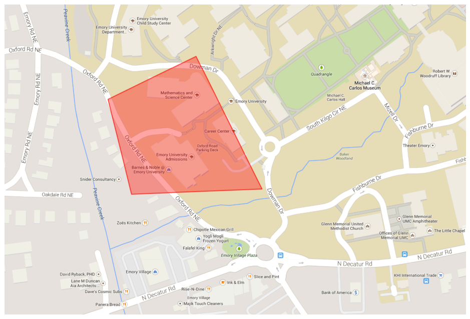


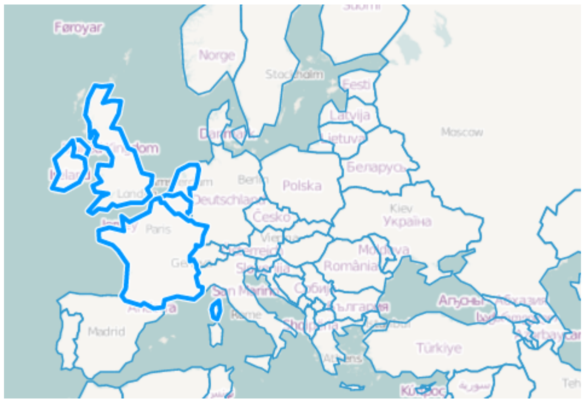
Figure 1.2: Examples spatial query cases in pathology imaging

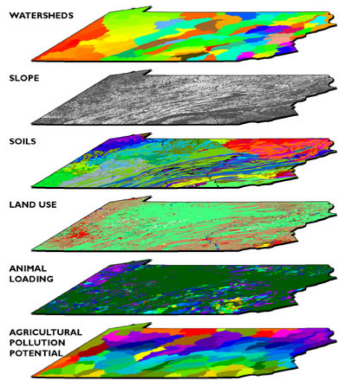
1.2.2 GIS and Social Media Applications

Volunteered geographic information (VGI) further enriched GIS world with massive amounts of user generated geographical and social data. VGI is a special case of the larger Internet phenomenon — user-generated content — in the GIS domain. Everyday Internet users can provide, modify, and share geographical data using interactive online services such as OpenStreetMap [3], Wikimapia, GoogleMap, GoogleEarth, and Microsofts Virtual Earth. The spatial information need to be constantly analyzed, and corroborated to track changes, and understand the current status. Most often, a spatial data management system is used to perform such analysis. Figure 1.3 shows an


(a) Point query


(b) Window query


(c) Containment query



(d) Join query

Figure 1.3: Example spatial query cases in GIS and geosocial applications

example of most common spatial queries in a typical GIS setting.

Recently, the explosive growth of social media applications contributed massive amounts of user-generated geographic information in the form of tweets, status updates, check-ins, Waze, and traffic reports. Furthermore, if such geo-spatial information is not available, automated geo-tagging/coding tools can infer and assign an approximate location to those contents. Analysis of such large amounts of data has implications for many applications — both commercial and academic. In [39] authors used such information to investigate the relationship between the geographic location of protestors attending demonstrations in the 2013 Vinegar protests in Brazil and the geographic location of users that tweeted the protests. Another example is location based targeted advertisement [106] and recommendation [80]. Nowadays social media applications are increasingly *location-aware*. Upon realizing a potential customer has walked into a specific super market (cellphone geolocation information), the owner can market certain products or coupons to the user. Those online services and GIS systems are backed by conventional spatial database systems that are were tailored to a different set of applications.

1.3 Spatial Queries

There are five major categories of queries : i) feature aggregation queries (non-spatial queries), for example, queries for finding mean values of attributes or distribution of attributes; ii) fundamental spatial queries, including point based queries, containment queries and window queries; iii) complex spatial queries, including spatial cross-matching or overlay (large scale spatial join) and nearest neighbor queries; iv) integrated spatial and feature queries, for example, feature aggregation queries in certain spatial regions; and v) global spatial pattern queries, for example, queries on finding high density regions, or queries to find directional patterns of objects. Figure 1.2 illustrates several examples from medical imaging domain, and Figure 1.3 shows examples from a geo-social application and a GIS application.

1.3.1 Characteristics of Modern Spatial Analytics Applications

The spatial analytics applications are different from conventional warehousing applications in several aspects. They involve:

- **Large Volumes of Multidimensional Data** Conventional warehousing applications deal with data generated from business transactions. As a result, the underlying data (such as numbers and strings) tend to be relatively *simple* and *flat*. However, this is not the case for the spatial applications which deals with massive amounts of geometry shapes and spatial objects. For example, a typical whole slide pathology contains more than 100 billion pixels, millions of objects , and 100 million derived image features. A single study may involve thousands of images analyzed with dozens of algorithms - with varying parameters - to generate many different result sets to be compared and consolidated, at the scale of tens of terabytes. A moderate-size healthcare operation can routinely generate thousands of whole slide images per day, leading to petabytes of analytical results per year. A single 3D pathology image could come from a thousand slices and take 1TB storage, containing several millions to ten millions of derived 3D surface objects.

- **High Computation Complexity** Most spatial queries involve geometric computations that are often compute-intensive. While spatial filtering through minimum bounding boxes (MBBs) can be accelerated through spatial access methods, spatial refinements such as polygon intersection verification are highly expensive operations. For example, spatial join queries such as spatial cross-matching or spatial overlay can be very expensive to process. This is mainly due to the polynomial complexity of many geometric computation methods. Such compute-intensive geometric computation, combined with the large volumes of big data requires a high performance solution.

- **Complex Spatial Queries** Spatial queries are complex to express in current spatial data analytics systems. Most scientific researchers and spatial application developers often interested in running queries that involve complex spatial relationships such as nearest neighbor query, and spatial pattern queries. Such queries are not well supported in current spatial database systems. Frequently, users are forced to write database user defined functions to be able to perform the required operations. SQL — Structured Query Language — has gained tremendous momentum in the relational database field and become the de facto standard for querying the data. While most spatial queries can be expressed in SQL, due to the structural differences in the programming model, translating complex spatial SQL queries into efficient MapReduce programs requires considerable optimization efforts.

1.4 Dissertation Goals

The fundamental goals of this dissertation is to address the research challenges for delivering a high performance software system for spatial queries and analytics of spatial big data, and provide an open source implementation for use. Surrounding this central goal, there are three sub-goals that we intended to achieve in this dissertation.

1.4.1 Exploring the Principles of Spatial Query Processing on MapReduce

MapReduce provides two simple programming API — *map* and *reduce* to run queries on a large set of commodity clusters. While the simplicity of the programming model has contributed its success and wider adoption, it is too simple and cumbersome to express certain queries. Therefore, it has been criticized as "Assembly language of parallel processing" [4].

Most currently available spatial query processing algorithms assume a single threaded serial execution model. Although a number of parallel spatial algorithms were proposed in the past, they were mostly designed for a shared memory parallel processing architecture. Recently, there are few research efforts [124] that explores the possibility of running spatial queries on MapReduce. However, they mostly ignore the architectural design issues, and only provide ad-hoc solutions to specific use cases and queries. The lack of principled query processing approaches has been a major obstacle for realizing a large scale spatial query system on MapReduce.

Consequently, in this dissertation we explore how to use MapReduce to express variety of spatial queries, how to handle boundary objects and partitions, and how to support basic spatial measurement functions and operators.

1.4.2 Exploring the System Architecture

Three are three approaches to build a MapReduce based spatial data warehousing system. The first approach is to completely rewrite a MapReduce implementation [56]to accommodate the complexities of spatial query processing. This includes customized storage model for spatial data types, specialized processing model to perform efficient spatial filtering and processing. In this approach, one have much freedom to arbitrarily optimize an existing MapReduce implementation for the best query performance. However, there are two disadvantages of this approach. The first disadvantage is incompatibility. As all major cloud computing service providers offers standard MapReduce framework for parallel processing, a code specifically written for spatially optimized MapReduce framework may not run on standard cloud platforms. The second disadvantage is lack of expressive query language for workload specification. Specifically, application developers have to understand the API for writing the spatial MapReduce code, and implement every spatial queries as a set of MapReduce jobs. Clearly, such "one code for one query" approach is not a optimal for developer productivity, and decades of Database research suggests that simplified expressive query language can greatly improve system usability and efficiency.

The second approach is to extend an existing MapReduce framework with a spatial database at the engine level [27, 96]. Specifically, each parallel processing node is equipped with an spatial database engine, and MapReduce worker. In such architecture, MapReduce engine and database engines are tightly integrated, closely coordinate with each other for query processing. MapReduce system mostly serves as task distributor and job coordinator, and database system serves as query processor and storage engine. This approach can achieve good performance and also provides structured query language for workload specification.

The third approach is to integrate the MapReduce framework with a spatial query engine, and provide a translation mechanism for SQL queries to be translated to MapReduce programs that run on such implementation. In particular, such system implements a mapping mechanism that maps a HDFS partition to spatial data partition, and a translation mechanism that translates structured queries into MapReduce query operators. Advantages of this approach are loose decoupling from the underlying MapReduce engine, backward compatibility with existing MapReduce platforms, and structured query language for workload specification.

The popularity of structured query languages such as Pig [59] and Hive [116], and increasingly effective SQL-to-MapReduce translation techniques [78] have motivated us to choose the third design approach. We explore how our design choice effects later query processing efficiencies; how easy it is to extend the system with new query operators; how such system can be integrated with hardware accelerators to co-process queries.

1.4.3 Improving the Query Processing Performance

The second goal of this dissertation is to study major performance problems in MapReduce based query processing system, and introduce novel approaches to further improve the query performance. In a system, there are many components that can be optimized, and connections between those components also present optimization opportunities. In this dissertation, we approach this problem from three different directions.

Query Optimization

Data skew is a major performance bottleneck in MapReduce based query systems [60, 71, 75, 76, 81]. Spatial data skew could cause unbalanced tasks and long query response time. Traditional approaches are static in nature that they use quantified dataset characteristics , such as selectivity and cost estimation, to avoid the occurrence of a skewed task assignment. A more adaptive approach is to recursively repartition the “straggler” tasks [75] to mitigate the skew. However, such adaptive approach requires a substantial modifications to the MapReduce scheduler framework. More recently, there are proposals to mitigate the data skew problem by integrating such skew handling mechanism into the declarative query language itself [5, 6]. However, such approach effectively relegates the optimization responsibility to the application developer.

Our study shows that a simple linear cost model to re-partition tasks for spatial join can reduction in response time[33]. We will approach the skew problem systematically at two levels. At the higher level, we will research an iterative feedback based approach. The query optimizer tries to generate a sound execution plan, and during the execution, the system monitors query performance and corroborates such information with original estimation. Such feedback is fed into the query optimizer for correction and knowledge update. At the lower level, we will provide balanced task scheduling through optimized task balancing. The goal is to achieve roughly equal amount of workload at each worker, assuming that we can estimate the cost of each subtask. Thus there are two problems to solve: load-balancing and cost modeling. Optimal solutions are NP-hard and we can approximate through a greedy approach. For the later, we will investigate two possible approaches: speculative execution through Hadoop and conventional cost estimation approaches.

Better Data Partitioning

Data partitioning is a powerful mechanism for improving efficiency of data management systems. Data partitioning improves the overall manageability of large datasets, and it improves query performance in two ways. First, partitioning the data into smaller units enables processing of a query in parallel, and henceforth the improved throughput. Second, with a proper partitioning schema, I/O can be significantly reduced by only scanning a few partitions that contain relevant data to answer the query. Therefore, a partitioning approach – that evenly distributes the data across nodes and facilitates parallel processing – is essential for achieving fast query response and optimal system performance.

In large data warehousing systems, data tables are horizontally or vertically partitioned, and partitions are distributed across processing nodes. Range partition, list partition and hash partition are well known approaches that are extensively studies in both research community and industry.

However, several pitfalls of spatial data partitioning make this task particularly challenging. First, data skew is very common in spatial applications. To achieve best query performance, data skew need to be reduced to the minimum. Second, spatial partitioning approaches generate boundary objects that cross multiple partitions, and add extra query processing overhead. Therefore, boundary objects need to be minimized. In this dissertation, we intend to study different approaches and provide a systematic evaluation. Our main objective is to provide a comprehensive guidance for optimal spatial data partitioning to support scalable and fast spatial data processing on MapReduce.

Hardware Accelerated Query Processing

We propose to use GPU based algorithms to accelerate spatial operations that can be integrated into MapReduce based spatial data processing pipelines. Our effort in this plan focuses on how to optimize spatial query execution by embedding GPU programs in MapReduce programs.

We design an efficient bridge interface between the MapReduce program and the GPU program. Since a GPU has a disconnected memory space from a CPU, input data organization for GPU is needed to compensate the long latency of host-device communication. Many small tasks sent to GPU may incur much overhead on communication and subdue the benefit of GPU. We will investigate a prediction model to decide the granularity of tasks for GPU invocation, by considering both data communication cost and execution cost for different types of spatial operations. Prior work [120] uses a simple data batching technique to combine and batch process multiple tiles in a single GPU program, which is not generalized. We will consider multiple factors such as tile sizes, object counts, and average object sizes.

We will research solutions to achieve load balancing and data/operation aware task assignment in the CPU/GPU hybrid environment. Most prior work on MapReduce for CPU/GPU hybrid environments [67, 69] focuses on general programming models or systems support. To divide load between GPUs and CPUs, these systems often use the fixed relative GPU/CPU performance as a metric to estimate the percentage of tasks that should be routed from execution with each device. We argue, however, that the GPU/CPU relative performance in the execution of internal map and reduce tasks created in spatial query.

1.5 Dissertation Contributions and Outline

1.5.1 Contributions

In summuray, we have developed *Hadoop-GIS* [7, 31, 32, 33, 34, 50, 117] – a spatial data warehousing system over MapReduce. The goal of the system is to deliver a scalable, efficient, expressive spatial querying system for efficiently supporting analytical queries on large scale spatial data, and to provide a feasible solution that can be afforded for daily operations.

Hadoop-GIS provides a framework on parallelizing multiple types of spatial queries and having the query pipelines mapped onto MapReduce. Hadoop-GIS provides space partitioning to avoid skewed regions and achieve task parallelization, an indexing-driven spatial query engine to process various types of spatial queries, implicit query parallelization through MapReduce, and boundary handling to generate accurate results. By integrating the framework with Hive, Hadoop-GIS provides an expressive spatial query language by extending HiveQL [116] with spatial constructs, and automates spatial query translation and execution. Hadoop-GIS supports fundamental spatial queries such as point, containment, join, and complex queries such as spatial cross-matching (large scale spatial join) and nearest neighbor queries. Structured feature queries are also supported through Hive and fully integrated with spatial queries.

The main intellectual contributions of this dissertation are :

- The first principled study of MapReduce based spatial query processing approaches.
- A comprehensive study of two-way, multi-way spatial join techniques on MapReduce, and their evaluation on real datasets.
- A detailed bottom-up description of a MapReduce based spatial query system and architecture.
- Study of query optimization techniques for spatial query processing on MapReduce, and cost based task repartition.
- A systematic study of six spatial partition techniques, a general categorization of those approaches, and a detailed evaluation.
- Study of sampling based spatial partitioning approaches and their performance.
- Study of parallel spatial partitioning techniques on MapReduce and their evaluation.
- Study of hybrid system architecture that can utilize GPU for query co-processing, and spatial query processing techniques on such hybrid system.
- An open source implementation of MapReduce based spatial query system.
- Introduction of the pathology analytical imaging application use case to the spatial and data management community.

1.5.2 Outline

This dissertation has four components. Chapter 2 is the first component in which we describe the of state-of-the-art approaches to spatial query processing, existing systems, and related work. The second component, the core of this dissertation, is an overview of the proposed spatial query processing framework, and corresponding system implementation. Chapter 3 and 4 are dedicated to this topic. The third component is the study of spatial partitioning methods for spatial query processing, and their performance consideration. Chapter 5 and chapter 6 are intend to address those problems. Last component is the GPU extension of the system and the query engine, and chapter 7 provides a detailed study on this topic. In chapter 8, we describe the integration of the system with a popular SQL-to-MapReduce solution — Hive, and this concludes the dissertation.

Chapter 2

Approaches to Large Scale Spatial Data Analytics

There are two mainstream approaches for large scale data analysis. Namely, parallel database systems and MapReduce based systems. Practically, these systems share some common design elements: they both employ a shared-nothing architecture[108], and deployed on cluster of independent nodes via a high speed interconnecting network; both achieve parallelism by partitioning the data and processing the query in parallel based on such partitioning. However, neither one of them alone is the "silver bullet" to the large scale data analysis. Both systems have their advantages and disadvantages, and analysis tasks are increasingly complex such that "one size fits all" systems no longer meet the efficiency requirement. Recent trends, that major parallel database systems vendors tend to package MapReduce functionality into their existing flagship warehousing products, may explain such market reality.

2.1 Spatially Extended Relational Database Systems

Parallel database systems [53] have been the primary choice for large scale data analysis for nearly two decades. As of this writing, there are dozens commercial systems are available in the marketplace, which claims to be a scalable and high performance enterprise data warehousing (EDW) solution. Theses systems include, Greenplum [13], Teradata/Aster Data [20], Netezza [14], DB2 (via DPF) [11], Microsoft SQL Server [85], Oracle Exadata, Vertica [22], MonetDB [15], and several others. Most of the modern parallel database systems are structured in a shared-nothing architecture [108] in which a cluster of nodes independently work together to process a query. Each processing unit has its own disk, CPU, and memory, and nodes are interconnected with a high bandwidth network for communication. Figure 2.1 shows a simplified version of such architecture.

In shared-nothing architecture, upon receiving a query, DBMS generates an execution plan based on the data distribution and locality information; then it notifies each worker node to process parts of data which are local to that node. DBMS can use scatter and gather operations to synchronize, aggregate or redistribute intermediate data. Large scale data warehousing systems use a partition function to horizontally/vertically partition data tables, and distributed those data chunks across processing nodes. Range partition, list partition, and hash partition are well known approaches for table partitioning that are extensively studies in both research community and industry.

Many applications use spatial database management systems (SDBMSs) for managing and querying spatial data. SDBMS is an extension of the relational database technology, and researchers have devoted much effort to better support spatial applications in the early days of object-relational era. Rather than building a specialized system for specific class of applications, those approaches [38, 40, 46, 63, 65, 101, 111] mostly focused on constructing a generalized DBMS that can support diverse applications — geographical, spatial, and CAD — very easily, using the the existing object-relational model. Proponents of such approaches added new constructs [92, 110, 112], abstract data types and indexes, to support the multidimensional nature of those applications. For example, geometry data types such as *point*, *line*, *polygon*, *multiline*, *multipolygon*,

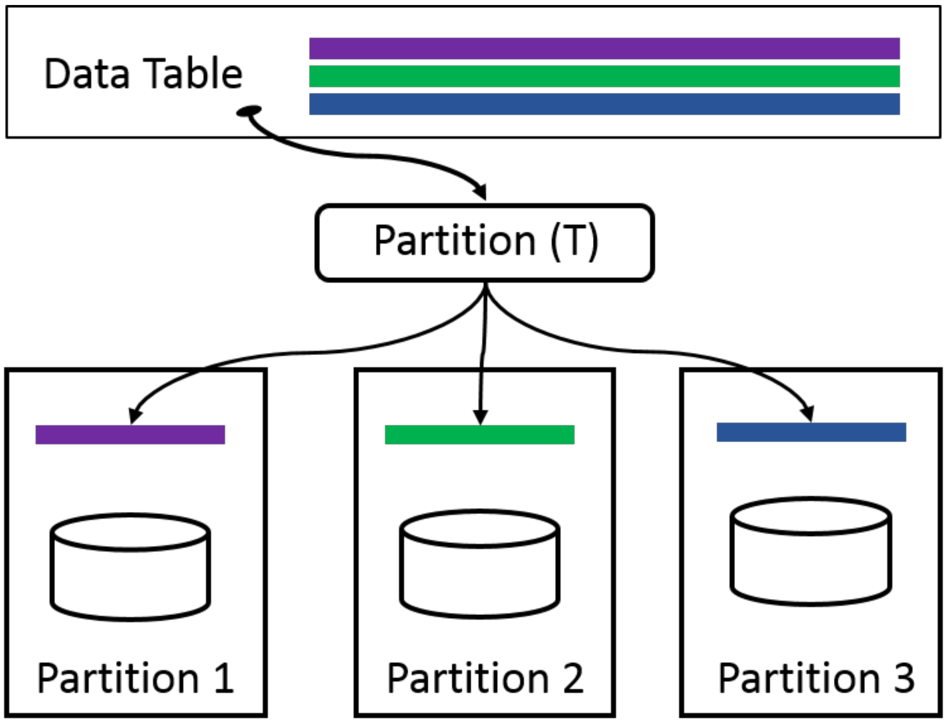


Figure 2.1: A shared-nothing parallel database architecture

and their 3D variations are available in major spatial database systems. Reference [70] provides a comprehensive survey on this subject.

Due to their tight integration with the existing relational database systems, spatial databases can benefit from advancement in the relational field. Specifically, using the parallelization mechanism that the relational database provides, we can setup a parallel spatial database system. Commercial examples of such parallel systems include Oracle Spatial [17], IBM DB2 Spatial Extender [11], Microsoft SQL Server Spatial [85] and Greenplum Database [13]. Therefore, it is natural to ask if we can provide a high performance spatial data management solution for large scale spatial data analytics.

In the past, we have developed and deployed a parallel spatial database solution named *PAIS* [25, 118, 119]. However, our experience indicates that there are several problems with parallel database system based solutions.

For example, in a business data processing and warehousing scenario, parallel database systems can easily partition data records by the *date* column, so that records that belong to a specific date end up in the same partition, and each node has some collections of such partitions to manage. There are many advantages of such data partitioning, among which are improved storage manageability, increased data parallelism, and more opportunities for query optimization. Specifically, for a query that only requests data from a specific date, the DBMS can quickly identify the corresponding data file, rather than performing a brute-force whole table scan operation. Figure 2.2 shows an example for such case in which *sales* table is range partitioned by the *date* column.

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
END (date '2009-01-01') EXCLUSIVE
EVERY (INTERVAL '1 day') );
```

Figure 2.2: A SQL query for creating partitioned table on Greenplum parallel database

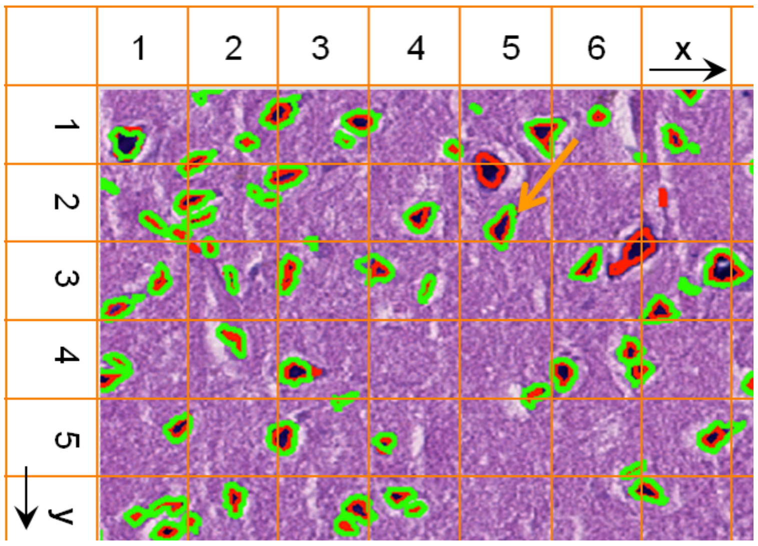


Figure 2.3: An ad-hoc grid partitioning of a pathology image

However, such partitioning operation is not available for the spatial data types. While spatial indexes can certainly help reduce the amount of data need to be processed for answering a query, absence of the *spatial partitioning* feature effectively reduces the parallel database system performance to the one of a single node system. One ad-hoc approach to go around such software limitation is to partition the spatial data with a regular fixed grid approach, and associate each grid cell with a simple numeric/string ID. Then use some kind of translation mechanism or pre-processing to translate those IDs into actual grid partitions boundaries which in turn can be used for region filtering. Figure 2.3 shows an example of such approach. Even after all those trouble, the *boundary object problem* haunts back such ad-hoc approach, and we will explain in more detail in Chapter 3.

Second, parallel database systems are not optimized for spatial analytics tasks. Parallel spatial data base systems have poor performance on complex analytical tasks, and require sophisticated tuning and maintenance efforts [94]. The high overhead of data loading is another major bottleneck for parallel SDBMS based solutions [94], and this is not acceptable for certain high data volume applications. The specific reasons why parallel database system vendors did not provide an efficient spatial analytics solution is out of the scope of this dissertation. Our conjecture is that major commercial vendors have targeted and optimized their flagship products to the larger OLTP/OLAP market, and the niche spatial analytics market have not received much attention.

Third, database approach is highly expensive on software licensing and dedicated hardware. Scalability of parallel database system comes at a high price. For example, a major commercial vendor prices a parallel data warehousing system in the range of $34 - 69$K \$ per Terabytes of data.

2.2 GIS systems

A geographic information system provides an end-to-end solution to capture, manage, manipulate and visualize geographical data, and it often uses a SDBMS as the back-end spatial engine [10]. Many GIS systems focus on specific applications and visualization, and have limited query capabilities to support complex spatial data analytics at massive scale.

2.3 MapReduce based Systems

MapReduce [52] is a very scalable parallel processing framework. Hadoop [26] — an open source implementation of MapReduce — is widely used in many data analytics tasks to support the modern data intensive applications. MapReduce is based on a highly scalable Shared Nothing parallel processing architecture which requires partitioning on the input data, and processes partitions in parallel. In each MapReduce task, Mappers/Reducers work on different logical portions of an input file, called input splits. Splits typically, but not always, correspond to physical data chunks.

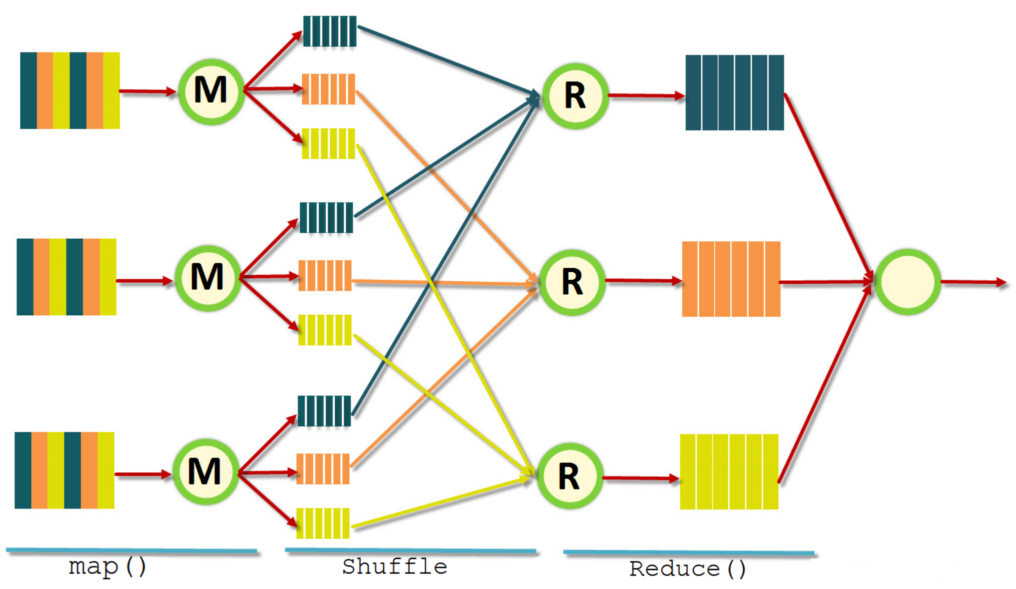


Figure 2.4: Overview of MapReduce execution

Figure 2.4 shows an overview of dataflow in MapReduce based data processing. A typical MapReduce job has three phases: *Map*, *Shuffle-and-Sort*, and *Reduce*. In the Map phase, a set of Mappers run in parallel on different TaskTrackers over different logical portions of an input file, called input splits. During the Map phase, Mappers write the intermediate results to local disks. These local intermediate results would be sorted and shuffled during the Shuffle phase, based on a partition key. Then Reduce phase starts, and each reducer will work on a set of shuffled results, and write the final output to the HDFS. MapReduce tasks run in parallel without any central orchestration or coordination.

2.4 Summary

Parallel database approach has major limitations on managing and querying spatial data at massive scale. Parallel DBMSs tend to reduce the I/O bottleneck through partitioning of data on multiple parallel disks and are not optimized for computational intensive operations such as spatial and geometric computations. Partitioned parallel DBMS architecture often lacks effective spatial partitioning to balance data and task loads across database partitions. While it is possible to induce a spatial partitioning, fixed grid tiling for example, and map such partitioning to one dimensional attribute distribution key, such an approach fails to handle boundary objects for accurate query processing. Scaling out spatial queries through a parallel database infrastructure is possible and we have explored this approach in our previous work [118, 119]. However, it comes at a very high cost (millions of dollars) and the scalability is rather limited. As the cloud based cluster computing technology gets mature and economically scalable, the massive scalability of MapReduce based systems offer an alternative solution for data-and-compute intensive spatial analytics at large scale.

Chapter 3

HadoopGIS – A Framework for Spatial Query Processing with MapReduce

In this chapter, we present a general overview of the proposed research –Hadoop-GIS – a system for scaling spatial queries on MapReduce.

3.1 Overview of the Framework

The main goal of Hadoop-GIS is to develop a highly scalable, cost-effective, efficient and expressive spatial query processing system for data and compute intensive spatial applications, that can take advantage of MapReduce on large number of commodity clusters. One of our main objectives is to identify time consuming spatial querying components, break them down into smaller tasks, and have them run in parallel.

An intuitive approach is to partition the data through partitioning the space, and assigning spatial objects to partitioned regions (or tiles). The generated tiles would form a parallelization unit for query processing. The query processing problem then reduces to designing efficient querying methods, opaque to the space partitioning, that can run on these tiles independently.

In MapReduce environment, we propose the following steps on running a typical

Algorithm 1: Typical workflow of spatial query processing on MapReduce

