TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Datenbanksysteme

DOCTORAL THESIS

Advancing Spatial Analytical Database Systems

Varun Pandey

TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Datenbanksysteme

DOCTORAL THESIS

Advancing Spatial Analytical Database Systems

Varun Pandey

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Pramod Bhatotia
Prüfer der Dissertation:	1.	Prof. Alfons Kemper, Ph.D.
	2.	Prof. Dr. Florian Matthes
	3.	Prof. Mohamed Sarwat Abdelghany Aly Elsayed, Ph.D. (Arizona State University)

Die Dissertation wurde am 11.01.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.04.2021 angenommen.

Abstract

Spatial data is pervasive. Over the last decade, we have observed a rise in amount of spatial data that is generated everyday. It comes from a plethora of sources, such as GPS-enabled devices in the form of cell phones, cars, sensors, and from various consumer-based applications such as Uber, Foursquare, location-tagged posts in Facebook, Twitter, and Instagram. This exponential growth in spatial data has led the research community to focus on building systems and applications that can process spatial data efficiently. At the same time, advances in machine learning enable researchers and practitioners to build techniques and systems that push the limit of the state-of-the-art.

This thesis makes three contributions to the design and implementation of systems that handle spatial data. First, we study the big-data spatial analytics systems that have emerged in recent years. We thoroughly compare these systems empirically for all major features that they support using various queries and real-world datasets. Second, we carry out a study of the state-of-the-art spatial libraries that are used in many big-data systems and services, many of which are multi-million dollar industries. These systems rely on spatial processing and indexing capabilities of these libraries to build efficient solutions. We empirically compare these libraries based on four popular spatial queries using two real-world datasets. Third, we propose an approach to apply learned indexes to five classical spatial indexes in order to improve spatial query processing on location-data. We show that learned index outperform binary search for searching within a spatial partition and that spatial index structures require tuning for various datasets and query workloads. In addition, we also integrate spatial query processing capabilities in a state-of-the-art main-memory database system, HyPer, invented at TU Munich.

Zusammenfassung

Geodaten sind allgegenwärtig. In den letzten zehn Jahren haben wir einen enormen Anstieg der täglich generierten Geodaten beobachtet können. Diese stammen aus einer Vielzahl von Quellen wie zum Beispiel GPS-fähige Geräten in Form von Mobiltelefonen, Autos, Sensoren und auch aus verschiedenen verbraucherorientierten Anwendungen wie Uber, Foursquare und standortbezogenen Beiträge auf Facebook, Twitter und Instagram. Dieses exponentielle Wachstum der Geodaten hat die Forschungsgemeinschaft veranlasst, sich auf die Entwicklung von Systemen und Anwendungen für die effiziente Verarbeitung von Geodaten zu konzentrieren. Gleichzeitig ermöglichen Fortschritte beim maschinellen Lernen den Forschern und Praktikern, Techniken und Systeme zu entwickeln, die die Grenzen des Standes der Technik vorantreiben.

Diese Arbeit leistet drei Beiträge zum Entwurf und zur Implementierung von Geodaten Systemen. Zunächst untersuchen wir, die in den letzten Jahren entstandenen Big Data Analysesysteme für Geodaten. Mithilfe verschiedener Abfragen und realer Datensätze erstellen wir einen empirischen Vergleich der wichtigsten Funktionen dieser Systeme. In einem zweiten Schritt führen wir eine Studie über die neuesten Programmbibliotheken für Geodaten durch. Diese Bibliotheken werden in vielen Big-Data-Systemen und -Diensten, von denen viele mehrere Millionen Dollar kosten, verwendet. Diese Systeme stützen sich auf räumliche Verarbeitungs- und Indizierungsfunktionen, welche von den Programmbibliotheken bereitgestellt werden, um effiziente Lösungen zu erstellen. Wir vergleichen diese Bibliotheken empirisch anhand von vier gängigen räumlichen Abfragen unter Verwendung von zwei realen Datensätzen. Drittens, schlagen wir einen Ansatz vor, um gelernte Indizes auf fünf klassische räumliche Indizes anzuwenden, um die räumliche Abfrageverarbeitung für Standortdaten zu verbessern. Wir zeigen, dass der gelernte Index den binären Suchalgorithmus für die Suche innerhalb einer räumlichen Partition übertrifft und dass räumliche Indexstrukturen eine Anpassung für verschiedene Datensätze und Abfragen erfordern. Darüber hinaus haben wir Funktionen zur Verarbeitung räumlicher Abfragen in das an der TU München erfundene moderne Hauptspeicher-Datenbanksystem, HyPer, integriert.

Acknowledgments

It gives me immense joy to express my gratitude towards everyone who contributed and advised me over the years of my Ph.D. journey.

First and foremost, I would like to thank my advisor, Prof. Alfons Kemper. He has always been very supportive of every work that I was part of, and was always present whenever I needed his help or advice. I would also like to thank Prof. Thomas Neumann on being there whenever we needed his guidance, especially while implementing HyPerSpace.

I am also very grateful to the whole thesis committee. I am very thankful to Prof. Florian Matthes and Prof. Mohamed Sarwat for serving on my thesis committee and for their invaluable feedback on my work. I am also very grateful to Prof. Pramod Bhatotia for chairing my thesis committee. I would also like to thank Frau Elisabeth Sommer, and Frau Manuela Fischer, who made the whole process of submission, defence, and publication of the thesis very simple.

A special thanks goes to Angelika Reiser and Silke Prestel. They were very helpful when I first moved to Germany, and advised me about so many aspects of life in the country. Angelika is also the best reviewer I know, who critically refined so many of our publications. Next, I would like to thank all of my colleagues in the database group, especially Andreas Kipf, and Alexander van Renen who I worked closely with. Both of you have been great friends as well as colleagues. I would also like to thank all of my co-authors in every publication. None of the work would be as refined without your invaluable feedback, and guidance.

I would also like to thank all of my friends from India who are now all over the world. Next I would like to thank all the friends I made in Germany, who made my day to day life easy. I would like to thank Romain, Oliver, Sanjay, and Chinmay for being great flatmates. I thoroughly enjoyed spending my time with all of you. I would also like to thank Vicky, Swetha, Anmol, Mathieu, Deepesh, Advait, Adya, Niko, Maria, Francesco, Kiwon, Narasimha, Narendra and many more for all the great times together. I would also like to thank all of my teammates from the football team, especially Ian, Youssef, Jose, Cladio, Ignacio, Nikita, and many more. Thank you all for the great memories, on the field as well as outside it.

I would also like to thank my parents, Jayant and Kavita, and my sister, Priyakshi, for their outstanding support. They all taught me to value education, to always help and be there for others, and to never give up. I wouldn't have been able to complete this thesis without your guidance and support. I would also like to thank all the other members of my family, my uncles, aunts, and cousins. Finally, I would also like to thank all of my grandparents who remain a great source of inspiration to me.

Funding. This work has been partially supported by the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

Preface

Excerpts of this thesis have been published in advance.

Chapter 2 is drawn from the following publications with minor modifications to the description of "HyPerSpace":

> Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. "High-Performance Geospatial Analytics in HyPerSpace". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 2016, pp. 2145–2148

Chapter 4 is drawn from the following publications with minor modifications:

> Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. "An Evaluation Of Modern Spatial Libraries". In: *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 21-24, 2020, Proceedings, Part II*. vol. 12113. Lecture Notes in Computer Science. Springer, 2020, pp. 157–174

> An extended version appeared in Data Science and Engineering (DSE) (Special Issue of DASFAA 2020):
> Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. "How Good Are Modern Spatial Libraries?" In: *Data Sci. Eng.* 6.2 (2021), pp. 192–208

Chapter 3 is drawn from the following publications with modifications to the description and additional unpublished results:

> Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "How Good Are Modern Spatial Analytics Systems?" In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1661–1673

Chapter 5 is drawn from the following publications with modifications to the description with additional algorithms, and unpublished results:

> Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. "The Case for Learned Spatial Indexes". In: *AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan*. 2020

Chapters 1 and 6 also draw from these publications, but also contain novel, unpublished material. In addition to these publications, the author of this thesis also co-authored the following related work, which is not part of this thesis:

Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* 2017, pp. 49–60

Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: *ACM Trans. Database Syst.* 44.1 (2019), 1:1–1:35

Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Approximate Geospatial Joins with Precision Guarantees". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 2018, pp. 1360–1363

Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, 2020, pp. 347–358

Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Geospatial Joins for Modern Hardware". In: *CoRR* abs/1802.09488 (2018)

Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. "The Case for Distance-Bounded Spatial Approximations". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021

All of the publications listed above are marked with an asterisk (*) in the bibliography in compliance with § 6 Abs. 6 Satz 3 Promotionsordnung der Technischen Universität München.

Contents

1 Introduction **1**
1.1 Big Data And Challenges . . . 1
1.2 Big Data Ecosystem . . . 3
1.2.1 Workloads . . . 3
1.2.2 Data Source . . . 4
1.2.3 Data Storage . . . 4
1.2.4 Distributed SQL Query Engines . . . 6
1.2.5 Big Data Computing Frameworks . . . 6
1.3 Spatial Data . . . 8
1.4 The Learned Era . . . 10
1.5 Contributions . . . 10

2 HyPerSpace **13**
2.0.1 Introduction . . . 13
2.0.2 HyPerSpace . . . 15
2.0.3 Evaluation . . . 15
2.0.4 Visualization using HyPerSpace . . . 18
2.0.5 Take-away message . . . 18

3 Modern Spatial Systems **21**
3.1 Introduction . . . 21
3.2 Motivation . . . 22
3.3 Queries . . . 23
3.3.1 Range Query . . . 23
3.3.2 k Nearest Neighbors Query . . . 23
3.3.3 Spatial Join . . . 24
3.3.4 k Nearest Neighbors Join . . . 24
3.4 Spatial Analytics Systems . . . 24
3.4.1 Hadoop-GIS . . . 24
3.4.2 SpatialHadoop . . . 26
3.4.3 SpatialSpark . . . 27
3.4.4 GeoSpark . . . 28
3.4.5 Magellan . . . 28
3.4.6 SIMBA . . . 28
3.4.7 LocationSpark . . . 29
3.5 Experimental Setup . . . 30
3.5.1 Cluster Setup And Tuning Spark . . . 30
3.6 Tuning Amazon EMR and Apache Spark . . . 31
3.6.1 Datasets . . . 33
3.6.2 Spark Memory Management Model and Caching RDDs 34

3.6.3 Performance Metrics 35
3.7 Evaluation 36
3.7.1 Memory Costs 36
3.7.2 Range Query Performance 37
3.7.3 kNN Query Performance 39
3.7.4 Distance Join Performance 41
3.7.5 Spatial Joins Performance 43
3.7.6 kNN Join Performance 48
3.7.7 US Census TIGER Dataset 50
Distance Join Performance 51
kNN Join Performance 51
Spatial Joins Performance 53
3.8 Conclusions And Future Work 53

4 Modern Spatial Libraries **59**
4.1 Introduction 59
4.2 Background 60
4.2.1 Geometry Models 60
4.2.2 When Can Things Go Wrong In Planar Geometries? . . 62
4.3 Queries 63
4.3.1 Range Query 63
4.3.2 Distance Query 64
4.3.3 *k*-nearest neighbors Query 64
4.3.4 Spatial Join 64
4.4 Libraries 64
4.4.1 ESRI Geometry API 64
4.4.2 Java Spatial Index 65
4.4.3 JTS Topology Suite and Geometry Engine Open Source 66
4.4.4 Google S2 Geometry 66
4.4.5 Vantage Point Tree 67
4.5 Methodology 67
4.6 Evaluation 69
4.6.1 Indexing Costs 70
4.6.2 Range Query 72
4.6.3 Distance Query 74
4.6.4 *k*-NN Query 75
4.6.5 Point-In-Polygon Join Query 76
4.7 Discussion 77
4.7.1 Why Refinement Should Be Looked At? 77
4.7.2 Distributed Spatial Analytics Systems 79
Spatial Partitioning 80
4.8 Related Work 81
4.9 Conclusions 81

5 The Case For Learned Spatial Indexes **85**
5.1 Introduction . . . 85
5.2 Approach . . . 87
5.2.1 Partitioning Techniques . . . 87
Fixed and Adaptive Grid . . . 87
Quadtree . . . 88
K-d tree . . . 88
Sort-Tile-Recursive (STR) packed R-tree . . . 88
5.2.2 Building Index . . . 89
5.2.3 Range Query Processing . . . 89
Search Within Partition . . . 91
5.2.4 Distance Query Processing . . . 92
5.2.5 Join Query Processing . . . 93
5.3 Evaluation . . . 94
5.3.1 Datasets . . . 94
5.3.2 Range Query Performance . . . 95
Tuning Partitioning Techniques . . . 95
Query Performance . . . 98
5.3.3 Distance Query Performance . . . 100
Tuning Partitioning Techniques . . . 100
Query Performance . . . 102
5.3.4 Join Query Performance . . . 104
5.3.5 Indexing Costs . . . 106
5.4 Related Work . . . 106
5.5 Conclusions and Future Work . . . 108

6 Future Work **111**

List of Figures

1.1 Growth in sales of handheld and wearable devices, and the expected growth of connected IoT devices 8

2.1 *HyPerSpace* vs. related systems: throughput of *ST_Covers* using lat/long co-ordinates 16
2.2 Microbenchmark results: throughput of *ST_Covers* using lat/long co-ordinates 17
2.3 Interactive visualization of a real-time replay of NYC taxi rides using *HyPerMaps* 19

3.1 A generalized indexing scheme for distributed spatial analytics systems 26
3.2 *maximizeResourceAllocation* deployment vs a better deployment 31
3.3 Memory footprint for various datasets 36
3.4 Indexing costs 37
3.5 Range query performance on a single node for different selection ratio (σ) 38
3.6 Range query performance for all geometric objects scaling up the number of nodes [selection ratio (σ) = 1.0] 38
3.7 Range query performance scaling up the number of nodes for different selection ratio (σ) on different datasets 40
3.8 kNN query performance varying k 41
3.9 kNN query scalability with $k = 10$ 41
3.10 Distance join cost breakdown scaling up the number of nodes 42
3.11 Distance join scalability 43
3.12 Distance join shuffle costs 43
3.13 Scalability of all spatial joins for different systems while scaling up the number of nodes 44
3.14 Spatial joins peak execution memory consumption 45
3.15 Spatial joins shuffle read costs 45
3.16 Spatial joins shuffle write costs 46
3.17 Total runtime cost breakdown for spatial joins between various geometric objects on a single node 46
3.18 *Point-Rectangle* spatial join cost breakdown scaling up the number of nodes 47
3.19 kNN join cost breakdown scaling up the number of nodes . . 49
3.20 kNN join scalability 49
3.21 kNN join shuffle costs 49
3.22 Distance join cost breakdown scaling up the number of nodes 51
3.23 Distance join scalability 51

3.24 Distance join shuffle costs 52
3.25 kNN join cost breakdown scaling up the number of nodes . . 52
3.26 kNN join scalability 52
3.27 kNN join shuffle costs 53
3.28 Scalability of all spatial joins for different systems while scaling up the number of nodes 54
3.29 Spatial joins peak memory consumption 55
3.30 Spatial joins shuffle read costs 55
3.31 Spatial joins shuffle write costs 56
3.32 Total runtime cost breakdown for spatial joins between various geometric objects on a single node 56
3.33 *Point-Rectangle* spatial join cost breakdown scaling up the number of nodes 57

4.1 Datasets: NYC Taxi trips are clustered in central New York while Tweets are spread across the city 70
4.2 Index sizes for the two datasets 71
4.3 Index building times for the two datasets 71
4.4 Range query performance varying the number of points and selectivity of the query rectangle for NYC Taxi and Twitter Datasets 73
4.5 Distance query performance varying the number of points and selectivity of the query rectangle for NYC Taxi Dataset and Twitter Datasets 74
4.6 kNN query performance varying the number of points and k for NYC Taxi and Twitter Datasets 75
4.7 Join query performance for NYC Taxi and Twitter Datasets . . 76
4.8 Refinement costs for Midtown Manhattan Polygon for NYC Taxi Dataset using various contains functions in JTS 78

5.1 Machine Learning vs. Binary Search. For low selectivity (0.00001%), the index and refinement phases dominate, while for high selectivity (0.1%), the scan phase dominates (parameters are tuned to favor Binary Search) 86
5.2 An illustration of the different partitioning techniques 86
5.3 Datasets: (a) Tweets are spread across New York, (b) NYC Taxi trips are clustered in central New York, and (c) All Nodes dataset from OSM 95
5.4 Range query configuration - ML vs. BS for low selectivity (0.00001%) 96
5.5 Effect of number of cells and number of points scanned for Fixed-grid on Taxi Trip dataset for skewed queries (0.00001% selectivity) 97
5.6 Effect of number of cells and number of points scanned for Quadtree on Taxi Trip dataset for skewed queries (0.00001% selectivity) 98
5.7 Total range query runtime with parameters tuned on selectivity 0.00001% 99

5.8 Distance query configuration - ML vs. BS for low selectivity (0.00001%) 103
5.9 Total distance query runtime with parameters tuned on selectivity 0.00001% 105
5.10 Join query performance for the three datasets 107
5.11 Index build times and sizes for the three datasets 108

List of Tables

3.1 Overview of features in spatial analytics systems 25
3.2 Evaluated systems, their compatible Spark version, and defaults for the experiments 30
3.3 Spark configuration parameters 32
3.4 Details of the datasets used for evaluation 33
3.5 Strengths and Weaknesses 57

4.1 Selected features of the libraries 65
4.2 Selected features of all indexes 68
4.3 CPU Counters - Range query datasize = 50M tweets, selectivity = 0.1 %, 1 thread, normalized by the number of range queries. All values are in millions except IPC. 73
4.4 Strengths/Weaknesses of the Libraries 82

5.1 Total range query runtime (in microseconds) for both RadixSpline (ML) and binary search (BS) for Taxi Rides dataset on skewed and uniform query workloads (parameters are tuned for selectivity 0.00001%) 101
5.2 Average number of partitions intersected for each partitioning scheme for selectivity 0.00001% on Taxi Rides and OSM datasets 102

Chapter 1

Introduction

The most valuable goal of data exploration is to extract information and make meaningful inferences [68]. Visualization is one of the most powerful and intuitive interactive analysis tools that facilitates data exploration and knowledge discovery [64]. The goal of interactive analysis tools is to empower data analysts to formulate and assess hypotheses in a rapid and iterative manner. Data exploration and visualization have thus become one of the major research areas in the era of *Big Data*.

In 2007, Jim Gray, a pioneering computer scientist, coined the term *The Fourth Paradigm*: Data-Intensive Scientific Discovery. He puts forward the case for one of the biggest challenges for 21st-century science: the new era of *big* and *data-intensive* science. He suggested that to tackle big data there is a need for a set of tools and technologies that help in data visualization and exploration.

1.1 Big Data And Challenges

A recent study [141] by International Data Corporation (IDC) predicts that the global datasphere will grow from 33 zettabytes to 175 zettabytes by 2025. Today, data is being generated at unprecedented rates, and it comes from a variety of sources. Square kilometer Area (SKA) radio telescope, will be the world's largest radio observatory and is expected to produce 700 terabytes per second [152, 111]. In a few days, the data generated would eclipse the current size of the world wide web. The world wide web at the time of writing consists of at least 5.5 billion[1] web pages [174].

Existing literature have investigated multiple definitions of big data. Big data is generally characterized by the 3V's [97]: Volume, Variety, and Velocity. These characteristics also layout the challenges that they bring in storing, managing, and processing the data. Over the years, more characteristics (or V's) have been added, namely, Veracity, Value, Variability, and Visualization [162].

[1] The figure is an underestimation as it is estimated from three search engines, Google, Bing and Yahoo Search. The approximate size of Google's index from the same source stands at 55 billion webpages.

- **Volume**: Volume refers to the sheer size of the large-scale dataset. Social media data alone (Facebook, LinkedIn, Twitter, Strava, etc.) is enormous. YouTube currently sees upload of 500+ hours of video everyday [199], Google conducts 1.2 trillion searches [57], Facebook generates 500+ petabytes of data everyday, Strava dataset consists of trillions of GPS data points [43]. The data sources are usually heterogeneous, ubiquitous, and dynamic in nature, which, along with the large size of the data makes storing, retrieving, and processing the data a challenging task. This also requires changes to existing data mining algorithms and novel approaches to handle the large size of the data [215].

- **Variety**: Variety indicates different the types of data, which include unstructured and semi-structured (audio, video, webpage, text, etc.) as well as traditional structured data. The data is generated from various sources which include, user generated contents (e.g. tweeets, blogs, photos/videos shared by users), transactional data (web logs, business transactions, feeds of moving objects, sensor networks etc.), scientific data (celestial data, genome data, health care data), to web data from search engines and graph data from social networks and RDF knowledge bases [25].

- **Velocity**: Velocity refers to speed at which the data is generated which can be real-time or nearly real-time. To utilize the commercial value of the data, it has to be processed and analyzed in a timely manner i.e, in real-time. This introduces another challenge because of the rate and the amount of inserts or updates that needs to be handled in real-time. Velocity brings challenges to every part of a data management system, and both storage and the query processing layer needs to be extremely fast and scalable [25].

- **Veracity**: [147] defines veracity to coping with the biases, doubts, ambiguities, and inaccuracies in data. Veracity can be caused by a variety of factors: collection errors, entry errors, system errors, spammers, rumors and many more. Web is also a soft medium to publish and announce falsified information across multiple mediums (Twitter, Facebook, Blogs, etc.). Moreover, customer opinion on different social media networks and web is different and unclear in nature as it involves human interaction [161]. Big data can be noisy, and thus requires validation to isolate high quality data from low quality data.

- **Value**: Value refers to extraction of knowledge or value from large amount of structured and unstructured data. The big data in it's original form has less value, but by applying data analytics it can be converted to a high-value asset. For example, values extracted from a stream of web clicks by internet users is driving the internet economy today, but organizations are still faced with challenges of storing, managing, and most importantly extracting value from the data [1].

- **Variability**: Variability indicates the variation in the data flow rates [52]. Often, the velocity of the big data is not consistent, and has periodic peaks and troughs. One example is load on social media websites which causes peaks in data ingestion as a result of an event. Variability also refers to data whose meaning is constantly changing [77]. For example, data from a source could potentially offer different meaning every time it is mined. Thus, organizations need to develop sophisticated techniques in order to understand the context in data and decode it's exact meaning.
- **Visualization**: Visualization refers to representing the key information and knowledge from large datasets in an instinctive and effective way by using various visual formats such as pictorial, or graphical layouts [171]. The data can have multiple dimensions and succinctly representing the data in a visual format for ease of knowledge extraction remains a big challenge.

1.2 Big Data Ecosystem

There are several layers or components of Big data. In this section we will describe the various layers of big data ecosystem.

1.2.1 Workloads

One of the key issue with large-scale data processing, whether be it in enterprise, science, medicine etc., is that there can be a wide variety of workloads which have the potential to be compute intensive. The various types of workloads can be:

- **Batch-Oriented**: these are recurring tasks such as large-scale data mining or aggregation.
- **OLTP**: OLTP stands for online transaction processing and is focused on transaction oriented tasks, which includes inserts, updates, and deletes from a database. These workloads support the daily operational needs of the business enterprises.
- **OLAP**: OLAP stands for online analytical processing, and queries large amount of historical data, aggregated from various sources for data mining, analytics, and business intelligence purposes. Utilizing business intelligence tools, like Tableau, on top of an OLAP engine turns raw data into insights allowing enterprises to make informed decisions.
- **Stream Processing**: Streaming data is continuously generated data from hundreds of different sources, where the data should be processed sequentially and incrementally on a record-by-record (also called events) basis or over a window of records or time. Tweets in Twitter are an example of streaming workload.

- **Pattern Search**: over structured, semi-structured, and unstructured data.