```
1  A. Data/space partitioning;
2  B. Data storage of partitioned data on HDFS;
3  C. Pre-processing queries (optional);
4  for $tile$ in $input\_collection$ do
5      Indexing building for objects in the tile;
6      Tile based spatial querying processing;
7  E. Boundary object handling;
8  F. Post-query processing;
9  G. Data aggregation;
10 H. Result storage on HDFS;
```

spatial query, as shown in Algorithm 1. In step A, we perform effective space partitioning to generate tiles. In step B, spatial objects are assigned tile UIDs, merged and stored into HDFS. Step C is for pre-processing queries, which could be queries that do global index based filtering, queries that does not need to run in tiles as the queries may not fit into the partitioning processing model, or queries that are fast enough and no tile based processing is needed. Step D does tile based spatial query processing independently which will be parallelized through MapReduce. Step E provides handling of boundary objects when needed, which can run as another MapReduce program. Step F does post-query processing, for example, joining spatial query results with feature tables, and can also be another MapReduce application. Step G performs data aggregation on final results if needed, and final results are written to HDFS in step H.

Figure 3.1 shows the architectural components of Hadoop-GIS. The system two sub-components, *data storage* and *query execution*. The storage sub-component uses HDFS for distributed data storage, and it is mainly responsible for retrieving data blocks to support efficient query processing. The query execution component includes the query interface for workload composition, the query translation module for translating declarative queries into HadoopGIS query operators, and the spatial query engine that executes query operators. HadoopGIS uses Hadoop as the MapReduce engine for query processing.

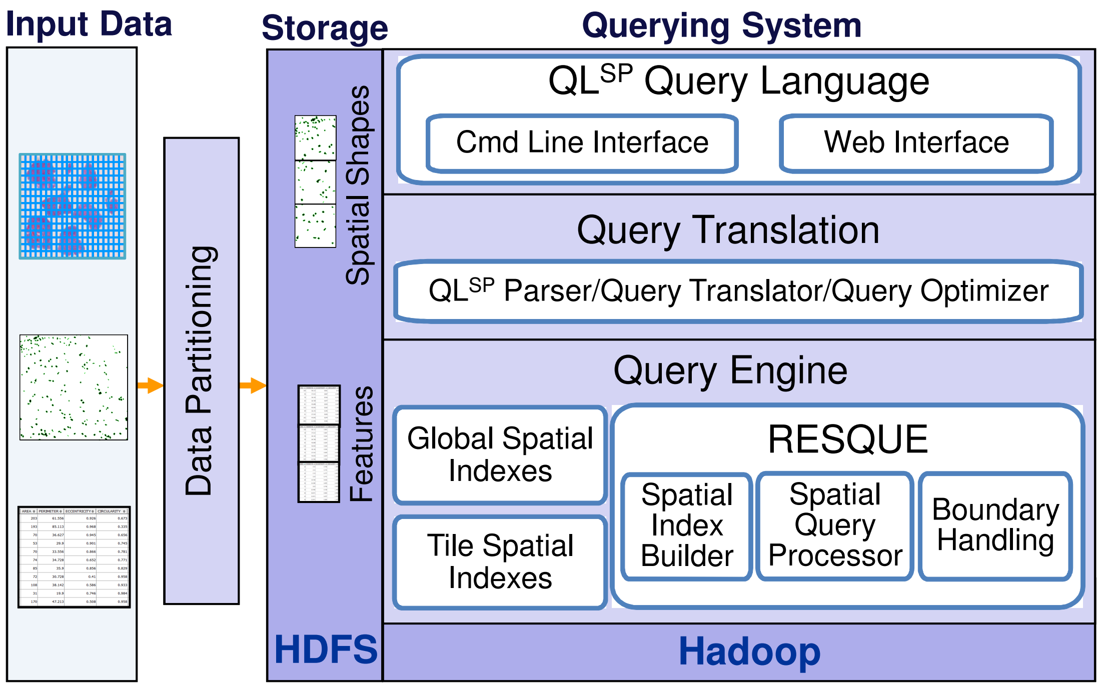


Figure 3.1: Architectural overview of Hadoop-GIS

In the following sections, rather than explaining each component in isolation, we give an overview of fundamental research problems and techniques in building such system. Those include data partitioning and storage, MapReduce based spatial query parallelization, multi-level spatial indexing, and query processing cost model.

3.2 Data Partitioning and Storage

Spatial data partitioning is an essential initial step to define, generate and represent partitioned data. Effective data partitioning is critical for task parallelization, load balancing, and directly affects system performance. Previously, Paradise [93] – a parallel spatial database system, used a *regular fixed grid* partitioning for parallel join processing. However, there are several problems with this approach as described in the original work. i) As spatial objects (e.g. polygons and polylines) have extent, regular grid based spatial partitioning would undesirably produce objects spanning multiple cell grids, which need to be replicated and post processed. If such objects account for a considerable fraction of the dataset, the overall query performance would suffer from such boundary handling overhead. ii) Fixed grid partitioning is skew averse, whereas data in most real world spatial applications are inherently highly skewed. For example, in OpenStreetMap (OSM), certain regions have more data compared to others due to enthusiastic data contributors. If OSM is partitioned with fixed grid approach with 1000x1000 grids, the average count of objects per tile is 993, but the tile with most objects contains $794,429$ objects. In such case, it is very likely that parallel processing nodes assigned to process those dense regions will become the *stragglers*, and the overall query processing efficiency will be severely affected [105].

In this dissertation we propose *SATO* — an effective and scalable partitioning framework [117] that can partition a geospatial dataset into *balanced regions* while *minimizing the number of boundary objects*. The partitioning methods are designed for scalability, which can be easily parallelized for high performance, for example, running on MapReduce. SATO stands for four main steps in this framework for spatial data partitioning: **S**ample, **A**analyze, **T**ear, and **O**ptimize. First, we sample a small fraction of the dataset to identify overall global data distribution with potential dense regions. Next we analyze the sampled data with a *partition analyzer* that produces a coarse partition scheme in which each partition region is expected to contain roughly equal amounts of spatial objects. Then we pass these coarse partition regions into the partitioning component that *tears* the regions into more granular partitions that are data skew aware and meet partition requirements. Finally, we analyze the generated partitions to produce multi-level partition indexes and additional partition statistics which can be used to *optimize* queries. SATO is also implemented for parallelization. We developed SATO as a standalone program for partitioning spatial datasets, and we also integrated the it with the Hadoop-GIS.

3.3 MapReduce Based Parallel Query Execution

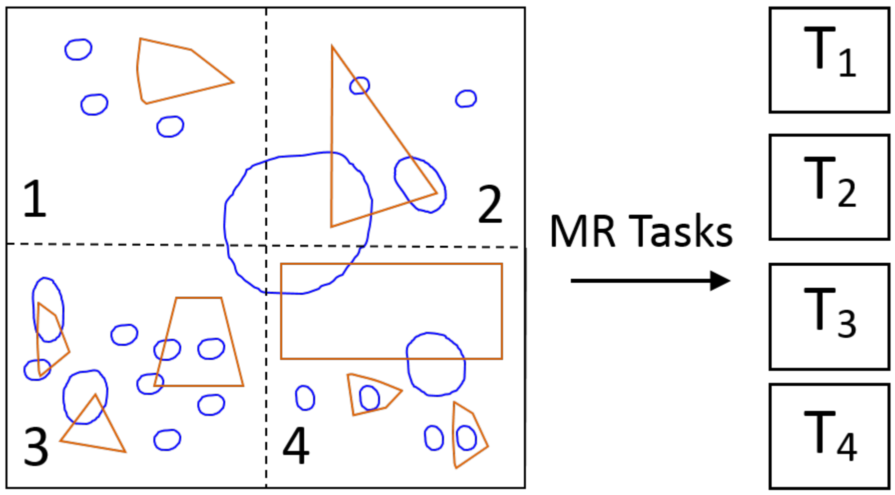


Figure 3.2: An example of MapReduce based spatial query parallelization

Instead of using explicit spatial query parallelization as summarized in [45], we take an implicit parallelization approach by leveraging MapReduce. This will much simplify the development and management of query jobs on clusters. As data is spatially partitioned, the tile name or UID forms the key for MapReduce, and identifying spatial objects of tiles can be performed through mapping phase. Figure 3.2 shows a simple example where the dataset is partitioned into four tiles (dotted lines depict partition boundary). To process a spatial query such as *find object pairs that intersect with each other from two datasets*, a single MapReduce job can be started where each tile is processed by a single mapper (T_1, T_2, T_3, T_4).

Depending on the query complexity, spatial queries can be implemented as map functions, reduce functions or combination of both. Based on the query type, different query pipelines are executed in MapReduce. As many spatial queries involve high complexity geometric computations, query parallelization through MapReduce can significantly reduce query response time. We devote the entire chapter 4 to this subject.

3.4 Boundary Object Processing

There is one specific problem that is endemic to spatial partitioning – *boundary objects*. Due to their multidimensional nature, some spatial objects may span more than one partition. For example, in Fig. 3.2 the big round object in the middle crosses all tile boundaries. As a result, $R_i \cap R_j \neq \emptyset$ for $i \neq j$, and it requires the spatial query processing framework to be able to handle such cases.

Parallel spatial query processing algorithms remedy the boundary object problem in two ways. Namely *multi-assignment, single-join* (MASJ) and *single-assignment, multi-join* (SAMJ) [83, 126]. In MASJ approach, each boundary object is replicated to each tile that overlaps with the object. During the query processing phase, each partition is processed only once without considering the boundary objects. Then a *de-duplication* step is initiated to remove the redundancies that resulted from the replication. However, in SAMJ approach, each boundary object is only assigned to one tile. Therefore, during the query processing phase, each tile is processed multiple times to account for the boundary objects.

Both approaches introduce extra query processing overhead. In MASJ, the replication of boundary objects incurs extra storage cost and computation cost. In SAMJ, however, only extra computation cost is incurred by processing the same partition multiple times. Hadoop-GIS takes the MASJ approach [34] and the original work pointed out that: (a) In practice, the MASJ approach is proven to be significantly efficient than the SAMJ approach [126] (b) MASJ approach allows higher degree of parallelization such that, for large datasets, the query processing efficiency can be greatly improved, and de-duplication cost can be very small.

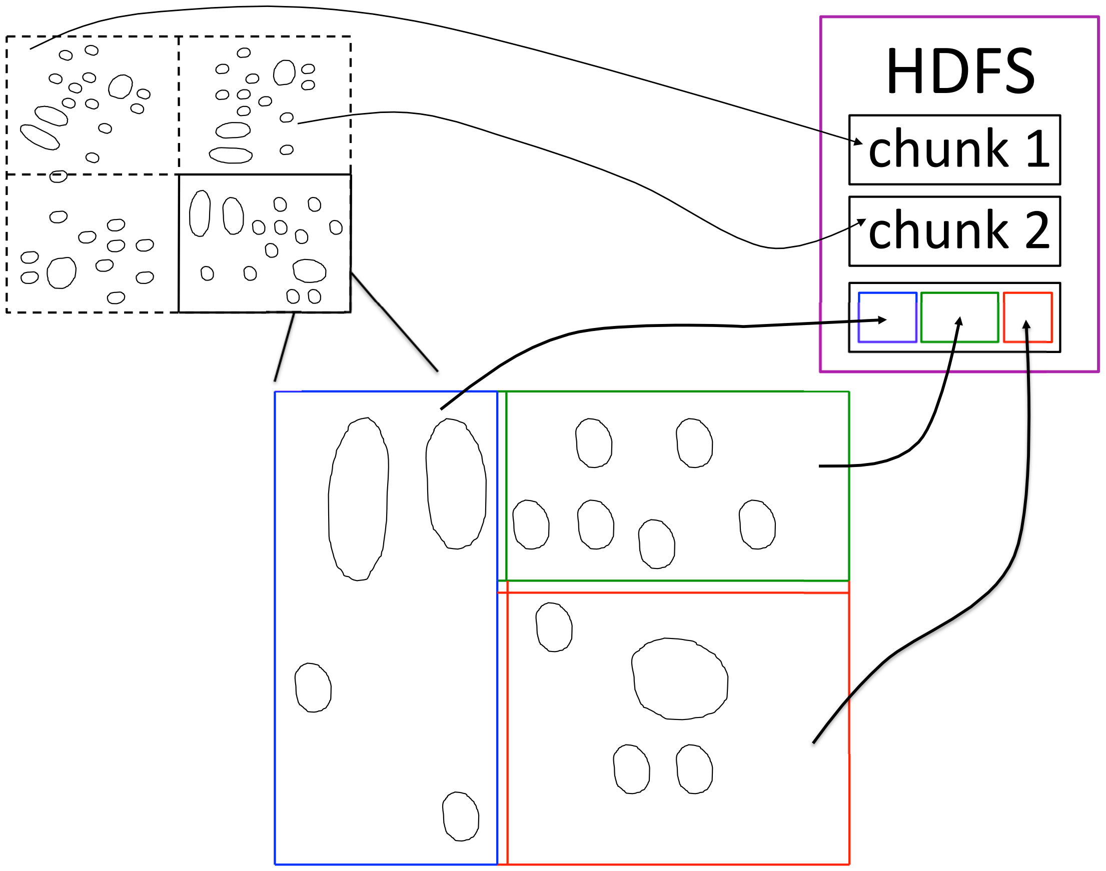


Figure 3.3: Multi-Level region based spatial index

3.5 Multi-Level Spatial Indexing

DBMS systems support both *scan* and *index* based query processing. Indeed utilizing indices (when such indices exist) for query processing are proven to improve query processing speed and efficiency. However MapReduce is a *scan* based data processing framework which does not utilize any form of disk based index. How to support DBMS like general data index on MapReduce is a hot topic in the database research community, and we explore how multidimensional spatial indices can be utilized for query processing in Hadoop-GIS. The main idea is to use a region based hierarchical space partitioning and MBB based region filtering. Figure 3.3 illustrates an example of such design.

On the top left is the spatial universe which contains large number of spatial objects.

The universe can be partitioned into independent *regions* and the level of partition can be very coarse. Here, for the sake of simplicity, the spatial universe is partitioned into 4 square regions. Region extent can be regular (rectangular) or irregular polygons which also may depend on the application requirement. In Figure 3.3, the region size are decided such that each region would fit into a single HDFS file chunk. Recursively, each region can be further partitioned into even smaller regions – tiles, as show on the bottom of the figure. In Figure 3.3, a single region is partitioned into 3 tiles (green, yellow, blue). Such multilevel indexing schema could save large I/O cost, improve query performance, and energy efficient. For example, image a containment query which asks for the spatial objects contained with a query window which totally fits into a partition region. In such scenario, a naive MapReduce solution would scan all the partitions, i.e. 4 HDFS file chunks, to answer the query. On the other hand, an index aware MapReduce solution – Hadoop-GIS, would only need to scan one HDFS file chunks. If the query window is about the size of a tile, the query processing system can take advantage of the fact that only single tile need to be processed and can skip other tiles.

3.6 Query Processing Cost Model

In HadoopGIS framework, the cost of processing a query involves non-boundary objects processing cost, duplicated boundary objects processing cost, and object de-duplication cost. A simple modeling approach can help us better understand overall query processing overhead and provide theoretical guidelines for optimizing spatial partitioning for improved query performance. Next, we use a simple example to illustrate the cost model.

Given two datasets R and S, a spatial join query finds all the pairs $r_i \in R, s_j \in S$ that satisfies a spatial topology relationship $F(r_i, s_j) = 1$. Selection of the spatial topology function can be arbitrary and without loss of generality, we use `st_intersects` as an example throughout the paper. This query simply finds all the intersecting object pairs from the datasets. Let us assume that, datasets are merged and co-partitioned with a partition schema which results partitions $R = \cup_{i=1}^{k} R_i$ and $S = \cup_{i=1}^{k} S_i$. Following the MapReduce based query processing framework , we have:

$$R \overset{F}{\bowtie} S = \bigcup_{i=1}^{N} R_i \overset{F}{\bowtie} S_i \tag{3.1}$$

Now, let us make few assumptions to simplify the analysis. First, let us assume that in each dataset data is uniformly distributed. Therefore, without considering boundary objects, each partition of the datasets contains roughly $|R|/k$ and $|S|/k$ objects. Second, the ratio of boundary objects due to the multiple assignment is α. Hence, each partition contains $(1+\alpha) * |R|/k$ objects. Third, overall query processing cost is the sum of partitioned query cost C_1 and duplicate elimination cost C_2 that depends on the query output and dataset size. Following the equation (3.1), cost if processing the spatial join query is:

$$\begin{aligned} C(R \overset{F}{\bowtie} S) &= \sum_{i=1}^{k} C_1(R_i \overset{F}{\bowtie} S_i) + C_2 \\ &= \sum_{i=1}^{k} \frac{(1+\alpha)|R|}{k} \frac{(1+\alpha)|S|}{k} + \beta(|R| + |S|) \\ &= \frac{(1+\alpha)^2 |R||S|}{k} + \beta(|R| + |S|) \end{aligned} \tag{3.2}$$

It is clear from the above simple cost model that, partition granularity is a double-edged sword. On one hand, a finer level of partition (larger k) is favorable as it improves the query performance. On the other hand, intuitively, a finer level of partition would generate many boundary objects (larger α); consequently it is detrimental to the query performance. Clearly, there is a sweet spot for the partitioning granularity such that it yields the best query performance. Finding the optimal partition granularity is non-trivial as it depends on the dataset characteristics and query type.

3.7 Related Approaches

Partitioning based approach for parallelizing spatial joins is discussed in [126], which uses the multiple assignment, single join approach with partitioning-based spatial join algorithm. The authors also provide re-balancing of tasks to achieve better parallelization. We take the same multiple assignment approach for partitioning, but use index based spatial join algorithm, and rely on MapReduce for load balancing. R-Tree based parallel spatial join is also proposed in early work [45] with a combined shared virtual memory and shared nothing architecture.

Spatial support has been extended to NoSQL based solutions, such as neo4j/spatial [16] and GeoCouch [12]. These approaches build spatial data structures and access methods on top of key-value stores, thus take advantage of the scalability. However, these approaches support limited queries, for example, GeoCouch supports only bounding box queries, and there is no support of the analytical spatial queries for spatial data warehousing applications.

In [47], authors propose an approach for bulk-construction of R-Trees through MapReduce. In [124], a spatial join method on MapReduce is proposed for skewed spatial data, using an in-memory based strip plane sweeping algorithm. It uses a duplication avoidance technique which could be difficult to generalize for different query types. Hadoop-GIS takes a hybrid approach on combining partitioning with indexes and generalizes the approach to support multiple query types. Besides, our approach is not limited to memory size. VegaGiStore [125] tries to provide a Quadtree based global partitioning and indexing, and a spatial object placement structures through Hibert-ordering with local index header and real data. The global index links to HDFS blocks where the structures are stored. It is not clear how boundary objects are handled in partitioning, and how parallel spatial join algorithm is implemented. Work in [19] takes a fixed grid partitioning based approach and uses sweep line algorithm for processing distributed joins on MapReduce. It is unclear how the boundary objects are handled, and no performance study is available at the time of evaluation. The work in [62] presents an approach for multi-way spatial join for rectangle based objects, with a focus on minimizing communication cost. A MapReduce based Voronoi diagram generation algorithm is presented in [35]. Closest to our work is SpatialHadoop [56], Niharika [96], and Parallel SECONDO [84].

Comparisons of MapReduce and parallel databases for structured data are discussed in [52, 94, 109]. Tight integration of DBMS and MapReduce is discussed in [27, 121]. MapReduce systems with high-level declarative languages include Pig Latin/Pig [59, 90], SCOPE [49], and HiveQL/Hive [116]. YSmart provides an optimized SQL to MapReduce job translation and is recently patched to Hive. Hadoop-GIS takes an approach that integrates DBMS's spatial indexing and declarative query language capabilities into MapReduce.

3.8 Summary

Our work was initially motivated by the use case of pathology imaging. We started from a parallel SDBMS based solution [119] and experienced major problems such as the data loading bottleneck, limited support of complex spatial queries, and the high cost of software and hardware.

Hadoop-GIS provides a generic framework for supporting multiple types of spatial applications, and a systematic approach for data partitioning and boundary handling. HadoopGIS combines the benefit of scalable and cost-effective data processing with MapReduce, and the benefit of efficient spatial query processing with spatial access methods. Hadoop-GIS achieves the goal through spatial partitioning, partition based parallel processing over MapReduce, effective handling of boundary objects to generate correct query results, and multi-level spatial indexing supported customizable spatial query engine

Chapter 4

MapReduce based Spatial Query Processing

In this chapter, we present our work on scaling spatial queries with MapReduce framework, and we experimentally evaluate our approaches on large datasets.

4.1 Real-time Spatial Query Engine

To support high performance spatial queries, a standalone spatial querying engine is needed. RESQUE is developed for such purpose and it supports: i) effective spatial access methods to support diverse spatial query types; ii) efficient spatial operators and measurement functions to provide geometric computations; iii) query pipelines to support diverse spatial queries with optimal access methods; and iv) the ability to run with decoupled spatial processing in a distributed computing environment. We have adopted a set of open source spatial and geometric computation libraries to support diverse access methods and geometric computations, including SpatialIndex [8], Computational Geometry Algorithms Library (CGAL) [21], GEOS [24], Boost [23], and build additional ones such as Hibert R-Tree [73]. Diverse spatial query pipelines are developed to support different types of queries based on the query type and data characteristics.

4.1.1 Index Supported Spatial Queries

One essential requirement for spatial queries here is fast response. This is important for both exploratory studies on massive amount spatial data with a large space of parameters, and algorithms, and decision making in enterprise or healthcare applications. Using spatial indexing to support spatial queries is a common practice for most SDBMS systems. However, the mismatch between the large data blocks in HDFS for batching processing and the page based storage and retrieval of spatial indexes makes it difficult to pre-store spatial indexes on HDFS and retrieve it later for queries. While some effort has been made on this [125], the approaches are not flexible and the pre-generated indexes might not be suitable to support dynamic spatial query types. To support indexing based spatial queries, we combine two approaches: i) global spatial indexes for regions and tiles; and ii) on demand indexing for objects in tiles.

For global spatial indexes, we manage the indexes through Hadoop distributed cache thus the indexes are easily shared across all cluster nodes. Given that the index granularity is very coarse, resulting index file size is generally very small. Global spatial indexing will facilitate region level data filtering. For an example, a point or window query can quickly lookup the global index to identify the tiles that are relevant for the query region.

We propose an approach on building indexes on-demand combined with data partitioning to process spatial queries. Our extensive profiling of spatial queries shows that index building on modern hardware is not a major bottleneck in large scale spatial query processing. Using dynamically built local indexes for objects in tiles could efficiently support spatial queries. To provide page based spatial index search, the built indexes can be stored in a local file system, for example, the local file system in Hadoop, or cached in memory depending on the availability of such resource during the query processing stage. Our tests show the indexing building time using RESQUE with R*-Tree based join takes a very small fraction of the overall response time (Section 4.1.3).

As indexes are read-only and no further update is needed, bulk-loading based index building techniques [42] are used. To minimize the number of pages, the page utilization ratio is also set to 100%. We also provide alternative indexing methods such as Hibert R-Tree, as it provides high efficient bulk loading (with slightly slower index searching performance). In addition, we provide compression to reduce leaf node shape sizes through compact chain code based representation: instead of representing the full coordinates for each x,y coordinate, we use offset from neighboring point to represent the coordinates. The simple chain code compression approach can save 40% space for the pathology imaging use case.

4.1.2 Spatial Query Workflows in RESQUE

Based on the query type, RESQUE can generate a query workflow which is optimized for that query type. Next we briefly describe various query workflows.

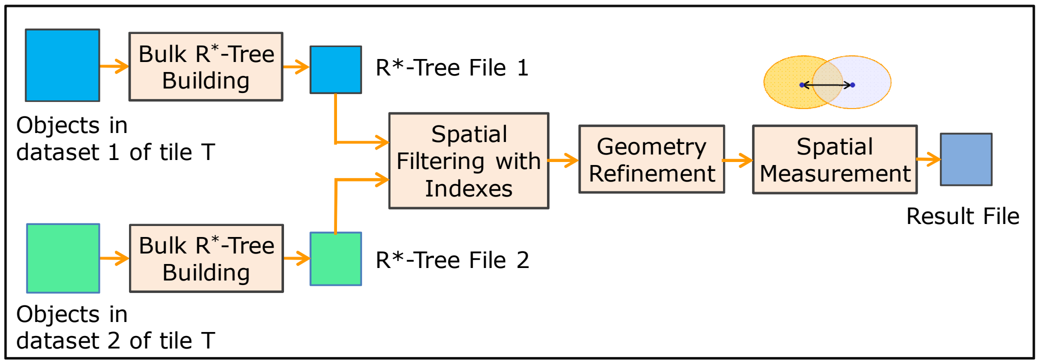


Figure 4.1: Spatial join query workflow in RESQUE

Spatial Join Query Workflow

Figure 4.1 illustrates workflow of a spatial join, where two datasets from a tile T are joined to find cross-matching of polygon objects. The SQL expression for this query is shown in Figure 4.3. Bulk spatial index building is performed on each dataset to generate index files – here we use R*-Trees [41]. (We also use Hilbert R-Tree when the objects are in regular shapes and relatively homogenous distribution.) The R*-Tree files are stored as local files and contain MBRs in their interior nodes and polygons in their leaf nodes, and will be used for further query processing. The spatial filtering component performs MBR based spatial join filtering with the two R*-Trees, and refinement on the spatial join condition is further performed on the polygon pairs through geometric computations. The spatial measurement step is performed on intersected polygon pairs to calculate results required, such as overlap area ratio for each pair of intersecting markups. Other spatial join operators such as *overlaps* and *touches* can be run in a similar way.

For each pair of markup polygons whose MBRs intersect, they are decoded from the representation, and geometry computation algorithm is used to check whether the two markup polygons actually intersect. If so, the spatial measurements are computed and returned. We rely on an open source libraries Boost [23] and *Computational Geometry Algorithms Library (CGAL)* [21] for computing the refinement and measurements. Spatial refinement based on geometric computation often dominates the query execution cost in data-intensive spatial queries, and could be accelerated through GPU based approach [32, 120].

Spatial Containment Query Workflow

Spatial containment queries have a slightly different workflow. The containing spatial object (e.g., a polygon) of a spatial containment query may span only a single tile or multiple tiles. Thus, an initial step will be to identify the list of intersecting tiles by looking up the global tile index, which could filter a large number of tiles. The MBR intersecting tiles will then participate the spatial join through the approach above, where only a single index is used in the spatial filtering phase. For an extreme case where the containing shape is small and lies within a tile, only a single tile is identified for creating the index. For a point query – given a point, find the containing objects, only a singe tile is needed thus it has similar workflow as the small containment query.

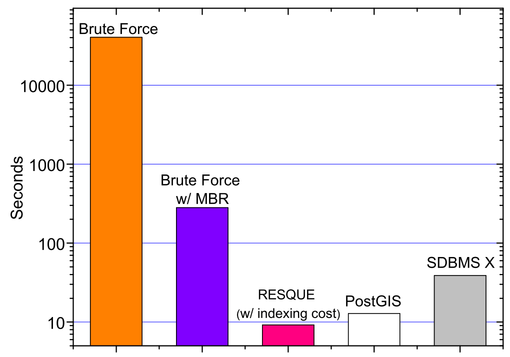
Nearest Neighbor Query Workflow

Nearest neighbor queries are complex to process and requires a different workflow. Specifically, the local nearest neighbors found within a spatial partition can be different from the ones found by searching whole spatial space. Therefore the partitioned parallelization approach need to be modified. In practice, however, we found that the number of the source objects – objects for which nearest neighbors need to be found, are significantly larger than the target objects – objects which constitute the nearest neighbors. For example, in a query setting such as "*for each cell, return nearest blood vessels and the distances*", there are far more cells than blood vessels. Therefore, we simplified nearest neighbor query in following way. We replicate the target objects (blood vessels) to each partition, and each partition is processed independently. Then a spatial index, for example R-Tree or Voronoi, is built to facilitate the search. This generates a local nearest neighbor query results set within each partition. Union of the local nearest neighbors from all partitions are consolidated as the final result.

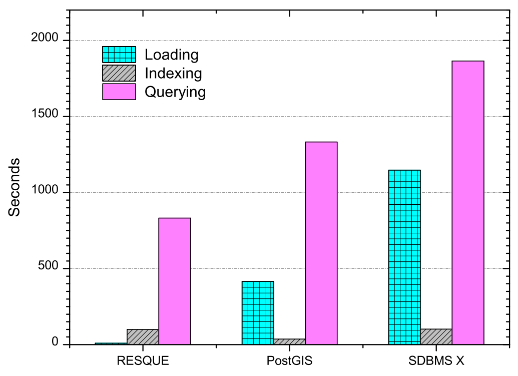
4.1.3 Query Engine Performance

RESQUE

An efficient query engine is a critical building block for a large scale system. To test the standalone performance of RESQUE, we run it on a single node as a single thread



(a) Single tile



(b) Single image

Figure 4.2: Performance of RESQUE

application. We use a spatial join query which finds all the intersecting polygons from two sets of spatial objects. We first test the *effect of spatial indexing*, by taking a single tile with two result sets (5506 polygons vs 5609 polygons), and the results are shown in Figure 4.2(a).

A *brute-force approach* compares all possible pairs of boundaries in a nested loop manner without using any index, and takes 673 minutes. Such slow performance is due to polynomial complexity on pair-wise comparisons and high complexity on geometric intersection testing. An *optimized brute-force approach* will first eliminate all the non-intersecting markup pairs by using a MBR based filtering. Then it applies the geometry intersection testing algorithm on the candidate markup pairs. This approach takes 4 minutes 41 seconds, a big saving with minimized geometric computations. Using RESQUE with indexing based spatial join, the number of computations is significantly reduced, and it only takes 16 seconds. When profiling the cost for RESQUE, we observe that reading and parsing cost is 30%, R*-Tree construction cost is 0.2%, MBR filtering cost is 0.67%, and spatial refinement and measurement cost is 69.13%. With fast development of CPU speed, spatial index construction takes very little time during the query process, which motivates us to develop an index-on-demand approach to support spatial queries. We can also see that geometric computation dominates the cost, which could be accelerated through parallel computation on a cluster environment.

Data Loading Performance

Another advantage of RESQUE is the light data loading cost compared to SDBMS approach. We run three steps to get the overall response time (data loading, indexing and querying) on three systems: RESQUE on a single slot MapReduce with HDFS, PostGIS and SDBMS-X with a single partition. The data used for the testing is two results from a single image (106 tiles, 528,058 and 551,920 markups respectively). As shown in Figure 4.2(b), data loading time for REQUE is minimal compared to others. Such light weight loading overhead is very important for certain spatial applications, as it greatly reduces the *data-to-query* time. NoDB [36] is one such example in which authors are mainly focused on reducing such data loading overhead.

4.2 MapReduce Based Spatial Query Processing

RESQUE provides a core query engine to support spatial queries, which enables us to develop high performance large scale spatial query processing based on MapReduce framework. Our approach is based spatial data partitioning, tile based spatial query processing with MapReduce, and result normalization for tile boundary objects.

4.2.1 Spatial Join

Spatial joins play an important role in various spatial applications, and it is a high cost query. A pairwise spatial join or *two-way spatial join* combines two datasets with respect to some spatial predicates. Next we discuss how to map spatial join queries into MapReduce computing model. We first show an example spatial join query for spatial cross-matching in SQL, as shown in Figure 4.3.