1.2.2 Data Source

Big data comes from a variety of sources [25] and can be classified into several categories:

- **User-generated**: It comes from a variety of users who voluntarily contribute data, information, or media that then appears before others is a meaningful or engaging way [93]. Examples include tweets, social media posts, blogs, photos/videos posted by user on online platforms.
- **Scientific Data**: is collected from data-intensive experiments or applications. Example of scientific data includes, celestial data, high-energy physics data, genome data, health-care data, biomedical/bio-informatics data, pharmaceutical data, biometric data, radiology data and many more. It can be structured (e.g., time series), semi-structured (e.g., XML files) or unstructured (e.g., images).
- **Transactional data**: is generated by large-scale systems from thousands of transactions processed by the system. Example can include data from financial institutions, stock markets, enterprise data, web logs, and business transactions. It usually is structured with pre-defined schemas.
- **Machine-generated**: is generated from a variety of sources and includes examples such as web server logs, application server logs, network logs, records of user activity etc.
- **Internet of Things (IoT)**: refers network of physical objects that are embedded with sensors for the aim of connecting and exchanging data with other devices and systems. IoT includes a wide array of applications, from consumer based applications such as connected vehicles, smart bikes, smart e-scooters, home automation, wearable technology, connected health, elderly care, to enterprise, industrial, environmental, and military applications such as manufacturing, agriculture, smart cities, energy management, environment monitoring, wildlife monitoring, military drones, robots, human-wearable biometrics etc.

1.2.3 Data Storage

Big data requires scaling, more specifically horizontal scaling. Horizontal scaling (also called scaling out) refers to adding more machines to the pool of resources. Big data, thus, once generated from various data sources, requires storage across multiple machines. For a system to obtain a holistic view of the data spread across multiple servers or machines, it requires *data management* [140]. Data Storage can be divided into two categories: *data storage formats* and *data storage systems*. Just as a single machine requires a file

system to control how data is stored and retrieved from a storage device, distributed *data stores* requires specialized techniques to manage data across multiple machines. Distributed storage formats is broadly of three types:

- **File**: File storage is the oldest and most commonly used storage technology where the data is organized in a hierarchical manner, and the basic unit of data is a file. File storage in a distributed environment requires a distributed file system in order to provide a unified view of the data. Some of the popular distributed file systems are Hadoop Distributed File System (HDFS) [158], Google File System (GFS) [56], Amazon EFS [5], IBM General Parallel File System (GPFS) [13], Quantcast File System (QFS) [71], Gluster File System (GlusterFS) [34], and many more.

- **Block**: Block storage provides fixed size raw-storage capacity in units called blocks. Every block is assigned a unique address, which is the only metadata that is attached to the block. Block storage is usually used for relational databases, NoSQL databases, virtual machines, containerized applications etc. Block storage has also become very popular among cloud providers today as a part of IaaS, such as Amazon Elastic Block Storage (EBS) [4], Google Persistent Disk [135], Azure Disk Storage [11] among many others.

- **Object**: Object storage is data object, file system metadata, and custom metadata combined together. It allows user to add custom metadata to the objects, and thus help in building many applications on top of it such as distributed warehouses (e.g., Snowflake [31], and Amazon Redshift [58]), data archive and backup, data lakes etc. Some popular cloud native block storage offerings are Amazon S3 [7], Azure Blob [16], IBM Cloud Object Storage [66], etc.

Distributed storage, on a high level, is an abstraction of how the data is stored on multiple machines in the cluster or on the cloud. Distributed *data stores* are responsible for placing the actual data in the distributed storage. These stores or systems can be broadly subdivided into two categories: *NoSQL* systems, and relational database systems (*RDBMS*)

- **NoSQL Systems**: NoSQL (or non-relational) systems were developed partly as a response to the unstructured data that originated from the web which required distributed and faster processing. These systems promised massive scalability and low latency (even under high load), but loosened the ACID guarantees, and mostly followed the CAP theorem [20]. The NoSQL systems can be further subdivided into key-value, document oriented, column-oriented, and graph databases. Some of the most popular NoSQL systems are BigTable [24], DynamoDB [163], Cassandra [96], HBase [187], Redis [23], MongoDB [28], Voldemort [137], CouchDB [9], and several others.

- **Databases**: Relational databases have been studied and researched for more than 40 years. But over the years many distributed database systems have emerged that are relational and especially built to scale out

such as Amazon RDS [6], Google Spanner [30] (evolved from a key-value store to a relational database system), CockroachDB [29], Azure SQL [12] among several others. At a high level of abstraction, these systems shard the data across many sets.

1.2.4 Distributed SQL Query Engines

The data in distributed storage can be structured, semi-structured, or unstructured. Moreover, many of the systems, especially NoSQL systems lead to development of SQL-like languages that could query the underlying data, for e.g. Apache Pig for Hadoop[10], HiveQL [176] for Hive. Many distributed SQL query engines thus also emerged in the past decade, which are essentially an abstraction layer on top of the distributed storage in order to run interactive SQL queries, in particular ad hoc queries, on the distributed data. Majority of these engines were built to address the non-SQL nature of the NoSQL systems, and in some cases also to completely remove SQL-like languages that were invented for each of these systems. This made it easier for practitioners to query the underlying data based on a standard and robust query language, instead of learning multiple languages for multiple systems. Presto [155] is layer on top of Cassandra and fully supports the relational model. It has now evolved to query over heterogeneous data stores, including Hive, HDFS, relational databases and even proprietary stores. Similarly, Impala [89], Apache Drill [62] are SQL engines for Hadoop in order to fully leverage the flexibility and scalability of Hadoop. Google Spanner [30] also evolved from a key-value store to a full featured SQL system, with query execution tightly integrated with the other architectural features of Spanner.

1.2.5 Big Data Computing Frameworks

The main big data computing frameworks can be categorized as batch processing frameworks, and stream processing frameworks. Here we discuss two popular batch processing frameworks, and briefly touch upon various streaming frameworks.

- **Apache Hadoop**: There are many ideas that Apache Hadoop combined effectively in order to organize, manage, and process large-scale data. The first observation was related to the storage capacity and the disk seek speeds. Although, the disk capacities increased tremendously over the years, the disk access speed lagged behind. For, example terabytes of disk space became the norm a decade ago, yet the disk access speed was around 100 MB/s. This meant that it took more than two hours to read the whole data from the disk. One way to mitigate this problem is to read from multiple disks at the same time, which is what effectively formed the core of Hadoop. HDFS [158] was inspired from the Google File System [56], which essentially distributed the data to multiple machines and exposed a unified view of the data. Since the data is distributed, the analysis of the data required combining data from multiple disks (reads), and writing out the subsequent results to

disks (writes). MapReduce [33], provides a programming model that abstracts this problem of disk reads and writes for the end user by transforming it into a computation over a set of keys and values. Hadoop runs the job by dividing it into two tasks: *Map* and *Reduce*. Initially, the data is processed by the map function and produces an intermediate result in the form of key-value. Next, the intermediate result is sorted using a phase called shuffle, followed by the reduce function. The reduce function performs a summary operation that processes the sorted intermediate results, and generates the final output. MapReduce is essentially a *batch* query processor, and provides the ability to run ad hoc queries against the whole dataset. Apache Hadoop ecosystem consists of many components: MapReduce, HDFS, Apache Pig (a data flow language running on top of MapReduce and HDFS), Apache Hive (a large-scale distributed data warehouse, which manages data on top of HDFS), HBase (a distributed, column-oriented database on top of HDFS that supports batch computation using MapReduce, and has supports for random reads or point queries), ZooKepeer (a distributed highly available coordination service, which provides services such as distributed locks), Sqoop (a tool for moving data between RDBMS and HDFS), etc.

- **Apache Spark**: Spark [209] is a unified engine for large-scale distributed data processing. Spark has a programming model similar to MapReduce, but it extends it to include a data-sharing abstraction which are known as *Resilient Distributed Datasets* (RDDs). RDDs are fault tolerant collections of objects that are partitioned across a cluster. RDDs are parallel data structures, which can be explicitly persisted in memory, and allows user to control partitioning for optimal data placement. RDDs can be created using operations known as *transformations* (map, filter, groupBy) to the data. RDDs are *ephemeral* by default which means that they are computed on the fly whenever an action (such as count) is called on them, but the user has the ability to persist them in memory. In case the memory is not sufficient, Spark spills to the disk. These persisted RDDs can be subsequently called by another action or transformation (data-sharing). While Hadoop writes the map's output to disk, before the shuffle and the reduce phase reads it back into memory buffers, Spark allows persisting RDDs in memory. This powerful yet simple abstraction allowed Spark to capture a wide range of processing workloads, include SQL, machine learning, graph processing, and streaming.

- **Streaming Frameworks**: While in batch processing, the queries are evaluated over a distinct set of data which is divided into batches, in streaming, the queries are processed using a progressive time window, a count-based window, or on just arrived data records. The streaming systems usually follow either of the two computational models: tuple-at-a-time, or micro-batch. The natural approach is to process the streams continuously (tuple-at-a-time), however, the events can also be

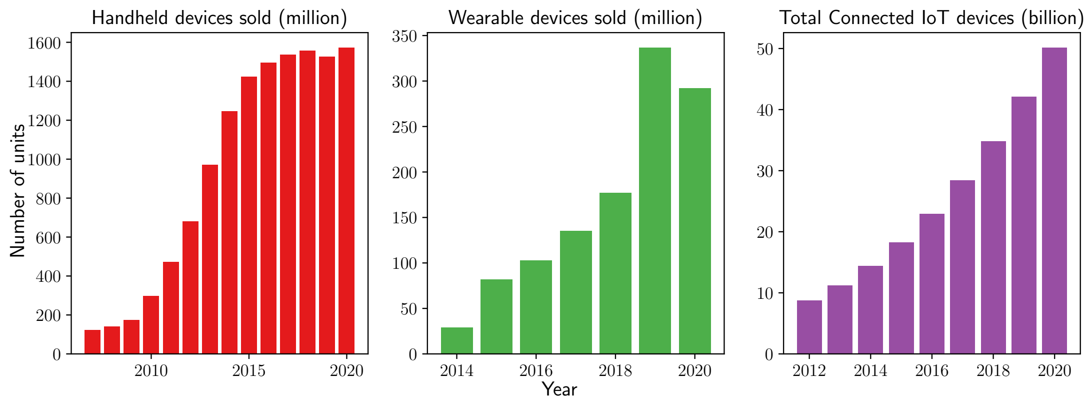


Figure 1.1: Growth in sales of handheld and wearable devices, and the expected growth of connected IoT devices

batched and processed as small chunks of data (micro-batch). Based on the computation model, the events are handled as they come into the system. Computation results are immediately available and continually updated whenever new data arrives. The streaming engines are able to ingest and aggregate large amount of events from different data sources. Some of the popular streaming frameworks are, Apache Flink [22], Apache Samza [120], Spark Streaming [210], and Apache Heron [94].

1.3 Spatial Data

A research [27] carried out by Pitney Bowes claims that **80%** of all data stored and maintained by organizations has a location component. Spatial data has become pervasive. There has been an explosion in the amount of spatial data being generated at the moment. It comes from the web, billions of phones, sensors, cars, satellites, delivery drones [190] and a huge array of various other sources. For example, NASA [116] provides climate projections since 1950 until 2100 for conducting studies of climate change impact. The data is captured using satellites and provides information about changes in the terrain etc. over the years. The dataset is approximately 17 TB in size.

At the same time, there has been an emergence of data-driven applications and location-based services. These services generate a large amount of location data on a daily basis and have a need to create real-time applications, including alerting systems, that consider the most current state of their data, enabling *real world awareness*. NYC Taxi Rides open dataset [122] consists of pickup and drop-off locations of more than 2.7 billion rides taken in the city since 2009. This represents more than 650,000 taxi rides every day in one of the most densely populated cities in the world, but is only a sample of the location data that is captured by many applications today. Foursquare, a popular cell phone application, has over 12 billion check-ins and has more than 105 million venues mapped around the world [48]. Uber,

a Transportation Network Company (TNC), recently reported completing 10 billion rides [183] till date, more than doubling the reported 5 billion rides completed the year before. Lyft, another TNC, now serves 1 million rides a day [123]. Twitter, a popular social media giant, generates approximately 10 million [105] geo-tagged tweets everyday. Another example is the popular human exercise tracking application which also incorporates social network features called Strava. The Strava dataset [43] comprises trillions of GPS data points. Most of these applications or data capture sources are consumer-based. These applications capture the location data using handheld-devices such as cell phones and tablets, or wearable technologies such as smart-watches and fitness-bands. Figure 1.1 shows the growth in sales of hand-held devices [121], wearable technologies [157], and the expected growth in number of connected IoT devices [65] over the years.

Business Intelligence has redefined multiple industries in various ways by effectively utilizing insights from the data. We argue that coupling *Business Intelligence* with *Location Intelligence* could potentially be a game-changer. This is also reflected by the fact that many cloud based data warehousing companies have started rolling out spatial functionalities including Red-shift [17], Snowflake [125], Amazon , Google's BigQuery GIS [70] etc.

Spatial data is also not only limited to only consumer data, or business related data. Spatial data is as prolific in scientific, medical, and environmental studies. The point-cloud dataset of Netherland consists of 640 billion data points [124], while the aforementioned NASA climate projections data consists of billions of data points. Square kilometer Area (SKA) radio telescope will generate 700 terrabytes of data per second [152, 111]. The data captured would be in the form of an image cube: consisting of two spatial dimensions, and a spectral frequency. Moreover, pathology data from sources such as microscopy scanners, generate images in extremely high resolution, where a 3D tissue volume typically generates hundreds of slices, and contains tens of millions of 3D biological objects. Such data also require complex 3D spatial processing, such as discovering and verifying spatial patterns among 3D biological objects like blood vessels, and cells. Finding patterns in such data has the potential to play a pivotal role in understanding hundreds of diseases.

Following this exponential growth in spatial data, and emergence of big data frameworks, many big data spatial analytics systems also emerged. These systems inlcude: HadoopGIS [2] and SpatialHadoop [40] based on Hadoop; GeoSpark [202, 203, 204], SpatialSpark [197], LocationSpark [173], and Simba [192] based on Apache Spark; GeoFlink [156] based on Apache Flink and a distributed spatial index [212] based on Apache Storm.

1.4 The Learned Era

Machine Learning and Artificial Intelligence have been extensively studied over the past few decades. Over the years, machine learning has made breakthroughs mostly due to three driving forces: large-scale datasets, new algorithms, and readily available high computing power. Promising breakthroughs in machine learning lead the database researchers to explore applying machine learning to designing the data systems. There are many reasons behind it: data keeps growing, hardware landscape keeps changing, new applications appear frequently, and it is hard to design one system that meets the requirements of all such trends [67], as we have seen earlier with multiple systems for big data. This leads to a new design space, tailor the system to requirements of the applications and the workloads. Machine learning is proven to be well-equipped to learn patterns and it seems like a natural step to utilize machine learning in design of data systems.

At the time of writing, machine learning has been applied to many areas of data systems design, and these areas can be summarized as follows [217]:

- **Learning-based Database Configuration**: Both databases and big data analytics systems consist of countless tunable knobs such as cache size, deadlock timeout, number of concurrent disk I/Os etc. Realizing this, many researchers proposed to apply machine learning to tuning the knobs. These research works are based on search-based tuning [218], traditional ML-based tuning [3, 51], and reinforcement learning ([102, 213]). Moreover, machine learning has also been applied to index selection [134, 150], and view selection [75, 74].

- **Learning-based Database Optimization**: Learning-based database optimization tries to address some of the most critical and hard problems in database optimization, such as cardinality estimation using supervised [38, 81, 133] and unsupervised [195] models, join order selection [92, 109, 205], and a full fledged query optimizers [110].

- **Learning-based Database Design**: Machine learning has also been applied to the area of designing various data structures for databases. This includes, replacement for traditional index structures by learned index structures such as the RMI-index [91], Fitting-tree [50], and an updatable learned index called ALEX [35]. Machine learning has also been applied to spatial data [103, 130, 138, 188], and multi-dimensional indexes [32, 117]. There has been a work on learning key-value store design [69] and a work that proposes a full fledged learned database system [90].

1.5 Contributions

In this thesis, we contribute to multiple areas to improve spatial query processing in the big spatial data and the learned era.

- First, we implement spatial datatypes, and spatial query processing in a state-of-the-art main-memory database system (MMDB), namely HyPer. HyPer belongs to an emerging class of hybrid databases, which enable *real world awareness* in real time by evaluating OLAP queries directly in the transactional database. In HyPer, OLAP is decoupled from mission-critical OLTP either by using the *copy on write* feature of the virtual memory management or *multi version concurrency control* [118]. We compare HyPerSpace with 3 state-of-the-art database systems, and show that even by generating a spatial index on-the-fly, HyPerSpace still outperforms these database engines.

- Second, we carry out an extensive study of the big data spatial analytics systems that have emerged in recent years. We first study the various features of these systems, and then we thoroughly compare them based on all of the features that these systems support. We compare these systems experimentally using five different spatial queries (range query, kNN query, spatial joins between various geometric datatypes, distance join, and kNN join) and four different datatypes (points, linestrings, rectangles, and polygons). We utilize two real-world datasets in order to accomplish the aforementioned tasks. This work, will help researchers in comparing their approaches with the existing systems, as well as help practitioners in choosing a system that suits their analytical needs.

- Third, we compare the modern spatial libraries that are used by hundreds of systems and applications to introduce spatial data processing. We argue that the systems or applications that are based on these libraries will be limited by the performance of these libraries, and the spatial indexes that are available in these libraries. We experimentally compare these libraries using two large real-world datasets, four popular spatial queries (range query, distance query, kNN query, and a spatial join query), and a variety of workloads that cover a large range of selectivity. This work will help researchers and practitioners alike in choosing a spatial library that best suits their needs.

- Lastly, we propose an approach to apply learned indexes to five classical spatial indexes in order to improve spatial query processing on location-data. We show that learned index outperform binary search for searching within a spatial partition and that spatial index structures require tuning for various datasets and query workloads for optimal performance. We also compare the performance of the learned spatial indexes utilized in this work with two state-of-the-art indexes, namely, S2PointIndex, and JTS STRtree. We show that learned indexes are 1.81× to up to 53.34× faster than these indexes across various queries. The learned indexes are also smaller in size, and faster to build than the aforementioned indexes. The spatial learned indexes can act as drop in replacement for read-only workloads in a variety of systems for big data, especially the analytical big data systems.

Chapter 2

HyPerSpace

2.0.1 Introduction

There has been a rapid advancement in research areas such as machine learning and data mining, which can be attributed to the growth in the database industry and advances in data analysis research. This has resulted in a need for systems that can extract useful information and knowledge from data. Data scientists use various data mining tools on top of databases for this purpose. To achieve lower latencies and minimize transmission costs between the database and external tools, it is necessary to move computation closer to the data. The current trend in database research is to integrate these various analytical functionalities that are useful for knowledge discovery into the database kernel. The goal is to have a full-fledged general-purpose database that allows big data analysis along with conventional transaction processing.

At the same time, there has been an emergence of data-driven applications. Companies like Uber, Lyft, and Foursquare have a need to create real-time applications, including alerting systems, that consider the most current state of their data, enabling *real world awareness*. Some of these applications have been enabled by the advent of the Internet of Things and the massive amounts of geotagged sensor data it generates.

There are publicly available datasets that can help in geospatial exploration. The New York City (NYC) Taxi Rides [122] dataset is a good example, but is only a sample of what is captured by the aforementioned companies. The dataset contains approximately 2.7 billion taxi rides taken in the city since 2009. This represents about 650,000 taxi rides everyday in one of the most densely populated cities in the world. Uber, a popular on demand car service available via a mobile application, has also made a subset of the taxi rides available for the cities of San Francisco and NYC. For NYC, Uber published data containing around 19 million rides for the periods from April to September 2014 and from January to June 2015. Ever since the datasets were published, there have been multiple static analyses on these datasets [191, 47, 153]. The authors of [45] present a comprehensive system built from scratch for storing, querying, and visualizing geospatial data using kd-trees. Their system takes two seconds to execute a query that returns 100,000 taxi trips, which is too slow to address real-time workloads. MemSQL has some real-time capabilities [112] and is one of the first main-memory database systems (MMDBs) to deeply integrate geospatial support. The current database systems do not offer the performance required by real-time applications, and

companies are often forced to build their own solutions [113]. We estimate that a 10x performance improvement is needed in general-purpose database systems to enable such applications/analytics.

We want to offer high-performance geospatial processing in a general-purpose database system that meets the requirements of real-time workloads, which can be used by emerging applications and data scientists alike without having to build their own system or use external tools for data analytics. Recent advancements in MMDBs research make it possible to efficiently create snapshots of the current database state. With our proposed system called *HyPerSpace*, we built a first prototype into that direction. Our goal is to drastically improve the performance of geospatial data processing in relational database systems by carefully using advanced encoding schemes and index structures. In our demo, we will present a web-based prototype called *HyPerMaps* that shows that it is possible to have an interactive analysis on geographical data using a general-purpose database system instead of a custom hand-written solution.

PostGIS [136] is a spatial database extension for the PostgreSQL object-relational database system. It adds support for geographic objects allowing users to formulate geospatial queries in SQL. PostGIS adds two popular spatial datatypes to PostgreSQL: *geometry* and *geography*. The *geometry* datatype treats the earth as a two dimensional flat surface. The earth is projected onto a plane and geographical co-ordinates are mapped to a two dimensional cartesian co-ordinate system. When evaluating spatial predicates such as *ST_Covers*, *ST_Intersects* and spatial measurements such as *ST_Area*, *ST_Distance*, this datatype allows for high efficiency, however, it comes with a drawback. Since it treats the earth as a two dimensional plane, the computations are not precise over a large area as the spherical nature of the earth is not considered. To put this into context, consider an example of the shortest distance between two points. In a 2D cartesian plane, the shortest distance between two points is a straight line, while on a spheroid the shortest distance between two points is a geodesic (shortest distance on the great circle). In contrast to *geometry*, the *geography* datatype treats the earth as a three dimensional spheroid and all the computations are based on the spheroid. The computations are precise since they are done on a spheroid, but they are very slow compared to those on *geometry*.

We implemented the *geography* datatype and the corresponding geospatial predicates, such as *ST_Covers*, in the high-performance MMDB HyPer [78][1]. HyPer belongs to an emerging class of hybrid databases, which enable *real world awareness* in real time by evaluating OLAP queries directly in the transactional database. In HyPer, OLAP is decoupled from mission-critical OLTP either by using the *copy on write* feature of the virtual memory management or *multi version concurrency control* [118]. These snapshotting mechanisms enable *HyPerSpace* to evaluate geospatial predicates on rapidly changing datasets. We achieve much better performance compared to an open-source database PostgreSQL, a commercially available MMDB (System

[1]When saying HyPer, we are referring to the research version of HyPer developed at the Technical University of Munich.

A), and a successful key-value store (System B). This demonstration presents *HyPerSpace* and showcases that an interactive analysis of huge amounts of rapidly changing geospatial data is indeed possible.

2.0.2 HyPerSpace

Similar to what PostGIS is to PostgreSQL, *HyPerSpace* is a geospatial extension to HyPer. For geospatial data processing in *HyPerSpace*, we make use of the Google S2 geometry library[2]. This is not novel, since System B also uses the S2 library for evaluating geospatial predicates. The novelty of our system is the integration of geospatial functionalities into a high-performance MMDB with snapshotting mechanisms which makes it possible to evaluate geospatial predicates on rapidly changing datasets.

HyPerSpace supports the three geospatial datatypes `Point`, `LineString`, and `Polygon`. Most of the geospatial processing is done using the S2 library.

S2 decomposes the earth into a hierarchy of cells. It considers earth of radius 1, and encloses it in a cube that completely covers it. S2 projects a point on the earth's surface onto one of the cube's faces and finds the cell that contains it. The faces of the cube are the top level cells, which can be recursively divided into four children to obtain lower level cells. There are 30 levels in total, and cells at the same level cover equivalent areas on earth (e.g., level 30 cells cover approximately $1cm^2$ each). The cells are enumerated using the Hilbert space-filling curve. The Hilbert curve is hierarchical in nature and fits well with the decomposition of earth into cells. Hilbert space-filling curves are fast to encode/decode and they have a very desirable spatial property: they preserve spatial locality. This means that the points on earth that are close to each other are also close on the Hilbert curve. The enumeration of the cells gives a compact representation of each cell in a 64 bit integer called *CellId*. A *CellId* thus uniquely identifies a cell in the cell decomposition. Similarly, other spatial datatypes like `LineString` and `Polygon` can be approximated using cells.

The enumeration of cells in S2 is hierarchical, which means that a parent cell shares its prefix with its children. To check if a cell is contained in another, we simply need to compare their prefixes, which is a bit operation. This enables one to index points based on their *CellIds* and thus be able to retrieve points contained in a certain cell by performing a prefix lookup on the index. B tree data structures are a good choice to index *CellIds*, since they support fast prefix lookups (essentially range scans). Additionally, B trees allow for high update rates, which is an essential requirement for real-time workloads.