```
SELECT
  ST_AREA(ST_INTERSECTION(ta.polygon,tb.polygon))/
    ST_AREA(ST_UNION(ta.polygon,tb.polygon)) AS ratio,
  ST_DISTANCE(ST_CENTROID(tb.polygon),
    ST_CENTROID(ta.polygon)) AS distance,
FROM markup_polygon ta JOIN markup_polygon ON
    ST_INTERSECTS(ta.polygon, tb.polygon) = TRUE
WHERE ta.algrithm_uid='A1' AND tb.algrithm_uid='A2' ;
```

Figure 4.3: A spatial join (cross-matching) query commonly used in analytical imaging study

This query finds all intersected polygon pairs between two result sets generated from an image by two different methods, and compute the overlap ratios (intersection-to-union ratios) and centroid distances of the pairs. The table *markup_polygon* represents the boundary as *polygon*, algorithm UID as *algrithm_uid*, and image UID as *pais_uid*. There is also an attribute *tile_uid* to represent the tile an object belongs to. The SQL syntax comes with spatial extensions such as spatial relationship operator *ST_INTERSECTS*, spatial object operators *ST_INTERSECTION*, *ST_UNION*, and spatial measurement functions *ST_CENTROID*, *ST_DISTANCE*, and *ST_AREA*.

In this example, table (*markup_polygon*) is a spatial table that represents markups. This table has three major columns, namely *pais_uid*, *tile_uid*, and *polygon*, respectively. In each record, *algorithm_uid* represents algorithm UID, *pais_uid* represents image UID, and *tile_uid* is the UID of a tile the polygon is contained. This is a self join of the same table *markup_polygon* by selecting polygons generated from the same image 'IMG1' and produced by different algorithms 'A1' or 'A2'. In the SELECT clause of this query, we calculate intersection-to-union ratios and centroid distances of the polygon pairs with a few computational geometry functions.

For simplicity, we first present how to process the spatial join above with MapReduce, by ignoring boundary objects, and then we come back to the boundary handling in Section 4.3. A MapReduce program for spatial join query (Figure 4.3) will have similar structure as a regular relational join operation, but with all the spatial part executed by invoking RESQUE engine within the program. According to the equal-join condition, the program uses the Standard Repartition Algorithm [43] to execute the query. Based on the MapReduce structure, the program has three main steps: i) In the map phase, the input table is scanned, and the WHERE condition is evaluated on each record. Only those records that can satisfy the WHERE condition will proceed to the next step; ii) In the shuffle phase, all records with the same *tile_uid* would be shuffled to be the input of the same reduce function, since the join condition is based on *tile_uid*. iii) In the reduce phase, the join operation is finished by the execution of the reduce function. The spatial component is executed by invoking the RESQUE engine in the reduce function.

Algorithm 2: MapReduce Program for Spatial Join

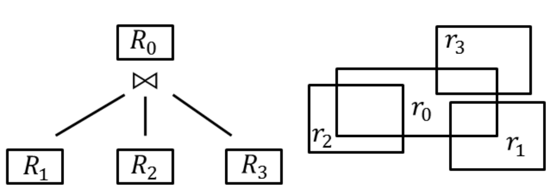
```
function Map($k$,$v$):
    $\_tile\_id$ = projectKey($v$);
    $join\_seq$ = projectJoinSequence($v$);
    $record$ = projectRecord($v$);
    $v$ = concat($join\_seq$,$record$);
    emit($\_tile\_id$ , $v$);

function JoinReduce($k$,$v$):
    /* arraylist holds join objects */
    $join\_set$ = [ ] ;
    for $v_i$ in $v$ do
        $join\_seq$ = projectJoinSequence($v_i$);
        $record$ = projectRecord($v_i$);
        if $join\_seq$ == 0 then
            $join\_set$[0].append($record$);
        if $join\_seq$ == 1 then
            $join\_set$[1].append($record$);
    /* library call to RESQUE */
    $plan$ = RESQUE.genLocalPlan($join\_set$);
    $result$ = RESQUE.processQuery($plan$);
    for $item$ in $result$ do
        emit($item$);
```

The workflow of the map function is shown in the map function in Algorithm 2. Each record in the table is converted into the map function input key/value pair (k_i, v_i), where k_i is not used by the program and v_i is the record itself. Inside the map function, if the record can satisfy the select condition, then an intermediate (k_m,v_m) is generated. The key k_m is the value of *tile_uid* of this record, and the value v_m is the values of required columns of this record. There are two remarkable points. First, since (k_m,v_m) will participate a two-table join, a tag must be attached to v_m in order to indicate which table the record belongs to. Second, since the query for this case is a self-join, we use a shared scan in the map function to execute the data filter operations on both instances of the same table. Therefore, a single map input key/value could generate 0, 1 or 2 intermediate key/value pairs, according to the SELECT condition and the values of the record.

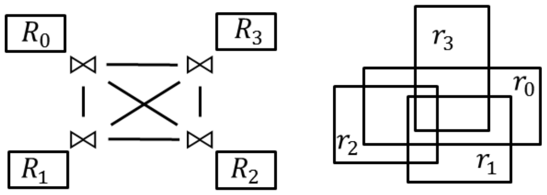
The shuffle phase is controlled by Hadoop itself, and groups data by tile UIDs. The workflow of the reduce function is shown in the reduce function in Algorithm 2. According to the main structure of the program, the input key of the reduce function is the join key (*tile_uid*), and the input values of the reduce function are all records with the same *tile_uid*. In the reduce function, we first initialize two temporary files, then we dispatch records into corresponding files. Then, we invoke RESQUE engine to build R*-tree indexes and execute the query. The execution result data sets are stored in a temporary file. Finally we parse that file, and output the result to HDFS. Note that the function *RESQUE.execute_query* here performs multiple spatial functions together, including evaluation of WHERE condition, projection, and computation (e.g., *ST_intersection* and *ST_area*), which could be customized.

Other join algorithms such as Improved Repartition Join [43] will not help on improving the query performance due to the characteristics of spatial join, where geometry computation is computational intensive and dominates the query execution time.

4.2.2 Multiway Spatial Join



(a) star join



(b) clique join

Figure 4.4: Spatial join query types

Spatial joins play an important role in effective spatial query processing for analytical pathology imaging. A pairwise spatial join or two-way spatial join combines two datasets with respect to some spatial predicates. *Multiway spatial join*s involve more than two spatial inputs and an arbitrary number of join predicates. For example, in Figure 4.4, the spatial relation R_0 is joined with three other relations with a predicate of *intersects*.

Depending on the actual join condition, the query graph may take different shapes, such as: i) chain ii) star iii) clique and iv) combination of the above. The shape of the query graph dictates the complexity of join processing. Queries with complex topological relationships are more expensive to evaluate. Here, we mainly focus on *star* and *clique* joins as shown in Figure 4.4. The reason is twofold. First, our experience indicates that star and clique queries are very common in spatial cross-matching and other spatial analytical tasks. Secondly, a complex query graph can be decomposed into a combination of several star and clique query graphs. Thus developing effective query evaluation techniques for these two types of queries can serve as a building block towards more complex query evaluation. The query pipeline and algorithm for processing multiway spatial joins are very similar to the two-way spatial join algorithm, and algorithm described in 4.2.1 can be extended to process multiway spatial join queries.

4.2.3 Nearest Neighbor Query

Spatial access methods are widely used to support point NN queries and a number of algorithms have been developed. Generally, these algorithms rely on the clustering properties of neighboring points and try to prune search space to quickly arrive at the neighborhood of the query point. In HadoopGIS, we provide two algorithms for efficient nearest neighbor query support.

NN Search with R*-Tree

In R-Tree, two metrics are defined to speed up the nearest neighbor search process, namely *mindist* and *maxmindist*. These metrics are used to prune as much of the R-Tree nodes as possible during both the downward searching process and the upward refining process. Details of the algorithm can be found in [99].

An approach similar to the R*-Tree join processing can be used to support nearest neighbor queries in MapReduce. However, tile based partitioning is not applicable in this scenario. Specifically, after such a partition, nearest neighbor of one object may reside in another tile. Thus if the nearest neighbors are processed independently we may not get the correct result. There are multiple ways to remedy this problem. One approach is to process the query in multiple passes such that in the first pass, only the index building process is initiated. In the second pass, partial indexes from the first pass are merged and replicated to other nodes. Thus, after several passes, each node would gather enough information to answer the query.

In the analytical pathology imaging setting, generally there are fewer target objects which are returned as the nearest neighbors, than the source objects. Consider the example of querying nearest neighbor blood vessel for each cell. The number of blood vessels (hundreds or thousands) is much smaller compared to the number of cells (millions). In this case, locating the target nearest neighbor is very fast whereas most of the query time is spent on iterating over millions of source objects. Therefore, we

Algorithm 3: Reduce Function

```
  Input: $k_i$, $v_i$
1 $tile$ = extract_source_objects($v_i$);
2 $k$ = get_K($v_i$);
3 $tar$ = read target objects from HDFS;
  // build R*-Tree index on target objetcs
4 $idx$ = $RESQUE$.build_index($tar$);
  // execute queries using spatial indexes
5 $result$ = $RESQUE$.execute_kNN_query($idx$,$tile$,$k$);
  // final output
6 output $result$ to HDFS;
```

take a simple approach in which only the source object set is partitioned, and the target object set is replicated and distributed to cluster nodes. Thus, each partition has a "global view" of the target search space and can carry on the nearest neighbor search without any communication overhead between nodes. In the Map phase, source objects are partitioned and target objects are replicated. The reduce phase of the algorithm is described in Algorithm 3.

Voronoi Diagram

Voronoi diagram [89] has been extensively studied in computational geometry and spatio-temporal database settings to support nearest neighbor queries. Given a set of input sites, typically points on the plane, *Voronoi diagram* divides the space into disjoint polygons where the nearest neighbor of any point inside a polygon is the site which has generated this polygon. These polygons are called *Voronoi polygons* and edges on adjacent Voronoi polygons define equidistance regions between two polygons. A number of algorithms are proposed to compute Voronoi diagrams and the best known algorithm has a lower bound complexity of $O(n\, log\, n)$, where n is the number of input line segments needed for computing the Voronoi diagram.

To answer the example nearest neighbor query, target objects (blood vessels) are replicated among cluster nodes for index construction. Source objects (cells) are par-

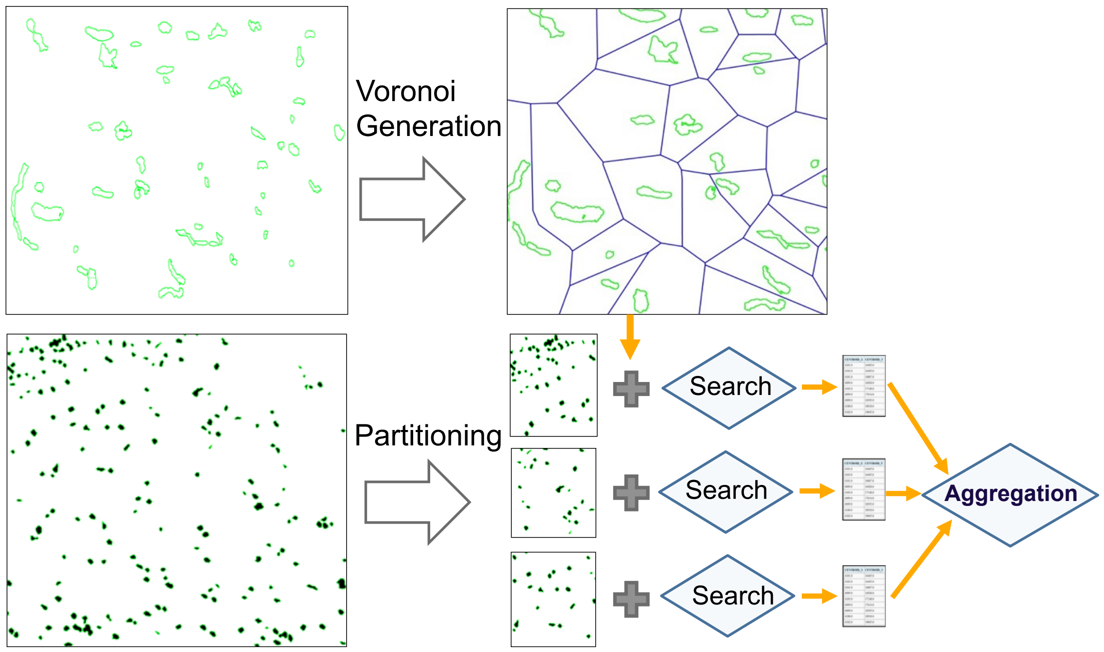


Figure 4.5: Nearest neighbor query workflow in RESQUE

titioned with tiling and distributed among the nodes participating the computation. Similar to the R*-Tree nearest neighbor query processing, a reducer first builds the Voronoi diagram for blood vessels which are represented as a set of line segments. Then for each cell in a given partition, the reducer queries the nearest blood vessel segments and computes the distance. To efficiently locate the Voronoi polygons, Voronoi diagrams are clipped to the size of a tile on each reducer. Figure 4.5 illustrates the workflow we described above.

The replication of target objects and computation of Voronoi diagram for the same set of objects may seem to cause extra overhead. There are two reasons why we do not also partition the target objects to achieve higher level parallelism. First, construction of Voronoi diagram is fairly fast due to the fact that the number of target objects is much less than the number of source objects. In our current dataset, target objects – blood vessels – roughly account for 0.1% of all spatial objects. In this case, the extra effort to parallelize the Voronoi diagram construction process may not justify itself. Even if the Voronoi diagrams are built in parallel, extra post-processing is needed to merge the partial Voronoi diagrams. Second, it complicates the SQL-to-MapReduce translator. Given these considerations, we do not parallelize the index building process in our current system.

4.2.4 Spatial Selection and Aggregation

Spatial selection/containment is a simple query type in which objects geometrically contained in selection region are returned. For example, in a medical imaging scenario, users may be interested in the cell features which are contained in a cancerous tissue region. Thus, a user can issue a simple query as shown in Figure 4.6, to retrieve cancerous cells. Since data is partitioned into tiles, containment queries can be processed in a *filter-and-refine* fashion. In the filter step, tiles which are disjoined from the query region can be filtered with MBR based testing. In the refinement step, the candidate objects are checked with precise geometry test. The query would be translated into a map only MapReduce program shown in Algorithm 4.

```
SELECT * FROM markup_polygon m, human_markup h
WHERE h.name='cancer' AND ST_CONTAINS(h.region, m.polygon) = TRUE;
```

Figure 4.6: A containment query commonly used in analytical imaging study

4.3 Boundary Handling

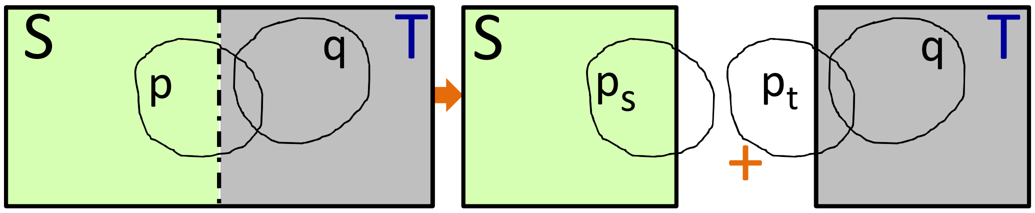


Figure 4.7: Boundary crossing objects

Algorithm 4: MapReduce Program for Containment Query

```
function Map(k,v):
    /* a arraylist holds spatial objects */
    $candidate\_set$ = [ ] ;
    $\_tile\_id$ = projectKey($v$);
    for $v_i$ in $v$ do
        $record$ = projectRecord($v_i$);
        $candidate\_set$.append($record$);
    $tile\_boundary$ =getTileBoundary($\_tile\_id$);
    if queryRegion.contains($tile\_boundary$) then
        emitAll($candidate\_set$);
    else
        if queryRegion.intersects($tile\_boundary$) then
            for $record$ in $candidate\_set$ do
                if queryRegion.contains($record$) then
                    emit($record$);
```

Tile is the basic parallelization unit in Hadoop-GIS. Space partitioning can be a simple fixed grid based partitioning where the data distribution is relatively balanced. A recursive partitioning can be applied to high density tiles to achieve a balanced partition. However, in such tile based partitioning, inevitably some objects would lie on the tile boundary. We define such a object as *boundary object* of which spatial extent crosses multiple tile boundaries. For example, in Figure 4.7 (left), the object p is a boundary object which crosses the tile boundaries of tiles S and T. In general, the fraction of boundary objects is inversely proportional to the size of the tile. As tile size gets smaller, the percentage of boundary objects increases.

There are several heuristics for the boundary object problem which may depend on the application requirements. One solution is to discard any boundary objects which arises from the tile based partitioning. While simple, this approach partially remedies the boundary problem and is suitable for certain application scenarios. For example, in pathology imaging, there are millions of spatial objects and the boundary objects only accounts for a very small fraction the data (generally less than 1%). Moreover, in

Algorithm 5: Boundary Aware Spatial Join

```
1  function Map(k,v):
2      _tile_id = projectKey(v);
3      record = projectRecord(v);
4      if isBoundaryObject(record, _tile_id) then
5          tiles = getCrossingTiles(record) ;
           /* replicate to multiple tiles                              */
6          for tile_id in tiles do
7              emit(tile_id , v);

8  function JoinReduce(k,v):
       /* arraylist holds join objects                                */
9      join_set = [ ] ;
10     for $v_i$ in v do
11         join_seq = projectJoinSequence($v_i$);
12         record = projectRecord($v_i$);
13         if join_seq == 0 then
14             join_set[0].append(record);
15         if join_seq == 1 then
16             join_set[1].append(record);
       /* library call to RESQUE                                      */
17     plan = RESQUE.genLocalPlan(join_set);
18     result = RESQUE.processQuery(plan);
19     for item in result do
20         emit(item);

21 function Map(k,v):
22     $uid_1$ = projectUID(v,1);
23     $uid_2$ = projectUID(v,2);
24     key = concat($uid_1$,$uid_2$);
25     emit(key,v);
   /* Hadoop sorts records by key and shuffles them                   */
26 function Reduce(k,v):
27     for records in v do
28         if isUniq(record) then
29             emit(v);
```

most warehousing applications, users are interested in statistical or aggregate results which are derived from large number of records. Therefore, the missing boundary objects would not pose any challenge to the correctness of the query results. Whereas in many other applications, query accuracy is critical and the boundary objects need to be handled correctly.

Hadoop-GIS remedies the boundary problem in a simple but effective way. If a query requires to return complete query result, Hadoop-GIS generates a *boundary-aware* query plan which has a pre- and post-processing task. In the pre-processing task, the boundary objects are duplicated and assigned to multiple intersecting tiles (multiple assignment). When each tile is processed independently during query execution, the results are not yet correct due to the duplicates. In the postprocessing step, results from multiple tile based query processing will be normalized, e.g., to eliminate duplicate records by checking the *object uid*s, which are assigned internally and globally unique during partitioning phase. For example, when processing the spatial join query, the object p is duplicated to tiles S and T as p_s and p_t (Figure 4.7 right). Then the same process of join processing follows as if there are no boundary objects. In the postprocessing step, objects will go through a filtering process in which duplicate records are eliminate in a sort-merge fashion.

One may argue that such approach would incur extra query processing cost due to the preprocessing and postprocessing steps. However, this extra cost is very small compared to the query processing time, which we will justify in Section 4.5.

4.4 Query Optimization Approaches

To show the adverse effects of spatial data skew, we run a join performance test where a set of images are partitioned into 40 reduce tasks and the completion time for each individual task is measured. As shown in Figure 4.8 (binned for visualization purpose), the actual completion time for each partition (brown bars) differs significantly and the overall system performance is largely affected by the longest running tasks. To

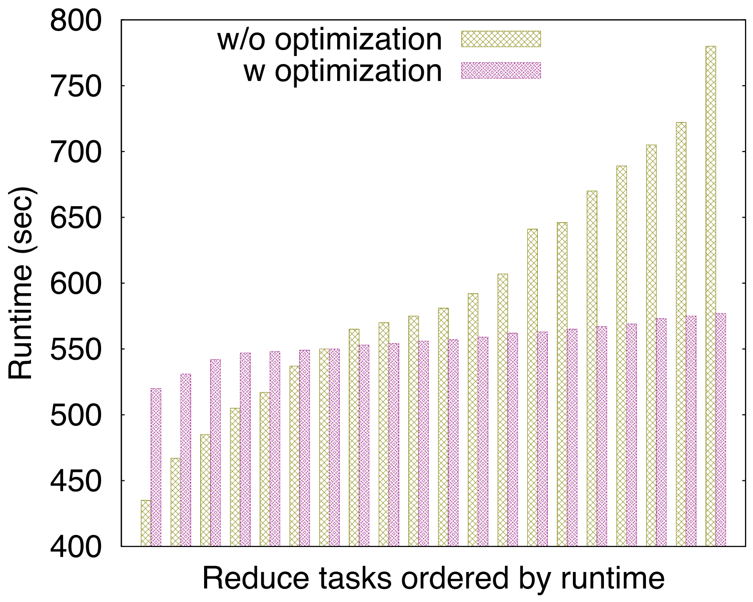


Figure 4.8: Spatial skew in query processing

remedy the skew problem, we take a cost-based greedy task partition approach in which each reducer is assigned roughly equal amount of work to balance the load for all the reducers. Consider a simplified version of the query 4.3 where two datasets are join with a predicate `intersects`, i.e., $Q = R \overset{intersects}{\bowtie} S$. To process this query Q, our system partitions each image into N tiles indexed by $I = \{1, 2, ..i..N\}$ and each pair of tiles would be assigned to a *reducer* node for join processing. When all tile pairs are processed, a final aggregation step will be performed.

$$Q = R \overset{intersects}{\bowtie} S = \bigcup_{i=1}^{N} R_i \overset{intersects}{\bowtie} S_i \tag{4.1}$$

Each *reducer* node will process a set P of tiles indexed by $\bigcup_{i \in P} i$ which we call the workload of a reducer node. Therefore, the query optimizer should generate a query plan that partitions the tiles indexed by I into k workloads such that $I = \bigcup_{i=1}^{k} P_i$ and the maximum workload is minimized. There are two problems need to be solved here. First, how to estimate the runtime of each workload ? Second, assuming that we know the completion time W_i for each workload, how to solve the partition problem?

Set partition problem is NP-Hard and an approximate solution would suffice in our case. Therefore, we take a simple approach in which tiles are *greedily* assigned to k partitions. However, the question of how to estimating completion time of each tile remains. Following the table statistics based cost estimation philosophy in modern database systems, we use following formula to estimate cost of processing each individual workload.

$$W_j = \sum_{i \in P_j} Cost(R_i \bowtie S_i) \tag{4.2}$$

$$Cost(R_i \bowtie S_i) = \alpha |R_i| + \beta |S_i| + \gamma \tag{4.3}$$

The coefficients α, β are introduced to reflect individual dataset characteristics. If we make no assumption about the dataset involved in query processing, we can set them to a constant value. The constant cost γ is introduced to take the cost of transferring the tile contents from other nodes across the network. While it can be set to an educated value, we simply set it to zero here, as in the join processing, the predicate cost dominates overall computation time and the I/O cost can simply be ignored.

4.5 Experiments

Next we describe the system performance in detail. We study the performance of Hadoop-GIS versus parallel SDBMS, scalability of Hadoop-GIS in terms of number of reducers and data size, and query performance with boundary handling. The tests are based on two real world datasets, with a set of representative benchmark queries.

4.5.1 Experimental Setup

Compared Systems

Hadoop-GIS: We use a cluster with 8 nodes and 192 cores. Each of these 8 nodes comes with 24 cores (AMD 6172 at 2.1GHz), 2.7TB hard drive at 7200rpm and 128GB memory. A 1Gb interconnecting network is used for node communication. The OS is CentOS 5.6 (64 bit). We use the Cloudera Hadoop 2.0.0-cdh4.0.0 as our MapReduce platform, and Apache Hive 0.7.1 for Hive^{SP}. Most of the configuration parameters are set to their default value, except the JVM maximum heap size which is set to 1024MB. The system is configured to run a maximum of 24 map or reduce instances on each node. Datasets are uploaded to the HDFS and the replication factor is set to 3 on each datanode.

DBMS-X: To have a comparison between Hadoop-GIS and parallel SDBMS, we installed a commercial DBMS (DBMS-X) with spatial extensions and partitioning capabilities on two database nodes. Each DB node comes with 32 cores, 128GB memory, and 8TB RAID-5 drives at 7200rpm. The OS for the nodes is CentOS 5.6 (64 bit). There are a total of 30 database partitions, 15 logical partitions on each node. With the technical support from the DBMS-X vendor, the parallel SDBMS has been tuned with many optimizations, such as co-location of common joined datasets, replicated spatial reference tables, proper spatial indexing, and query hints. For RESQUE query engine comparison, we also install PostGIS (V1.5.2, single partition) on a cluster node.

Dataset Description

We use two real world datasets: pathology imaging, and OpenStreetMap.

Pathology Imaging (PI). This dataset comes from image analysis of pathology images, by segmenting boundaries of micro-anatomic objects such as nuclei and tumor regions. The images are provided by Emory University Hospital. Spatial boundaries have been validated, normalized, and represented in WKT format. We have dataset sizes at 1X (18 images, 44GB), 3X (54 images, 132GB), 5X (90 images, 220GB), 10X (180 images, 440GB), and 30X (540 images, 1,320GB) for different testings. The average number of nuclei per image is 0.5 million, and 74 features are derived for each nucleus. For nearest neighbor query performance test, we use 50 images (42 GB) from TCGA. Both datasets have similar characteristics. The first dataset comes with polygons of nuclei and spatial features. The second dataset comes with polygons of nuclei and other spatial objects such as blood vessels.

OpenStreetMap (OSM). OSM [3] is a large scale map project through extensive collaborative contribution from a large number of community users. It contains spatial representation of geometric features such as lakes, forests, buildings and roads. Spatial objects are represented by a specific type such as points, lines and polygons. We download the dataset from the official website, and parse it into a spatial database. The table schema is simple and it has roughly 70 columns. We use the polygonal representation table with more than 87 million records. To be able to run our queries, we construct two versions of the OSM dataset, one from a latest version, and another smaller one from an earlier version released in 2010. The dataset is also dumped into text format for Hadoop-GIS.

Query Benchmarks

We take three typical queries for the benchmark: spatial join (spatial cross-matching), spatial selection (containment query), and aggregation. Many other complex queries can be decomposed into these queries, for example, a spatial aggregation can be run in two steps: first step for spatial object filtering with a containment query, followed by an aggregation on filtered spatial objects.

The spatial join query on PI dataset is demonstrated in Figure 4.3 for joining two datasets with an *intersects* predicate. Similar spatial join query on OSM dataset is also constructed to find changes in spatial objects between two snapshots. We constructed a spatial containment query, illustrated in Figure 4.6 for PI use case, to retrieve all objects within a region, where the containment region covers a large area in the space. A similar containment query is also constructed for OSM dataset in which spatial objects within a large query region are retrieved. For aggregation query, we compute the average area and perimeter of polygons of different categories, with 100 distinct group labels.

4.5.2 Performance of Hadoop-GIS

Hadoop-GIS v.s. Parallel SDBMS

For the purpose of comparison, we run the benchmark queries on both Hadoop-GIS and DBMS-X on the PI dataset. The data is partitioned based on tile UIDs – boundary objects are ignored in the testing as handling boundary objects in SDBMS is not supported directly. Figures 4.9 and Figure 4.10 show the performance results. The horizontal axis represents the number of parallel processing units (PPU), and the vertical axis represents query execution time. For the parallel SDBMS, the number of PPUs corresponds to the number of database partitions. For Hadoop-GIS, the number of PPUs corresponds to the number of mapper and reducer tasks.

Figure 4.9 shows the benchmark results for the spatial join query. Both systems exhibit good scalability, but overall Hadoop-GIS performs much better compared to DBMS-X, which has already been well tuned by the vendor. Across different number of PPUs, Hadoop-GIS is more than a factor of two faster than DBMS-X. Given that DBMS can intelligently place the data in storage and can reduce IO overhead by using in-

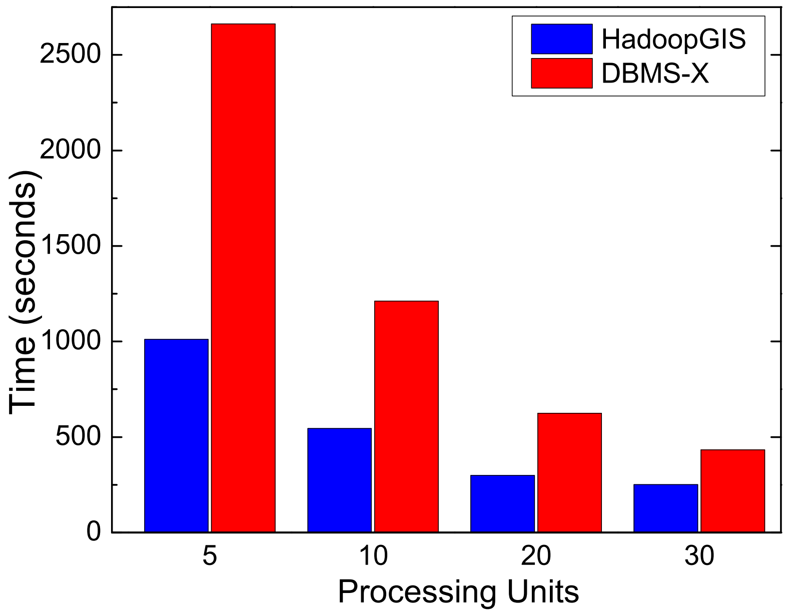
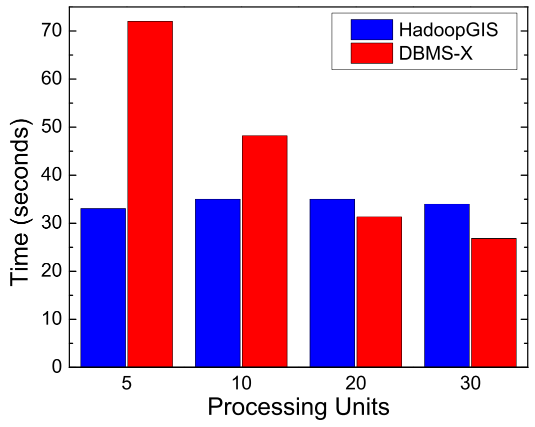


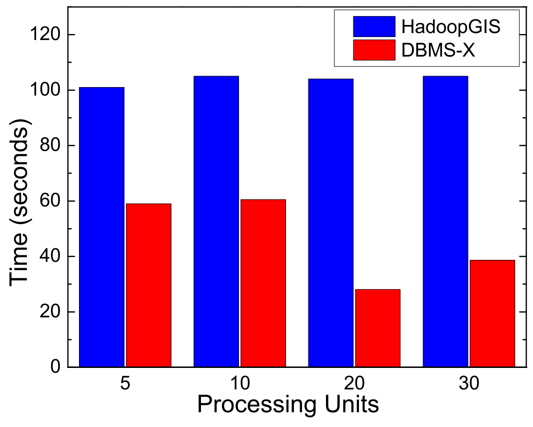
Figure 4.9: Spatial join query performance

dex based record fetching, its expected to have performed better on IO heavy tasks. However, spatial join involves expensive geometric computation, and the query plan generated database is suboptimal for such tasks. Another reason for the performance of DBMS-X is because of its limited capability on handling computational skew, even though the built-in partitioning function generates a reasonably balanced data distribution. Hadoop has an on-demand task scheduling mechanism which can help alleviate such computational skew.

For containment queries, shown in Figure 4.10(a), Hadoop-GIS outperforms DBMS-X on a smaller scale and has a flat performance across different number of parallel processing unit. However, DBMS-X exhibits better scalability when scaled out with larger number of partitions. Recall that, in Hadoop-GIS, a containment query is implemented as a Map only MapReduce job, and the query itself is less computationally intensive compared to the join query. Therefore, the time is actually being spent on reading in a file split, parsing the objects, and checking if the object is contained in the containing region. On the other hand, DBMS-X can take advantage of a spatial index and can quickly filter out irrelevant records. Therefore it is not surprising that DBMS-X has



(a) Containment query



(b) Aggregation query

Figure 4.10: Performance of containment and aggregation queries

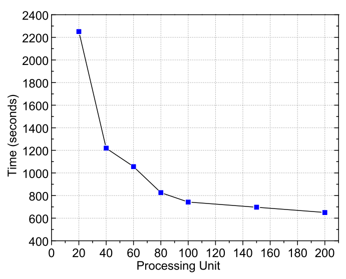
slightly better performance for containment queries.

Figure 4.10(b) demonstrates that DBMS-X performs better than HadoopGIS on aggregation task. One obvious reason for this is that Hadoop-GIS has the record parsing overhead. Both systems have similar query plans - whole table scan followed by aggregation operation, which have similar I/O overhead. In Hadoop-GIS, however, the records need to be parsed in real-time, whereas in DBMS-X records are pre-parsed and stored in binary format.

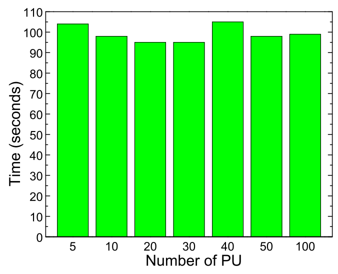
In a summary, Hadoop-GIS performs better in compute-intensive analytical tasks and exhibits nice scalability - a highly desirable feature for data warehousing applications. Moreover, it needs much less tuning effort compared to the database approach. However, MapReduce based approach may not be the best choice if the query task is small, e.g., queries to retrieve a small number of objects.

Performance on OpenStreetMap

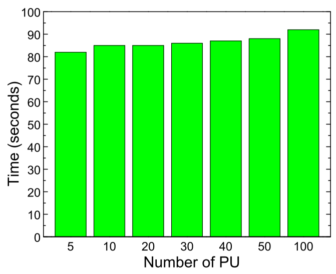
We also tested performance and scalability of Hadoop-GIS on a geospatial dataset – OSM. To test the systems scalability, we run the same types of queries as on the PI dataset. For the join query, the query returns objects which have been changed in the newer version of the dataset. Therefore, the join predicate becomes $ST\_EQUAL =$



(a) Spatial join query



(b) Containment query



(c) Aggregation query

Figure 4.11: Performance of HadoopGIS on OSM dataset

FALSE. Figure 4.11 shows the performance results for Hadoop-GIS on OSM the dataset. From the Figure 4.11(a), we can see that Hadoop-GIS exhibits very nice scalability on the join task. When the number of available processing units is increased to 40 from 20, the query time nearly reduced into half, which is almost a linear speed-up. With increase of the number of PPUs, there is a continuous drop on the query time. However, we can see a long tail in which there is no obvious speed, even the number of processing unit increases. This is mainly due to the heavy data skew in the OSM dataset. Recall that we induced a grid partition on the dataset, it is unavoidable that certain grids have significantly larger number of objects. While the on-demand scheduling mechanism of Hadoop may help to alleviate such skew, it can not provide a complete remedy. There some processing units become *straggelers* and reduces effective scalability of the system. How to effectively remedy such a skew problem is essential for query performance, and we will discuss several solutions to this problem in chapter 5.

For containment query, we randomly select a large region which contains several city boundaries, and query spatial objects contained in that region. Figure 4.11(b) illustrates the containment query performance on OSM dataset, with running time less than 100 seconds. The variance of query performance across different number of PPUs is flat, due to the nature of the query that containment queries are I/O intensive, and the query performance is bound by the number file blocks scanned by the system. The aggregation query performance on the OSM dataset is shown in 4.11(c). Similar to the containment query, even with the increased number of processing power (PPUs), the query performance is not increased at all. This is due to the nature of the aggregation query that the query is translated into a whole table scan operator followed by an aggregation operator. Therefore, the query has constant cost which is not affected by the number of processing units.

4.5.3 Scalability of Hadoop-GIS

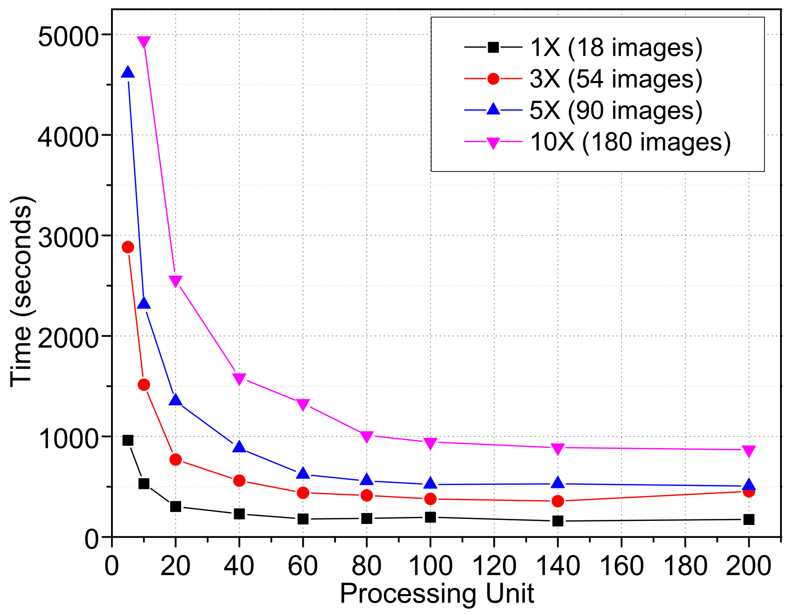
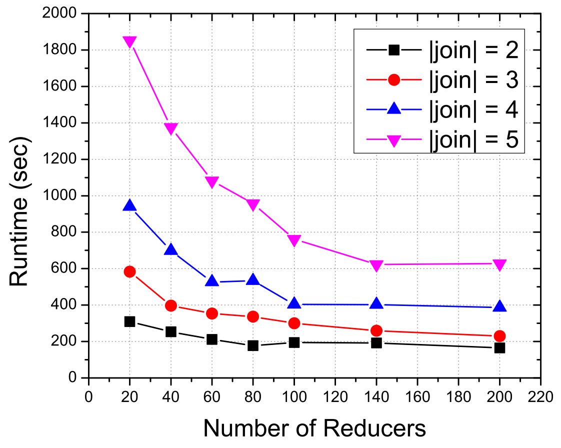


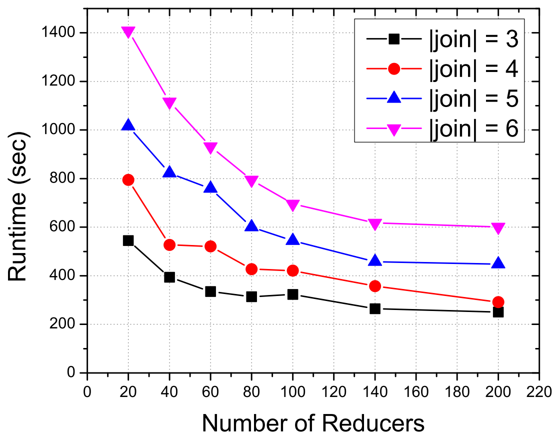
Figure 4.12: Scalability test with spatial join query on PI dataset

Figure 4.12 shows the scalability of the system on large dataset. Data sizes used include: 1X, 3X, 5X, and 10X data sets, with varying number of PPUs. We can see a continuous drop of time when the number of reducers increases. It achieves a near linear speed-up, e.g., time is reduced to by half when the number of reducers is increased from 20 to 40. The average querying time per image is about 9 seconds for the 1X dataset with all cores, comparing with 22 minutes 12 seconds in a single partition PostGIS. The system has a very good scale up feature. As the figure shows, query processing time increases linearly with dataset size. The time for processing the join query on 10X dataset is roughly 10 times of the time for processing a 1X dataset.

Multiway Join Query



(a) Star shaped join query



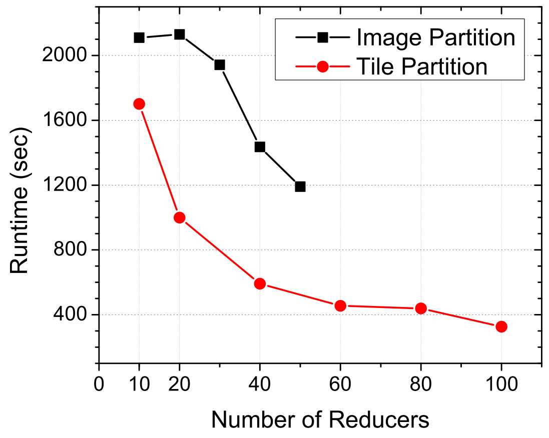
(b) Clique shaped join query

Figure 4.13: Multiway spatial join query performance

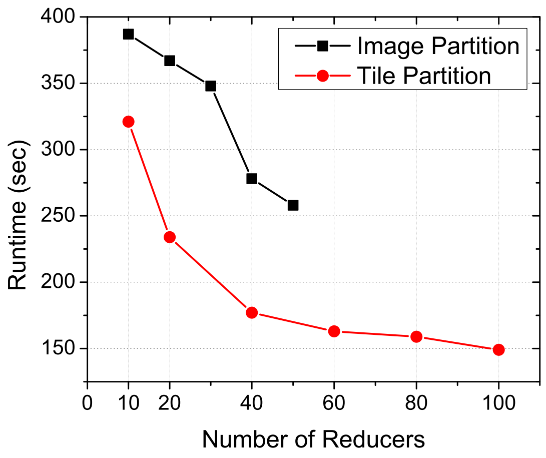
Figure 4.13 shows the query performance for star-shaped multiway spatial join with different join cardinality. The horizontal axis represents the number of reducers and the vertical axis represents query runtime. As the figure shows, the system exhibits good scalability. For both figures, it is noticeable that the query runtime drops linearly as the number of reducers increases. This effect is more pronounced in the region where the number of reducers ranges between 20 and 80. Interestingly, when the join cardinality increases, the linear relationship between runtime and the number of processing units becomes more apparent and it saturates as the number of reducers approaches to the maximal number of available cluster cores.

Nearest Neighbor Query

To test nearest neighbor query performance, we run the example query "*return the distance to the nearest blood vessel from each cell*" for each image in the TCGA dataset. This query can be parallelized at image level or at tile level. We report results for both



(a) R*-Tree



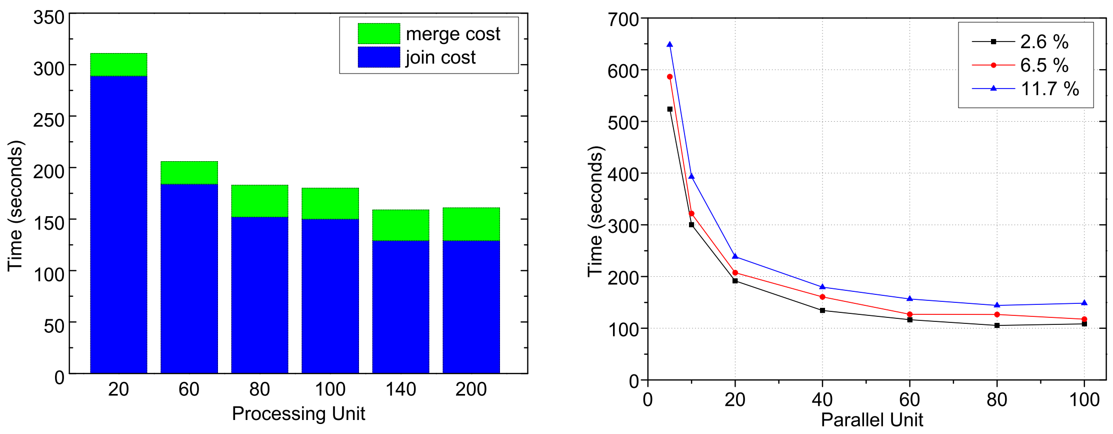
(b) Voronoi

Figure 4.14: Nearest neighbor query performance

levels of parallelism in Figure 4.14. Since the number of images used for this test is 50, more than 50 reducers would not help to increase the system performance. As it can be seen from both figures, it is clear that finer level of partition granularity offers higher level of parallelism which translates into better performance. The system also exhibits good scalability for tile level partitioning. In both figures, the execution time is reduced roughly by half when the number of processing nodes is doubled (from 10 to 20 and from 20 to 40). The experiments also show that Voronoi based nearest neighbor search is much faster than R*-Tree based approach.

4.5.4 Boundary Handling Overhead

We run the join query on PI dataset to measure the overhead in boundary handling step. Figure 4.15(a) shows the performance of two way spatial join query with boundary handling. The blue bars represent the cost of processing the query, and the green bars represents the cost of amending the results. As the figure shows, the cost of boundary handling is very small. Boundary handling overhead depends on two factors – the number of boundary objects and the size of the query output. If the number of objects on the tile boundary accounts for a considerable fraction of the dataset, the

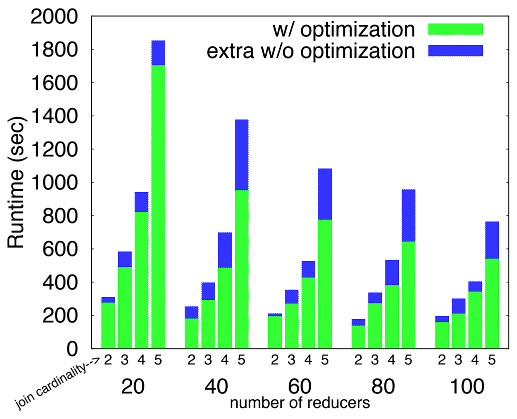


(a) Boundary-aware spatial join processing (b) Performance variation

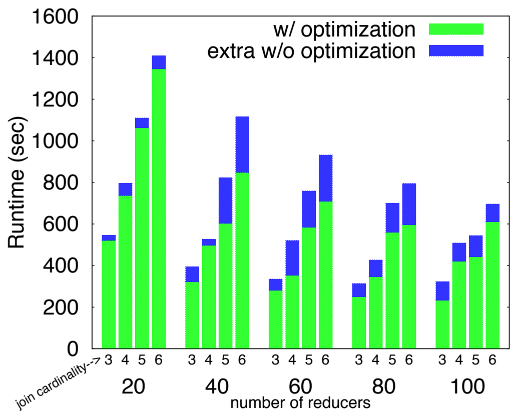
Figure 4.15: Boundary handling overhead in Hadoop-GIS

overhead should not dominate the query processing time. Therefore, we test the join query on the same dataset in which the number of boundary objects is deliberately increased. Figure 4.15(b) shows the spatial join query performance with different fraction of boundary objects. The lines represent query performance with varying percentage of boundary objects as shown in the legend. It is clear from the figure that, the boundary handling overheard increases linearly with the percentage of boundary objects.

While in Figure 4.15(b) we show that the percentage of boundary objects can be as high as 11.7%, in reality, the fraction of boundary objects are much smaller. We did an experiment with the OSM dataset in which we partitioned the dataset into 1 million tiles ($10^3 \times 10^3$ grid), and counted the number of boundary objects. Even at such a fine granular level of partitioning, the number of objects lying on the tile boundary is less than 2%.



(a) Query optimization for a star shaped spatial join query



(b) Query optimization for a clique shaped spatial join query

Figure 4.16: Spatial join query optimization through cost-based task partition

4.5.5 Effects of Query Optimization Approaches

Skew Mitigation

As Figure 4.8 shows, partition-optimized system not only performs better, it suffers less from the "straggler" problem. Each partition (indicated by the purple bars) finishes roughly at the same time. Figure 4.16(a) and Figure 4.16(b) show a comparison between optimized partition and original partition for different query types. Here, the first horizontal axis represents join cardinality, and the second horizontal axis represents number of reducers for that run. Generally, such optimization considerably reduces job completion time and increases query performance. Interestingly, when the job partitioned into smaller number of partitions, 20 for example, the performance improvements are not as significant as larger number of partitions. It seems the skew effects are less severe in smaller level of partitions, or the original hash partition is happen to be good enough for that particular number.

Partition Filtering with Multilevel Spatial Index

For containment and aggregation queries, the I/O is the main bottleneck. Therefore, number of HDFS blocks scanned by the system effectively determines the cost of the query. Hadoop-GIS uses a multilevel spatial index (section 3.5) structure to reduce the number of file blocks scanned by the system.

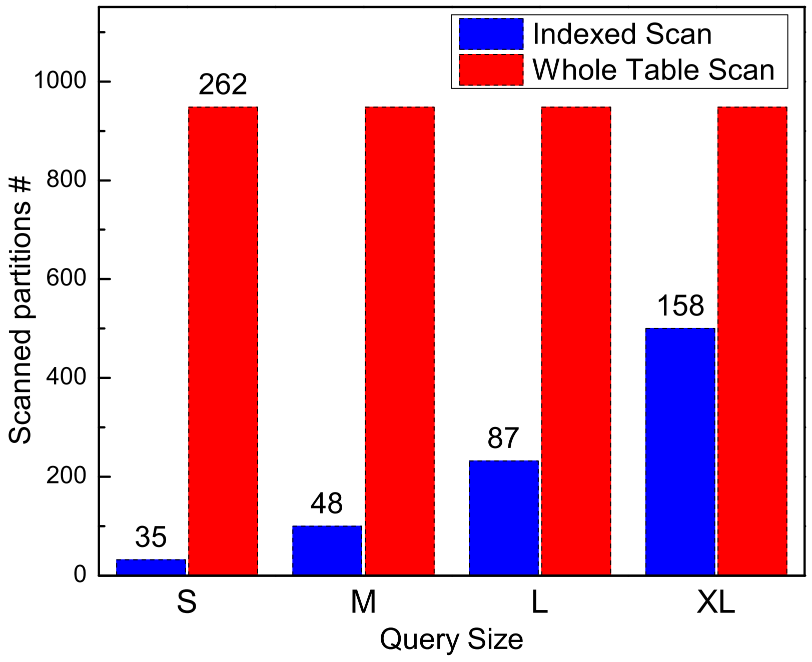


Figure 4.17: Performance of spatial containment query with multi-level indexing

Figure 4.17 shows the performance of a containment query that aided by such multilevel index. Here, the horizontal axis indicates containment query size; the vertical axis indicates number of scanned partitions to process a query; and column labels denote the query processing time. As the figure shows, Hadoop-GIS can effectively reduce the number of partitions scanned by using the multi-level spatial index. This is very helpful for small containment queries, such as point queries, as the number of records involved with the query is significantly small. However, for larger containment queries, performance gain from using the multi-level index is not as significant as for small containment queries.

4.6 Summary

The Hadoop-GIS framework provides a basic blueprint for implementing a MapReduce based spatial query system. Through the development and deployment of MapReduce based query processing, we are able to provide scalable query support with cost-effective architecture. Hadoop-GIS is based on a decoupled architecture in which the spatial query engine — RESQUE — provides essential spatial query processing capability, and MapReduce engine enables running partition based spatial queries on a massive scale.

In Hadoop-GIS, we address several fundamental problems for MapReduce based spatial query — spatial data partition, boundary object handling, index supported query processing, and query optimization. Experiments show that Hadoop-GIS is significantly faster than the parallel database approach for compute-intensive complex queries, and on par with parallel database approach for I/O intensive queries. Hadoop-GIS uses a *replicate-and-filter* strategy for handling boundary objects, and experimental results indicate that such approach incurs very small query processing overhead. Skew mitigation plays an important role in improving the query performance, and Hadoop-GIS employs a cost based tasks partition approach to mitigate the skew.

Chapter 5

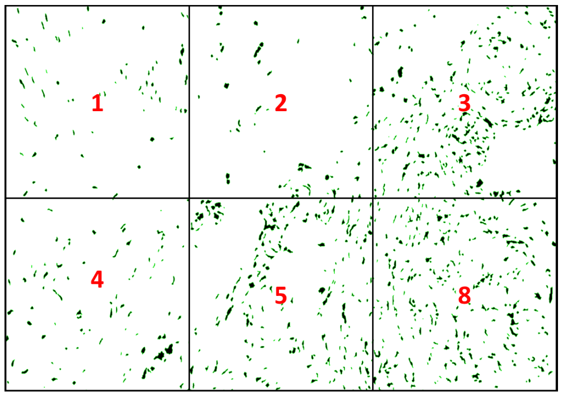
Effective Spatial Data Partitioning for Scalable Query Processing

5.1 Introduction and Related Approaches

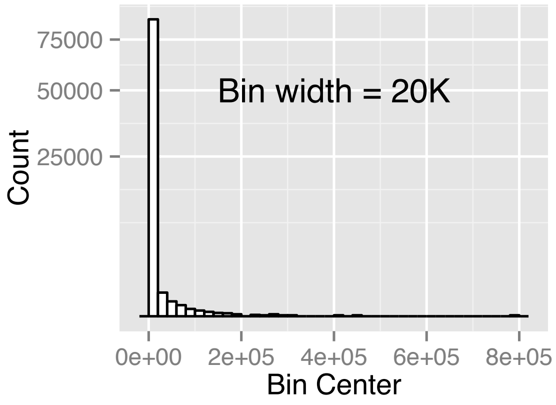
Data partitioning is a powerful mechanism for improving efficiency of data management systems, and it is a standard feature in modern database systems. In fact, state-of-the-art systems employ a shared-nothing architecture [108], and both MapReduce and parallel DBMS are examples of such architecture. Aside from the fact that data partitioning improves the overall manageability of large datasets, it improves query performance in two ways. First, partitioning the data into smaller units enables processing of a query in parallel, and henceforth the improved throughput. Second, with a proper partitioning schema, I/O can be significantly reduced by only scanning a few partitions that contain relevant data to answer the query. Therefore, a partitioning approach – that evenly distributes the data across nodes and facilitates parallel processing – is essential for achieving fast query response and optimal system performance.

5.1.1 Challenges in Spatial Partitioning

Spatial data partitioning, however, is particularly challenging due to several pitfalls that are endemic to spatial data and query processing.



(a) Fixed grid partition



(b) Spatial data skew

Figure 5.1: An example of fixed grid partition and spatial data-skew

Spatial Data Skew

Data skew is very common and severe in spatial applications. For example, in microscopic pathology imaging scenario, tumorous tissues contain far more spatial objects (segmented cells), whereas cells are more evenly distributed in healthy tissues. In geospatial applications (e.g., OpenStreetMap) some countries and regions have more detailed mapping information due to the enthusiastic data contributors. Figure 5.1 shows a simple fixed grid partitioning of a pathology image (a), and binned histogram of partition size from a fixed grid partitioning of OpenStreetMap data (b). Needless to say, data skew is detrimental to the query performance [105] and curtails system scalability [93]. Therefore, to achieve the best query performance, a spatial partition approach should try to avoid a skewed partitioning whenever it is possible.

Boundary Objects

Spatial partitioning approaches generate boundary objects that cross multiple partitions, thus violating the partition independence. As spatial objects have complex boundary and extent, imposing a rectangular region based partitioning on sufficiently large dataset would most certainly produce objects that cross multiple partition boundary. Spatial query processing algorithms get around the boundary problem by using a *replicate-and-filter* approach [93, 126] in which boundary objects are replicated to multiple spatial partitions, and side effects of such replication is remedied by filtering the duplicates at the end of the query processing phase. This process adds extra query processing overhead which increases along with the volume of boundary objects. Therefore, a good spatial partitioning approach should aim to minimize the number of boundary objects.

Performance

Spatial partitioning algorithms are expensive to compute compared to the conventional one dimensional table partitioning algorithms, such as hash and range partitioning, that can be done quickly on the fly. The multidimensional nature of spatial data entails that most spatial operators are of linear time complexity. The high computational complexity combined with massive amounts of data require an efficient approach for spatial partitioning to achieve overall fast query response. This is in particularly important for spatial-temporal data where new spatial data has to be partitioned and processed in a timely fashion.

To the best of our knowledge, no spatial database system provides a graceful approach to spatial partitioning. Previously, Paradise [93] – a parallel spatial database system – used a regular fixed grid partitioning for parallel join processing. Fixed grid partitioning is the basis of many spatial algorithms and it is easy to compute. However, as mentioned in the original work, fixed grid approach suffers from both *data skew* problem and *boundary object* problem.

5.1.2 Related Approaches

Data partition problem is discussed extensively in the context of database systems in the last few decades [48, 102]. FixedGrid spatial partition and its variations are used for spatial join processing in [93, 126]. Le et al. [77] studied the problem finding optimal splitters for large interval data. More recently, MapReduce based systems emerged as an effective solution to Spatial Big Data challenges [57]. HadoopGIS [34] is a high performance spatial data warehousing system that is based on a general spatial query processing framework. The system uses SQL as the query language and integrated into Hive[116]. SpatialHadoop [56] is an extension of Hadoop for spatial query processing and it also extends Pig [55] at the query language layer. Ray et al. [96] proposed a spatial data analysis infrastructure that uses a combination of cloud environment and relational database systems. Authors also briefly discussed their hybrid approach that uses Hilbert Curve and space partitioning for spatial join processing.

Spatial histogram construction is extensively studied in database settings, and it is widely used for approximate query processing. The main goal of spatial histogram construction is to partition the multi-dimensional data into buckets (most often a bucket represents a rectangular region), where data within buckets is uniformly distributed. In that sense, spatial histogram generation is relevant to spatial partitioning, but not the same. In [88], authors have showed that computing the non-overlapping rectangular partitioning with near-uniform data distribution within buckets is NP-hard. One of the pioneering works is [87], in which authors proposed to extend the concept of equi-depth histogram to multidimensional data. An in-memory data structure *hTree* is designed for storing the histograms. It constructs non-overlapping partitioning of multidimensional space based on object frequencies. However location of objects are not considered for histogram construction, which may result in skewed histograms.

MinSkew histogram [29] is proposed to remedy some of the disadvantages of hTree.

GenHist [61] is a recent approach which can identify high density regions for real valued attributes. However, in GenHist bucket rectangles may overlap, and the buckets can be contained in other buckets. It uses a fixed-size grid as the basis of histogram construction. More recently, an approach called *STHist* [97] is proposed to generate density aware histograms. In the basic STHist algorithm, decision about whether the region is dense is made by applying a sliding window over all dimensions, by approximating the frequency distribution by a marginal distribution. The dense regions called Hot-Spots, and the constructed histogram is represented as an unbalanced R- tree. In the advanced variant called STForest, the algorithm first computes coarse partitions according to the object skew, and then applies a sliding window algorithm to them. The idea behind this is that, if the region is already uniformly distributed, further partitioning is unnecessary. Moreover, the coarse regions are merged together if the skew of merged buckets decreases. While STHist is better than proposed methods, STHist has a time complexity of $\mathcal{O}(n^2)$ for 2-dimensional and $\mathcal{O}(n^3)$ for 3-dimensional data.

A convenient approach to obtain a spatial histogram is to generate it using a spatial index structure like R-Tree [64], R*-Tree [41], R+−Tree [103] etc. *RK-Hist* [54] is an example of such approach which is based on R-tree bulk-loading procedure. The data is presorted according the Hilbert space-filling-curve. After the leaf nodes are generated, a histogram can be generated by packing nodes according to the sorting order in equi-sized histogram buckets. However, this may not necessarily generate a good partitioning. Specifically, for approximately uniformly distributed data equi-sized partitioning wastes buckets for regions with a high object density and produces high overlap between buckets. Therefore, the authors proposed a greedy algorithm utilizing a sliding window of pages along the Hilbert order. The algorithm is parametrized with a number of buckets that should be considered for a split. A bucket-split is applied if it leads to an improvement according to the proposed cost function. More recently, a new approach *R-V* [28] is proposed to overcome skewed-data distribution problem.

5.2 Classification of Spatial Partition Algorithms

In this dissertation we study six spatial partition algorithms that are representative of different classes of approaches. Before we delve into the technical details, it would be more interesting to give a high-level view to help readers understand how these algorithms are related, and what their major differences are. Here, we attempt to categorize those algorithms along three dimensions, and Table 5.1 summarizes such classification. The algorithmic details will be discussed next, in section 5.3.

Dimension	**Category**	BSP	FG	SLC	BOS	STR	HC
Partition Boundary	overlapping					√	√
	non-overlapping	√	√	√	√		
Search Strategy	top-down	√	NA				
	bottom-up		NA	√	√	√	√
Split Criterion	space-oriented	√	√				
	data-oriented			√	√	√	√

Table 5.1: A general classification of spatial partition algorithms

5.2.1 Partition Boundary

We start with whether the spatial partition boundaries overlap with each other.

Non-overlapping Partitions

Algorithms in this category generate spatial partitions of which boundaries do not overlap with each other. Non-overlapping partitioning is ideal for most query processing tasks as it does not incur any extra storage or computation overhead other than replicated boundary objects. Due to the same reason, in this paper we mostly focus on this class of algorithms which includes FG, BSP, SLC, and BOS.

Overlapping Partitions

Algorithms in this category relax the non-overlapping boundary condition, and allow generated partitions to overlap with each other. Most spatial index construction algorithms [100] are based on the similar idea, and the packing algorithms such as STR [79] and Hilbert Curve [73] belong to this class. Since the partitions may overlap with each other, some fraction of objects would be present in multiple partitions. Those multi-partition objects would be replicated and assigned to each of the overlapping partitions. As a result, in this class of approaches the replication factor α can be high which consequently increases the deduplication cost factor β. However, if a *good* partitioning can be *quickly* obtained by allowing the partitions to overlap, then the extra cost can be compensated by the improved query performance.

5.2.2 Search Strategy

The second dimension we consider is the search strategy which focuses on how the partitions are generated.

Top-down

This class of algorithms generate partitions in top-down manner. Specifically, given a dataset and an expected partition payload b (number of objects assigned to that partition), a top-down approach recursively splits the dataset into k sub-partitions, and examines if any sub-partitions has more than b objects. If a sub-partition has more than b objects, then it will be further partitioned, until the payload requirement is met. Most spatial indexes are constructed using similar procedure. While the value of the parameter k can be chosen arbitrarily, some specific values, such as $k = 2$ (BSP) and $k = 4$ (Quad-Tree), are used more frequently in practice. Depending on the split criterion, this class of algorithms can be implemented as either data-oriented or space-oriented, and we describe these categories in the next subsection.

Bottom-up

Rather than generating partitions in a recursive manner, this class of algorithms attempt to construct the final partitions as early as possible. Such approach bears some resemblance to the spatial packing algorithms. The general idea is to use proximity information of spatial objects to group them into partitions. Since there is no spatial proximity preserving total ordering for multi-dimensional objects, Space Filling Curves are used to generate approximate one dimensional ordering. Then, objects are packed into partitions by grouping them based on such ordering.

5.2.3 Partition Criterion

Finally, splitting an oversized partition into smaller ones is a core subroutine in spatial partitioning, and algorithms may have different criterion for this task. For example, consider a simple case where a partition with payload w need to be partitioned into two sub-partitions. There will be two strategies: space oriented, and data oriented.

Space Oriented

This class of algorithms generate sub-partitions by *spatially decomposing* the current partition boundary into two equal sub-spaces. As the split decision is made solely based on the space, this approach suffers from data skew. If the data distribution is uniform, we would expect to get two sub-partitions where each of them has a payload of roughly $\frac{w}{2}$. However, if the data distribution is skewed, it is possible that one of the subpartitions still contains large fraction of objects in the original partitions, while the other contains only few objects.

Data Oriented

This class of algorithms generate sub-partitions by finding a *cut* such that each resulting sub-partitions contains roughly equal amounts of data ($\frac{w}{2}$). The cut position is derived based on the distribution of data objects rather than splitting the space. However, finding an optimal cut which generates an even partitioning requires significant computational effort. Furthermore, the algorithms also need to be judicious about the split position so that the number of boundary objects induced by such split is not very large.

5.3 Spatial Partition Algorithms

5.3.1 Preliminaries

We study the following partition problem: given a set of d-dimensional spatial objects $R = \{r_1, r_2, ..r_i..r_n\}$ ($|R| = N$), a partition algorithm partitions R into k partitions $P = \{p_1, p_2, ..p_j..p_k\}$, where each partition is size bounded $|p_j| \leq b$, and the number of partitions k is minimized. Without loss of generality, we consider the case where $d = 2$ and a spatial object is approximated by its MBR (Minimum Bounding Rectangle), and each rectangle is represented by $r_i = (x_i, y_i, u_i, w_i)$.

Partitioning of one dimensional data ($d = 1$) has been extensively studied in the past, and it is shown that the optimal solution can be obtained in polynomial time [72]. However, for higher dimensions, even for a simple case $d = 2$, the problem becomes intractable. Previously, a simpler version of the problem, known as rectangle tiling, was studied. The main objective of rectangle tiling is to partition a matrix of integers into tiles, and it was proven to be NP-Hard [58, 74] for cases $d \geq 2$.

5.3.2 Methods and Details

Next we discuss each of the six algorithms in detail. While some of these algorithms have been partially studied in the earlier research, others are rarely utilized for parallel spatial query processing.

Fixed Grid Partitioning (FG)

Fixed grid partitioning is a simple space-oriented, non-overlapping partitioning approach in which the spatial universe is partitioned into k equal sized grids. A major assumption behind this approach is that data follows a uniform distribution. Therefore, if the data distribution is close to a uniform distribution, FixedGrid is expected to generate a reasonably good partitioning. While the partition process is very simple, for the sake of clarity, details of this approach are described in Algorithm 6.

Algorithm 6: Fixed grid partition (FG)