2.0.3 Evaluation

All experiments were run *single threaded* on an Ubuntu 15.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM and all reported performance results are averages over ten runs.

[2]`https://code.google.com/archive/p/s2-geometry-library/`

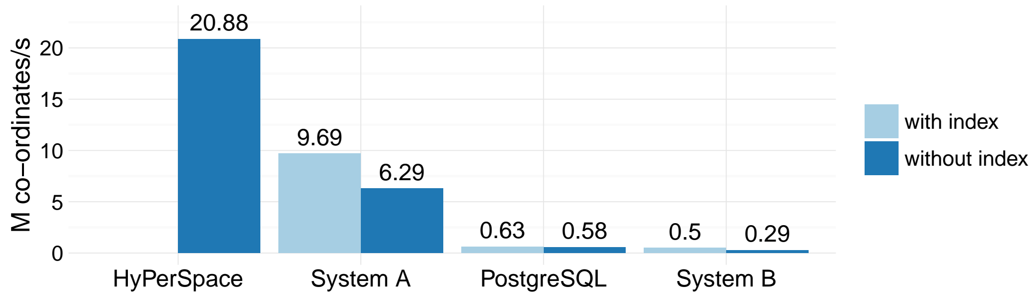


Figure 2.1: *HyPerSpace* vs. related systems: throughput of *ST_Covers* using lat/long co-ordinates

For evaluation, we used the NYC Taxi Rides dataset consisting of approximately 1.1 billion rides taken in the city from January 2009 until June 2015. The dataset includes the pickup and dropoff locations (latitudes and longitudes), pickup and dropoff times, and various details about the trip, such as distance, payment type, number of passengers, various taxes, tolls, surcharge, tip amount, and total fare. For privacy reasons, it does not contain details about drivers or passengers. The exact route taken for the trip is also not available. We needed to clean the dataset as some of the pickup or dropoff locations did not make sense as they were way outside NYC. We cleaned such records from the dataset and only considered rides that originated between longitude values -70.00 and -80.00, and latitude values 35.00 and 45.00. For evaluation, we made use of the taxi data for the month of January 2015. The cleaned dataset for January 2015 contains a total of 12505344 records.

We compared *HyPerSpace* with the following related systems: System A, System B, and PostgreSQL 9.4.5 (postgis-2.2.0). Since PostgreSQL does not support intra-query parallelism, we configured all systems to run single threaded. For evaluation purposes, we find how many rides originated from Midtown Manhattan in January 2015. In SQL notation, the following query is issued:

```
select count(*)
from   nyc,pickups_jan_2015
where  ST_Covers(nyc.geog,pickups_jan_2015.geog)
       and borough='Manhattan'
       and neighborhood='Midtown';
```

With the exception of System B, with NoSQL syntax, the query looks similar on all systems.

Figure 2.1 shows the throughput of the *ST_Covers* predicate for all of the systems. System A, System B, and PostgreSQL achieve better performance when using appropriate index structures. Particularly System B, which also makes use of the Google S2 geometry library, benefits from its index on points. System B's index is basically a B tree on the 64bit *CellIds*. System B computes an exterior covering of the polygon using the S2 library. That covering consists of cells at various levels (i.e., of different sizes). For each

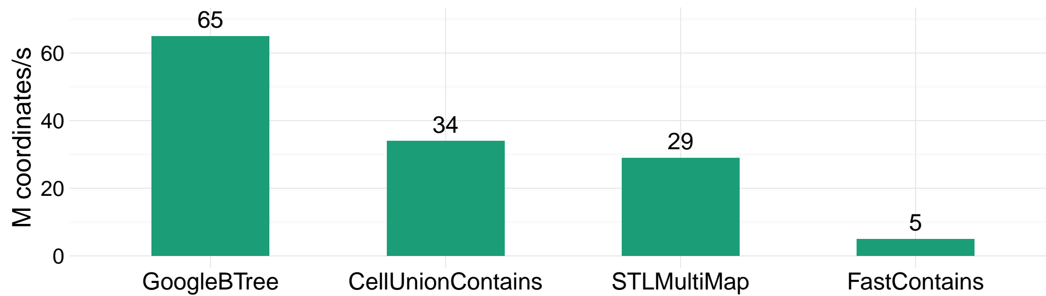


Figure 2.2: Microbenchmark results: throughput of *ST_Covers* using lat/long co-ordinates

cell of this covering, it then performs a prefix lookup in the B tree (essentially a range scan) and evaluates qualifying points for actual containment in the polygon. System B suffers heavily from its document-based storage layout, since it needs to parse GeoJSON documents at runtime.

HyPerSpace completes the query in 550ms and thus achieves more than twice the performance of its closest competitor, which is System A with an index on points (1290ms). We have not evaluated *HyPerSpace* with an index on points yet, but ran multiple microbenchmarks outside of *HyPerSpace*. All microbenchmarks were implemented in C++11 and compiled with gcc 4.9.2 with `-O3` and `-march=native` settings. We compared the implementation *CellUnionContains* that we used in *HyPerSpace* as well as *FastContains*, which is a modified version of the `S2Loop.Contains` implementation that skips the initial bounding box check, to the two index-based implementations *GoogleBTree* and *STLMultiMap*.

Figure 2.2 shows the throughput of the *ST_Covers* predicate for the different implementations. *GoogleBTree*, which is an implementation similar to System B's index, completes the workload in 191ms. In the *GoogleBTree* implementation, we first compute exterior and interior coverings for the given polygon and then perform a range scan in a Google B tree[3] for each cell of the exterior covering. For each qualifying point, we check whether the point is contained in the interior covering, which is essentially a binary search on a sorted vector of *CellIds*. Only if a point qualifies the exterior, but not the interior covering, an exact containment check using our modified implementation of the `S2Loop.Contains` function needs to be performed. The other index-based implementation *STLMultiMap* takes twice as long (425ms) as *GoogleBTree* to complete the workload, even though it uses the same approach. In C++11, the `stl::multi_map` interface that we used in this case is implemented by a RB tree, which is less efficient for range scans. It is well known that a B+ tree would yield even higher rates for range scans than a B tree. However, for the sake of expediency and reproducibility of our measurements, we have used the B tree implementation provided by Google instead of a custom B+ tree implementation. Once we integrate this approach into *HyPerSpace*, we will make use of an optimized B+ tree implementation. The difference in performance between the two implementations *GoogleBTree*

[3] `https://code.google.com/archive/p/cpp-btree/`

and *STLMultiMap* shows that the overall runtime of this approach is heavily influenced by the actual index structure used.

The approach *CellUnionContains* completes the workload in 367ms, compared to 550ms when implemented within *HyPerSpace*. The overhead is mostly caused by function calls that are issued for each of the 12M points. *CellUnionContains* is a straightforward approach. It first computes the bounding box and exterior and interior coverings for the given polygon. For each of the points, *CellUnionContains* then performs the following steps: First, it checks whether the point is within the bounding box. If that is the case, it checks for containment in one of the cells of the exterior covering. Analogous to the containment check for the interior covering, this essentially comes down to a binary search. Then the *CellUnionContains* approach continues analogous to the *GoogleBTree* approach by checking the interior covering and performing the exact containment check if necessary. By properly using the S2 mechanisms, our *CellUnionContains* approach achieves a slightly better performance than the index-based *STLMultiMap* approach, even though we have to loop over all of the 12M points.

2.0.4 Visualization using HyPerSpace

We created an interactive web interface, called *HyPerMaps*, that demonstrates the outstanding geospatial processing performance of *HyPerSpace* on the NYC Taxi Rides dataset. The user interaction concept of *HyPerMaps* is designed to minimize the requirement of users' expertize with the explored data. The ability of *HyPerSpace* to answer queries with typically sub-second latency enables tight feedback loops. It supports users during query formulation and encourages an iterative approach. During filtering of the data, users can rely on datatype dependent elements, which provide context-based information like value distributions or geographic locations in real time. Users can draw polygons on the map to filter points geographically. Subsequently, users can combine different graphical and textual representations to create an informative and intuitive visualization. During this data exploration process, *HyPerMaps* will automatically compute updated results reflecting the current state of the user interface as well as the underlying dataset.

Figure 2.3 shows *HyPerMaps* visualizing the taxi dataset. On the left, various tiles allow users to specify filters on the data, which will be immediately translated into SQL code as illustrated on the top. This binding works in both directions—manually written SQL code will be translated into corresponding tiles. Users can choose between a heat map and pins to display selected points on the map. On the right, *HyPerMaps* shows aggregated information about selected points in tabular or in chart form.

2.0.5 Take-away message

In this chapter, we presented *HyPerSpace*, a geospatial extension to the MMDB HyPer. Our implementation of the *ST_Covers* predicate achieves a much

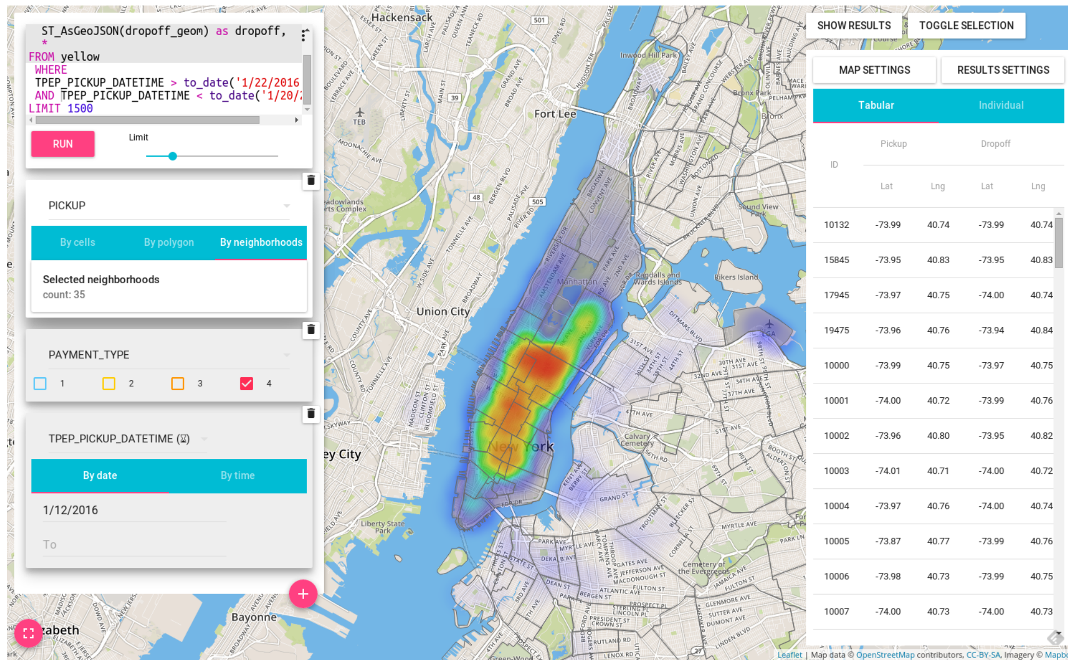


Figure 2.3: Interactive visualization of a real-time replay of NYC taxi rides using *HyPerMaps*

lower latency than corresponding implementations in related systems, without using any index structures. Additionally, we found that using index structures optimized for range scans such as B trees or B+ trees on *CellIds*, can yield even lower latencies. In this work, we have shown that it is indeed possible to build real-time visualizations on geographical data using a general-purpose database system instead of a custom hand-written solution that takes much longer to build and is harder to maintain. The novelty of our system is the integration of geospatial functionalities into a high-performance MMDB that allows for efficient snapshotting of the current database state. Our contribution also includes the careful use of the features of the Google S2 geometry library, thereby achieving much lower latencies than related systems. This makes it possible to evaluate geospatial predicates on high throughput data streams in real time. To demonstrate this, we created a web interface that allows users to interactively explore the NYC Taxi Rides dataset while the data is being replayed at various speeds. Our work also shows that features (such as value distributions) of the entire dataset, including the most current data, can be used to populate UI elements, thereby supporting users in creating meaningful (aggregated) real-time visualizations.

Chapter 3

Modern Spatial Systems

Excerpts of this chapter have been published in [128].

3.1 Introduction

The era of big spatial data has lead the research community to focus on developing systems that can efficiently analyze and process spatial data. Systems to manage and analyze big data have existed for a long time (Hadoop [10], Impala [89], Spark [209]), however, spatial support in these systems had not existed. This lead to various Hadoop based spatial systems being developed (HadoopGIS [2], SpatialHadoop [40]). Similarly, there have been plenty of spatial processing and analytics systems that have been developed for Spark (SpatialSpark [197], GeoSpark [203], Simba [192], LocationSpark [173], and Magellan [166]). Spatial extensions for databases, have seen a similar trend with Oracle Spatial [126], MemSQL [112], Cassandra [19], and HyPer[129]. The general approach of building such systems is *on-top*, *from-scratch* and *built-in* and has been well documented in [41].

In this chapter we present

- A brief survey of available modern spatial analytics systems, including two new systems that have not been covered in literature previously
- A thorough performance evaluation of the available systems using a real world dataset, focusing on major features that are supported by the systems

The rest of this chapter is structured as follows: Section 3.2 gives the motivation to carry out this study. Section 3.3 presents the spatial queries domain explaining which queries we consider for this study. Section 3.4 summarizes a broad variety of existing big spatial data analytics systems. Section 3.5 gives the details about the experimental setup and datasets used for evaluation. Section 3.7 gives the details about the performance evaluation of the systems which is followed by the conclusions in Section 3.8.

3.2 Motivation

The aim of our study is to compare five Spark based systems namely, SpatialSpark, GeoSpark, Simba, LocationSpark, and Magellan, using four different datatypes (*points*, *linestrings*, *rectangles*, and *polygons*) and five different spatial queries (range query, kNN query, spatial joins between various geometric datatypes, distance join, and kNN join). Although we include SpatialHadoop and HadoopGIS in the brief survey of modern big data spatial analytics systems, we decided to omit them from evaluation. We only consider spatial analytics systems based on Spark for evaluation since Hadoop based systems like SpatialHadoop and HadoopGIS have consistently been shown to perform poorly compared to Spark based systems in existing work.

There have been multiple studies which compare these systems based on various queries and performance metrics but all of them are incomplete or only compare a limited features of the systems. SpatialSpark [197] implements two join algorithms, **point-in-polygon** and **point-to-polyline** distance join, and evaluates the two implementations. In the extended study [198], **point-in-polygon** and **polyline-with-polyline** intersection join performance is evaluated for Hadoop-GIS, SpatialHadoop, and SpatialSpark. In [203], GeoSpark compares itself with SpatialHadoop for **linestring-polygon** intersection join and kNN query performance. In [173] LocationSpark compares the kNN join performance against the state-of-the-art kNN join algorithms. Simba [192] evaluates itself with a variety of systems including Hadoop-GIS, SpatialHadoop, SpatialSpark, GeoSpark, and the state-of-the-art kNN join algorithms only on the point data type. Both Simba and LocationSpark support kNN joins but they have not been evaluated against each other. Simba does not support linestring and polygon datatypes yet. The join and range query performance comparison for these geometric objects are missing. Also, Simba only considers a small window of **selection ratio** for range queries, and only compares itself with SparkSQL variant for these windows. Moreover, all the performance comparison in the aforementioned studies were done using a **large cluster**, and a **scalability** study of these systems is missing.

Meanwhile, some of these systems have been actively developed and many optimizations have been added. Since the previous studies, GeoSpark has introduced many new datatypes and has also added a query optimizer. Also, Magellan [166] has gathered attention in the Free And Open Source Software for Geospatial[1] (FOSS4G) committee and has not been evaluated in any existing study.

To summarize, these are some open ended questions missing in the existing literature:

- How do the modern *in-memory* spatial analytics systems perform for all the **major** features that they **support**?
- How do these systems perform for **all** possible spatial join combinations of various geometric data types?

[1] http://foss4g.org/

- **Where** is the time actually spent during various join queries?
- How well do these systems perform for **different** selection ratios for range queries for **different** geometric objects?
- What are the **memory costs** related to the systems?
- Do the memory costs have any **impact** on query performance?
- How well do these systems **scale** for the queries that they support?

We aim to fill this gap and compare the modern in-memory spatial analytics systems to present a complete study, while the experiment files and setup provided will make it easier for researchers to benchmark these systems against future spatial analytics systems or spatial algorithms.

3.3 Queries

For the queries we consider four geometric features or datatypes: *points*, *linestrings*, *rectangles* and *polygons* subsets (or all) of which are supported in most of the evaluated systems.

The queries considered for evaluation are: single relation operations (range query, kNN query) and join operations (distance join, spatial joins and kNN join). There can be other spatial queries such as computational geometry operations, spatial data mining operations, and raster operations. These queries are well-defined in [41]. We do not consider these queries since the evaluated systems do not support these queries and evaluating systems that do is out of the scope of this chapter. We will now briefly describe the set of queries that we consider for evaluation.

3.3.1 Range Query

A range query takes a range R and a set of geometric objects S, and returns all objects in S that lie in the range R. Formally,

$$Range(R, S) = \{\ s | s \in S,\ s \in R\ \}.$$

3.3.2 k Nearest Neighbors Query

A kNN query takes a set of points R, a query point q, and an integer k $\geq$ 1 as input, and finds the k nearest points in R to q. Formally,

$$kNN(R, q) = \{T \subseteq R, |T| = k \wedge \forall t \in T,$$
$$r \in R - T : d(q, t) \leq d(q, r)\}.$$

3.3.3 Spatial Join

A spatial join takes two input sets of spatial records R and S and a join predicate θ (e.g., overlap, intersect, contains, within, withindistance) and returns a set of all pairs (r,s) where r $\in$ R, s $\in$ S, and θ is true for (r,s). Formally,

$$R \bowtie_{\theta} S = \{ \ (r,s) \mid r \in R, \ s \in S, \ \theta(r,s) \ is \ true \ \}.$$

A distance join is a special case of spatial join where the join predicate is *withindistance*. For the sake of clarity, we will refer to distance join as is and do not include it in spatial joins.

3.3.4 k Nearest Neighbors Join

A kNN join takes two input sets of spatial records R and S and an integer k $\geq$ 1 and returns for all objects r $\in$ R theirs k closest neighbours in S. Formally,

$$R \bowtie_{kNN} S = \{ \ (r,s) \mid r \in R, \ s \in kNN(S,r) \ \}.$$

3.4 Spatial Analytics Systems

In this section, we briefly review the cluster-based systems that support spatial data management, queries and analytics over distributed data using a cluster of commodity machines. We study the various features, data partitioning and indexing schemes, and queries that are supported in these systems. Table 3.1 gives an overview of the features of the different spatial analytics systems.

An important point to make here is that distributed systems, generally, use a two level indexing scheme consisting of a global index in the master node and multiple local indices in the slave nodes. Figure 3.1 shows the generalized indexing scheme. The input file is first partitioned based on a partitioning scheme, each partition is then indexed using a specialized spatial index (e.g., R-tree, R+-tree, Quadtree etc.), and finally these local indices are indexed in a global index on the master node. This is also known as the preprocessing phase wherein the data is loaded into the distributed file system, and data is partitioned logically or physically which is useful for query processing. The quality and performance of partitioning techniques have been thoroughly covered in [39].

3.4.1 Hadoop-GIS

Hadoop-GIS [2] is a scalable and high-performance spatial data warehousing system for running large-scale spatial queries on Hadoop. It was the first system based on Hadoop to support spatial queries. Hadoop-GIS treats Hadoop as a black box and relies on underlying architecture for processing. For partitioning, Hadoop-GIS uses a *uniform grid* to partition the space first and then map the objects to the tiles. If partitioning creates some high density

Table 3.1: Overview of features in spatial analytics systems

	Hadoop-GIS	**Spatial Hadoop**	**SpatialSpark**	**GeoSpark**	**Magellan**	**SIMBA**	**Location Spark**
In-Memory Processing	No	No	Yes	Yes	Yes	Yes	Yes
Language	HiveSP	Pigeon	N.A.	N.A.	Extended SparkSQL	Extended SparkSQL	N.A.
Partitioning Techniques	SATO Framework (Multiple partitioning techniques)	Quad, STR, STR+, K-d, Hilbert, Z-curve	Uniform, Binary-Split, STR	Quad, KDB, R-tree, Voronoi, Uniform, Hilbert	Z-curve	STR	Uniform, R-tree, Quad
Index	R*-tree	R-tree	R-tree	R-tree, Quadtree	None	R-tree	R-tree, Quadtree, IR tree
Datatypes	Point, Rectangle, Polygon	Point, Rectangle, Polygon	Point, LineString, Rectangle, Polygon	Point, Rectangle, Polygon, LineString	Point, LineString, Polygon, MultiPoint, MultiPolygon	Point	Point, Rectangle
Queries	Range, Spatial Joins	Range, kNN, Spatial Joins	Range, Spatial Joins	Range, kNN, Spatial Joins, Distance Join	Range, Spatial Joins	Range, kNN, Distance Join, kNN Join	Range, kNN, Spatial Join, Distance Join, kNN Join

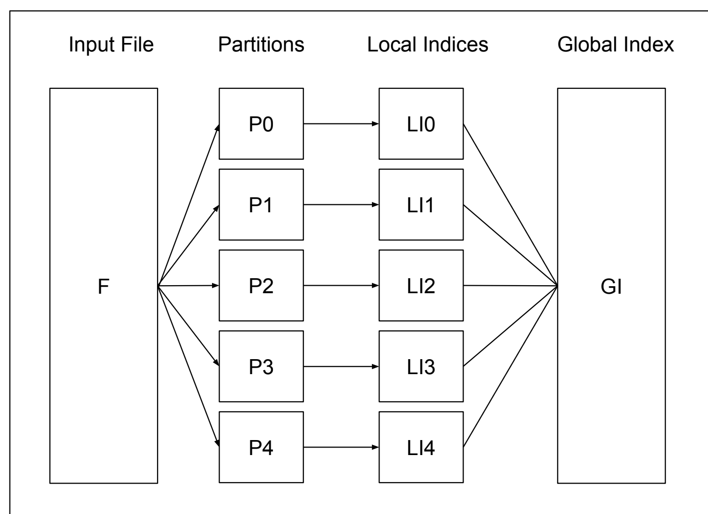


Figure 3.1: A generalized indexing scheme for distributed spatial analytics systems

tiles, these tiles are broken down into smaller tiles to handle this data skew. In [186], Hadoop-GIS added more partitioning techniques to provide flexibility to the system. Here, the input data is partitioned in four steps: Sample, Analyze, Tear and Optimize (SATO). 1-3% of the data is sampled and the density distribution of the dataset is computed. The Minimum Bounding Rectangles (MBR) from the sampled dataset are fed to the Analyzer which decides the optimum global partitioning scheme for the global partitions. In the Tear phase each global partition is further partitioned to create local partitions. The physical partitioning takes place in this step. In the Optimize phase the data is re-scanned and statistics about the partitions are collected to build the multi-level index. This is an example of dynamic partitioning and indexing, which takes into consideration the distribution and skew of spatial data. These indices are used to process the queries supported: range and spatial join queries. Hadoop-GIS supports points, rectangles, and polygons. Hadoop-GIS extends HiveQL with spatial query support and integrates the spatial query engine into Hive.

3.4.2 SpatialHadoop

SpatialHadoop [40] is a full-fledged MapReduce framework with native support for spatial data. Unlike Hadoop-GIS, SpatialHadoop is *built-in* Hadoop.

It enriches Hadoop with spatial constructs and awareness of spatial data inside the core functionality of Hadoop and is thus able to obtain better performance than Hadoop-GIS since it has to deal with no layer overhead. SpatialHadoop partitions the dataset into n partitions that confirm to three conditions (i) each partition should fit one HDFS block (64MB), (ii) the objects close to each other in space should be assigned to same partition and, (iii) all partitions should be of similar size for load balancing purposes. The input dataset can be partitioned and indexed using either Grid Index, R-tree or R+-tree. Since, R-tree performs the best in most cases as reported in the publication, we will describe the partitioning phase using R-tree. SpatialHadoop bulk loads a sample from the input dataset into an in-memory R-tree using the Sort-Tile-Recursive (STR) algorithm. It computes the number of partitions, n, based on the size of the input file. It then fills the R-tree with degree d ($\sqrt{n}$) using the STR algorithm. The STR algorithm ensures that the tree is balanced and the degree d of the tree ensures that there are at least n nodes in the second level of the tree. The second level of the tree is used to physically partition the input dataset. In the physical partitioning step, each input record is assigned to a partition which requires the least enlargement to cover the record. After physical partitioning, each partition is bulk loaded into an R-tree using the STR algorithm and dumped to a file. The block in local index file is annotated with the MBR of its content. In the global indexing phase, all local indexed files are concatenated and the global index is created by bulk loading all the blocks into an R-tree using the annotated MBR as the index key. SpatialHadoop extends *FileSplitter* and *RecordReader* in Hadoop to support spatial records. *SpatialFileSplitter* uses the global index to prune out blocks that do not contribute to the query result. *SpatialRecordReader* exploits the local index in the partitions received from *SpatialFileSplitter* to efficiently process the query. It also extends Pig Latin, called Pigeon, with spatial support. SpatialHadoop supports range queries, kNN queries, and spatial joins. It has support for point, rectangle, and polygon datatypes.