```
Input: a set of spatial objects $R$
Input: partition payload $b$
$m = \lceil \sqrt{|R|/b} \rceil$;
$U$ = spatialUniverse($R$);
$G$ = split $U$ into $m$ by $m$ grid;
for $r_i$ in $R$ do
    $g$ = grids intersects with $r_i$;
    assign $r_i$ to each grid in $g$;
end
```

Binary Split Partitioning (BSP)

Binary split partitioning is a top-down approach that generates partitions by recursively dividing a given spatial partition into two non-overlapping subpartitions until the payload requirement is met. Given a expected partition payload b, BSP recursively creates subpartitions if the number of objects inside a partition exceeds the specified payload (Algorithm 7). The split point is chosen to be the median of object centroids in that

Algorithm 7: Binary split partition (BSP)

```
Input: a set of spatial objects $R$
Input: partition payload $b$
$U$ = spatialUniverse($R$);
while $r$ in $R$ do
    $n$ = node($U$);
    addObject($n$, $r$);
end
function addObject(n,r):
    if $n$ is leafNode then
        $n$.objectList.add($r$);
    end
    if size(n.objectList) $\leq c$ then
        compute $median\_x$ and $median\_y$ split ;
        $split$ = argmax(Product of children areas);
        $child1$, $child2$ = children($n$, $split$);
        if $child1$ intersects with $r$ then
            addObject($child1$, $r$);
        end
        if $child2$ intersects with $r$ then
            addObject($child2$, $r$);
        end
    end
```

partition. The direction of the split (horizontal or vertical) is dependent on the relative ratio of areas of subpartitions. The split direction is chosen so that the relative area difference between children nodes are minimized based on a probabilistic expectation.

Strip Partitioning (SLC)

Strip partitioning is a non-overlapping, data oriented partitioning approach that has some resemblance to *slicing* a cake. In this approach, rather than defining a fixed space, we slice off a rectangular region from the spatial universe where each region contains approximately b objects. Then similar process is continued on the rest of the data and repeated until we generate all the partitions. Details of this approach are described in Algorithm 8.

Algorithm 8: Strip partition (SLC)

```
Input: a set of spatial objects $R$
Input: partition payload $b$
Input: partition dimension $d$
/* sort objects by mbr center in dimension $d$ */
sort ($R$,$d$);
$U$ = spatialUniverse($R$);
while $R$ is not empty do
    $s$ = cutStrip($U$, $R$, $b$);
    for $r_i$ in $R$ do
        if not $r_i$ intersects with $s$ then
            break;
        end
        assign $r_i$ to partition $s$ ;
        if $s$ contains $r_i$ then
            remove $r_i$ from $R$;
        end
    end
end
```

Boundary Optimized Strip Partitioning (BOS)

Algorithms described above do not explicitly consider the boundary object problem, although the partition payload is guarenteed to be balanced. As a result, we may still get a partitioning that are balanced but has a higher deduplication cost. BOS is a boundary object aware extension of SLC that minimizes the number of boundary objects while still generating a balanced partitioning. While performing the strip based partitioning, BOS has two dimensions (d dimensions in general) to choose at each step. BOS calculates the partitioning in both dimensions, and selects the one which induces smaller number of boundary objects. Algorithm 9 describes the details of this approach.

Hilbert Curve Partitioning (HC)

Space filling curves used in many application to obtain a locality preserving approximate total ordering for multidimensional data. Commonly used space filling curves in-

Algorithm 9: Boundary optimized strip partition (BOS)

```
   Input: a set of spatial objects $R$
   Input: partition payload $b$
 1 $U$ = spatialUniverse($R$);
 2 while $R$ is not empty do
       // cost in dimension $x$
 3     $cx$ = getCost($U$, $R$, $b$, $x$);
       // cost in dimension $y$
 4     $cy$ = getCost($U$, $R$, $b$, $y$);
 5     if $cx \leq cy$ then
 6         strip partition in $x$ dimension;
 7     end
 8     else
 9         strip partition in $y$ dimension;
10     end
11 end
```

clude Z-curve, Gary-coded curve, and Hilbert curve. Among those approaches, Hilbert curve is shown [86] to have better clustering property for two dimensional objects. In our implementation, we use Hilbert curve to map the centroid of the spatial objects to obtain the curve value, and sort the dataset based on the curve value. Then, we group each consecutive b objects together to form a spatial partition, and the union of their extent is the final partition boundary. However, as the final partition boundaries may overlap with each other, we rescan the dataset to perform the replication process.

Algorithm 10: Sort-Tile-Recursive partition (STR)

```
   Input: a set of spatial objects $R$
   Input: partition payload $b$
 1 $m = \lceil \sqrt{|R|/b} \rceil$;
   // $m$ strips in dimension $x$
 2 $S$ = stripPartition ($R$, $x$);
 3 for $i \leftarrow 1$ to $m$ do
       // $m$ strips in dimension $y$
 4     $t$ = stripPartition ($S[i]$, $y$);
 5 end
```

Sort-Tile-Recursive Partition (STR)

Packing spatial objects for bulk loading spatial index can be regarded as a "mini-partition" step. Most often the leaf nodes are pre-packed in order to generate low level nodes of the index, and higher level index nodes are constructed from the leaf nodes. Similarly, we can use packing algorithms to generated spatial partitions such that we only generate the lowest level index nodes, and the node boundary serves as partition boundary. STR [79] first partitions the spatial universe into large vertical strips, then each strip is further partitioned in the horizontal direction. Algorithm 10 illustrates the partition process.

Figure 5.2 shows a simple example in which a set of 32 randomly distributed spatial objects are partitioned with different spatial partition algorithms we described above.

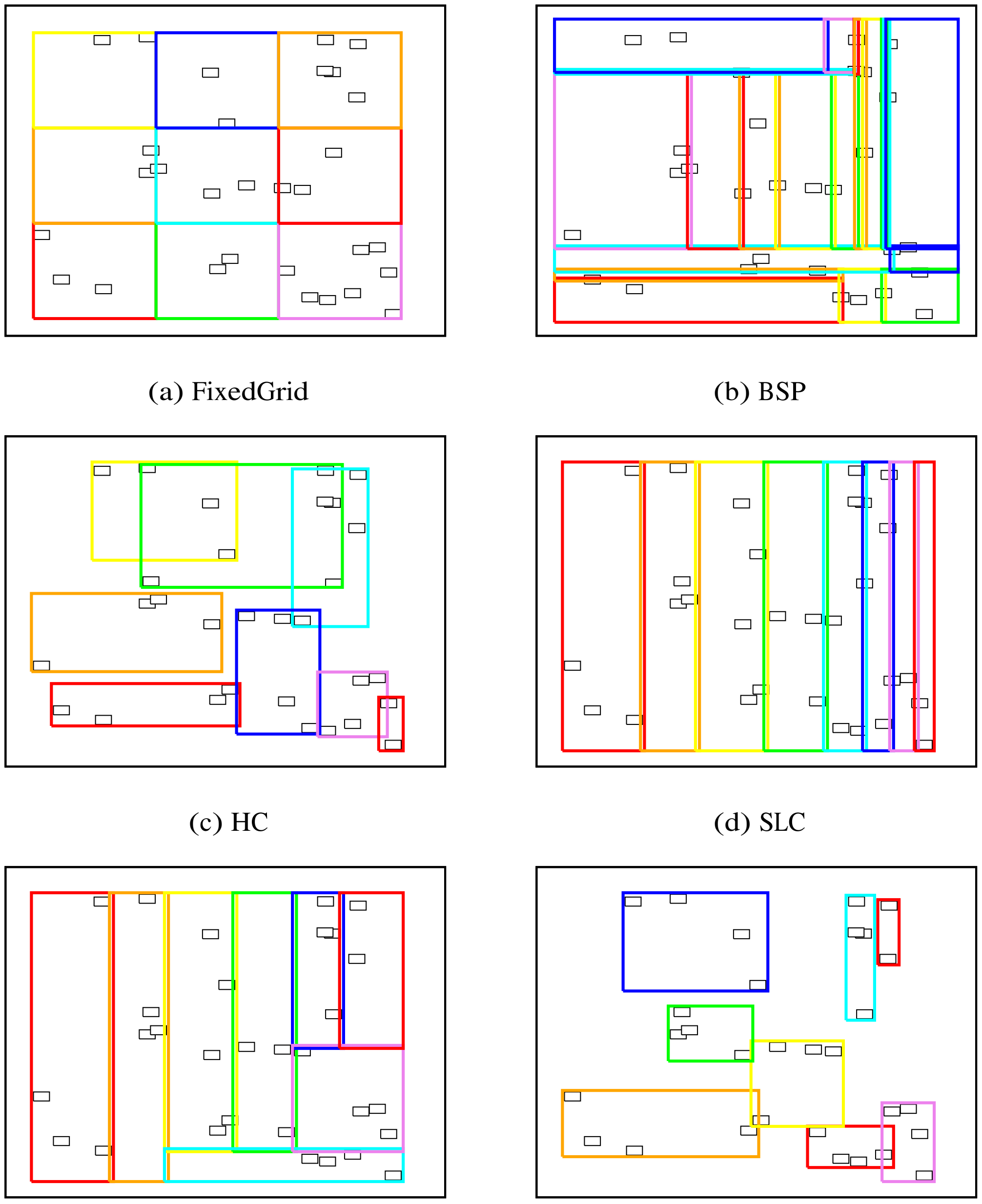
5.4 Experiments

We use Amazon EMR for our benchmarking tasks. For single thread benchmarking of partition algorithms, we use a large memory physical machine that comes with 128 GB memory. For spatial join query scalability tests, we use general purpose extra-large instance as our core and task nodes. Each extra-large instance is equipped with 15 GB memory, 4 virtual cores and 4 disks with 1680 GB storage (4×420 GB). The Amazon Machine Images (AMI) version we used for the cluster nodes is 3.0.2. Amazon S3 is used as the primary data storage for data serving.

5.4.1 Parameters and Metrics

Partition Payload

The two datasets, OSM and PI, are from different application domains, and they have different characteristics. Therefore, using the same parameter to partition both datasets


(a) FixedGrid

(b) BSP

(c) HC

(d) SLC

(e) BOS

(f) STR

Figure 5.2: Spatial partitions generated by different algorithms (the bigger rectangles in colors represent partition boundaries, and the small rectangles represent the spatial objects)

may be problematic. For example, if we partition the smaller dataset with an expected payload of c – a perfect parameter for this dataset that yields best query performance, it might be a too fine granular partitioning for the larger dataset. To be able to make the results comparable, we define the partition payload relative to the dataset size. We use a wide range of *fractions* that will be multiplied with the dataset size to obtain the actual partition payload. Table 5.2 shows those numbers.

f	0.001	0.005	0.01	0.02	0.05	0.1	0.2	0.5	1.0	5.0

Table 5.2: Partition parameters: fraction ($\times 10^{-2}$)

Boundary Object Ratio

We define a simple metric to study the relationship between partition granularity and partition quality in terms of boundary objects. For a dataset R that partitioned into k partitions $P = \{p_1, p_2, ..p_i..p_k\}$, we define the boundary object ratio as:

$$\lambda = \frac{\sum_{i=1}^{k} |p_i|}{|R|} - 1 \tag{5.1}$$

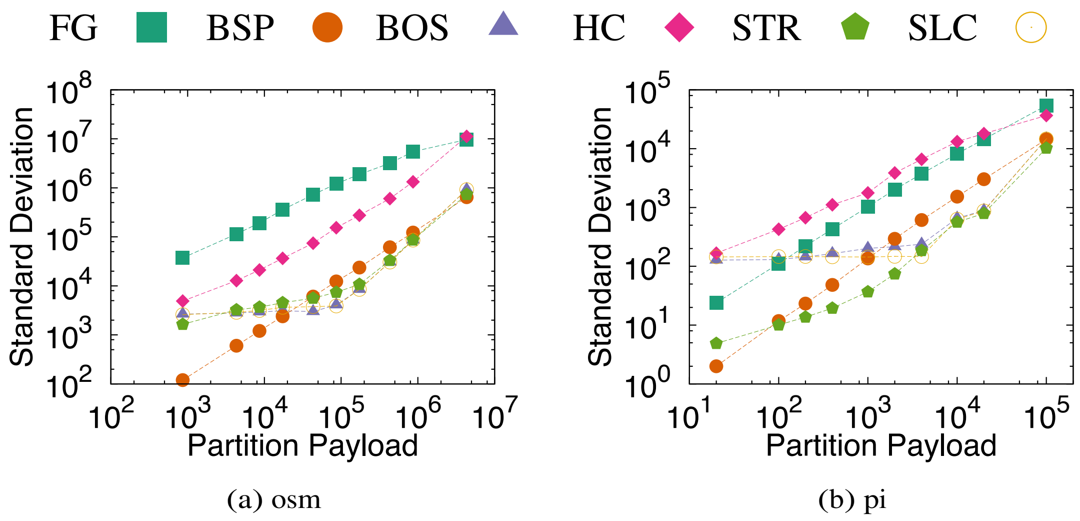
λ is a real value that lies in the interval $[0, \infty)$. If a spatial partitioning does not induce any boundary objects, the value of λ would be 0.

5.4.2 Comparison of Partition Quality

Before we evaluate the partition results with real queries, we present some statistical properties of the generated partitions which can provide us insights on the partition algorithm behavior and quality.

Partition Balance

Figure 5.3 shows standard deviation of generated partitions for different partition algorithms on two datasets. Here, we use standard deviation as a measure of partition skewness. The horizontal axis represents the expected partition payload — a granularity value that we use to partition the datasets. The vertical axis represents the standard deviation of generated partition payloads. Two conclusions can be made from the figure. First, as the partition granularity increases, the skew tends to increase very quickly for all methods. Therefore, a very coarse level spatial partitioning should be avoided for parallel processing tasks that suffer from data skew. Second, not surprisingly, FG generates significantly skewed partitions compared to other approaches.



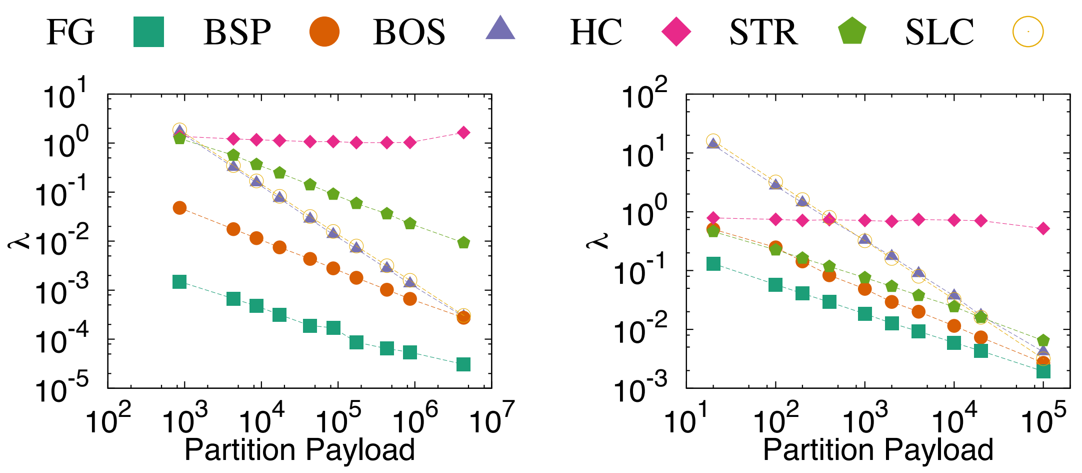
(a) osm (b) pi

Figure 5.3: Standard deviation of partition results

If we compare the same approach across two datasets with the same parameter setting, we will find that the partitions generated from the OSM dataset is more skewed than the partitions generated from the PI dataset. That means that, overall, the inherent skew in OSM is much severe than the PI dataset. Furthermore, the FG partitioning for PI dataset is considerably better than the FG partitioning of OSM dataset. Therefore, we can conclude that, for a evenly distributed dataset, the FG approach can generate a reasonably well partitioning. However, if the dataset is highly skewed, FG approach may generate a very low quality partitioning.

Adaptive approaches, such as STR, BOS and SLC, should be able to handle certain level of data skew as they can make smarter data oriented partition decision. We can see from the figures that corresponding lines for those approaches are relatively flat until the partition granularity gets large. However, as the partitions get larger, the adaptability of those algorithms also approaches their limitations.

One interesting result we did not expect to see is that partitions generated by HC approach are also as skewed as FG partitions, and for the PI dataset HC is not even as good as FG. As HC approach is a data oriented approach that traditionally used for bulk loading spatial indexes, it is surprising that the partitioning from HC has such high imbalance.



(a) osm (b) pi

Figure 5.4: Ratio of boundary objects

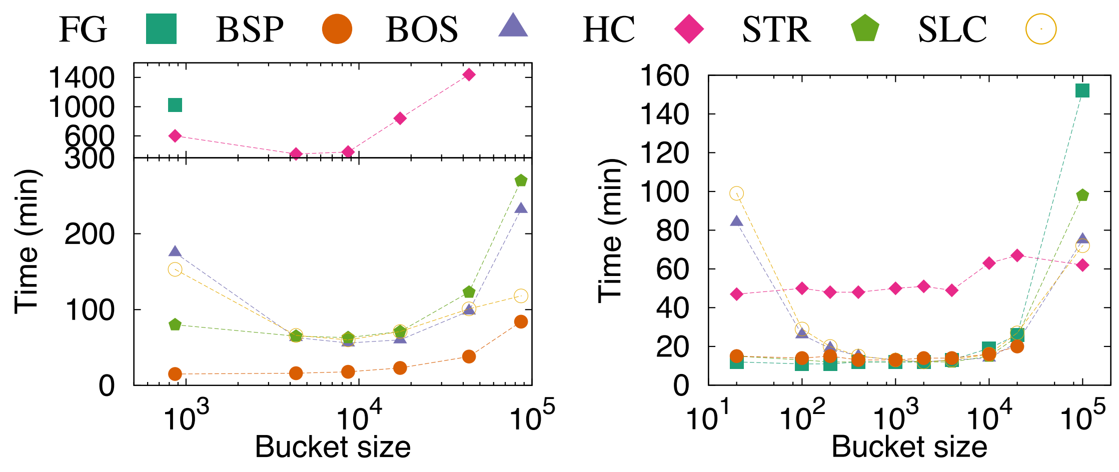
Boundary Objects

Figure 5.4 shows the ratio of boundary objects generated by different algorithms. We can see the overall trend that, for both datasets, as the partition granularity increases the ratio of boundary objects decreases. FG seems to be a good algorithm if our main objective is to have less boundary objects. However, as both figures show, a very fine granular partitioning is problematic as it significantly increases the dataset size, and in certain cases such increase can be dramatic. For example, if we look at the λ value for the first horizontal axis data point in Figure 5.4(a), for Strip partitioning (SLC) the boundary object ratio is 1.86, whereas the same data point value is 16.1 in Figure 5.4(b). Such a large increase in data size is certainly not acceptable, and we can conclude that a very fine granular partitioning is not a practical approach for large scale query processing.

Interestingly, in both figures, the lines for the *slicing* approaches, SLC and BOS, have higher slopes than other approaches. This indicates that, for those partitioning algorithms, even a slight increase in the partition payload can contribute to significantly less number of boundary objects. Therefore, in practice, those partition methods should be configured to generate a relatively larger size partitions so that the number of boundary objects are reasonably small.

5.4.3 Effects of Partitioning on Query Performance

In this section, we empirically evaluate partition algorithms on different configurations to study how a specific partitioning affects the query performance, and investigate the relationship between partition granularity and query performance. The experiments are performed on a 50 node Amazon AWS MapReduce cluster, and general purpose AWS instances are used as compute nodes and storage nodes. Each experiment is conducted three times, and average of those three runs is used to account for performance variations in cloud environment.



(a) osm (b) pi

Figure 5.5: Spatial join query performance

Figure 5.5 shows the performance of the spatial join query on two datasets. The horizontal axis represents the partition granularity, and vertical axis represents the query performance. Clearly, neither a very fine or very coarse partitioning yields the optimal query performance. For a fine granular partitioning, the main cause can be attributed to the high boundary object ratio which not only increases the I/O overhead, but also the extra computation overhead. For a very coarse granular partitioning, however, the root cause is the data skew between partitions.

Recall that, earlier in section 3.6, our cost analysis framework suggests that there is a point of optimal partition granularity that yields best query performance. The performance numbers on both datasets support such case. As the figures show, overall, query performance is close to the optimal in mid-range of horizontal axis, and performance starts to degrade as the partition granularity increases. However, if we compare different algorithms over a wide range of partition granularities, it is difficult to generalize such statement. Specifically, BSP and STR have relatively better performance on a wider range of partition granularities, and the performance starts to suffer only after the partition granularity becomes too large. This can be attributed to the properties of these algorithms that they can adaptively handle data skew and boundary objects.

Performance variance between datasets. In Figure 5.5(a), the performance of different approaches are tiered. FG and HC have similar performance, and their performance are almost orders of magnitude worse than other approaches (due to the long query runtime, we only report one data point for FG). While performance of HC is still the worst on PI dataset as shown in Figure 5.5(b), performance of FG, however, is almost optimal for most cases. Clearly, specific characteristics of a dataset are contributing to such difference. Our observation indicates that PI dataset consists of large number of *small* objects that are fairly evenly distributed across space, whereas OSM dataset consists of *variety of objects* of all sizes that are clustered around a number of hotspots. If we simply consult to the statistical properties from the previous subsection

5.4.2, we can also see that FG partitioning of PI dataset is less skewed compared to the OSM dataset. Moreover, the number of boundary objects from FG partitioning is very small on all partition granularity. Due to those reasons, on PI dataset, FG partitioning achieves a balanced partitioning for "free", and has an unfair advantage over other approaches.

5.5 Summary

A good spatial partitioning schema is essential for optimal query performance and system efficiency. In this paper, we formally introduced the spatial partition problem, and presented a comprehensive study of six different partitioning algorithms. We categorized the algorithms along three dimensions, and provide a systematic evaluation of the algorithms on two real world datasets from different domains. Our study reveals several insights on how partitioning effects query performance and what factors should be considered for effective spatial partitioning. The results provide practical guidelines for designing spatial partitioning for large scale parallel spatial query processing.

Chapter 6

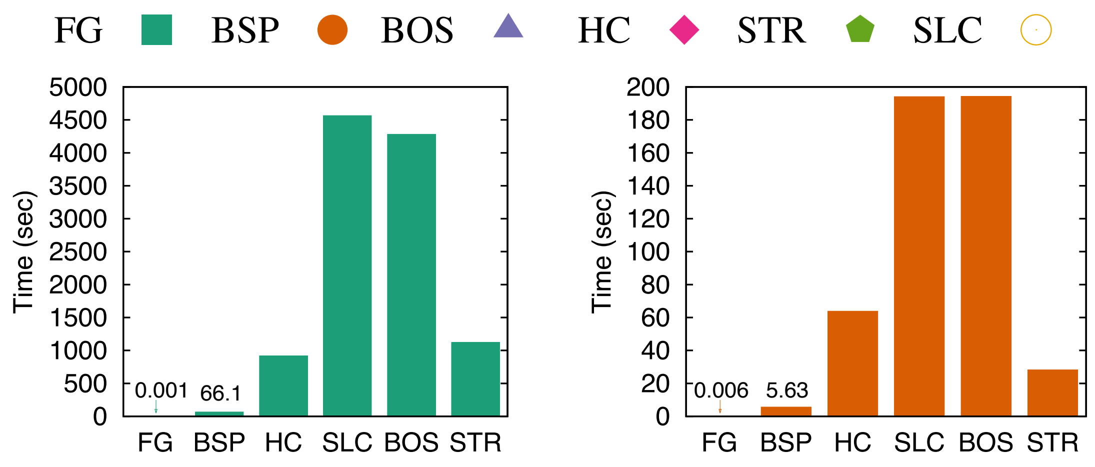
Efficiency Improvements for Spatial Data Partitioning

In this chapter we study the partition efficiency of different algorithms, characterize the spatial partition algorithms in terms of runtime performance. We provide a parallelization approach on MapReduce to improve spatial partition performance, and discuss a sampling based approach that is inexpensive to compute.

6.1 Runtime Cost of Spatial Partitioning Algorithms

Computational complexity of finding an optimal spatial partitioning is NP-hard. For spatial query processing tasks, performance of a query on an optimal partition layout may not be so different than the one on a suboptimal partitioning. Therefore, finding a reasonably well partitioning in an efficient manner has practical implications for many real world applications. In modern database systems, partitioning can be easily achieved by simply computing a hash function on some data fields. However, spatial partitioning can be expensive to compute.

Figure 6.1 shows the runtime cost of partition algorithms on two datasets. To perform a fair comparison, the time for reading the dataset from the disk, and writing the

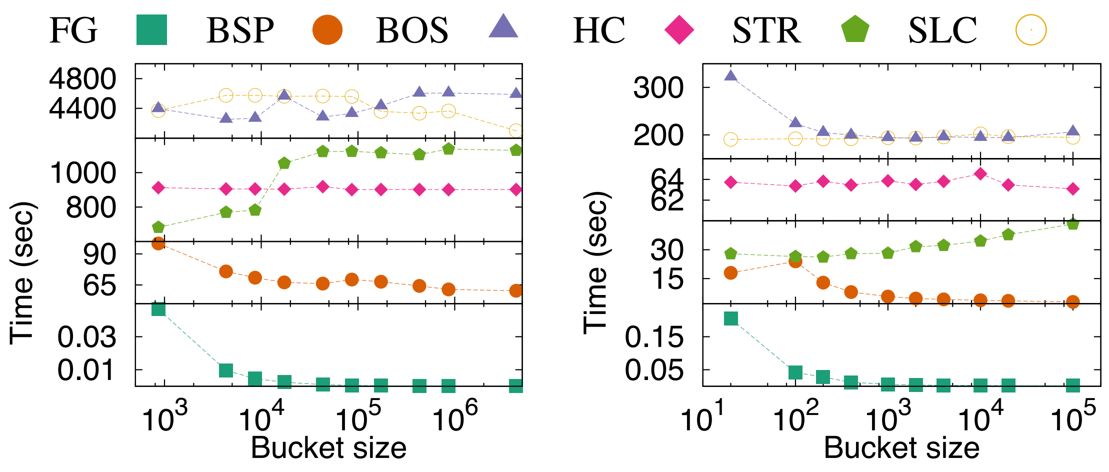


(a) osm (b) pi

Figure 6.1: Spatial partition performance

partition results to the disk is not included in the performance measurement. The performance time only includes the time for deriving the actual partition boundaries after the dataset is loaded into the main memory of a single machine. Depending on the actual runtime performance, algorithms can be roughly categorized into three categories – fast (FG, BSP), medium (HC, STR), and slow (SLC, BOS). For both datasets, FG partition has the lowest runtime cost which is only in the range of milliseconds, and BSP has the second best performance. However, other four algorithms require considerable amounts of time to generate partitions. Specifically, the space slicing approaches – SLC and BOS, require more than an hour to derive a partitioning on OSM. This is mainly due to the nature of the algorithms that SLC and BOS not only sort the dataset on one dimension, they also perform lots of boundary object examination. The main cost of HC is the Hilbert Curve calculation and sorting based on the curve value. The performance of the algorithms on different datasets are roughly similar, with the exception of HC that has a slightly slower performance on the PI dataset compared to the OSM.

Figure 6.2 shows the runtime performance of the algorithms over different partition granularity. While the performance of the algorithms do not depend too much on the partition granularity, there are noticiable differences. Intuitively, a finer granular-



(a) osm

(b) pi

Figure 6.2: Spatial partition performance variance

ity partitioning entails more cpu cycles, and therefore it is expected that algorithms run slower for small payload values. Performance numbers of FG and BSP show such tendency. However, depending on the algorithm and dataset characteristics such hypothesis may not hold true. For example, the main cost in HC partitioning comes from calculating and sorting the spatial objects based on the Hilbert curve value. Regardless of partition granularity, such cost is constant. Therefore, as the figure shows, performance of HC does change with partition granularity. Interestingly, STR has lightly degraded performance on a larger partition granularity on OSM dataset. The specific reasons are not completely clear to us, and we are planning to investigate such problem in future work.

If we compare relative performance of the algorithms across the two datasets, the lines for PI dataset is more smooth and predictable. For example, on OSM dataset, SLC and BOS have an irregular runtime performance over different partition payloads. However, those algorithms do not exhibit the same behavior in the PI dataset. Given the dataset characteristics we discussed earlier, we can conclude that dataset characteristics have implications for the algorithm performance.

In a spatial data warehousing scenario, the underlying dataset is large and relatively stable, and queries run on the same dataset many times. In such case, an approach that produces a balanced partitioning but requires significant computational resources may be acceptable as it improves the query performance in the long run. However, in some other application scenarios such as scientific data exploration and simulation, queries consume large amounts of intermediate data that are generated quickly, and most queries run only once as the data being generated. In such cases, a fast partitioning algorithm is critical for achieving overall fast query response. Here, we explore two different approaches towards improving spatial data partitioning efficiency, namely *parallel* spatial partitioning and partitioning with *sampling*.

6.2 Two Approaches for Improving Efficiency

6.2.1 Parallel Partitioning with MapReduce

Spatial partitioning is a time consuming process. As the performance numbers in previous section show, spatial partitioning on a moderately big dataset may take hours. We developed MapReduce based spatial partitioning to improve the performance of spatial query and spatial ETL process. Our parallelization approach is based on following two insights. First, spatial partition algorithms involve some kind of sorting based on a derived spatial value, and MapReduce can perform such task very efficiently. We can tweak the shuffle-and-sort phase of MapReduce to perform such task for (almost) free. Second, as different regions of a spatial dataset can be partitioned independently, rather than changing the algorithms for parallelization, we can run the partition algorithms on different regions of the dataset in parallel. Although the generated partition layout may be different from the one generated by a single thread partitioning program, it is acceptable as long as the partitioning is reasonably well.

We propose following approach for MapReduce-based parallelization of spatial partition algorithms. First, similar to Hadoop Terasort [91], we sample the dataset to gen-

Algorithm 11: MapReduce based spatial partition