3.4.3 SpatialSpark

SpatialSpark [197] is a lightweight implementation of spatial support in Apache Spark. It targets in-memory processing for higher performance. SpatialSpark supports multiple geometric objects including points, linestrings, polylines, rectangles, and polygons. It supports multiple spatial partitioning schemes fixed grid, binary split and STR partitioning. For indexing, SpatialSpark uses an R-tree. SpatialSpark offers a variety of operations on spatial datasets including range queries on all types of geometric objects, spatial joins between various geometric objects and distance joins. It supports 1NN queries but does not support kNN queries and kNN joins.

3.4.4 GeoSpark

GeoSpark [203] is an in-memory cluster computing framework based on Apache Spark for processing large spatial data. It consists of three layers: (i) Apache Spark Layer, (ii) Spatial RDD Layer, and (iii) Spatial Query Processing Layer. GeoSpark extends the core of Apache Spark to support spatial datatypes, indexes, and operations. GeoSpark extends the resilient distributed datasets (RDDs) to support spatial datatypes. Apache Spark Layer is responsible for native functions that are supported by Spark such as load/save data to persistent storage. Spatial RDD layer extends Spark with spatial RDDs (SRDDs) that can efficiently partition SRDD elements across machines and also introduces parallelized spatial transformations. GeoSpark introduces support for various types of spatial objects: points, linestrings, rectangles, and polygons. It also provides a Geometrical Operations library which has geometrical operations such as *Ovelap()* (find overlapping objects), *MinimumBoundingRectangle()* which returns the MBR of either every object in the SRDD or largest MBR encompassing every object in the SRDD, *Union()* which returns the union of all polygons in the SRDD. GeoSpark also comes with a query optimizer. GeoSpark supports multiple partitioning schemes including, Quadtree, KDB tree, R-tree, Voronoi, fixed grid, and Hilbert partitioning. GeoSpark has two indexes available, R-tree and Quadtree. GeoSpark has support for range queries, spatial join queries, and kNN queries. GeoSpark does not support kNN joins.

3.4.5 Magellan

Magellan [166] is a distributed execution engine for spatial analytics on big data. It leverages modern database techniques in Apache Spark like efficient data layout, code generation, and query optimization in order to optimize spatial queries. Magellan extends SparkSQL to accommodate spatial datatypes, geometric predicates, and queries. Magellan has support for points, linestrings, rectangles, polygons, multipoints, and multipolygons. Magellan supports range queries and spatial joins but does not support kNN queries, distance joins, and kNN joins. Magellan also adds geometric predicates such as intersects, within, and contains. Magellan uses on the fly indexing of the geometrics objects but can also leverage the indices if they were persisted earlier. Magellan uses Z-order curve for indexing and appends a column to the dataset with the Z curve values. To perform join queries efficiently, Magellan intercepts the query plan and overwrites it to use Z-curve index. It uses an inner join on the Z-curve and a predicate filter on top of the inner join, instead of a cross join between two input datasets. Magellan does not support spatial partitioning and leverages Spark's built-in partitioner to partition the dataset.

3.4.6 SIMBA

Simba [192] (Spatial In-Memory Big Data Analytics) is a distributed in-memory analytics engine that supports spatial queries and analytics over

big spatial data in Spark[2]. Simba extends Spark SQL to support spatial operations. Simba also adds spatial indices in RDDs and SQL context, which helps in reducing query latencies and increasing analytical throughput by executing queries in parallel. It also introduces logical and physical optimizers to select better query plans. Tables are represented as RDDs of records of the table, thus indexing records of the table means indexing elements of the RDDs. To partition the data, Simba uses a similar strategy as SpatialHadoop where an R-tree is constructed by sampling the input dataset and filled using the STR algorithm to get the first level of the tree that represents the partition boundaries. These boundaries are only the MBR of the sampled set, which are extended as each record is added to the partition. Simba provides flexibility to the user to specify its own partitioning scheme as well, since the *Partitioner* abstract class in Spark allows users to specify their own partitioning strategy. For indexing within an IndexRDD, Simba uses an R-tree by default. Finally, a global index is constructed by using the partition boundaries from the partitioner and statistics from the local index. Simba supports a variety of queries which include, range (rectangle and circle) queries, kNN queries (on points), distance joins (between points), and kNN joins (between points). Simba does not support spatial joins.

3.4.7 LocationSpark

LocationSpark [173] is a spatial data processing system based on Apache Spark. It provides a wide range of spatial features. It supports a rich set of spatial queries: range queries, kNN queries, spatial joins and kNN joins. LocationSpark introduces a spatial RDD layer named *LocationRDD* which can be cached in memory. LocationSpark has a query scheduler component which can detect if there is a query skew, by actively collecting statistical information from each partition. If it detects a hotspot partition for a query, a cost model evaluates the overhead of repartitioning and takes suitable action. For data partitioning, similar to other systems, LocationSpark samples the input dataset and partitions data accordingly. A user has the flexibility to choose between either a uniform grid or Quadtree as the partitioning scheme. It also provides flexibility for local indices. A user can choose between Fixed-Grid, R-tree, Quadtree, or an IR-tree for indexing the data locally within a partition. Furthermore, LocationSpark also has a spatial bloom filter termed *sFilter* embedded with the global and local indices to prune out more partitions for a query, which helps in reducing network communication costs. LocationSpark supports range queries (on points), kNN queries (on points), spatial joins between points and rectangles, and kNN joins (between points).

[2]Note: The latest stable Simba release is the standalone package outside of Spark (i.e. a library running on top of Spark) and we benchmark it and not the version in the original publication which is built inside Spark core

Table 3.2: Evaluated systems, their compatible Spark version, and defaults for the experiments

System	Version	Amazon EMR and Spark Version	Spatial Partitioning	Index
SpatialSpark	1.0	emr-5.9.0 (2.2.0)	STR	R-tree
GeoSpark	v1.1.3	emr-5.9.0 (2.2.0)	Quadtree	R-tree
Magellan	v1.0.5	emr-5.9.0 (2.2.0)	Z-curve	Z-curve
LocationSpark	the first version	emr-4.9.3 (1.6.3)	Quadtree	Quadtree
Simba	Standalone package compatible with Spark 2.1.x	emr-5.7.0 (2.1.2)	STR	R-tree

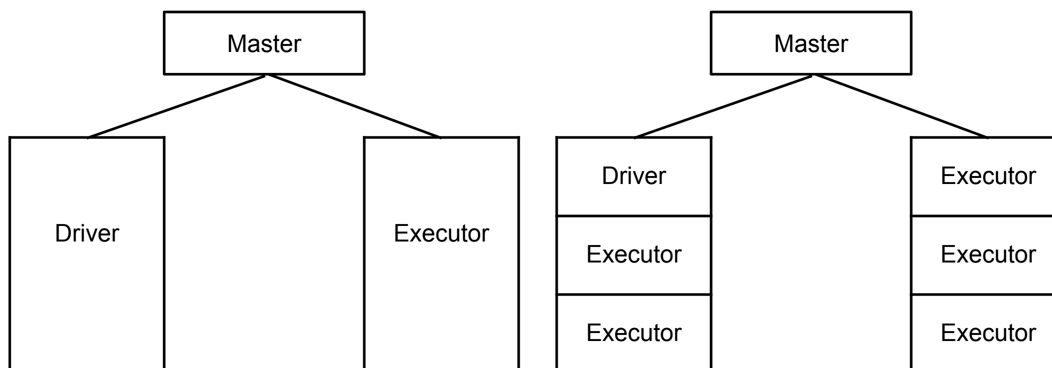
3.5 Experimental Setup

3.5.1 Cluster Setup And Tuning Spark

To evaluate the systems we deploy variable sized clusters on Amazon AWS. We make use of the Amazon Elastic Map Reduce (EMR) framework to deploy the Spark cluster. The master node which runs the YARN resource manager for the cluster is an EC2 instance of type `m4.xlarge` that has 8 vCPUs and 16 GB main memory. For slave nodes we make use of `r4.8xlarge` EC2 instances which have 32 vCPUs and 244 GB main memory. We also attach 100 GB general purpose SSDs to each slave node. We deploy 1, 2, 4, 8, and 16 slaves nodes in the cluster to evaluate the systems and their scalability. We will only count the slave nodes in the cluster since the master node only runs the resource manager and is in no way responsible for any computation for the applications. We deploy the Spark applications in *cluster-mode* where the Spark *driver* is deployed on one of the slave nodes as *Application Master*. Since not all systems were compatible with latest Spark release we deployed clusters in different EMR versions. Table 3.2 shows the different systems evaluated, their compatible Spark versions and the default values we used for the experiments. The default number of partitions in every system has been set to 1024 for every query unless stated otherwise.

Amazon EMR cluster model has a master node and slave nodes. The master node runs the resource manager, by default YARN, which manages the cluster resources. The Spark applications are deployed on the slave nodes. The Spark execution model has two main components, the *driver* and the *executors*. The *driver* breaks up the work into tasks and assigns them to the *executors*. By default, Amazon EMR launches the cluster with *maximizeResourceAllocation*, which means that if there are four slave nodes in the cluster, then one node is selected as the *driver* and the other three as *executors*. This means that 25% of the cluster resources are dedicated to the bookkeeping tasks that the driver performs and only 75% of the resources are available for processing data. Moreover, having only three *executors* with all cores per node assigned to these executors in the cluster is not the optimal setting and

Figure 3.2: *maximizeResourceAllocation* deployment vs a better deployment



often leads to poor HDFS throughput and failed Spark jobs[3]. We tuned every cluster in our experiments to utilize resources optimally by following the guidelines in the Cloudera Engineering Blog[4].

3.6 Tuning Amazon EMR and Apache Spark

As the resources increase in the cluster many decisions need to be made to utilize the resources optimally for Spark. These decisions include how many executors the cluster should have, how many cores and memory to assign to an executor. As we mentioned earlier in 3.5.1, we tuned every cluster to fully and optimally utilize all the resources using the guidelines in the Cloudera Engineering Blog [5]. Amazon EMR launches the cluster with *maximizeResourceAllocation*, which means that if there are two slave nodes in the cluster, then one node is selected as the *driver* and the other one as an *executor*. This means that 50% of the cluster resources are dedicated to bookkeeping tasks that the driver performs and only 50% of the resources are available for processing data. A better deployment would be to have multiple executors along with driver on one node. This is illustrated in Figure 3.2.

The parameters we are interested in computing are number of executors (*spark.executor.instances*), the number of cores to assign to each executor (*spark.executor.cores*), memory to assign to each executor (*spark.executor.memory*), number of cores to assign to the driver (*spark.driver.cores*), and memory to assign to the driver (*spark.driver.memory*). It has been shown canonically that assigning 4-6 cores to an executor is good to achieve maximum HDFS throughput. We will fix the cores assigned to each executor to 5. One core per node needs to be left for Hadoop and YARN daemons that run on each node.

[3]https://databricks.com/session/top-5-mistakes-when-writing-spark-applications

[4]http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[5]http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

Table 3.3: Spark configuration parameters

Parameter	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
spark.executor.instances	6	12	24	49	99
spark.executor.cores	5	5	5	5	5
spark.executor.memory	37G	37G	37G	37G	37G

Another thing to keep in mind is the *spark.yarn.executor.memoryOverhead* parameter which adds seven percent, by default, more memory from the resource manager when requesting for memory for an executor. So when specifying *executor-memory* parameter for Spark, it needs to be reduced by seven percent. Keeping these things in mind, let the number of nodes be *n*, number of cores in each node be *c*, and memory available per node be *m*. The total number of cores available in the cluster are thus *n**c*. The number of cores available for executors are *n**(c-1)* (keeping in mind one core per node is needed for Hadoop and YARN daemons). Since, we fix the number of cores per executor to 5 (for maximum HDFS throughput), the total number of executors in the cluster will then be *n**(c-1)/5* which we denote as *E*. The only parameter left to be computed is the memory to assign to each executor. There are *n* nodes in the cluster and the total number of executors are *E*. The number of executors that can run on one node are thus E/n (E_n). Each node has *m* amount of memory and *spark.yarn.executor.memoryOverhead* is 0.07 (seven percent). Thus memory that can be assigned to each executor is

$$\text{Memory per executor} = \frac{m}{E_n} - \left(\frac{m}{E_n} * 0.07\right) = \left\lfloor \frac{0.93 * m}{E_n} \right\rfloor$$

Note that the total number of executors also includes the *driver*, which runs as *Application Master* on YARN. The parameters *spark.driver.cores* and *spark.driver.memory* will be the same as that for the executors. Table 3.3 shows the parameter values for our clusters. To summarize, let

$$\text{Number of nodes} = \text{n}$$
$$\text{Number of cores per node} = \text{c}$$
$$\text{Total memory per node} = \text{m}$$

Table 3.4: Details of the datasets used for evaluation

Dataset	Geometry type	Number of records	Raw file size (GBs)	Total number of co-ordinates
OSM Nodes	Point	200 million	5.4	200 million
OSM Roads	LineString	70 million	18	803 million
OSM Buildings	Polygon	114 million	19	764 million
OSM Rectangles	Rectangle	114 million	14	573 million

then,

$$
\begin{aligned}
\text{Total number of cores} &= n * c \\
\text{Total number of cores for executors} &= n * (c - 1) \\
\text{Total number of executors (E)} &= \left\lfloor \frac{n * (c-1)}{5} \right\rfloor \\
\text{Number of executors per node } (\mathrm{E_n}) &= \left\lfloor \frac{E}{n} \right\rfloor \\
\text{Memory per executor} &= \frac{m}{E_n} - \left(\frac{m}{E_n} * 0.07 \right) = \left\lfloor \frac{0.93 * m}{E_n} \right\rfloor
\end{aligned}
$$

Also, Amazon EMR, by default takes the Task Configuration[6] values based on the master node (`m4.xlarge`). We had to overwrite these values so that the Task configuration values are based on slave nodes (`r4.8xlarge`).

3.6.1 Datasets

To evaluate the systems we make use of the Open Street Maps (OSM) dataset made available by [40]. The OSM dataset comprises of `All Nodes` (*Points*), `Roads` (*LineStrings*), and `Buildings` (*Polygons*) datasets. The full OSM dataset contains *2.3 billions* points on earth (`All Nodes`), *70 million* roads and streets around the world (`Roads`), and *114 million* buildings (`Buildings`). We sampled a subset of *200 million* points from `All Nodes` dataset. We sampled the points from the dataset because some join results can be arbitrarily large with the full dataset and will not fit entirely in *driver's* memory. To extract the subset we make use of the `shuf` command in Linux. In addition to these datasets, we also generated a *Rectangle* dataset which is generated from the `Buildings` dataset by computing the minimum bounding rectangles of the polygons. We also needed to clean the datasets since some of the geometric objects did not comply with the OGC standard for spatial objects. To clean the datasets we used *Java Topology Suite* (JTS[7]) library which is OGC compliant. It is important to point out here is that certain systems only expect

[6]https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-task-config.html
[7]https://github.com/locationtech/jts

geometries in a particular format (such as Well-Known Text) as input, so we had to pre-process files in order to make them suitable as inputs for the different systems. In some cases the files sizes reduced because we had to strip the metadata from the files. Table 3.4 shows the details of the datasets used for evaluation. The datasets used for evaluation are available on our server[8] and all experiment files are available on GitHub[9]. We will refer to the datasets as Point, LineString, Rectangle, and Polygon datasets from now on.

We also ran the experiments with the US Census TIGER dataset provided by [40]. We used the LineString dataset from the TIGER dataset which contains approximately *70 million* linestrings in US. There are other datasets in TIGER but they are limited in size (less than 2 million spatial objects). To have a larger dataset to join with, we generated a rectangle dataset by computing bounding boxes of these linestrings. We also sampled *170 million* points that are in US from the OSM dataset to join with these datasets.

3.6.2 Spark Memory Management Model and Caching RDDs

Spark's memory management model is split into two parts, storage memory and execution memory. The amount of memory assigned to Spark after reserving memory for internal data structures is split into execution and storage memory. Execution memory refers to the memory that is assigned for computations such as joins, aggregations, shuffles, and sorts. Storage memory is the amount of memory that is used for caching the user datasets in memory. The total assigned memory is split 50/50 between storage and execution memory and is managed by *spark.memory.storageFraction* parameter. If no execution memory is needed, storage can acquire all the memory and vice versa. Execution memory can evict (spill to disk) storage blocks in case more execution memory is needed, and execution memory is immune to eviction.

Spark allows the user to cache (or persist) the RDDs in memory if they are used multiple times for computation. If sufficient storage memory is available in the cluster it is advisable to cache such RDDs. One purpose of choosing the AWS instance `r4.8xlarge` is that it comes with large memory so the RDDs can be cached. Even if RDDs do not fit in the memory in deserialized form, they can be serialized and persisted in memory. Most of the evaluated systems come with a custom serializer for the spatial RDDs which is based on Spark's KryoSerializer. Caching such RDDs is system specific and differs quite a bit because of different design choices. GeoSpark has an abstract SpatialRDD layer. It consists of three RDDs: *RawSpatialRDD*, *SpatialPartitionedRDD*, and *IndexedRDD*. When a SpatialRDD is initialized (e.g., new PointRDD()), the *RawSpatialRDD* gets populated. *SpatialPartitionedRDD* can be initialized by specifying the spatial partitioning technique, and then calling some action on the RDD. Initially for every type of query we keep the *RawSpatialRDD* in `MEMORY_ONLY` persistence level. Once the *SpatialPartitionedRDD* is generated, we unpersist the *RawSpatialRDD* as it is not needed in any query operation and keeping it in memory even in serialized form

[8] http://osm.db.in.tum.de/
[9] https://github.com/varpande/spatialanalytics

just incurs extra memory cost. We then populate the *IndexedRDD* and keep it along with the *SpatialPartitionedRDD* in memory at all times for all query operations. In Magellan, we make use of the dataframe API. We only persist a dataframe that contains the spatial object and the index for the object. LocationSpark has a *LocationRDD* and *QueryRDD* abstraction layer. We only cache these *LocationRDD* and *QueryRDD* for all query operations. In Simba, we make use of the dataframe API. For single relation operations we cache a dataframe with an index. For join operations we do not build an index on the dataframe, since Simba spatially partitions and indexes the datasets on the fly inside the join algorithms and does not utilize the persisted index. Spatially partitioning the data and building index on the dataframe is an overhead in case of join operations. For all operations, SpatialSpark first builds an RDD which simply reads the input dataset and parses the spatial object from the Well-Known Text (WKT) representation. It then builds a spatial RDD which has a unique ID for every spatial element. We cache these RDDs for all the queries in SpatialSpark.

3.6.3 Performance Metrics

To measure and compare performance for single relation queries, we submit a batch of 100 queries and compute the throughput of the systems in queries/minute.

To measure and compare the performance of different systems for join queries, we make use of six performance metrics:

- Preparation time: The *preparation time* is the total time spent by the system in reading the two datasets from HDFS, partitioning the input datasets, and indexing the partitions.
- Join time: The *join time* is the total amount of time spent by the system to complete the join query. This metric is a useful indication for use cases where the datasets are already indexed and the join queries need to run multiple times.
- Total runtime: The *total runtime* is the total of the two aforementioned performance metrics. The *total runtime* gives an end-to-end query performance of the query.
- Shuffle write costs: the *shuffle write* is the sum of all written serialized data on all executors before transmitting to other executors at the end of a stage.
- Shuffle read costs: the *shuffle read* is the amount of read serialized data on all executors at the beginning of a stage.
- Peak Execution Memory: the *Peak execution memory* is the maximum amount of memory used at any point in time for execution of a query.

In addition to the performance metrics, we also report the index sizes for the different datasets. Please note that we only report the cumulative sizes of

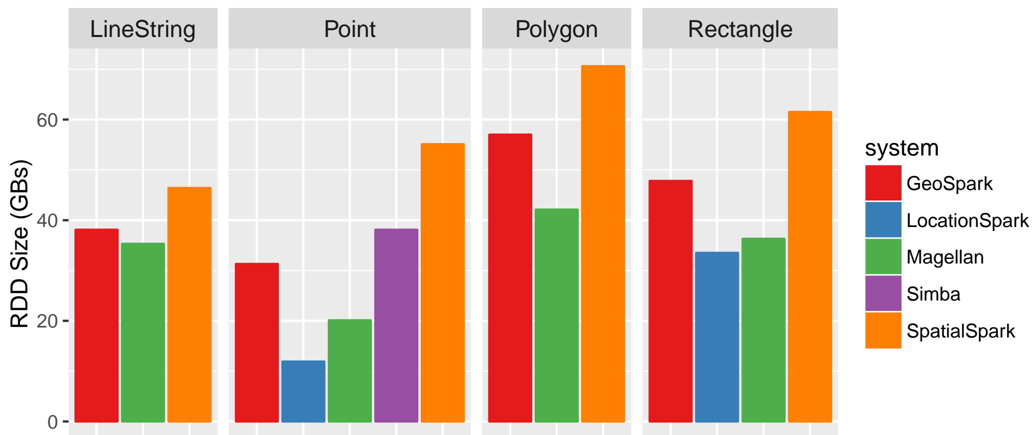


Figure 3.3: Memory footprint for various datasets

the local indices and chose to skip the size of the global index, which usually is very small (few KBs).

3.7 Evaluation

3.7.1 Memory Costs

In this section, we report the in-memory consumption of the various data structures by caching the respective data structures and observing the Storage tab in the Spark Web UI[10]. Note that memory consumption for RDDs in Spark cannot be obtained programmatically, as it can only report approximate memory consumption of RDDs, hence these values are not available in the Scala codes for the systems in the GitHub page.

Figure 3.3 shows the raw spatial RDD sizes for various datasets for the systems. It is normal for Java objects to consume more memory than raw file size on disk[11]. We see most of the systems consume almost 3x more memory for every dataset. Also spatial partitioned RDDs add additional overhead because most of them use a replication-based technique to handle boundary objects. As mentioned before, the common technique for these systems to generate partitions is to sample the dataset first and decide spatial boundaries for the partitions based on this sampled data. When the spatial objects are loaded from the file system, they are mapped to these partitions. When a spatial object overlaps with multiple partitions, it is replicated to these multiple partitions, which increases the memory cost. Another point to make here is that both GeoSpark and SpatialSpark store JTS Geometry objects in the raw spatial RDD. The difference in their memory consumption is because SpatialSpark also adds a unique ID to each element in the RDD, which accounts for a slightly higher memory usage.

[10]https://spark.apache.org/docs/latest/tuning.html
[11]https://spark.apache.org/docs/latest/tuning.html#memory-tuning

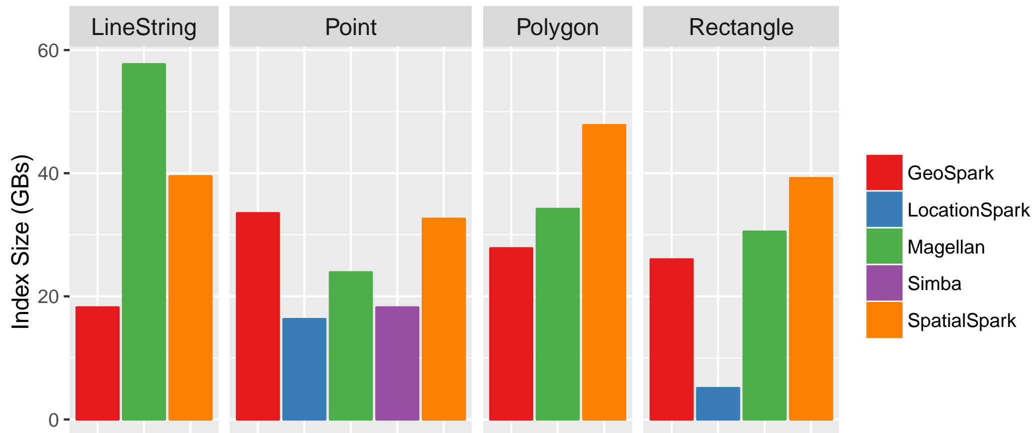


Figure 3.4: Indexing costs

Figure 3.4 shows the index sizes for various systems for the different datasets. Simba and LocationSpark have the lowest memory consumption for indices for the Point dataset. LocationSpark only keeps the point coordinates and its two MBR coordinates in the Quadtree, and thus the indexing cost is low. Simba serializes its index (default persistence level is *MEM_AND_DISK_SER*) and thus the index cost is very low. An unusual case is Magellan's LineString index which consumes close to 92 GB of main memory compared to its indices for other datasets. The reason is that Magellan generates Z-curve to approximate the shapes. For Points, it has to generate one cell value for each coordinate. Polygons and Rectangles can be approximated using large cells and hence the cell counts for these geometric objects is low. For LineStrings, Magellan ends up generating cells for each coordinate in each linestring record in the dataset. There are a total of 803 million coordinates in the LineString dataset and hence Magellan ends up generating the same amount of cells for the LineString dataset.

3.7.2 Range Query Performance

To evaluate range queries, we varied the selection ratio (σ) to cover a wide range for selection. We generated six ranges for each dataset that cover six selection ratios for each dataset. In this experiment, we loaded and indexed the datasets in every system and do not include the costs to prepare them. We submit a batch of 100 queries for each range for each type of dataset and evaluate the query throughput in queries/minute.

Figure 3.5 shows the range query performance for the different systems on a single node varying the selection ratio (σ) from 0.0001 to 100. Magellan does not have any optimization for range queries and ends up scanning all partitions for all selection ratios for all datasets. It serves as a baseline for other systems. For the point dataset, LocationSpark performs the best for selection ratios 0.0001 and 0.01. This is due to the *sFilter* (spatial bloom filter) that it puts on top of the global and local indices. The global index in the systems provide multiple overlapping partitions that intersect with the given

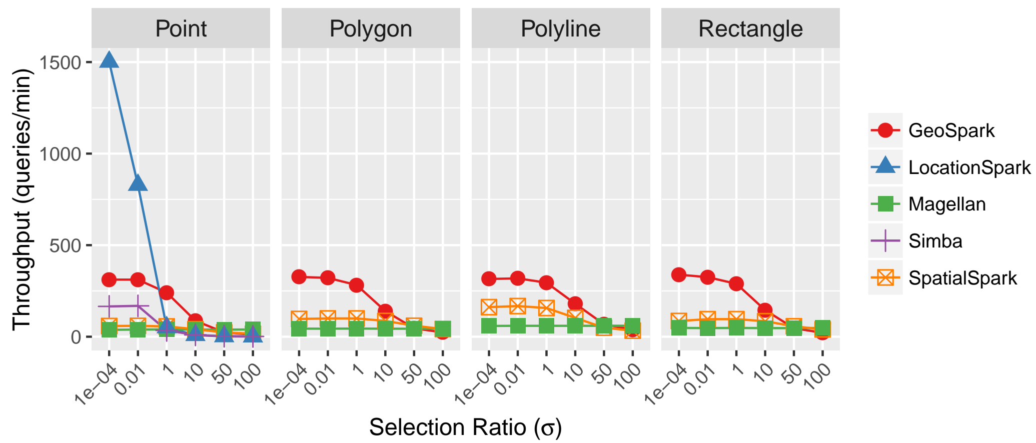


Figure 3.5: Range query performance on a single node for different selection ratio (σ)

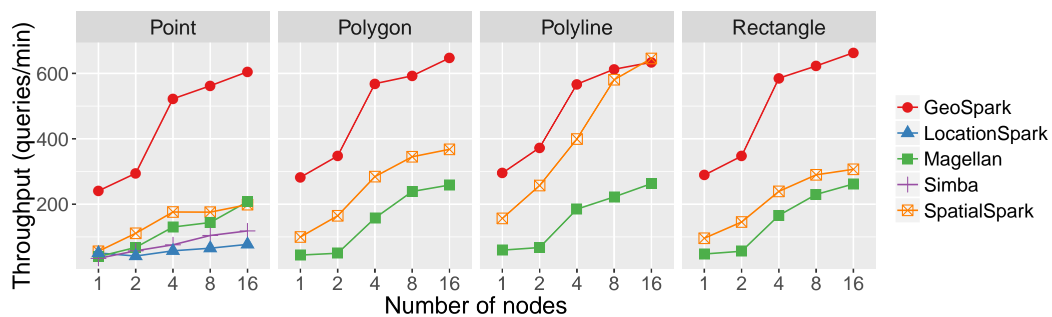


Figure 3.6: Range query performance for all geometric objects scaling up the number of nodes [selection ratio (σ) = 1.0]

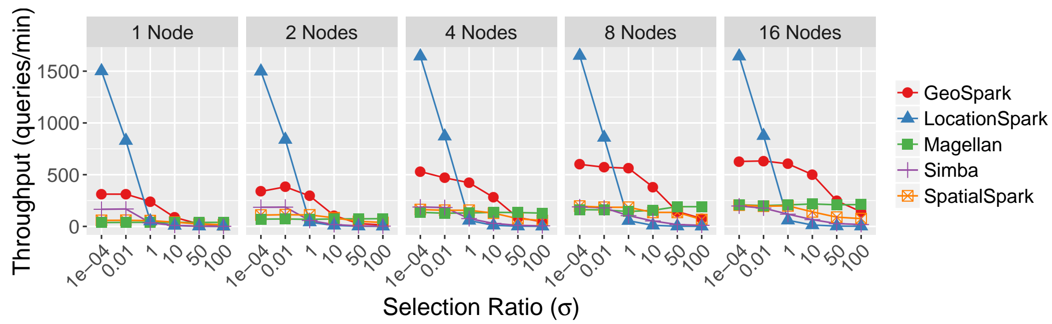
range. LocationSpark can further filter out partitions using the *sFilter* from the global index and local indices that do not contribute to the answer and avoids unnecessary scans of partitions and network costs. As the selection ratio increases, LocationSpark's performance degrades as well, similar to other systems, which is expected as more partitions need to be scanned for higher selection ratios. GeoSpark performs better than Simba, because it utilizes the deserialized *IndexedRDD* compared to the serialized index that Simba uses to minimize memory costs and for fault tolerance. Figure 3.6 shows the range query performance for different datasets, fixing the selection ratio (σ) to 1.0 and scaling up the number of nodes in the cluster. This experiment shows that all the systems scale well with the number of nodes. The scalability is not perfectly linear, but it is acceptable.

Figure 3.7 shows the range query performance for different datasets for all selection ratio (σ) while scaling up the number of nodes in the cluster. GeoSpark dominates in performance for all the datasets, except in Points dataset where LocationSpark performs 5x better for low selection ratios.

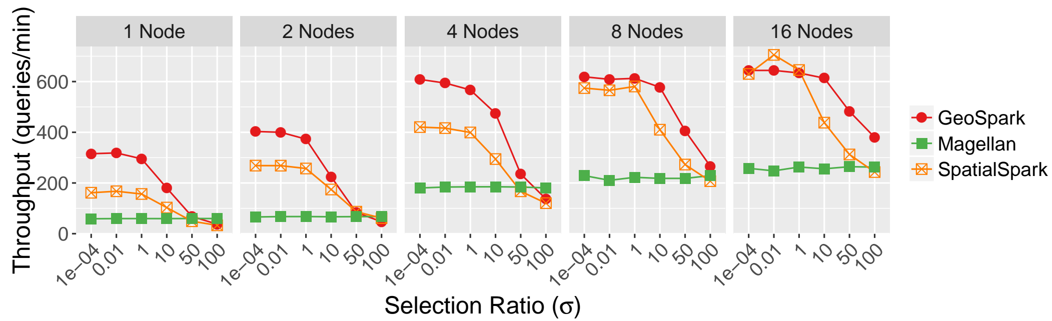
3.7.3 kNN Query Performance

To study kNN query performance, we generate 100 random location points in the longitude range (-180.0,180.0) and the latitude (-90.0,90.0) range, issue the random locations to the systems and measure the throughput of the systems in queries/minute. We also vary the value of k between 5 and 50. Only three systems support kNN queries: GeoSpark, Simba, and LocationSpark.

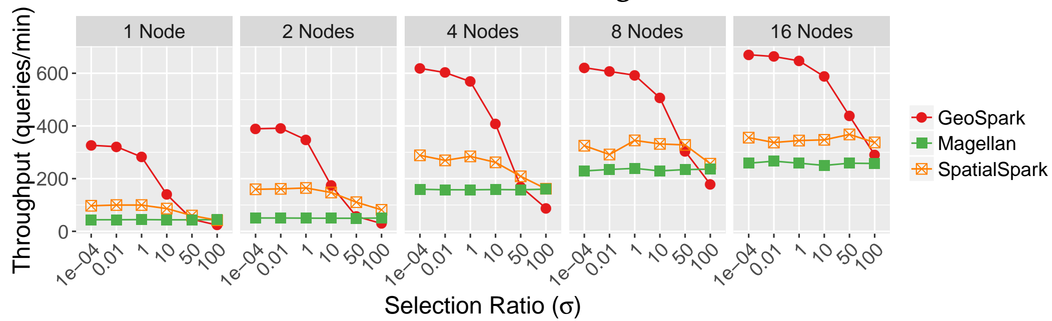
Figure 3.8 shows the kNN query performance varying the value of k on a single node. It can be seen that LocationSpark's throughput fluctuates a lot and is not as stable compared to Simba and GeoSpark. We repeated the experiments multiple times and encountered performance spikes for all values of k. This can be attributed to the *sFilter* that can significantly decrease the number of partitions to scan. GeoSpark utilizes the JTS library for most of its operations. GeoSpark uses *nearestNeighbour* function in JTS which uses the Branch-and-Bound tree traversal algorithm to provide efficient search for k nearest neighbor in the STRtree (*IndexedRDD* in GeoSpark). This means distance computation to other objects would only be limited to one (or at most two in case the query point overlaps with multiple partitions or is close to the boundaries of partitions). It then uses *takeOrdered* from the results to produce k nearest neighbors. Simba, on the other hand first computes a safe pruning bound to select partitions that contain at least k candidates. It then computes the tight pruning bound by issuing the kNN queries on the selected partitions. Again, similar to GeoSpark, the selected partitions are usually one or two for low values of k since most partitions would contain way more than k elements. Simba, also uses *takeOrdered* on distances to return the first k elements. The difference in their performance comes from the serialized index in Simba. Simba scans over the serialized index while GeoSpark has to scan the deserialized index. LocationSpark, can efficiently utilize the *sFilter* on global and local indices to reduce the distance computations to points in the



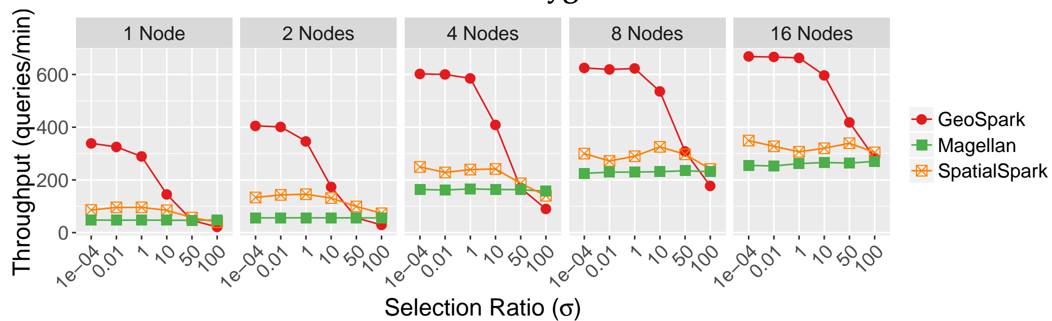
(a) Point



(b) LineString



(c) Polygon



(d) Rectangle

Figure 3.7: Range query performance scaling up the number of nodes for different selection ratio (σ) on different datasets

LocationRDD. The fluctuation in performance is due to periodic updates to the *sFilters*.

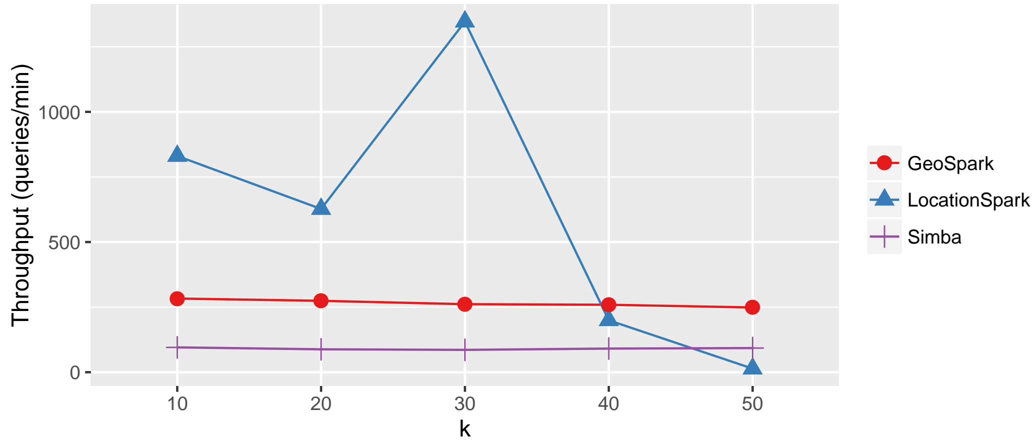


Figure 3.8: kNN query performance varying *k*

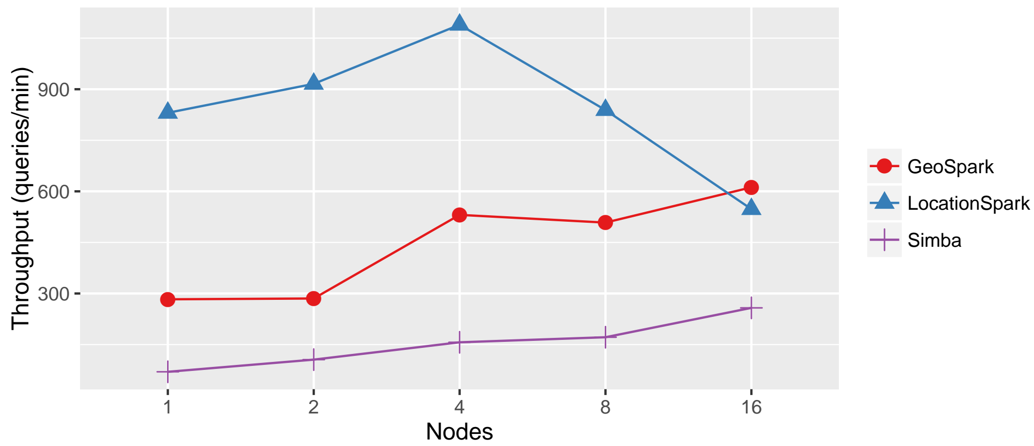


Figure 3.9: kNN query scalability with *k* = 10

3.7.4 Distance Join Performance

Only three systems support distance join queries: Simba, GeoSpark and SpatialSpark. Note that DJSpark (Distance Join) in Simba partitions the datasets inside the algorithm and thus we had to embed the timers to compute *Preparation Time* inside the join algorithm. This is the case with SpatialSpark as well. To measure the performance of distance joins we use the Points dataset. The distance for the query is set to 5 meters.

Figure 3.9 shows the kNN query performance scalability with value the of *k* fixed to 10.

Figure 3.10 shows the distance join cost breakdown for these systems while scaling up the number of nodes. For distance join, Simba samples both datasets and partitions the two datasets, *R* and *S*, using the STR algorithm. Simba then produces partition pairs (i,j) such that r $\in R_i$, s $\in S_j$ and distance(r,s) $\leq D$ (where D is the distance for the join). After generating these candidate pairs, Simba generates a combined partition P = (R_i,S_j) for each pair (i,j) and broadcasts them to the workers for local join processing. In local join processing, Simba creates local indices for S_j on the fly, and uses R_i

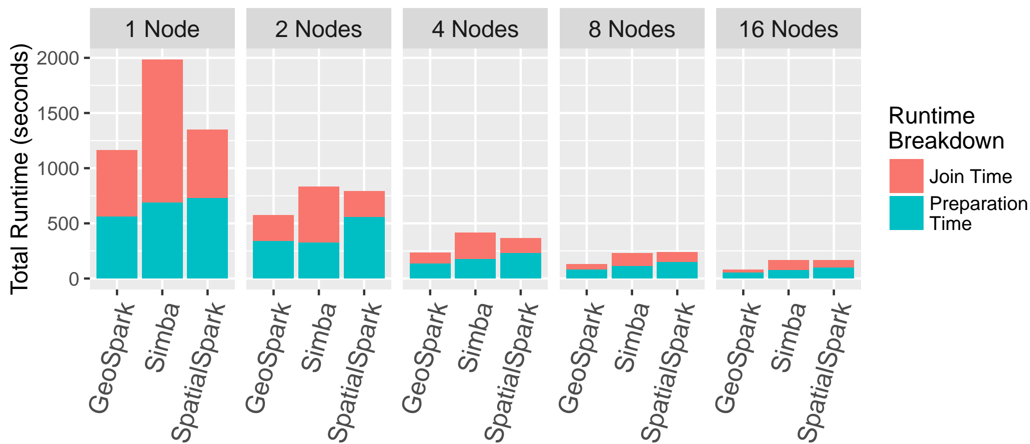


Figure 3.10: Distance join cost breakdown scaling up the number of nodes

to probe into the index to produce the final result. SpatialSpark samples data from only one input dataset and uses partition MBRs to build a spatial index to assign partition IDs for each data item on both sides of the join. This index is broadcasted to all nodes. The data items in both dataset are used to query the index to compute the partition ID that each data item should be assigned to. This assignment of the partition IDs is done using the STR algorithm. The partitioned data items are grouped based on the partition ID on both sides of the join using *groupByKey* function of the RDD. Then the partitions on the two sides are joined into one using the hash-based *join* on the partition IDs (one-to-one integer matching). Finally, these combined partitions are sent to the nodes for local join processing where local indices are built for the partitions and geometric refinement is done. This is how SpatialSpark handles all types of joins (including spatial joins). The spatial predicate (intersects or withindistance) for refinement is handled in the local join processing phase. GeoSpark handles the joins in a similar way. An advantage with GeoSpark is that user has more control since it exposes the spatial partitioning and index RDD APIs for applications. This means that distance join (or any join) can be called multiple times without incurring extra costs of partitioning and indexing the RDDs again. In case of SpatialSpark and Simba, the partitions and the indices are created on the fly which means that partitioning and indexing is tightly coupled with the join algorithm. This implies that the input datasets will be partitioned again in case distance join has to be invoked again. SpatialSpark and Simba can be tuned to reuse the partitions from the previous join query, but this would require changes to the systems source code rather than the application code. Figure 3.11 shows the scalability of the systems for distance join query based on *Total Join Time* and Figure 3.12 shows the shuffle read and shuffle write costs related to the systems. It can be seen that Simba has the highest shuffling costs. The peak memory consumption by GeoSpark, SpatialSpark, and Simba for distance join are **149 GB**, **287 GB**, and **211 GB** respectively.

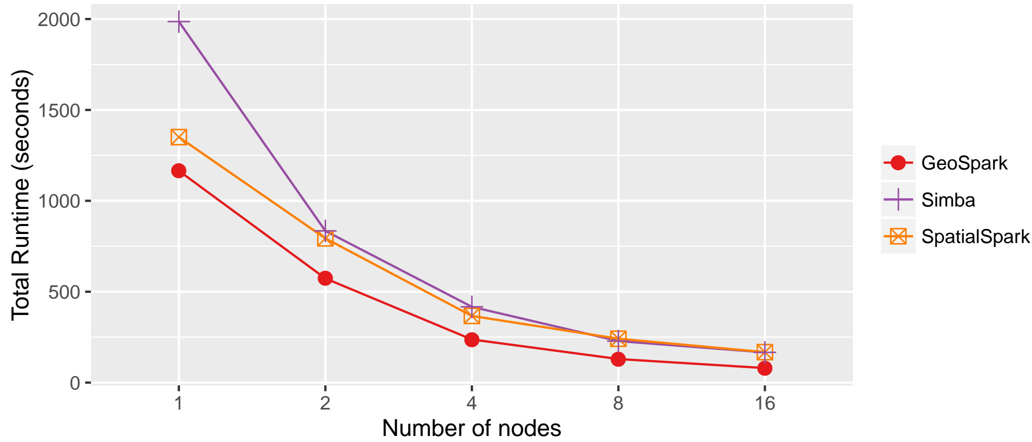


Figure 3.11: Distance join scalability

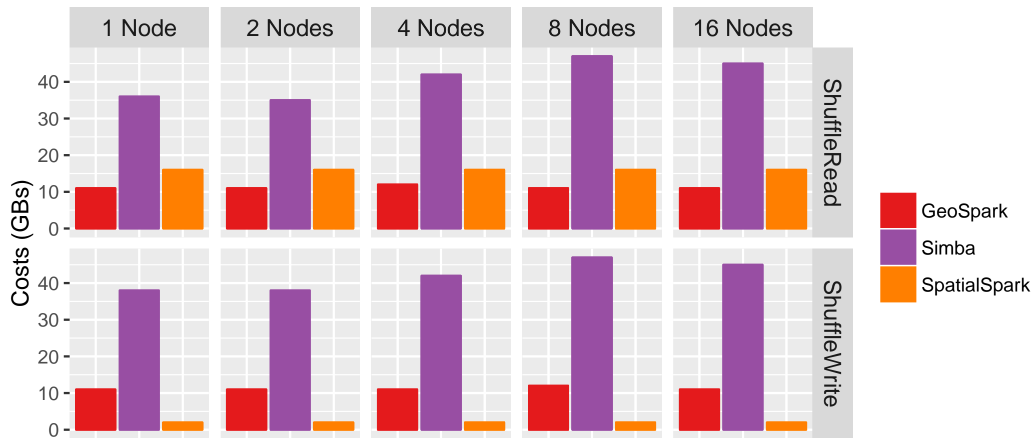


Figure 3.12: Distance join shuffle costs

3.7.5 Spatial Joins Performance

In this experiment, we measure spatial join performance for all possible combinations of geometric objects. To evaluate the systems, we make use of the *intersects* predicate in every case. We study the *Preparation Time*, *Join Time*, *Peak Execution Memory* consumption, *Shuffle Write* costs and *Shuffle Read* costs for evaluating join query performance. It is also important to mention here is that at the time of writing, Magellan does not have a full implementation of *Point-LineString* and *LineString-LineString* join. These joins only work for the filter phase where join partners can be filtered out based on the Z-curve value but no exact intersection test takes place. The results produced are only an approximation of the actual join.