```
   Input: a set of spatial objects $R$
   Input: partition payload $b$
1  $S$ = sample_for_partitioning(R);
2  function Map($k$,$v$):
3      $anchor$ = getAnchor($v$);
4      $key$ = calculateKey($anchor$,$S$);
5      emit($key$ , $v$);
   /* shuffle and sort by MapReduce                                  */
6  function Reduce($k$,$v$):
       /* partition the bucket with algorithm X                      */
7      $P$ = genPartitionX($v$);
8      emit($P$);
```

erate an anchor point list which will be utilized in the partition function of MapReduce for partition assignment. In the Map phase we calculate a spatial ordering anchor, such as geometrical center or Hilbert Curve value, and generate a key based on the sample points generated previously. Next, the MapReduce framework will partition the objects into groups based on their anchor location and sorts them on the anchor value. At this point, dataset is roughly partitioned into large spatial partitions. Later in section 7.4, we will discuss issues related to this coarse level partitioning. In the reduce phase, each reducer will work on a single large partition, and further partitions them into smaller partitions. Algorithm 11 gives a sketch of this approach.

6.2.2 Partitioning on Sampled Data

Efficiency of a partition algorithm is subject to a number of factors such as algorithm runtime complexity, dataset characteristics and size. In relational database systems, sampling is used in various tasks to avoid full dataset processing. For example, typical histogram construction algorithms work on a small fraction of sampled data, thus avoiding the expensive full dataset statistics. Such approach is shown to be practical and efficient for query processing and dataset approximation. Therefore it is natural to ask that if we can generate a spatial partition schema on a sampled dataset, which reasonably approximates a full dataset partitioning.

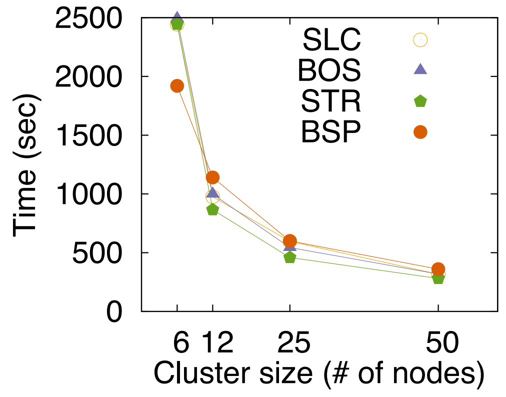
Specifically, given a sampling ratio γ, we uniformly sample the dataset to get a reduced dataset of size $\gamma|R|$, and run a partition algorithm on the this reduced dataset. Then, we map the generated partition layout onto the original dataset for final partition assignment and boundary object replication. Sampling ratio is the main control variable in the sampling based approaches. If the sampling ratio is too low, the resulting partition quality may suffer. On the other hand, if the sampling ratio is unnecessarily high, the partition efficiency may suffer while the partition quality is only marginally improved.

One problem with sampling based partitioning is that some approaches fail to generate an effective spatial partitioning on sampled dataset. For example, HC and STR generates the partition regions that may not cover the entire spatial universe (Fig. 5.2(c) and 5.2(f)) and the partition region MBRs are tight. In such case, the resulting partitions from the sampled dataset can not be used without further fix. How to adapt those approaches for spatial partitioning on sampled dataset is a problem we are planning to explore in our future work.

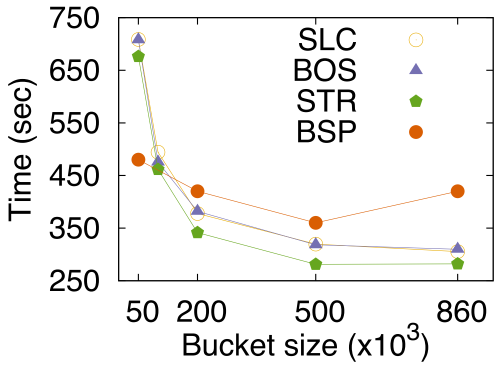
6.3 Experiments

6.3.1 MapReduce based Approach

To test efficiency and scalability of our MapReduce based parallel partitioning approach, we modified and tested selected set of four partitioning algorithms, namely BSP, SLC, BOS and STR. The rationale in such selection is that, 1) parallelization of FG and HC is straightforward, and 2) they generate suboptimal partitioning in most cases. Here, we select a set of three expensive spatial partitioning approaches (SLC, BOS, STR) to experiment. While BSP is reasonably fast, we also include it in our experiments to compare its performance with other approaches. Experiments are also performed on the Amazon EMR, and unlike the performance measurement in previous subsection, here the runtime performance includes both I/O cost and computation cost.



(a) scalability



(b) performance

Figure 6.3: Parallel partitioning performance

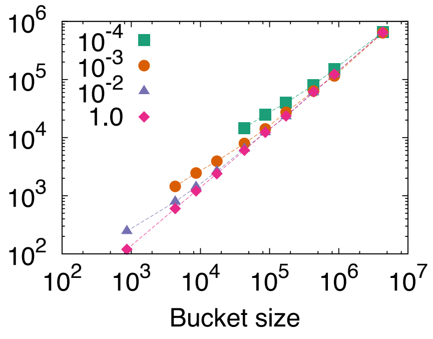
Figure 6.3(a) shows a scalability chart for the three MapReduce based parallel partitioning approaches on OSM dataset. The horizontal axis represents the number of nodes used for parallelization, and the vertical axis represents the partition runtime. The performance is measured with a top level coarse partition granularity of 500000. While this number seems to be arbitrary, our experiments show that the scalability is not affected by the coarse partitioning granularity. As the figure shows, the MapReduce based partitioning approach is very scalable and efficient. With the increased cluster capacity, the runtime performance improves almost linearly. With parallelization, the partition efficiency of the algorithms increased by an order of magnitude. For example, the runtime of BOS decreased from 4000 seconds to merely 300 seconds. Although the algorithms have very different runtime performance on a single thread implementation, the performance after parallelization seems to be homogeneous.

Recall that our parallelization algorithm performs partitioning in two steps. The top level coarse partitioning for parallel partitioning, and bottom level partitioning in which the coarse partitions are re-partitioned with specific spatial partition algorithms. Each step involves a partitioning granularity parameter which controls partition size. To study the effects of those parameters on parallel partitioning performance, we perform two seperate experiments. In the first experiment, we fix the coarse top level partitioning granularity and test the runtime performance with different bottom level partitioning granularity. Not surprisingly, the performance difference between different parameters are too little to be significant, and consequently we can conclude that the bottom level partitioning granularity has no noticeable effect on parallel partitioning performance.

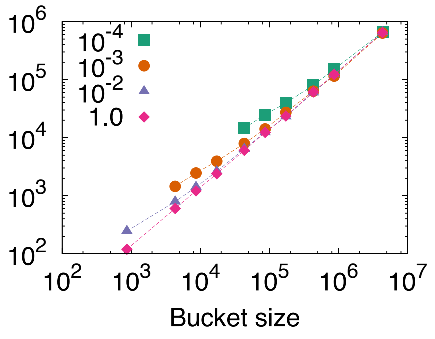
In the second experiment, we fix the bottom level partitioning, and change the top level partition granularity. Figure 6.3(b) shows performance variations of parallel partitioning for different partition granularity. We can see that as the top level partitioning granularity gets coarser, the performance gets better. Our profiling of the parallel algorithms provides following explanation. Like Terasort [91], the parallelization algorithms use a sampled data file for assigning the spatial objects into separate partition groups which has a global total ordering. In a finer granularity top level spatial partitioning, the total order based partition group assignment becomes the major bottleneck. Interestingly, the visualization of the partition boundaries show that spatial partition results from a larger top level partitioning has more resemblance to the partition results from a single threaded approach.

6.3.2 Sampling based Approach

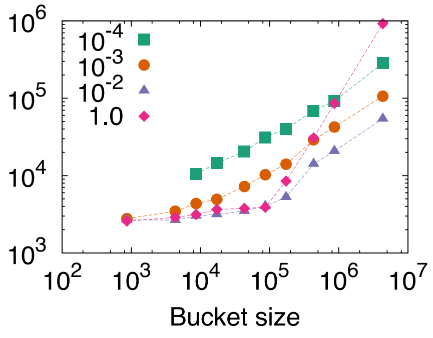
Figure 6.4 shows a statistical evaluation of three sampling based partitioning approaches on the OSM dataset. The figures on the left column show the standard deviation – measure of skewness – of generated partitions, and the figures on the right column show boundary object ratio. The full dataset is sampled with different sampling rate (shown in the legend of the figures), the resulting partitions from the sampled dataset are com-



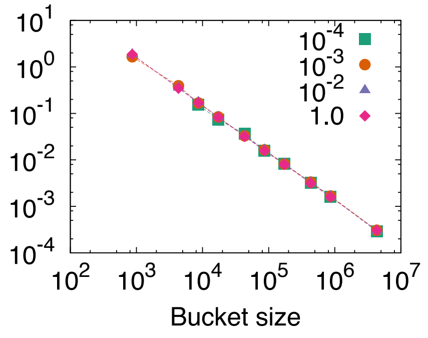
(a) BSP standard deviation



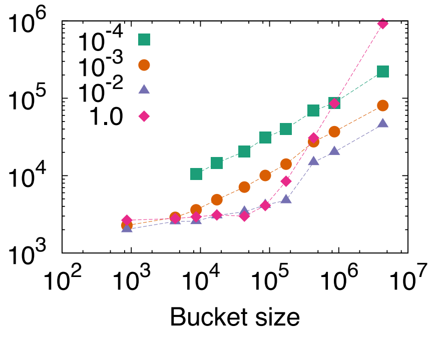
(b) BSP ratio of boundary objects



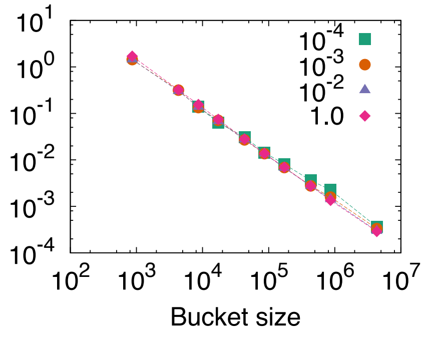
(c) SLC standard deviation



(d) SLC ratio of boundary objects



(e) BOS standard deviation



(f) BOS ratio of boundary objects

Figure 6.4: Quality of partitions generated by sampling based approaches

pared against the the partitioning generated from the full dataset. The sampling rate of 1.0 represents full dataset partitioning. From the figures we can see that sampling can be a very effective approach for spatial partitioning.

Intuitively, higher the sampling rate, the better we can preserve data distribution, and consequently the partitioning on the sampled dataset is of higher quality. If we look at the figures on the left column, we can see that partitions generated with higher sampling rate are less skewed compared to lower sampling rate partitioning. However, depending on the algorithm, partition skew can be different. For example, as BSP implicitly try to avoid a skewed partitioning, the impact of higher sampling rate is not significant. Whereas in SLC and BOS, higher sampling rate seems to be always beneficial. There is a minor exception to this case. Specifically, in SLC and BOS, if the partition payload is reasonably large, sampling based approaches can generate a less skewed partitioning than the full dataset partitioning. This is particularly interesting, and it has important implications for certain application scenarios. First, by using a sampling based approach we can significantly reduce the partition time. Second, aside from the improved performance, we can actually obtain a less skewed partitioning with the minor limitation of large partition size. Interestingly, the ratio of boundary objects generated by sampling based partition approaches is not completely dependent on the samping ratio. Overall, the sampling based partitioning approaches generate more boundary objects compared to the full dataset partitioning, although the variation is not significant.

Chapter 7

Haggis – Hardware Acceleration of Hadoop-GIS

GPUs have been successfully utilized in numerous applications that require high performance computation. Both approaches, GPU and MapReduce, have their own limitations and advantages, and have been separately utilized in spatial query processing tasks to boost application performance. However, it is unclear that how MapReduce and GPU, while being two vastly different parallelization strategies, can be fused together to effectively deal with the spatial big data challenges. In this chapter, we explore a such synergy of parallelization techniques for large scale spatial query processing.

7.1 Introduction and Related Approaches

Previous research on spatial query workload characterization [104] shows that spatial queries are a lot more compute-intensive compared to the conventional non-spatial query workloads. While MapReduce, with massive scalability, can effectively address data-intensive aspect of large scale spatial queries, it is not well suited to handle compute-intensive aspect of spatial queries. The multidimensional nature of spatial data analytics and the complexity of spatial queries requires a high performance approach that can leverage multiple parallelism mechanism for query processing. In recent years, GPGPU has become mainstream as multi-core computer architecture and programming techniques become mature. Now, a single machine can contain different parallel processors like multi-core CPUs or GPUs, and such hardware configurations is readily available on major cloud computing platforms. In the coming years, such heterogeneous parallel architecture will become dominant, and software systems must fully exploit such heterogeneity to deliver performance growth [44].

In this chapter, we explore opportunities for such synergy between two different parallelization techniques, MapReduce and GPU, for large scale spatial query processing. We develop *Haggis* – HadoopGIS accelerated with a GPU engine. Due to the difference in the parallelization model, there are several problems that need to be addressed and that is the focus of this work. First, integration of two programing models is not trivial. Previous research efforts [67] propose to modify the MapReduce programming model to bring those two parallelization frameworks together in a unified framework. In Haggis, we take a decoupled approach in which we use GPU as an engine accelerator without modifying underlying MapReduce programming model. Second, as GPU computation requires the data to be moved to the device memory which incurs off-chip memory movement cost. Such data movement need to be well orchestrated to reduce the I/O cost and fully exploit the computation power of GPU. Third, to achieve best system performance, the task scheduler need to judiciously place tasks on CPU or GPU, so that the system resource is fully utilized. There are several design decisions towards building an integrated system, and we explore various issues in detail.

Previously, in [47], an approach is proposed on bulk-construction of R-Trees through MapReduce. In [124], a spatial join method on MapReduce is proposed for skewed spatial data, using an in-memory based strip plane sweeping algorithm. It uses a duplication avoidance technique which could be difficult to generalize for different query types. Hadoop-GIS takes a hybrid approach on combining partitioning with indexes and generalizes the approach to support multiple query types. Besides, our approach is not limited to memory size. VegaGiStore [125] tries to provide a Quadtree based global partitioning and indexing, and a spatial object placement structures through Hilbert-ordering with local index header and real data. The global index links to HDFS blocks where the structures are stored. Work in [19] takes a fixed grid partitioning based approach and uses sweep line algorithm for processing distributed joins on MapReduce. The work in [62] presents an approach for multi-way spatial join for rectangle based objects, with a focus on minimizing communication cost. A MapReduce based Voronoi diagram generation algorithm is presented in [35]. In our work [33], we present results on supporting multi-way spatial join queries and nearest neighbor queries for pathology image based applications. Previously, we proposed Pixelbox [120] for accelerating cross-comparison queries for pathology image analysis. In [95, 122, 123], authors discuss a GPU accelerated approach for spatial query processing. However, none of those approaches are concerned with an integrated approach which combines both MapReduce and GPU.

There are several recent efforts on task scheduling for hybrid machines [37, 66, 68, 82, 98, 113, 114, 115]. Most of the previous works deal with task mapping for applications in which operations attain similar speedups when executed on a GPU vs a CPU. On the other hand, we are exploiting performance variability to better use heterogeneous processors.

7.2 GPU Accelerated Spatial Query Processing

7.2.1 Spatial Queries on CPU

Most spatial queries are compute-intensive [104] as they involve geometric computations on complex multi-dimensional spatial objects. Spatial objects have complex extent which generally described with multi-dimensional data points. For example, typical spatial objects such as lines and polygons are need to represented with several two dimensional points (in a 2D Euclidean coordinate system), and those data points are stored and processed together. Therefore, even simple operations on those objects become expensive. Geometric computation is not only used for computing measurements or generating new spatial objects, but also as logical operations for topology relationships.

To avoid the high cost of the geometry computation and reduce unnecessary disk I/O, spatial queries employ a *filter-and-refine* strategy in which queries are processed in two phase. During the *filter* phase, spatial objects are approximate with minimum bounding rectangles (MBRs), and objects that do not satisfy the query predicate even on the MBRs are eliminated. During the refinement step, candidate objects from the filter step are processed with real geometry operations, and the objects that satisfy the query predicate are reported as final query results. While spatial filtering through MBRs can be accelerated through spatial access methods, spatial refinements such as polygon intersection verification are highly expensive operations. For example, spatial join queries such as spatial cross-matching or overlaying multiple sets of spatial objects

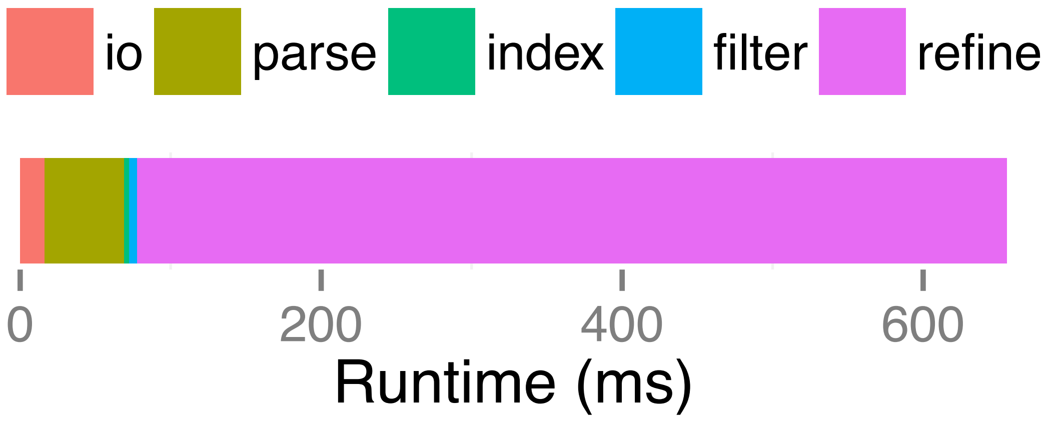


Figure 7.1: Spatial join query cost breakdown on CPU

To illustrate the high computational cost of spatial queries, we run a single threaded spatial join query on a small dataset. The join query here simply reads in the data from a file which resides on the secondary storage, builds an RTree index on the data, and performs an indexed spatial join operation, then calculates the area of intersection for intersecting polygons in the dataset. Figure 7.1 shows the cost of each query components. In the figure, the refinement operator includes both geometry based refinement operation for checking polygon intersection, and the measurement of intersecting area. Clearly, the computation component cost shadows other query components, and becomes the main performance bottleneck in large scale spatial query processing.

7.2.2 Spatial Queries on GPU

Graphics processing units (GPUs) have been successfully utilized in numerous applications that require high performance computation. Mainstream GPGPUs come with hundreds of cores, and can run thousands of threads in parallel. Compared to the multi-core computer systems (dozens of cores), GPUs can scale to large number of threads in a cost effective manner. Therefore, GPUs have the great potential to improve the performance of spatial queries by eliminating the computation bottleneck.

Most spatial algorithms are designed for executing on the CPUs, and the branch intensive nature of CPU based algorithms require the algorithm to be rewritten for GPUs for satisfactory performance. For such need, PixelBox is proposed in [120]. PixelBox is an algorithm specifically designed for accelerating cross-matching queries on the GPUs. It first transforms the vector based geometry representation into raster representation using a pixelization method, the performs operations on such representation in parallel. The pixelization method reduces the geometry calculation problem into simple pixel position checking problem, and it is very suitable for execution on GPUs. Since testing the position of one pixel is totally independent of another, it can parallelize the computation by having multiple threads process the pixels in parallel. Furthermore, since the position of different pixels are computed against the same pair of polygons, the operations performed by different threads follow the SIMD fashion, a parallel com-

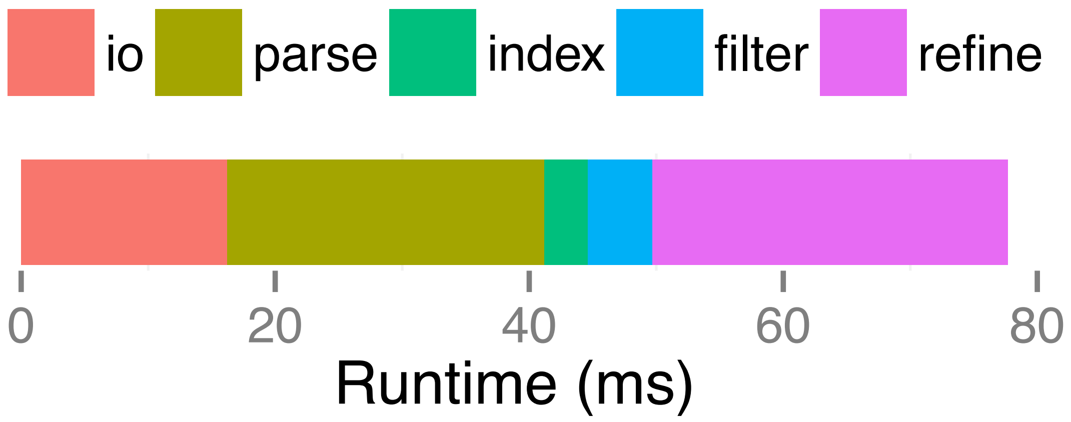


Figure 7.2: Spatial join query cost breakdown on GPU

Figure 7.2 shows the performance and query cost breakdown for the same spatial join query in Figure 7.1. As the figure shows, PixelBox achieves almost an orders of magnitude speedup compared to the CPU implementation, and significantly reduces the cost of computation. Note that, here we also utilize GPU for parsing the input data.

7.3 Implementation Details of Haggis

Haggis is an extension of HadoopGIS using GPUs. Haggis utilizes MapReduce for multi-node query parallelization, and accelerates queries using GPU within single node, thus elegantly employing two different parallelization paradigms for performance.

7.3.1 Architectural Details

Within a Mapper/Reducer task, HadoopGIS relies on RESQUE for spatial processing. Therefore, the MapReduce parallelization is decoupled from the actual spatial query operations, and we can easily extend the query engine for query performance.

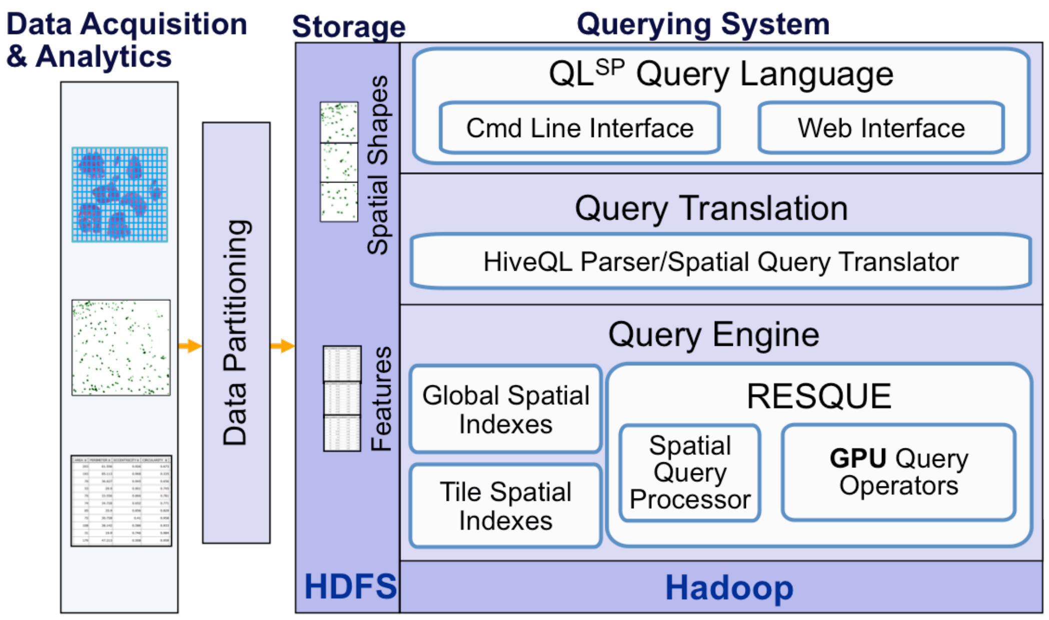


Figure 7.3: Architecture of HadoopGIS

Figure 7.3 shows an architecture overview of Haggis. Other than the query engine, the systems are identical. In *Haggis* extension, we implemented a number of GPU based spatial query operators, and integrated into the RESQUE engine. During the query processing phase, the RESQUE engine can arbitrarily choose CPU or GPU for executing the query, and such decision is transparent to the user. Query optimizer is responsible for making such decision to achieve optimal query performance. One major advantage of Haggis is flexibility. In a computer system equipped with GPUs, Haggis can utilize the extra computation power for performance, and if such resource is not available Haggis can simply rely on the CPU and move on.

7.3.2 Task Assignment

One critical issue for GPU based parallelization is task assignment. As Haggis uses MapReduce based parallelization at the higher level, tasks arrive in the form of data partitions along with spatial query operation on the data. Given a partition, the query optimizer has to decide which device should be assigned to execute the task. Such decision is not simple, and it depends on how much speedup we can get by assigning it to CPU/GPU. If we schedule a small task on GPU, we may not only get very little speedup, the opportunity cost of executing some other high speedup task on GPU can be high.

In Haggis, we use a predictive modeling approach for making such decision. Similar to the speculative execution model in Hadoop, we sample certain amounts of data (10% in our experiment) for performance profiling. We execute those data on both GPU and CPU, and measure the speedup. Then we use a simple polynomial line fitting algorithm to derive the performance model, and corresponding parameters on the sampled data. The for upcoming tasks, this model is used to predict to calculate the potential speedup factor. If the speedup factor is higher than certain threshold, we assign the task to GPU, otherwise to the CPU.

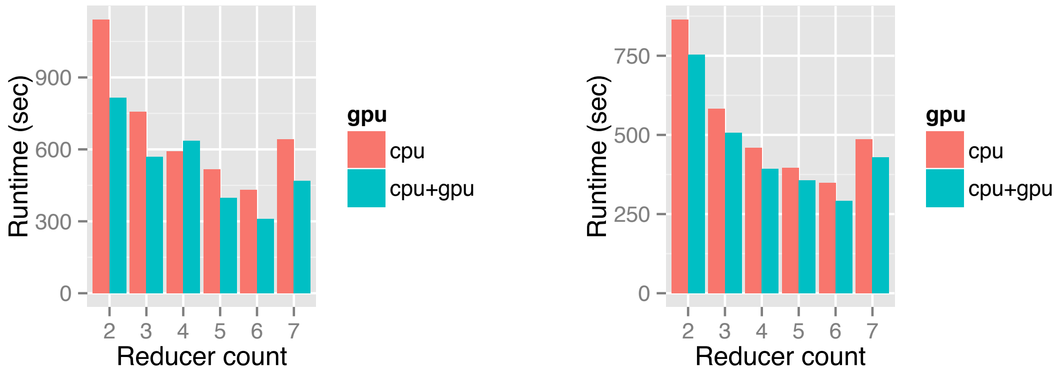
7.3.3 Effects of Task Granularity

Data need to be shipped to the device memory to be executed on the GPU device. Such data transfer incurs certain I/O cost. While the memory bandwidth between GPU and CPU is much higher compared to the bandwidth between memory and the disk, it should be minimized to achieve optimal performance. To achieve optimal speedup, the compute to transfer ratio needs to high for GPU applications. Therefore, Haggis need to adjust the partition granularity to fully utilize system resources. While larger partitioning is ideal for achieving higher speedup on GPU, it causes data skew which is detrimental for MapReduce system performance. At the same time, a very small partition is not a good candidate for hardware acceleration.

7.4 Experiments

7.4.1 Effects of CPU for co-processing

In the first experiment we study if the number of CPUs has an effect on the query performance. Specifically, the haggis scheduler only utilizes the given number of CPUs for co-processing. Ideally, if there are extra compute resources, the system should utilize such resource and try to improve the query performance. However, due to the scheduling issues, such objective is hard to achieve.

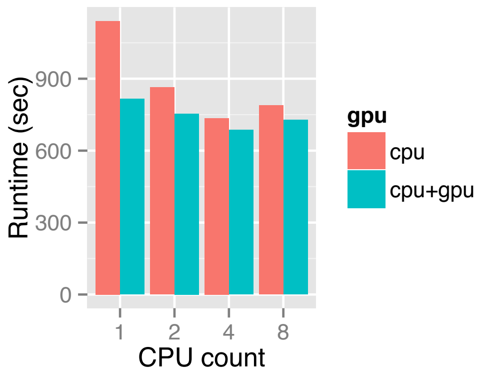


(a) Single CPU for co-processing (b) Eight CPUs for co-processing

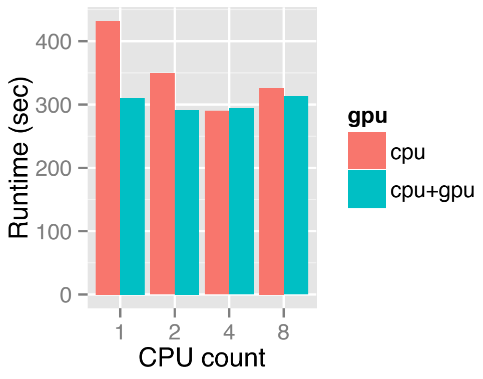
Figure 7.4: Effects of available CPUs for query processing

Figures 7.4 shows the performance of spatial join query on two different configurations. The horizontal axis represents MapReduce level parallelization, and consequently the number of reducers participated for query processing. The vertical axis represents the query runtime. In Figure 7.4(a), only single CPU is used for query co-processing as if the node is a single core machine, whereas in Figure 7.4(b), 8 CPU cores are used for query co-processing. As Figure 7.4(a) shows, GPU can be helpful for improving query performance, and we can see that with different MR parallelization granularity the GPU accelerated system outperforms the CPU only system. However, the speedup is not very high. We were expecting to see same speedup as we show earlier in section 7.2. While partly this can be attributed to the data skew, data transfer overhead, the result is not satisfactory for a GPU accelerated system. Furthermore, to our dismay, given that there are enough CPU cores for query processing, GPU seems to be less helpful as shown in Figure 7.4(b).

7.4.2 Effects of MR Parallelization



(a) Two reducer node



(b) Six reducer node

Figure 7.5: Effects of MapReduce parallelization

Next, we study how number of available reducer nodes effects query performance. Figure 7.5 shows the performance of spatial join query on different cluster settings. The horizontal axis represents the number of available cores for co-processing, and the vertical axis represents the query runtime. As the figures show, with higher number of parallel MapReduce nodes the query performance can be improved significantly. For example, with single CPU core for co-processing, the query performance reduced from 1211 to 424 seconds. This also illustrates the advantage of such hybrid system which can benefit from the scalability of MapReduce. With the help of GPU, this number further reduced to 309 seconds. However, as the number of CPU cores increases, the advantage of GPU is not so obvious.

7.5 Summary

Haggis – a hybrid system that combines the benefit of scalable and cost-effective data processing with MapReduce, and the benefit of efficient spatial query processing with GPU. Our preliminary experimental results demonstrate that Haggis provides a scalable and effective solution for analytical spatial queries over large scale spatial datasets. However, there are many challenges such as effective task scheduling and assignment, and mitigating suboptimal partition granularity for achieving better speedup.

Chapter 8

Hive^{SP} — An Implementation of Hadoop-GIS

We integrate the Hadoop-GIS framework into Hive for public release. Hive provides a declarative query interface – HiveQL, for running large scale MapReduce tasks. Currently we have a subset of spatial operators that we have integrated into Hive, and our development team is working towards implementing the complete spatial operators defined in SQL/MM [107] standard, and their corresponding SQL-to-MapReduce translators for Hive. Hive^{SP}, is fully compatible with Hive and it provides an easy way for querying massive spatial data. Given the large codebase, and difficulty of translating spatial SQL queries into efficient MapReduce operators, it may also lead to other interesting research questions on spatial query optimization and translation.

8.1 System Architecture

The core of Hadoop-GIS is a partition based spatial query engine [33, 34] that supports diverse spatial queries with optimal access methods in the MapReduce framework. Spatial queries are translated into series of MapReduce code, and can be run in parallel on a large number of cluster nodes.

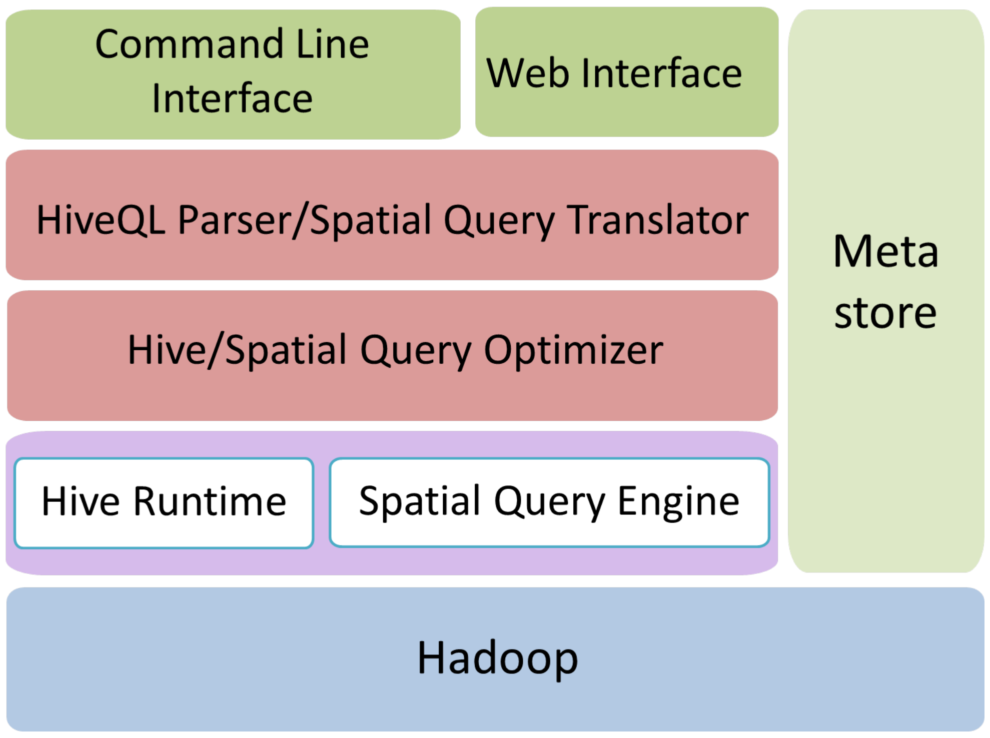


Figure 8.1: Architecture of Hive^{SP}

Figure 8.1 shows an architectural overview of Hadoop-GIS. Users interact with the system by submitting SQL queries either from a command line or web interface; the queries are parsed and translated into an operator tree by the spatial query translator, and the query optimizer applies heuristics optimization rules to generate an optimized query plan. For a query with spatial query operator, corresponding MapReduce code are generated, which call an appropriate spatial query pipeline supported by the spatial query engine. Generated MapReduce code are submitted to the execution engine for execution. Spatial data is partitioned based on the spatial attribute which specified in the table definition, and staged to the HDFS system for parallel access. There are several key components in Hadoop-GIS that are developed to provide spatial query processing capability.

- *Spatial Query Translator* parses and translates SQL queries into an abstract syntax tree. We extended the HiveQL translator to support a set of spatial query operators, spatial functions, and spatial data types.

- *Spatial Query Optimizer* takes an operator tree as an input and applies rule based optimizations such as predicate push down or index-only query processing.

- *Spatial Query Engine* is a stand-alone spatial query engine which supports following infrastructure operations: i) spatial relationship comparison, such as *intersects, touches, overlaps, contains, within, disjoint*, ii) spatial measurements, such as *intersection, union, convexHull, distance, centroid, area*, etc; iii) creating and querying spatial access methods, such as *R^*-Tree*, for efficient query processing.

8.2 Query Language

Declarative query language interfaces to MapReduce, such as Hive, Pig and Scope, have gained much momentum in recent years. Data scientists and applications developers are also more comfortable with SQL queries compared to pure programming interfaces. Moreover, most spatial applications are backed by spatial database systems which use SQL as the primary query interface. Therefore, we extended HiveQL, a SQL-like query language to Hive, to provide a high level and easy to use query interface to Hadoop-GIS.

The query language inherits major operators and functions from ISO SQL/MM Spatial, and extends it for more complex pattern queries and data partitioning constructs to support parallel query processing in MapReduce. Major spatial operations include spatial query operators, spatial functions, and spatial data types. The spatial query operators include topology based spatial relationships, such as `ST_INTERSECTS`, `ST_CONTAINS`, `ST_TOUCHES`, and nearest neighbor operator `ST_KNN`. They can be categorized into two main types: binary operators to find spatial topology relationship between two spatial objects, and aggregate operators to find spatial patterns among a group of spatial objects. The spatial functions include unary functions, for example `ST_AREA`; binary functions such as `ST_DISTANCE`, and aggregate functions. Currently supported spatial data types include `Point`, `Polygon`, `Box`, and `LineString`.

We also developed query constructs which facilitate the parallel query processing by partitioning the input data on spatial attribute. The query language provides a `PARTITION BY` clause, which specifies a partition attribute on which the data is partitioned, and a partitioning method by which the input data is partitioned. So far, the systems only supports regular fixed grid based tile partitioning. In future, we are planning to implement other optimal partitioning approaches.

Example queries include: i) Spatial feature aggregation: aggregation or summary statistics on computed features or spatial attributes. ii) Spatial range query: find spatial object(s) which contained in a specified space (possibly followed by an aggregation query). iii) Spatial join and spatial cross-matching: find correlations between multiple sets based on topology relationships of the spatial objects. In a typical spatial data warehousing scenario, user may also need to work with other business data sources that are warehoused in the same system with spatial data. In such case, users can write a mixture of spatial query and non-spatial query without switching to a different system. Hadoop-GIS supports nested and mixed queries in a streamlined fashion.

8.3 Storage Layer

At the storage layer, the original HDFS is kept intact to insure backward compatibility with existing Hadoop platforms. However, the internal organization of the spatial data is optimized to insure better performance. Specifically, after tile based partitioning, each record is assigned a internal *tile id* which indicates which tile this record belongs to. There are two advantages of such tile based storage organization. First, it servers as a logical parallelization unit for task creation and assignment. Second, similar to the range partitioning in parallel database systems, it works as a "filtering" mechanism to avoid redundant processing.

We also extend Hive index utility for creating persistent spatial index on the spatial columns. This index data is stored in separate internal table which later used by the query planner for query optimization. Since certain queries, such as window query, can be processed more efficiently with the help of a spatial index, this would increase the spatial query performance. Moreover, for queries that can be answered only using an index, the I/O overhead can be significantly reduced by only scanning the index data which has much smaller storage footprint compared to the whole table scan.

8.4 Query Processing

Hive^{SP} uses the traditional *plan-first, execute-next* approach for query processing. There are three key steps: query translation, logical plan generation, and physical plan generation. To process a query expressed in SQL, the system first parses the query and generates an equivalent abstract syntax tree representation of the query. Preliminary query analysis is performed in this step to ensure that the query is syntactically and grammatically correct, and table metadata information is valid. Next, the abstract syntax tree is translated into a logical plan which is expressed as an operator tree, and simple query optimization techniques such as predicate push down, and column pruning are applied in this step. Currently, system has a simple rule based query optimizer which has limited capability. How to incorporate a cost based query optimizer is an ongoing research effort. Then, a physical plan is generated from the operator tree which eventually consists of series of MapReduce jobs. Finally, the generated MapReduce jobs are submitted to the Hive runtime for execution.

Major differences between Hive and Hadoop-GIS are in the logical plan generation step. If a query does not contain any spatial operations, the resulting logical query plan is exactly the same as the one generated from Hive. However, if the query contains spatial operations, the logical plan is regenerated with special handling of spatial operators. Specifically, two additional steps are performed to rewrite the query. First, operators involving spatial operations are replaced with internal spatial query engine operators. Second, serialization/deserialization operations are added before and after the spatial operators to prepare Hive for communicating with spatial query engine.

An example query plan is given in Figure 8.2, which is generated by translating the SQL query in Figure 8.5. As the query plan shows, the query translator generates the *table scan* operators in the first step; during this process, query predicates are applied to filter the input data; then, upon determining this is a spatial join operator, the query translator generates a tile based join processing workflow, where each tile forms a simple join task; each such task is marked for executing in the spatial query engine, and appropriate input/output data formats are specified; each task is scheduled for execution on Hadoop; after the spatial query engine finishes processing a tile, the result is returned, and the execution privilege is returned to the Hive execution engine which continues the processing task. As the workflow shows, the interaction between spatial query engine and Hive execution engine is transparent to the user.

8.5 Software Setup

Hadoop-GIS is designed to be completely hot-swappable with Hive. Hive users only need to deploy the spatial query engine on Hadoop cluster nodes, and turn the spatial query processing switch on. To use the system, users can follow the same user guidelines of using Hive. Any query that runs on Hive, runs on Hadoop-GIS without any modification. Current version of Hadoop-GIS utilizes a set of UDFs from *ESRI GIS-tools-for-hadoop* [9] library.

Schema Creation: Users create all the necessary table schema depending on their warehousing need. The schema information is stored in the metastore. Spatial columns need to be specified with corresponding data types defined in ISO SQL/MM Spatial.

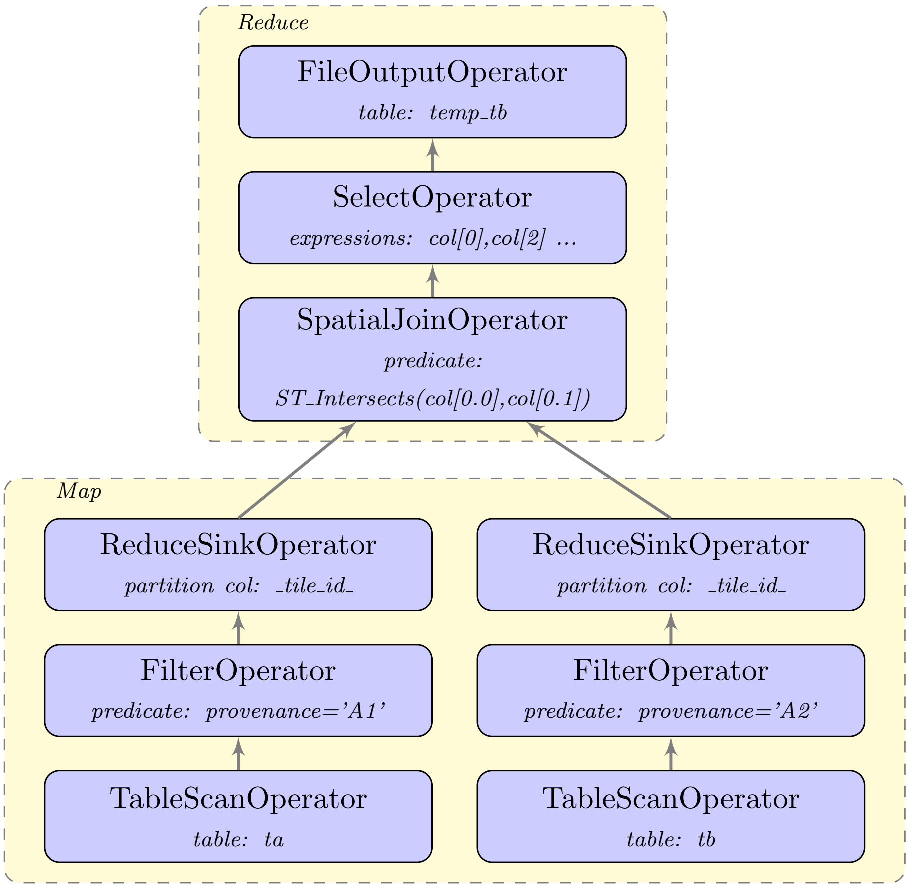


Figure 8.2: Two-way spatial join query plan in Hive^{SP}

Optionally users can specify spatial partition column to speed up query processing. An example SQL query for creating the example table schema is given in Figure 8.3.