Figure 3.13 shows the scalability of all possible spatial joins based on *Total Runtime*. Figure 3.14 shows the peak execution memory consumption, Figure 3.15 shows the shuffle write costs and Figure 3.16 shows the shuffle read costs related to the systems. Figure 3.17 shows the spatial joins cost breakdown and join performance for different systems on a single node and Figure 3.18 shows the *Point-Rectangle* join performance for different systems

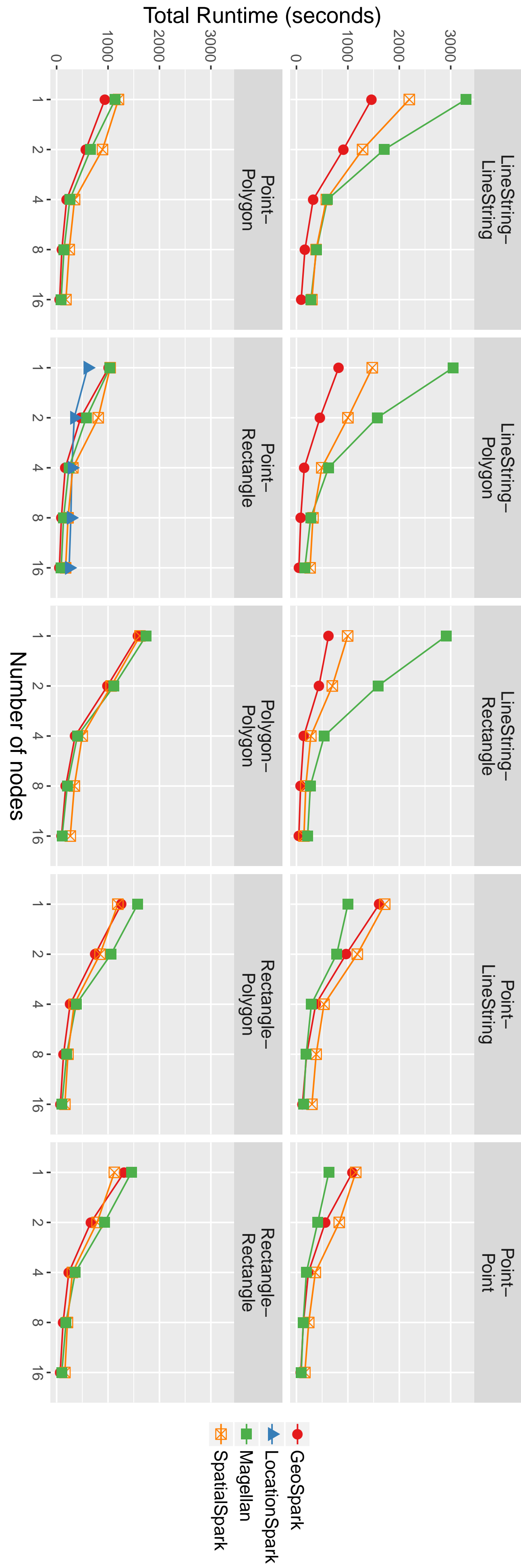


Figure 3.13: Scalability of all spatial joins for different systems while scaling up the number of nodes

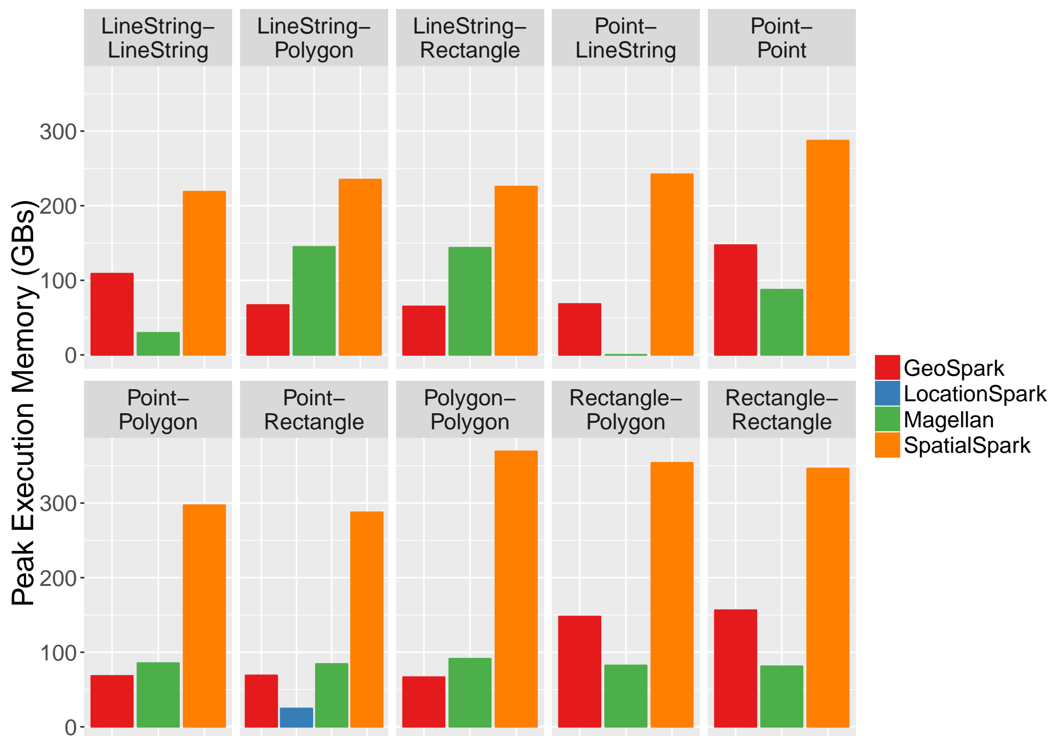


Figure 3.14: Spatial joins peak execution memory consumption

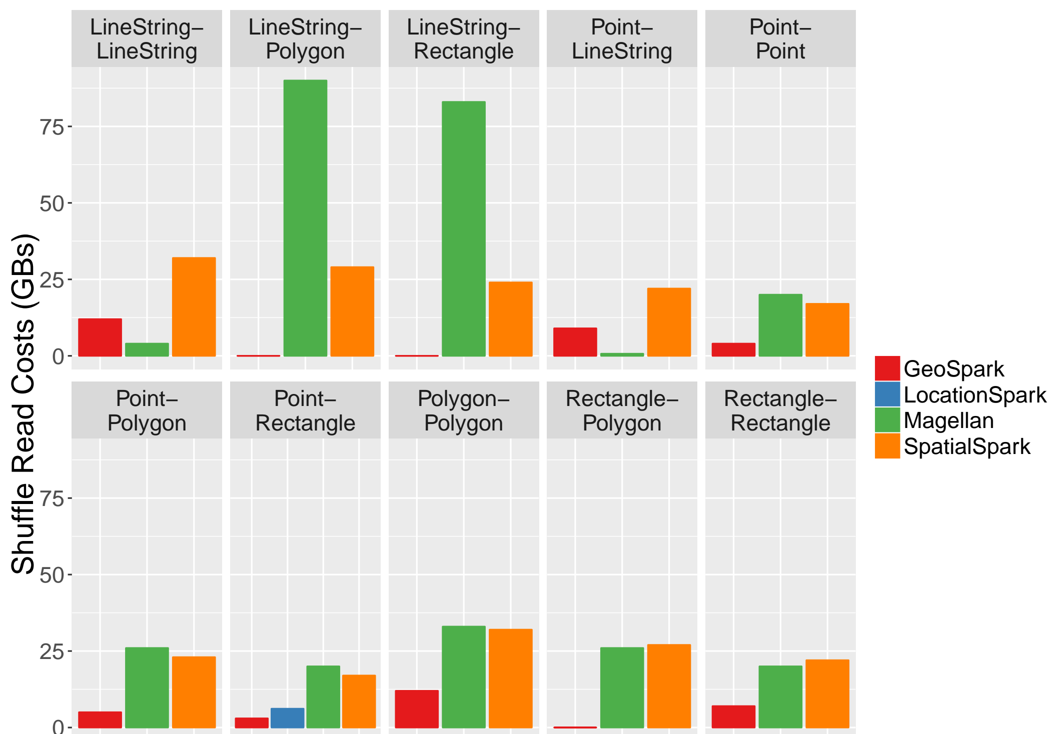


Figure 3.15: Spatial joins shuffle read costs

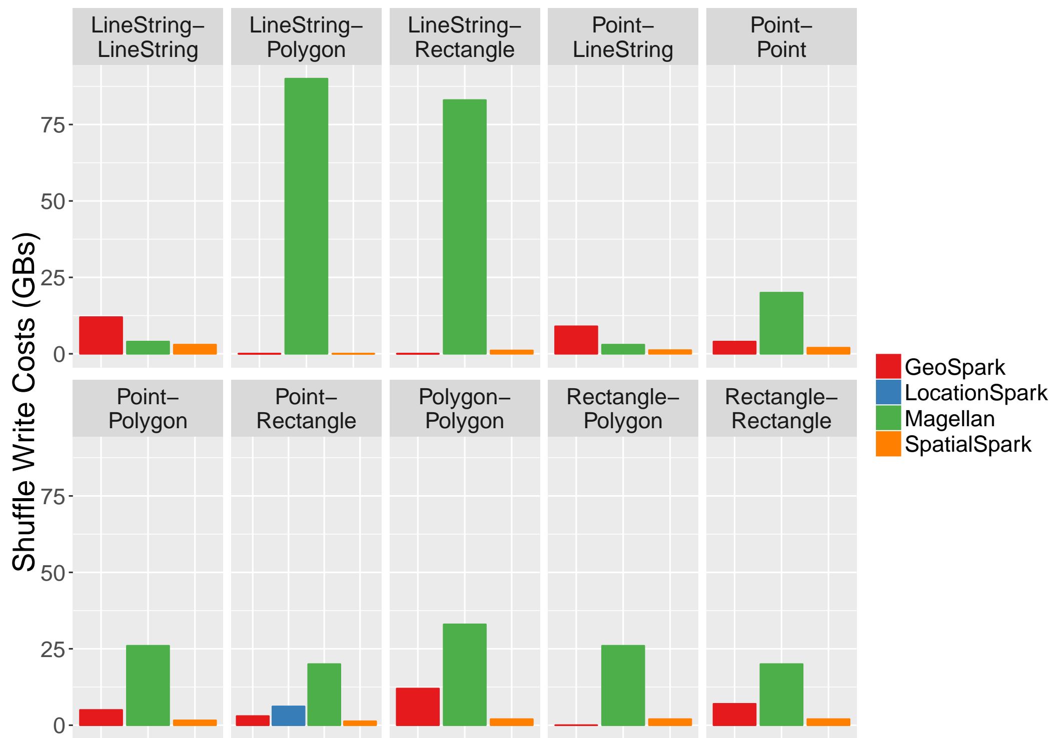


Figure 3.16: Spatial joins shuffle write costs

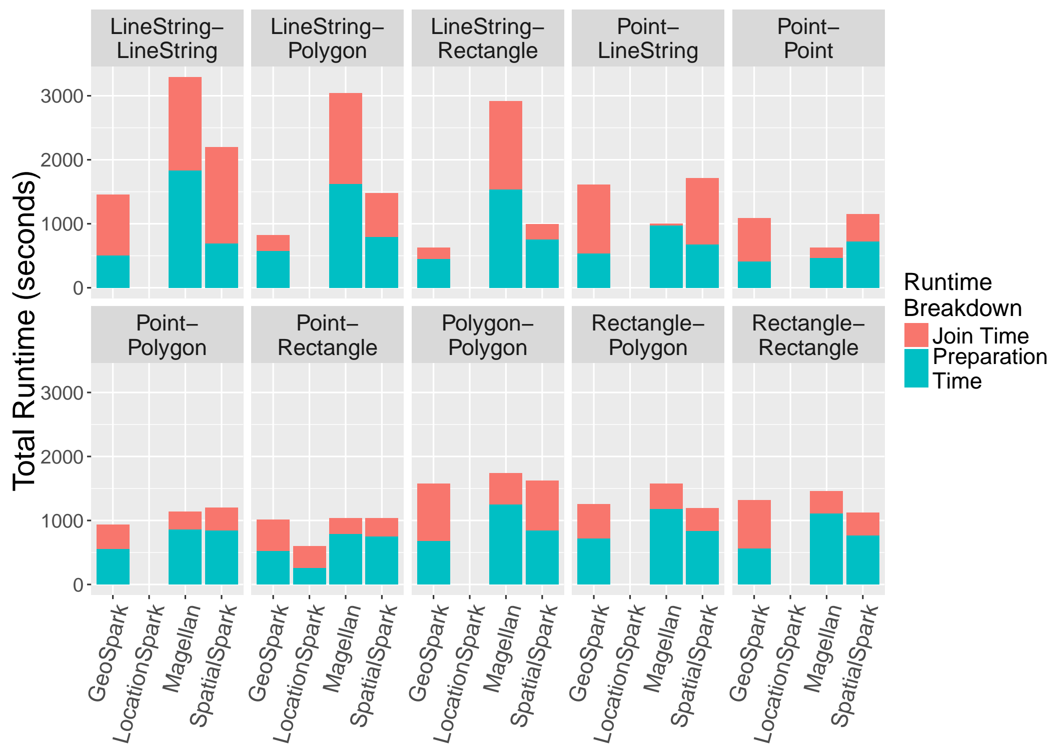


Figure 3.17: Total runtime cost breakdown for spatial joins between various geometric objects on a single node

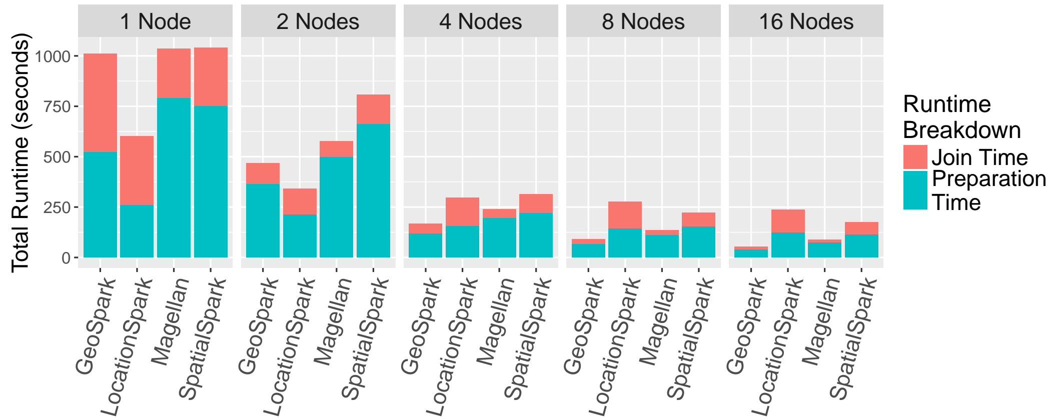


Figure 3.18: *Point-Rectangle* spatial join cost breakdown scaling up the number of nodes

while scaling up the number of nodes.

It can be seen that SpatialSpark has the highest *Peak Execution Memory* consumption, like in the case of distance join. It can also be seen that Magellan has high *Shuffling* costs compared to the other systems, especially in the case of joins in *LineStrings*. For the join, Magellan rewrites the plan, to use Z-curve value as the key and adds a *filter* that checks if the curves intersect or not. If the curves intersect, only then Magellan checks whether the spatial objects actually intersect or not. In the refinement phase Magellan ends up shuffling a lot of data. Note that no data is shuffled for *Point-LineString* and *LineString-LineString* joins, since these join only have the filter phase. It can also be seen that Magellan, has low *Join Time* for some type of joins. Although, Geospark has high memory consumption for input RDDs, it does not exhibit high *Peak Execution Memory* consumption or high *Shuffling* costs.

For spatial join in Magellan, we use a dataframe from different input datasets that contains tuples of the form spatial data, Z-curve value, and relation. In a tuple a relation represents the relationship (within, overlapping, or disjoint from the spatial object) of Z-curve value to the spatial object. The *Preparation Time* in Magellan is spent in generating these dataframes from the input datasets which only contains the spatial objects. For the join, Magellan rewrites the plan, to use Z-curve value as the key and adds a *filter* that checks if the curves intersect or not. If the curves intersect, only then Magellan checks whether the spatial objects actually intersect or not. It can be seen that GeoSpark also has high preparation costs in most cases. This is due to high memory costs in GeoSpark. Note that for spatial joins we have *SpatialPartitionedRDD* and *IndexedRDD* cached in memory for both datasets. GeoSpark serializes/deserializes these RDDs as needed to stay within the storage memory limit on one node. On one node these serialization/deserialization costs add up resulting in high preparation and join times. This is especially evident in Figure 3.18. It can be seen that when there is sufficient storage memory available (2 nodes upwards) for all four RDDs to be cached in memory the total run time reduces by 2.5x. This is also the case for Magellan, where the indexed LineString dataframe consumes the most memory (92 GB) and every

type of join involving the LineString dataset shows significant improvement as we scale up the number of nodes thereby increasing the amount of storage memory.

From the figures we can also see that LocationSpark outperforms the closest competitor Magellan for Point-Rectangle join (the only supported spatial join in LocationSpark) by 1.5x. This is due to couple of reasons. Firstly, its has low memory related costs. Secondly, LocationSpark has two abstract spatial layers, *LocationRDD* (for locations or points) and *queryRDD* (for rectangles), and a query scheduler. *LocationRDD* is globally and locally indexed along with their embedded *sFilters*. The query scheduler first estimates the cost of query runtime using sampled queries and partitions from *queryRDD* and *LocationRDD*. LocationSpark uses reservoir sampling [185] to sample queries from the *queryRDD* and partitions from the *LocationRDD* and estimates the runtime costs if queries are executed on the sampled partitions. The costs are estimated using techniques proposed in [95]. It then takes the partitions with high query runtime estimates and estimates the cost of repartitioning these partitions and computes the runtime costs to run sampled queries on repartitioned partitions. If the estimated cost of repartitioning and runtime is less than the previously estimated runtime costs, it adds the repartitioning step in the execution plan. This ensures that not only the partition sizes are well balanced but also the amount of work on the executors is also more or less well balanced. Secondly, LocationSpark also filters out multiple partitions for a tuple from the *queryRDD* to join against using its *sFilter*, in a similar way as it does it in the case of range and kNN queries.

3.7.6 kNN Join Performance

Only two systems support kNN join: Simba and LocationSpark. For kNN join query we fix the value of k to 5 and measure the join performance for the two systems. Another point to make here is that the kNN Join query for the Points datasets (200 million points) crashed in Simba. We will explain the reason later. Since, Simba [192] used a maximum of 10 million points (for both datasets) in their evaluation of kNN join, we decided to do the same. We sampled 10 million points from the Points dataset and then ran the kNN join query on them. Since we reduced the dataset to 10 million points for both datasets we had to run multiple experiments to determine a good number of partitions for Simba. LocationSpark does not require tuning the number of partitions for the *LocationRDDs* as the query scheduler and optimizer already does it and overwrites the number of partitions specified by the user. On the other hand Simba, by default, sets the number of join and index partitions to 200 each. We found that 50 partitions performed the best for Simba for 10 million points.

Figure 3.19 shows the kNN join cost breakdown and Figure 3.20 shows the scalability of the systems based on *Join Time*. LocationSpark balances the work among the Spark workers well, using the query cost estimation mentioned previously in Section 3.7.5. Simba is not able to do so, and ends up

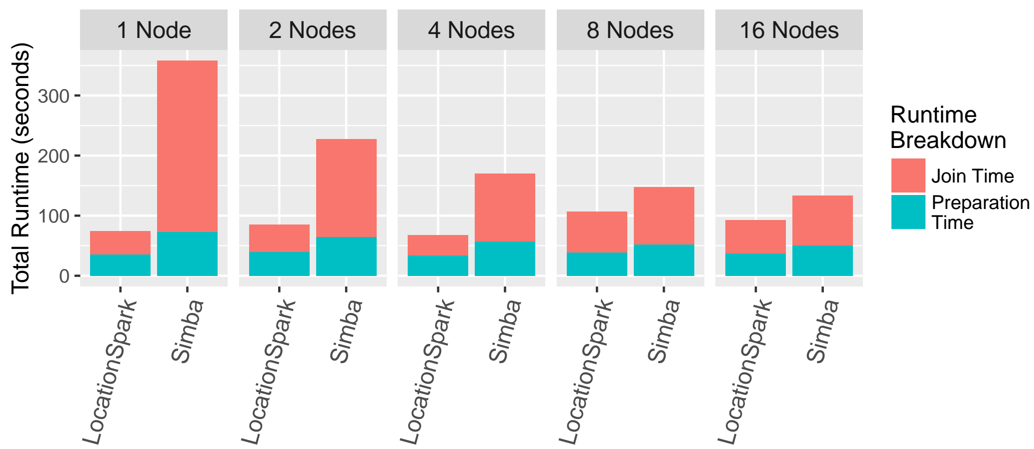


Figure 3.19: kNN join cost breakdown scaling up the number of nodes

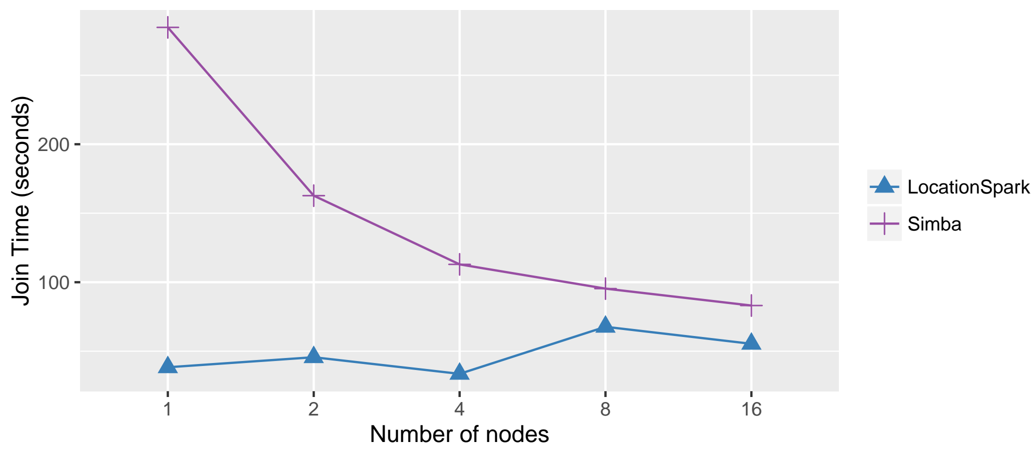


Figure 3.20: kNN join scalability

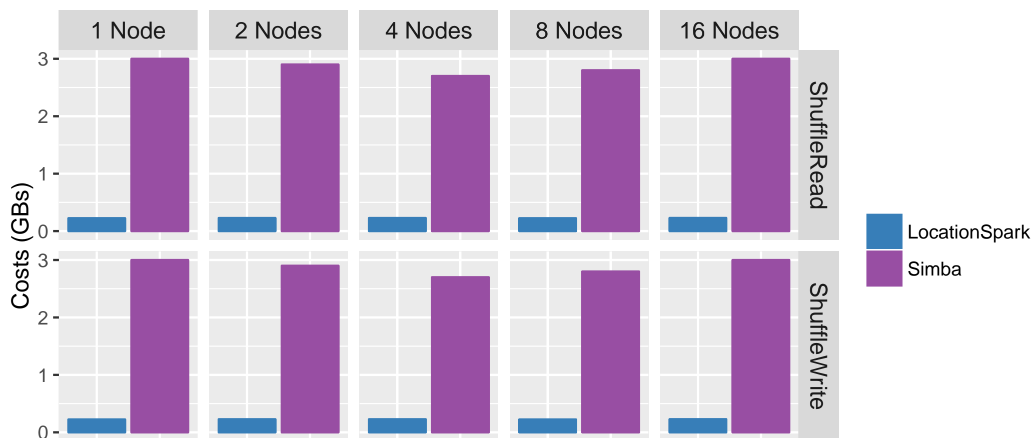


Figure 3.21: kNN join shuffle costs

creating overloaded partitions because of duplicated points. This can be attributed to how the kNN join algorithm (*RKJSpark*) works in Simba: Let the two datasets be *R* and *S*. RKJSpark algorithm tries to find *n* partitions of *S* to pair with *n* partitions of *R*, such that these paired partitions can be combined into one RDD partition using *zipPartitions* and then kNN join can be run on them locally. The pairing is done by computing distance bounds (γ). Simba partitions *R* into *n* partitions (R_n) and computes a distance bound (γ_i) for each partition R_i in two steps. First, for each partition R_i, the algorithm computes the distance of centroid (C_i) of the MBR (minimum bounding rectangle) of the partition to the furthest point in the partition (we denote this distance as D_{i1}). Second, it samples a set of points from *S* and builds an R-tree on the sampled dataset. It then computes the kNN of the centroid (C_i) of each partition (R_i) from the sampled dataset using the R-tree and selects the distance of the furthest *k*th neighbor (D_{i2}). The distance bound (γ_i) is then set to $2D_{i1}$ + D_{i2}. Note that the distance bound is different for each partition. The algorithm then partitions S into *n* partitions based on

$$S_i = \{ \; s | s \in S, \; distance(C_i, s) \leq \gamma_i \; \}$$

This means that for every s $\in$ S, *RKJSpark* includes a **copy** of s in S_i if distance(C_i,s) $\leq \gamma_i$. This creates a lot of duplicated points in the partitions for S and leads to more and redundant computations. This is also the reason, why Simba crashes for the Points dataset (200 million) where it simply runs out of heap space because of a lot of duplicated points.

Figure 3.20 shows the scalability of the systems for kNN join query based on *Join Time*. It can be noticed that LocationSpark shows a slight increase in runtime for 8 and 16 nodes. This is due to the communication cost where more *executors* return the local result to the *driver*.

Figure 3.21 shows the shuffle costs for each system. It can be seen that Simba has a higher *Shuffling* related costs as compared to LocationSpark. The peak memory consumption for LocationSpark and Simba is **2.24 GB** and **1.75 GB** respectively.

3.7.7 US Census TIGER Dataset

In these experiments, we used the TIGER Edges dataset which contains approximately *70 million* linestrings. As, the other datasets in the TIGER are limited in size (less than 2 million spatial objects), we generated a rectangle dataset from the Edges dataset by computing the bounding box of each linestring record. We also generated a Points dataset from the OSM `All Nodes` dataset, which comprises of 170 million points that are located in the US region. We ran the distance join, the spatial joins, and the kNN join queries on these datasets. These datasets are also available on our server[12].

[12] http://osm.db.in.tum.de/

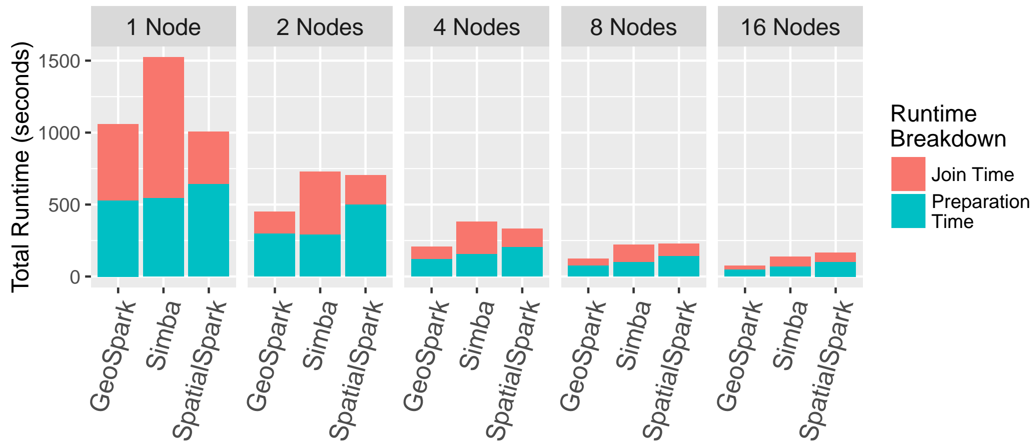


Figure 3.22: Distance join cost breakdown scaling up the number of nodes

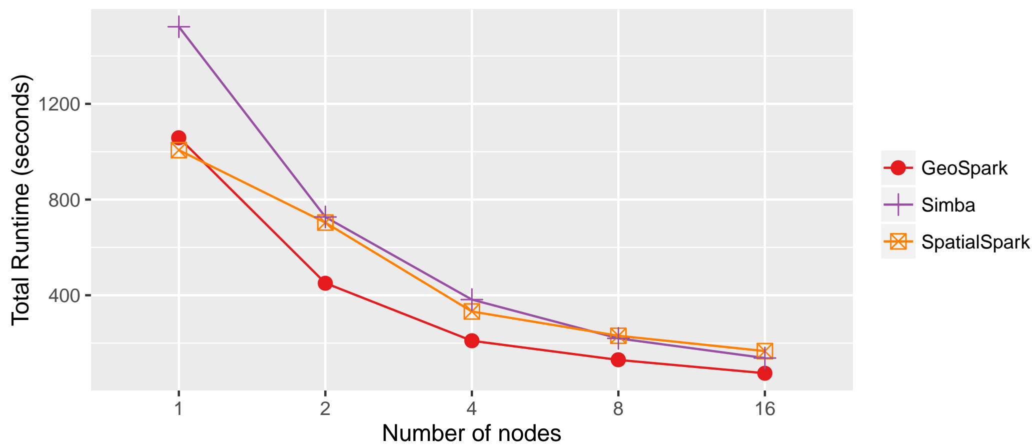


Figure 3.23: Distance join scalability

Distance Join Performance

To measure the performance of distance joins we use the US Points dataset. The distance for the query is set to 5 meters. It can be seen that the performance, and the join related costs are similar to those in the OSM datasets. Simba, again has high shuffling costs and is also the slowest among the three systems. Figure 3.22 shows the distance join cost breakdown for these systems while scaling up the number of nodes. Figure 3.23 shows the scalability of the systems for distance join query based on *Total Join Time* and Figure 3.24 shows the shuffle read and shuffle write costs related to the systems. The peak memory consumption by GeoSpark, SpatialSpark, and Simba for distance join are **129 GB**, **273 GB**, and **176 GB** respectively.

kNN Join Performance

As in the case of the OSM dataset, we again sampled 10 million points from the US points dataset and ran the kNN join query on them. kNN join query

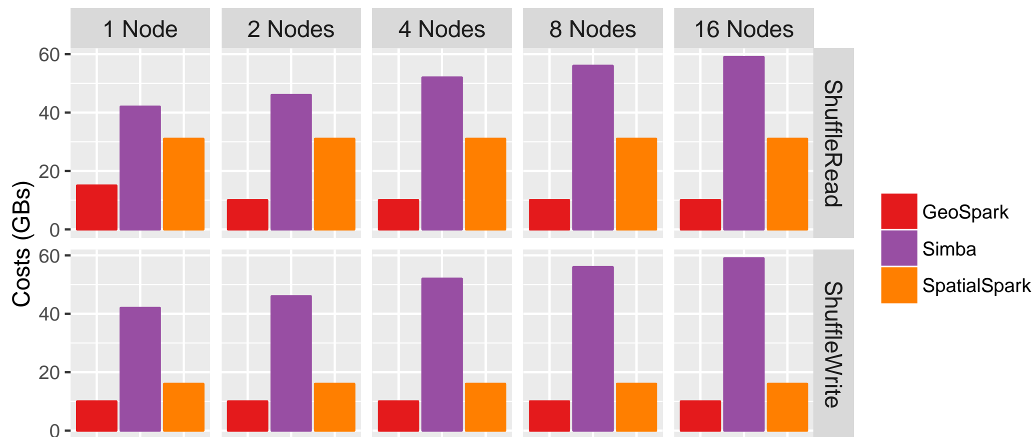


Figure 3.24: Distance join shuffle costs

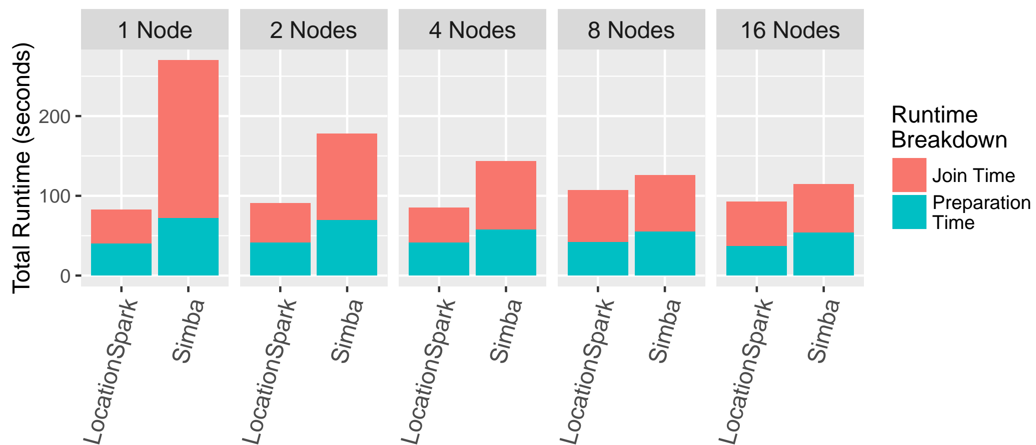


Figure 3.25: kNN join cost breakdown scaling up the number of nodes

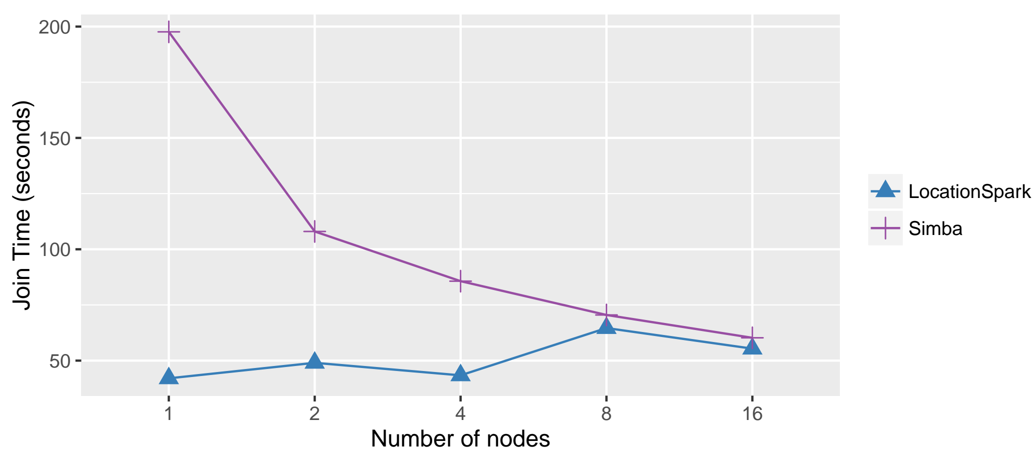


Figure 3.26: kNN join scalability

performance shows the same trend as in the OSM dataset.

Figure 3.25 shows the kNN join cost breakdown and Figure 3.26 shows

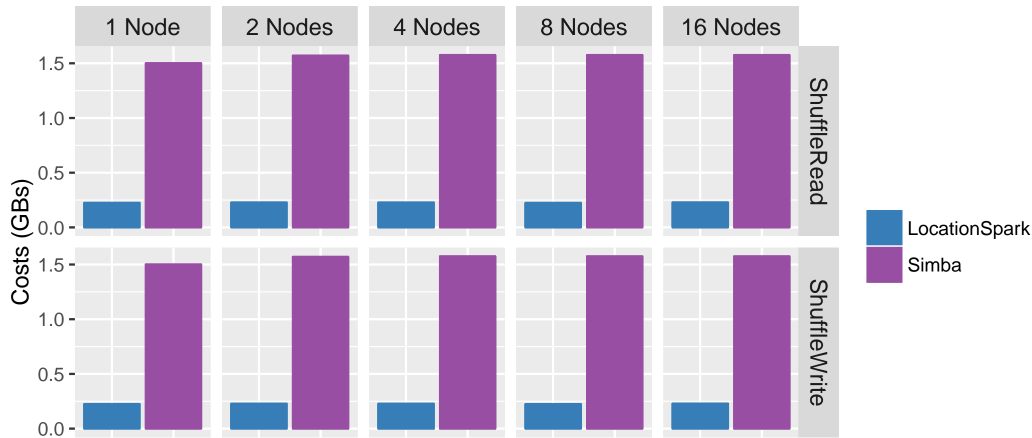


Figure 3.27: kNN join shuffle costs

the scalability of the systems based on *Join Time*. Figure 3.26 shows the scalability of the systems for kNN join query based on *Join Time*. Figure 3.27 shows the shuffle costs for each system. The peak memory consumption for LocationSpark and Simba is **2.24 GB** and **1.75 GB** respectively, exactly as in the case of the OSM dataset.

Spatial Joins Performance

Figure 3.28 shows the scalability of all possible spatial joins based on *Total Runtime*. Figure 3.32 shows the spatial joins cost breakdown and join performance for different systems on a single node and Figure 3.33 shows the *Point-Rectangle* join performance for different systems while scaling up the number of nodes. Magellan, again exhibits high shuffling costs. SpatialSpark has the highest *Peak Execution Memory* consumption. GeoSpark performs the best in almost all spatial joins.

3.8 Conclusions And Future Work

In this chapter, we evaluated five Spark based spatial analytics systems. We performed an experimental evaluation of these systems using real-world datasets. Table 3.5 summarizes the strengths and weaknesses of the systems. From our experience, GeoSpark comes close to a complete spatial analytics systems because of data types and queries supported and the control user has while writing applications. It also exhibits the best performance in most cases. There are a few drawbacks though. First, it consumes a large amount of memory for the input datasets. Second, GeoSpark does not support kNN joins yet. Magellan also exhibits good performance for some spatial joins especially if only *Join Time* is considered, but it does not have any optimization for range queries. Also, it does not support kNN queries, distance joins and kNN joins. Moreover, Magellan has very high shuffling related costs. An advantage of GeoSpark and Magellan is that they are actively under development. LocationSpark is interesting since it has a very good query

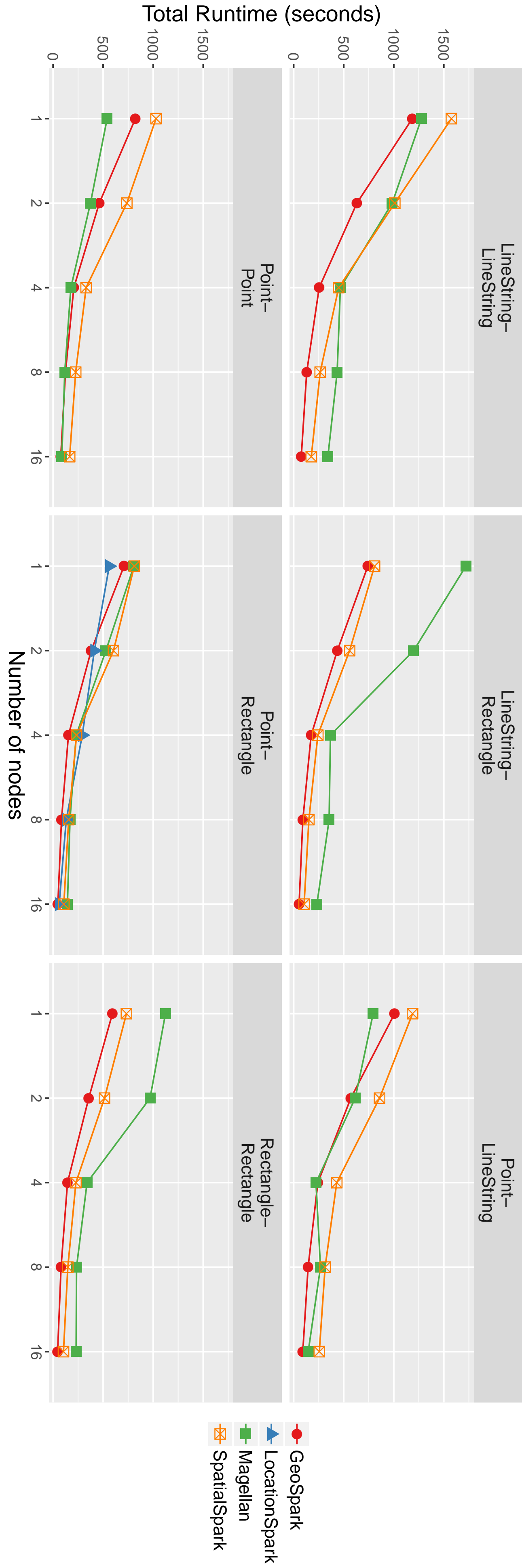


Figure 3.28: Scalability of all spatial joins for different systems while scaling up the number of nodes

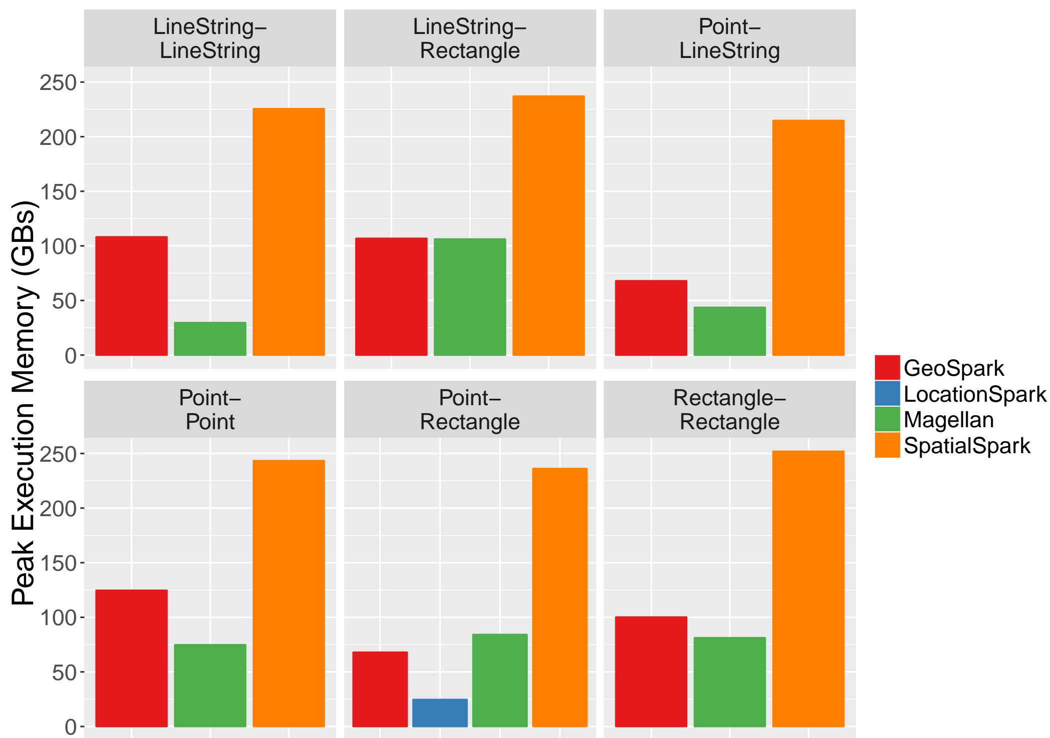


Figure 3.29: Spatial joins peak memory consumption

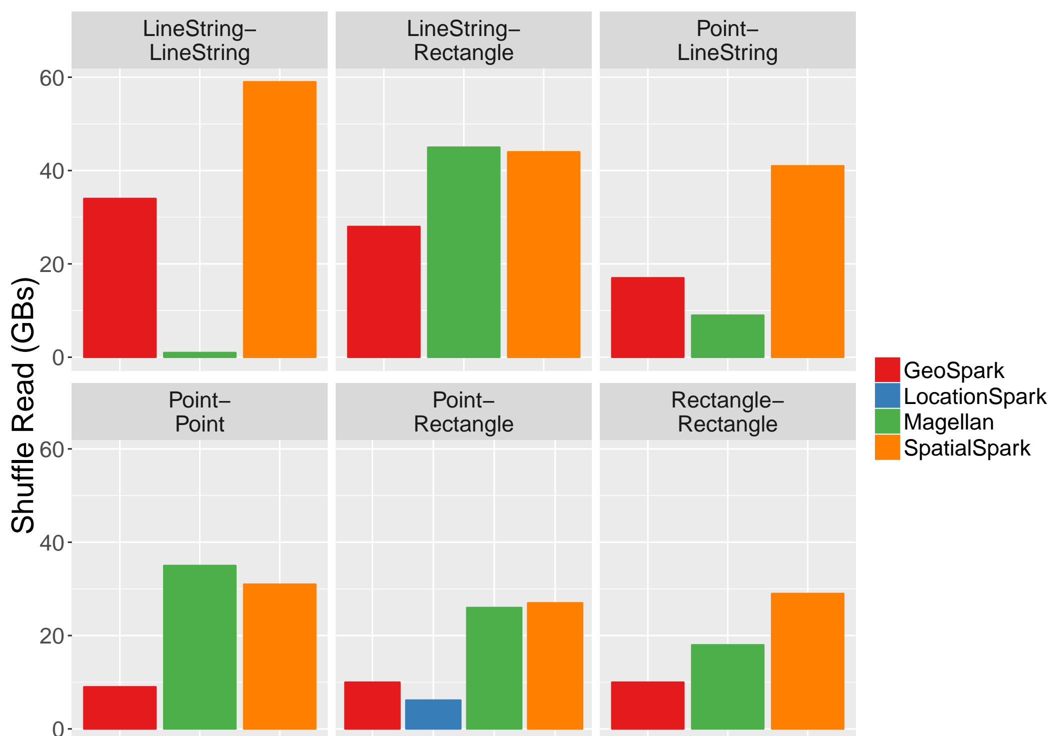


Figure 3.30: Spatial joins shuffle read costs

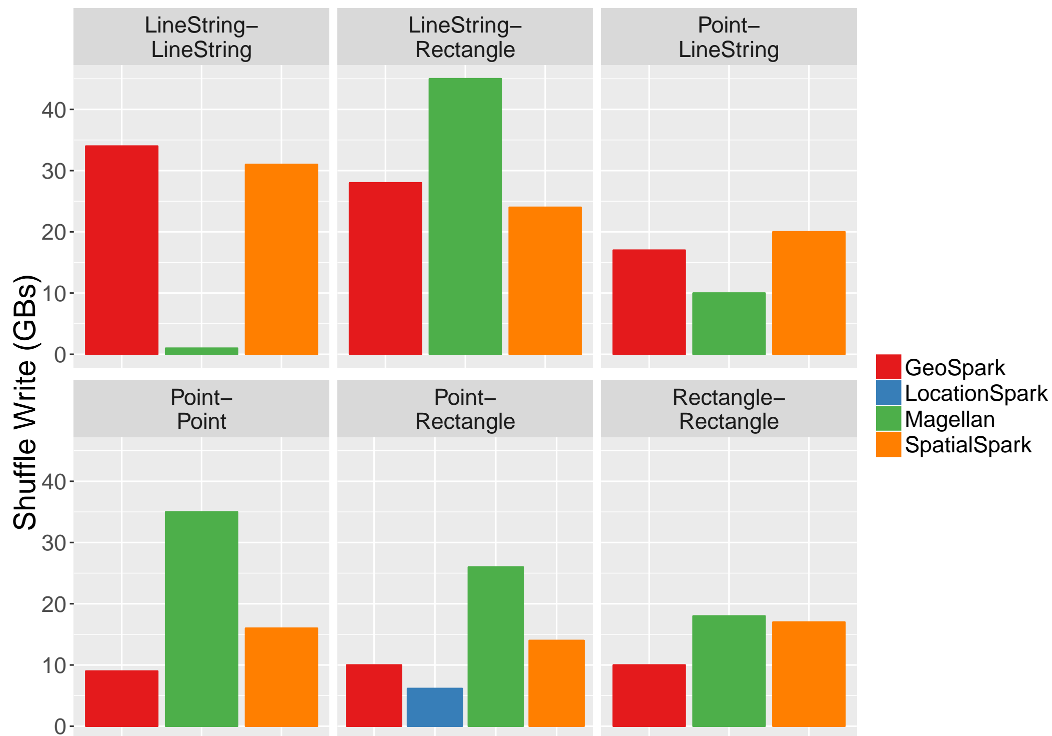


Figure 3.31: Spatial joins shuffle write costs

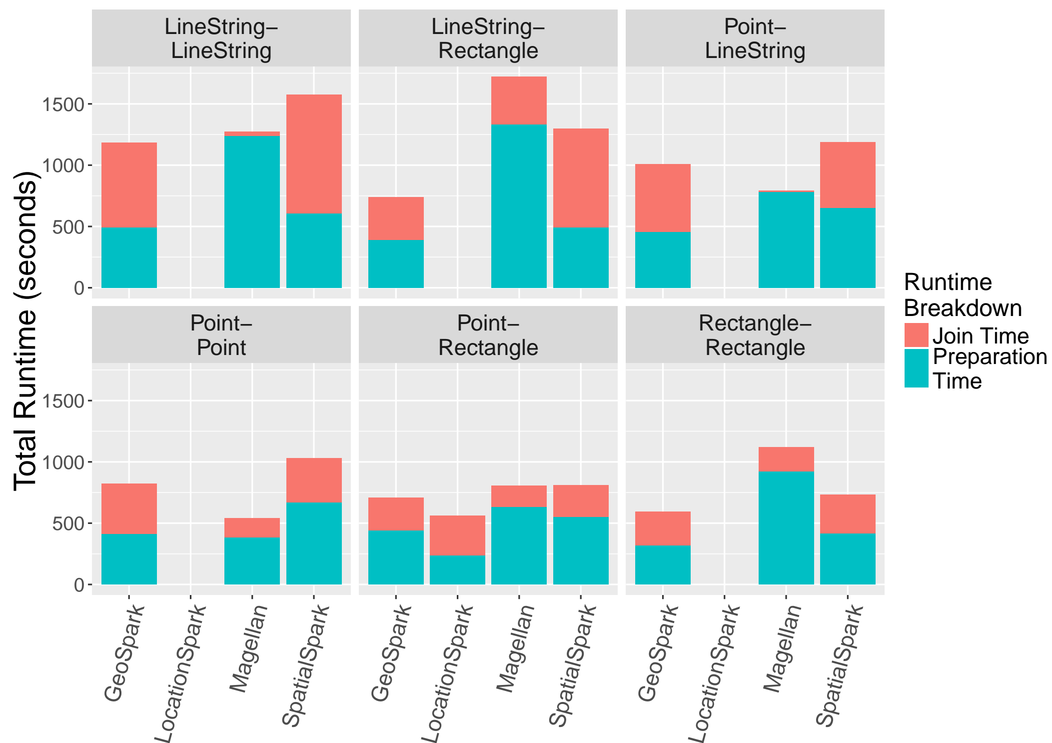


Figure 3.32: Total runtime cost breakdown for spatial joins between various geometric objects on a single node

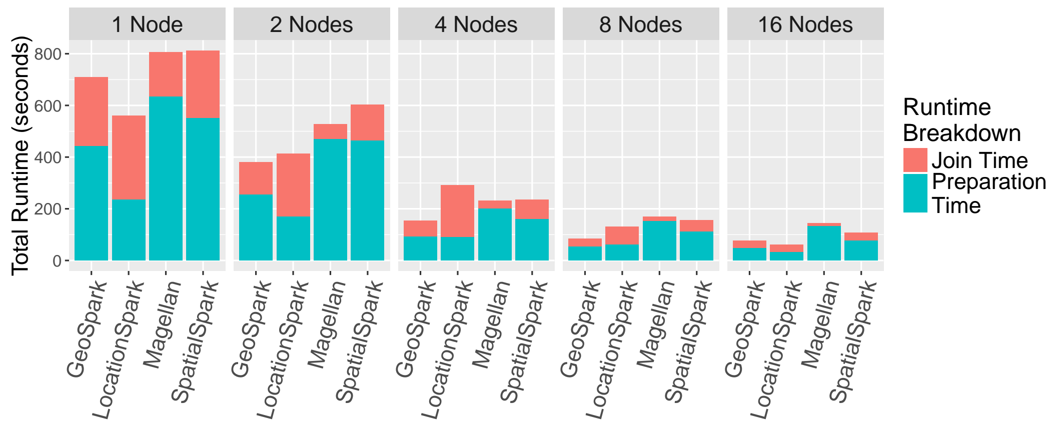


Figure 3.33: ***Point-Rectangle*** **spatial join cost breakdown scaling up the number of nodes**

Table 3.5: Strengths and Weaknesses

System	Strengths	Weaknesses
GeoSpark	Query optimizer Scales well Rich in features Active development	High memory costs No kNN join
Simba	Query optimizer Scales well	Limited data types No recent development
LocationSpark	Query optimizer and scheduler Spatial bloom filter	Limited data types No recent development
Magellan	Join query optimizer Low join time Scales well Active development	High shuffle costs High preparation times No range query optimization
SpatialSpark	Scales well	No recent development High memory costs

scheduler and optimizer. Also it has a spatial bloom filter *sFilter* which bring query costs down. The aforementioned systems may look to incorporate such filters in their system as well. Again, the limitation is that it has limited data types and there has not been any development recently. Simba, like LocationSpark, has very limited data types (only points) and does not support spatial joins. SpatialSpark is competitive but has high *Peak Execution Memory* consumption. Moreover, there has been no active development. We also see that all the systems evaluated scale pretty well with more resources.