```
CREATE TABLE tcga_markups ( markup_id BIGINT,
    provenance STRING, hand_marked BOOLEAN,
    center ST_POINT, polygon ST_POLYGON )
PARTITIONED BY TILE(polygon, 4096, 4096)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE ;
```

Figure 8.3: An example table schema

Data Loading: Then users can upload data which will be used to populate the schema. Figure 8.4 shows an example. Data can be read either from a local path or a HDFS path.

```
LOAD DATA [LOCAL] INPATH '/data/tcga.wkt' [OVERWRITE]
INTO TABLE tcga_markups ;
```

Figure 8.4: Data loading command

Querying: Users can submit SQL queries from the terminal within Hive Shell, or from a web interface which designed for interactively viewing and analyzing data. Here, Figure 8.5 illustrates a spatial join query for evaluating to two image segmentation results.

```
SELECT
   ST_AREA(ST_INTERSECTION(ta.polygon,tb.polygon))/
   ST_AREA(ST_UNION(ta.polygon,tb.polygon)) AS ratio,
   ST_DISTANCE(ST_CENTROID(tb.polygon),
   ST_CENTROID(ta.polygon)) AS distance,
FROM tcga_markups ta JOIN tcga_markups tb
   ON (ST_INTERSECTS(ta.polygon, tb.polygon) = TRUE)
WHERE ta.provenance='A1' AND tb.provenance='A2' ;
```

Figure 8.5: A spatial join query in HiveQL

Bibliography

[1] https://en.wikipedia.org/wiki/Satellite_imagery.

[2] https://foursquare.com/about.

[3] http://www.openstreetmap.org.

[4] http://craig-henderson.blogspot.com/2009/11/dewitt-and-stonebrakers-mapreduce-major.html.

[5] https://cwiki.apache.org/confluence/display/Hive/Configuratio+Properties.

[6] https://wiki.apache.org/pig/PigSkewedJoinSpec.

[7] https://web.cci.emory.edu/confluence/display/hadoopgis.

[8] http://libspatialindex.github.com.

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

[10] Arcgis. http://www.esri.com/software/arcgis.

[11] Db2 spatial. www.ibm.com/software/data/spatial/db2spatial.

[12] Geocouch. https://github.com/couchbase/geocouch.

[13] Greenplum database. http://www.greenplum.com/products/greenplum-database.

[14] Ibm netezza. http://www-01.ibm.com/software/data/netezza.

[15] Monetdb. https://www.monetdb.org.

[16] neo4j/spatial. https://github.com/neo4j/spatial.

[17] Oracle spatial and graph. http://www.oracle.com/us/products/database/options/spatial/overview/index.html.

[18] The sloan digital sky survey project (sdss). http://www.sdss.org.

[19] Spatialhadoop. http://spatialhadoop.cs.umn.edu.

[20] Teradata. www.teradata.com/products-and-services/teradata-geospatial.

[21] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[22] Vertica. http://www.vertica.com.

[23] Boost c++ libraries. http://www.boost.org/, 2013.

[24] Geos. http://trac.osgeo.org/geos, 2013.

[25] Pathology analytical imaging standards. https://web.cci.emory.edu/confluence/display/PAIS, 2013.

[26] Apache hadoop. http://hadoop.apache.org.

[27] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, Aug. 2009.

[28] D. Achakeev and B. Seeger. A class of r-tree histograms for spatial databases. In *SIGSPATIAL/GIS*, pages 450–453. ACM, 2012.

[29] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *ACM SIGMOD Record*, volume 28, pages 13–24, 1999.

[30] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Commun. ACM*, 53, June 2010.

[31] A. Aji. High performance spatial query processing for large scale scientific data. In *Proceedings of the on SIGMOD/PODS 2012 PhD Symposium*, pages 9–14. ACM, 2012.

[32] A. Aji, T. George, and F. Wang. Haggis: Turbocharge a mapreduce based spatial data warehousing system with gpu engine. In *ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '14, 2014.

[33] A. Aji, F. Wang, and J. H. Saltz. Towards Building A High Performance Spatial Query System for Large Scale Medical Imaging Data. In *SIGSPATIAL/GIS*, pages 309–318. ACM, 2012.

[34] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *Proc. VLDB Endow.*, 6(11):1009–1020, Aug. 2013.

[35] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *CLOUDCOM*, pages 9–16, 2010.

[36] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012 international conference on Management of Data*, pages 241–252. ACM, 2012.

[37] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In S. B. Jesper Larsson Träff and J. Dongarra, editors, *The 19th European MPI Users' Group Meeting (EuroMPI 2012)*, volume 7490 of *LNCS*, Vienna, Autriche, 2012. Springer.

[38] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system: the story of O2*. Morgan Kaufmann Publishers Inc., 1992.

[39] M. Bastos, R. Recuero, and G. Zago. Taking tweets to the streets: A spatial analysis of the vinegar protests in brazil. *First Monday*, 19(3), 2014.

[40] D. S. Batory, T. Leung, and T. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems (TODS)*, 13(3):231–262, 1988.

[41] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[42] J. V. d. Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB*, pages 461–470, 2001.

[43] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.

[44] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[45] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using r-trees. In *ICDE*, 1996.

[46] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The exodus extensible dbms project: An overview. *Readings in object-oriented database systems*, pages 474–499, 1990.

[47] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM*, pages 302–319, 2009.

[48] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136. ACM, 1982.

[49] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[50] X. Chen, H. Vo, A. Aji, and F. Wang. High performance integrated spatial big data analytics. In *ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '14, 2014.

[51] L. A. Cooper, A. B. Carter, A. B. Farris, F. Wang, J. Kong, D. A. Gutman, P. Widener, T. C. Pan, S. R. Cholleti, A. Sharma, et al. Digital pathology: Data-intensive frontier in medical imaging. *Proceedings of the IEEE*, 100(4):991–1003, 2012.

[52] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[53] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[54] T. Eavis and A. Lopez. Rk-hist: an r-tree based histogram for multi-dimensional selectivity estimation. In *CIKM*, pages 475–484. ACM, 2007.

[55] A. Eldawy and M. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, 2014.

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

[57] M. R. Evans, D. Oliver, X. Zhou, and S. Shekhar. Spatial big data. *Big Data: Techniques and Technologies in Geoinformatics*, page 149, 2014.

[58] R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information processing letters*, 12(3):133–137, 1981.

[59] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high level dataflow system on top of MapReduce: The Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[60] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533. IEEE, 2012.

[61] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *ACM SIGMOD Record*, volume 29, pages 463–474, 2000.

[62] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. Mohania. Processing multi-way spatial joins on map-reduce. In *EDBT*, pages 113–124, 2013.

[63] R. H. Güting et al. Gral: An extensible relational database system for geometric applications. In *VLDB*, volume 89, pages 33–44, 1989.

[64] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[65] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: as the dust clears [database project]. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):143–160, 1990.

[66] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Automatic dataflow application tuning for heterogeneous systems. In *International Conference on High Performance Computing (HiPC)*, pages 1–10, 2010.

[67] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[68] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Parallel Architectures and Compilation Techniques*, 2008.

[69] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010.

[70] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, 19(3):201–260, 1987.

[71] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE, 2010.

[72] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[73] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, 1994.

[74] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packing. In *ACM-SIAM symposium on Discrete algorithms*, pages 384–393, 1998.

[75] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[76] Y. Kwon, K. Ren, M. Balazinska, and B. Howe. Managing skew in hadoop. *IEEE Data Eng. Bull.*, 36(1):24–33, 2013.

[77] W. Le, F. Li, Y. Tao, and R. Christensen. Optimal splitters for temporal and multi-version databases. In *Proceedings of the 2013 international conference on Management of data*, pages 109–120. ACM, 2013.

[78] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, 2011.

[79] S. T. Leutenegger, M. A. Lopez, and J. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506. IEEE, 1997.

[80] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. Lars: A location-aware recommender system. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 450–461. IEEE, 2012.

[81] J. Lin et al. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, volume 1, 2009.

[82] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[83] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.

[84] J. Lu and R. H. Guting. Parallel secondo: Practical and efficient mobility data processing in the cloud. In *Big Data*, pages 107–25. IEEE, 2013.

[85] R. Mistry and S. Misner. *Introducing Microsoft SQL Server 2014*. Microsoft Press, Redmond, WA, USA, 2014.

[86] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[87] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. *ACM SIGMOD Record*, 17(3):28–36, 1988.

[88] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *ICDT*, pages 236–256, 1999.

[89] A. Okabe, B. Boots, and K. Sugihara. *Spatial tessellations: concepts and applications of Voronoi diagrams*. Wiley & Sons, 1992.

[90] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[91] O. OMalley. Terabyte sort on apache hadoop. http://sortbenchmark.org/Yahoo-Hadoop, 2008.

[92] S. L. Osborn and T. Heaven. The design of a relational database system with abstract data types for domains. *ACM Transactions on Database Systems (TODS)*, 11(3):357–373, 1986.

[93] J. Patel et al. Building a scaleable geo-spatial dbms: technology, implementation, and evaluation. In *SIGMOD*, pages 336–347, 1997.

[94] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[95] S. Puri and S. K. Prasad. Mpi-gis: New parallel overlay algorithm and system prototype. 2014.

[96] S. Ray, B. Simion, A. D. Brown, and R. Johnson. A parallel spatial data analysis infrastructure for the cloud. In *SIGSPATIAL*, pages 274–283. ACM, 2013.

[97] Y. J. Roh, J. H. Kim, Y. D. Chung, J. H. Son, and M. H. Kim. Hierarchically organized skew-tolerant histograms for geographic data objects. In *SIGMOD*, pages 627–638. ACM, 2010.

[98] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, 2011.

[99] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, New York, NY, USA, 1995. ACM.

[100] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[101] H.-J. Schek and W. Waterfeld. A database kernel system for geoscientific applications. In *Proc. 2nd Int. Symp. on Spatial Data Handling, Seattle, Washington*, pages 273–288, 1986.

[102] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *the VLDB Journal*, 7(1):48–66, 1998.

[103] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB endowments*, 1987.

[104] B. Simion, S. Ray, and A. D. Brown. Surveying the landscape: an in-depth analysis of spatial database workloads. In *SIGSPATIAL*, pages 376–385. ACM, 2012.

[105] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 6(14):1882–1893, 2013.

[106] J. Steenstra, A. Gantman, K. Taylor, and L. Chen. Location based service (lbs) system and method for targeted advertising, Mar. 23 2006. US Patent App. 10/931,309.

[107] K. Stolze. Sql/mm spatial: The standard to manage spatial data in relational database systems. In *Proceedings of the BTW*, 2003.

[108] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[109] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[110] M. Stonebraker et al. Inclusion of new types in relational data base systems. In *ICDE*, volume 262, page 269. Citeseer, 1986.

[111] M. Stonebraker and L. A. Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[112] M. Stonebraker, B. Rubenstein, and A. Guttman. The ingres papers: Anatomy of a relational database system. chapter Application of Abstract Data Types and Abstract Indices to CAD Data, pages 317–333. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[113] G. Teodoro, T. Hartley, U. Catalyurek, and R. Ferreira. Optimizing dataflow applications on heterogeneous environments. *Cluster Computing*, 15:125–144, 2012.

[114] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. M. Jr., U. Catalyurek, and R. Ferreira. Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications. In *IEEE Cluster*, pages 1–10, 2009.

[115] G. Teodoro, E. Valle, N. Mariano, R. Torres, J. Meira, Wagner, and J. Saltz. Approximate similarity search for online multimedia services on distributed CPU-GPU platforms. *The VLDB Journal*, pages 1–22, 2013.

[116] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[117] H. Vo, A. Aji, and F. Wang. Sato: A spatial data partitioning framework for scalable query processing. In *SIGSPATIAL/GIS*. ACM, 2014.

[118] F. Wang, J. Kong, L. Cooper, T. Pan, K. Tahsin, W. Chen, A. Sharma, C. Niedermayr, T. W. Oh, D. Brat, A. B. Farris, D. Foran, and J. Saltz. A data model and database for high-resolution pathology analytical image informatics. *J Pathol Inform*, 2(1):32, 2011.

[119] F. Wang, J. Kong, J. Gao, D. Adler, L. Cooper, C. Vergara-Niedermayr, Z. Zhou, B. Katigbak, T. Kurc, D. Brat, and J. Saltz. A high-performance spatial database based approach for pathology imaging algorithm evaluation. *Journal of Pathology Informatics*, 4(5), 2013.

[120] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *Proc. VLDB Endow.*, 5(11):1543–1554, 2012.

[121] Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *SIGMOD*, pages 969–974, 2010.

[122] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '13, pages 23–31, 2013.

[123] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–32. ACM, 2012.

[124] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, 2009.

[125] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards parallel spatial query processing for big spatial data. In *IPDPSW*, pages 2085–2094, 2012.

[126] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *GeoInformatica*, 2(2):175–204, 1998.