A recent development in the area of spatial joins have been in the area of approximate and adaptive joins with precision guarantees [206] [83] [84]. The motivation behind such joins is that many applications today do not require the join results to be accurate and only need an approximation to make certain decisions. The systems studied in this chapter may look to add such joins for such applications. Another interesting field is the area of trajectory similarity search, and an operator for such queries in these systems would be a welcome addition for many users. Also, Postgres with its extension PostGIS is rich with a variety of spatial operators that the Spark based spatial systems do not currently have and could be implemented in the future.

Chapter 4

Modern Spatial Libraries

Excerpts of this chapter have been published in [131, 132].

4.1 Introduction

In recent years, services such as recommending close-by social events, businesses, or restaurants as well as navigation, location-based mobile advertising, and social media platforms have fueled an exponential growth in location-enabled data. Industry giants like Google, Facebook, Uber, Yelp, and Foursquare are some of the various companies that provide such services. To handle location data from its users, these companies either build their own spatial data management systems from scratch, or rely on existing solutions.

The rise of location-based services has also led the research community to develop systems that can efficiently handle, process and analyze spatial data. HadoopGIS [2] and SpatialHadoop [40] were one of the first research efforts to focus on handling and processing spatial data at scale. Apache Spark and Impala saw a similar trend with a plethora of research introducing spatial support in the form of SpatialSpark [197], GeoSpark [203], Simba [192], Magellan [166], STARK [61], LocationSpark [173], Sphinx [42], SRX [175], STAR [26], and Amazon Redshift [17]. Popular database systems have also witnessed a similar trend with Oracle Spatial [126], HyPerSpace [129], MemSQL [112] and MongoDB [114].

Many of these systems or services use an open-source library to implement the basic geometry types, indexes, and algorithms for spatial processing. Some of the most popular libraries are: JTS Topology Suite (JTS), it C++ port Geometry Engine Open Source (GEOS), Google S2 (S2), ESRI Geometry API, and Java Spatial Index (JSI). Today, these libraries are being used in a variety of services and research projects alike. We will highlight the major services and research projects that use these libraries in Section 4.4. Many of the services that use these libraries are multi-million dollar business models, such as on-demand ride-hailing and dating applications. Moreover, many research efforts today in the systems community also use these libraries for their spatial-processing capabilities. Given how prevalent and relevant these libraries are in present-day services and systems, it becomes a necessity to evaluate these libraries.

In this work we take an application-oriented approach in evaluating these libraries. Many open datasets such as Open Street Maps or NYC taxi rides datasets provide location information using **raw GPS coordinates**. Moreover, millions of GPS devices in use today send location information in the form of GPS coordinates. Thus, unless stated otherwise, we assume that applications receive raw GPS coordinates and have to process spatial queries based on them.

In this chapter, we contribute:

- A study of problems arising in using planar geometry libraries directly with GPS coordinates.
- A survey of modern spatial libraries, highlighting their features and indexes.
- A thorough performance analysis of these libraries using four spatial queries: range, distance, *k*-NN, and a spatial join query.

The rest of the chapter is structured as follows: Section 4.2 discusses the background for planar and spherical geometry, and identifies potential pitfalls in incorrect usage of these libraries. Section 4.3 formally defines the spatial queries we considered for evaluation and presents practical examples of these queries. Section 4.4 introduces the modern spatial libraries. Section 4.5 presents the experimental setup used for evaluation, which is followed by the evaluation itself in Section 4.6. In Section 4.7 we highlight a potential research area and discuss how distributed spatial query processing can be implemented using the spatial libraries. Section 4.8 discusses related work and is followed by takeaways and conclusions in Section 4.9.

4.2 Background

The libraries evaluated in this chapter either use planar or spherical geometry. In this section, we will describe what these two terms mean and why a naive usage of planar geometry libraries can introduce unintended errors.

4.2.1 Geometry Models

Mathematician Carl Friedrich Gauss proved in Theorema Egregium [55] that a sphere and a plane are not isometric, i.e. distances cannot be preserved if one is transformed into the other. Thus as a corollary of Theorema Egregium, one cannot wrap a paper around a ball without crumpling it. Conversely, the surface of a sphere cannot be flattened into a plane without distorting the distances. This fact proved to be the basis of cartography, the study and practice of making maps. A map projection [193] is a systematic transformation of the latitudes and longitudes of locations from the surface of a sphere or an ellipsoid into locations on a plane. As the theorem implies that no planar (flat) map of Earth can be perfect, even for a portion of the Earth's surface, every cartographic projection necessarily distorts at least some distances. For

example, the Mercator projection distorts the size of geographical objects far from the equator. In the Mercator projection, Greenland appears roughly the same size as Africa, while in reality it is 14 times smaller than Africa. The Mercator projection is practically unusable at latitudes greater than 70°, north or south, and can never fully show polar areas. On the other hand, the Mercator projection preserve the shape of the countries and it preserves direction, and thus is very useful for navigation, i.e. 90°turns on roads in reality appear the same on the map. This is one of the reasons that a variant of Mercator projection, called Web Mercator, became a popular choice for various web services such as Google Maps[1], Bing Maps, and Open Street Maps. There are around five thousand [164] available projections today and they come with different trade offs in terms of four spatial properties: shape, distance, direction, and land area.

Earth can be projected onto many surfaces, but today the most widely adopted surfaces to project Earth on are planes and spheres.

Planar Geometry: is geometry on a plane. The basis of planar geometries is a plane, i.e., all the calculations on the geometries such as distance between geometries, area covered by a geometry, intersection between geometries is done on a plane using cartesian mathematics. In planar geometry, the distance between two points on a plane is a straight line distance between the points.

Spherical Geometry: is geometry on a sphere. The basis of spherical geometries is thus a sphere. On the sphere there are no straight lines as in case of a plane. In spaces involving curvature (such as spheres), straight lines are replaced by geodesics. The shortest distance between two points on the surface of a sphere is called the great-circle distance or orthodromic distance [189].

To make planar geometries work with geographic data, Earth has to be projected onto a plane. There are multiple projections available, some of which are based on the area that they cover such as city based, region based, country based, and even on continental and global scale but they all come with different trade-offs [115]. Most notably, there is no planar projection that preserves distance. Projections can only *minimize* distance distortion. When working with planar geometries, it thus becomes essential to choose the right projection that is best suited to the application concerned.

Spherical geometries on the other hand work on spherical projections, which maps the points on Earth's surface to a perfect mathematical sphere. As Earth is not a perfect sphere, spherical projections of the Earth also create distortions, but are limited to a maximum distortion of 0.56% [149]. Spherical projections also preserve the correct topology of the Earth with no singularities and low distortions everywhere. An even more accurate projection of Earth is on an ellipsoid, but operations on ellipsoids are orders of magnitude slower than on a sphere. Spherical geometry are also slower than their planar geometry counterparts usually since the computations are on a sphere rather than on a plane. But spherical geometry is generally considered better suited to work with geographic data on a *global* scale.

[1]Google Maps on a web browser now displays a globe if zoomed out sufficiently.

4.2.2 When Can Things Go Wrong In Planar Geometries?

In this section, we will show how applications can end up using planar geometry libraries in a wrong way. We motivate this by using an illustrative example of a ride-hailing application in two scenarios: operating in a city and on a global scale. We highlight potential pitfalls which can lead to applications getting wrong results.

Consider a ride-hailing application scenario in New York City that stores the location data as raw GPS coordinates (lat/long)[2], and matches riders with the nearest drivers using the *k*-NN query (we formally define *k*-NN query in Section 4.3.3). A part of *k*-NN query processing is the distance computation between two points, the user and the drivers in this case. Planar geometry libraries come with distance functions[3] that compute Euclidean distance. The application could naively compute Euclidean distance between two raw GPS coordinates, in which case, the distance would be in degrees and does not have any meaning. The correct approach is to project the raw GPS coordinates using a spatial reference system, such as EPSG:32118 [44] that *minimizes* the distance distortion for the New York area, and the measurement unit is in meters. The Euclidean distance can then be computed on the projected coordinates using the distance function in the planar geometry library. Another way is to compute the Haversine distance between the GPS coordinates, but it is slower to compute because it involves computing multiple sine and cosine operations.

Now as another example, consider the same application as in the previous example, but the application now operates at a global level and uses a planar geometry library. The application may naively start using EPSG:3857 [165] as the projected coordinate system, which projects the whole Earth onto a plane, and not just a city as in the case with EPSG:32118. In EPSG:3857, distances are only accurate along the equator, and the error increases with gain or loss in latitude. The application receives two ride requests, one in city A which lies on the equator, and the other in city B which is closer to the North (or the South) Pole where distance distortions are large (distances become larger than they actually are). While the distance computation will be correct for city A, for city B the distance distortions will be large. In EPSG:3857 the distance distortion can be significant. So if the application is using planar geometry, or more accurately using Euclidean distance, to compute the distance between the users and the drivers in city B, a user might not be assigned any driver as the application may wrongly interpret that the drivers are far away from the user, while in reality the driver might be parked next to the user. A better approach would be to detect during query processing that the user is in city B, and then transform coordinates into a reference system specific to the city as mentioned in the previous example to compute the distances.

[2] Many open datasets today provide location information in lat/long format.

[3] JTS/GEOS do not support geodetic operations: `https://locationtech.github.io/jts/jts-faq.html#geodetic_operations`.
ESRI geometry API has geodesic distance function: `https://github.com/Esri/geometry-api-java/wiki`.

A more **hidden** potential pitfall is while using a spatial index in a planar geometry library. Many popular spatial index structures in these libraries are either designed or implemented with Euclidean distance as a basis for distance computation during various types of index traversals, depending on the query. For example, the R-tree in Java Spatial Index (JSI) assumes Euclidean distance as the metric. So, if an application uses the R-tree to index GPS coordinates and issues a k-NN query to the R-tree, it is bound to get wrong results because the nearest-neighbor search algorithm in the index uses Euclidean distance. Similarly in JTS and GEOS, if a user does not provide a distance metric to the k-NN (or NN) query in the R-tree, the library uses Euclidean distance by default. As an example, consider the description of STR-Packed R-tree in Shapely[4], a popular python geospatial library, which uses GEOS internally. The description gives a simple example of R-tree for a nearest-neighbor query. The user might be using GPS coordinates in the R-tree, and might not be aware that the underlying library GEOS uses Euclidean distance as the metric and thus obtain an unintended error. The correct approach for using a spatial index that indexes geodectic coordinates is shown in [154].

4.3 Queries

In this work we have considered four queries, namely, range, distance, k-nearest neighbor (k-NN) and a spatial point-in-polygon join query. We selected these four queries based on recent research in systems [192] and applications [206]. Simba [192] is a big spatial data analytics system that is optimized for storing location-data and considers (1) range, (2) distance, and (3) k-nearest neighbors Query (k-NN) queries. [206] showcases multiple motivating examples of spatial point-in-polygon join queries which are particularly useful for visual exploration and analysis of urban data.

4.3.1 Range Query

A range query takes a range r (i.e., min/max values for all dimensions D) and a set of geometric objects S. It returns all objects in S that are contained in the range r. Formally:

$$Range(r, S) = \{\ s | s \in S\ \wedge\ \forall d \in D :\\ r\,[d]\,.\text{min} \leq s\,[d] \leq r\,[d]\,.\text{max}\ \}.$$

Practical Example: Retrieve all objects at current zoom level in a maps application (e.g., Google Maps) for a browser window.

[4]`https://shapely.readthedocs.io/en/latest/manual.html#str-packed-r-tree`

4.3.2 Distance Query

A distance query takes a query point q, a distance d, and a set of geometric objects S. It returns all objects in S that lie within the distance d of query point q. Formally:

$$Distance(q, d, S) = \{\ s | s \in S \ \wedge\ \text{dist}(q, s) \leq d\}.$$

Practical Example: Retrieve all dating profiles within 5 kilometers of a user's location.

4.3.3 *k*-nearest neighbors Query

A k-NN query takes a set of points S, a query point q, and an integer $k \geq 1$ as input, and finds the k-nearest points in S to q. Formally:

$$k\text{-}NN(q, k, S) = \{\ s | s \in T \subseteq S \ \wedge\ |T| = k \ \wedge\ \forall t \in T, \forall r \in S - T : d(q, t) \leq d(q, r)\}.$$

Practical Example: Find five closest pizzerias from a user's location.

4.3.4 Spatial Join

A spatial join takes two input sets of spatial records R and S and a join predicate θ (e.g., overlap, intersect, contains, within, or withindistance) and returns a set of all pairs (r, s) where $r \in R$, $s \in S$, and the join predicate θ is fulfilled. Formally:

$$R \bowtie_\theta S = \{\ (r, s) \mid r \in R,\ s \in S,\ \theta(r, s) \text{ holds } \}.$$

Practical Example: Given two datasets, taxi rides (R: points) and neighborhood boundaries (S: polygons), join the two datasets to find how many rides originate (θ: within) from each neighborhood.

4.4 Libraries

In the following section, we will describe the major features of the evaluated libraries. We will also highlight the major services, applications, and systems that use these libraries. Table 4.1 summarizes various features of the libraries, and Table 4.2 summarizes the features of the indexes found in these libraries.

4.4.1 ESRI Geometry API

ESRI Geometry API[5] is a planar geometry library written in Java. ESRI Geometry API comes with a rich support for multiple geometry datatypes, such as point, multipoint, line, polyline, polygon, and envelope and OGC variants

[5] https://github.com/Esri/geometry-api-java

Table 4.1: Selected features of the libraries

Features	S2	GEOS	ESRI	JTS	JSI	jvptree
Language	C++	C++	Java	Java	Java	Java
Indexes	ShapeIndex, PointIndex, RegionTermIndexer	STRtree, Quadtree	Quadtree	STRtree, Quadtree, k-d tree	R-Tree	Vantage Point Tree
Geometry Type	Spherical	Planar	Planar	Planar	Planar	Metric space
Geometry Model	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Area	Point
License	Apache v2.0	LGPL	Apache v2.0	Dual licence (EPL 1.0, BSD)	LGPL	MIT

of these datatypes. It has support for various topological operations, such as cut, difference, intersection, symmetric, union and various relational operations using DE-9IM matrix such as contains, crosses, overlaps etc. ESRI Geometry API also supports a variety of I/O formats, WKT, WKB, GeoJSON, ESRI shape and REST JSON. The geometry library also comes with Quadtree index which cannot be classified into a particular type from the Quadtree family. The key property of any Quadtree is its decomposition rule, in ESRI Quadtree, a leaf node splits into four when the node element count reaches 5 elements, and they are pushed to the children quads if possible.

ESRI Geometry API is used in a variety of products by ESRI such as ArcGIS, ESRI GIS tools for Hadoop, and various ArcGIS APIs. It is also used by the Hive UDFs and by developers building geometry functions for third-party applications such as Cassandra, HBase, Storm, and many other Java-based "big data" applications.

4.4.2 Java Spatial Index

The Java Spatial Index (JSI)[6] is a main-memory optimized implementation of the R-tree [59]. JSI relies heavily on the trove4j[7] library to optimize performance and reduce the memory footprint. The code is open-source, and is released under the GNU Lesser General Public License, version 2.1 or later. The JSI spatial index is limited in features, and only supports a few operations. It is a lightweight R-tree implementation, specifically designed for the

[6] `https://github.com/aled/jsi`

[7] `http://trove4j.sourceforge.net/html/overview.html`

following features (in order of importance): fast intersection performance by using only main memory to store entries, low memory footprint, and fast updates. JSI's R-tree implementation avoids creating unnecessary objects by using primitive collections from the trove4j library. JSI only supports rectangle and point datatypes, and has support for only two predicates for refinement, intersects and contains. The R-tree index can be queried natively for ranges and k-NN.

We could not find any reference of JSI being used in a major system or service, which we believe is mostly due to its limited capabilities. Although limited in features, JSI is still regularly utilized in diverse research areas [99, 100, 108, 107, 172].

4.4.3 JTS Topology Suite and Geometry Engine Open Source

The JTS Topology Suite (JTS) is an open-source Java library that provides an object model for planar geometry together with a set of fundamental geometric functions. JTS conforms to the Simple Features Specification for SQL published by the Open GIS Consortium[8]. GEOS (Geometry Engine Open Source)[9] is a C++ port of the JTS Topology Suite (JTS). Both JTS and GEOS provide support for basic spatial datatypes such as points, linestrings and polygons along with indexes such as the STR packed R-tree and MX-CIF Quadtree [104]. They also support a variety of geometry operations such as area, distance between geometries, length/perimeter, spatial predicates, overlay functions, and buffer computations. They also support a number of input/output formats including Well-Known Text (WKT), Well-Known Binary (WKB).

JTS is used in many modern distributed spatial analytics systems such as Hadoop-GIS [2], SpatialHadoop [40], GeoSpark [203] and SpatialSpark [197] and other research areas [170]. GEOS on the other hand is used in a number of database systems and their spatial extensions such as MonetDB, PostGIS, SpatiaLite, Ingres. GeoPandas and Shapely, two popular geospatial libraries in python, internally use GEOS. It is also used by a number of frameworks, applications and proprietary packages[10].

4.4.4 Google S2 Geometry

S2[11] is a library that is primarily designed to work with spherical geometry, i.e., shapes drawn on a sphere rather than on a planar 2D map, which makes it especially suitable for working with geographic data. S2 supports a variety of spatial datatypes including points, polylines, and polygons. It also has two index structures, namely (i) S2PointIndex to index collections of points in memory and is a variant of Linear Quadtree [104], and (ii) S2ShapeIndex to index arbitrary collections of shapes, i.e., points, polylines and polygons in

[8] `https://www.opengeospatial.org/standards/sfa`
[9] `https://trac.osgeo.org/geos/`
[10] `https://trac.osgeo.org/geos/wiki/Applications/`
[11] `https://github.com/google/s2geometry`

memory. S2 also defines a number of queries that can be issued against these indexes. Indexes also define iterators to allow for more fine-grained access. S2 also accepts input in lat/long (GPS) format.

In recent years, S2 has become a popular choice among various location-based services. It is used by Foursquare [177], on-demand ride-hailing services such as Uber [139] and GO-JEK [151], the location-sharing application Zenly [160] (recently acquired by Snap [63]), the location-based dating application Tinder [142], and by popular games such as Pokémon GO [148] and Ingress [211]. S2 is also used by many database systems, including MemSQL [112], MongoDB [114], HyPer's [78] geospatial extension HyPer-Space [129] and in other research areas [84, 83].

4.4.5 Vantage Point Tree

The vantage point tree [196] is based on metric space and has been well studied in image retrieval and nearest-neighbor search algorithms for high-dimensional data. It is a binary tree which is built recursively. At each node in the tree, the points are split into two equal-sized partitions, and are assigned to its two children. This process is repeated until no points are left or a certain threshold is reached. A node partitions its points by picking one point p at random, the vantage point. The points assigned to the node are then are sorted by their distance to the vantage point p. The resulting sorted array is then split in the middle and assigned to the two children. The distance of the split point from the vantage point p serves as the radius r for the node. All the points that are within the radius r (i.e., the left part of the sorted array) are assigned to the left child of the node, and the rest of the points are assigned to the right child. Based on this partitioning, the tree can then be traversed efficiently to answer distance and k-NN queries. We refer readers to [196] for more details on vantage point trees. We use the library jvptree[12] for an implementation of vantage point tree in our experiments.

4.5 Methodology

To benchmark the various libraries and measure memory costs, we use language specific open-source tools. For Java based libraries, we use the Java Microbenchmark Harness (JMH)[13], which is a framework for building, running, and analyzing benchmarks. To measure the memory consumption in Java, we use the Memory Measurer tool[14]. To benchmark C++ based libraries, we use Google Benchmark[15], and for memory consumption of the indexes in C++, we use the Heap Profiler in TCMalloc[16]. TCMalloc overrides the `malloc` and `new` implementations, and can thus track the memory usage of an application from the amount of memory allocated/deallocated.

[12] https://github.com/jchambers/jvptree
[13] https://openjdk.java.net/projects/code-tools/jmh/
[14] https://github.com/msteindorfer/memory-measurer
[15] https://github.com/google/benchmark
[16] https://github.com/gperftools/gperftools

Table 4.2: Selected features of all indexes

	S2	ESRI	JTS			JSI	jvptree
feature	Point Index	Quadtree	k-d tree	Quadtree	STRtree	R-tree	jvptree
Implementation	Linear Quadtree	Quadtree	k-d tree	MX-CIF Quadtree	STR packed R-tree	R-tree	VPTree
Geometry	Point	Rectangle	Point	Rectangle	Rectangle	Rectangle	Point
Native queries	Range, Distance, *k*-NN	Range	Range	Range	Range, *k*-NN	Range, *k*-NN	Distance, *k*-NN
Updates	Yes	Yes	Insert:Yes Delete:No	Yes	No insertion after build	Yes	No
Default Fanout	32	4	2	4	10	20-50	2

For evaluation, we used two location (points) datasets, the New York City Taxi Rides dataset [122] (NYC Taxi Rides) and geo-tagged tweets in the New York City area (NYC Tweets). NYC Taxi Rides contains 305 million rides from the years 2014 and 2015. NYC Tweets data was collected using Twitter's Developer API [182] and contains 83 million tweets. Figure 4.1 shows the distribution of the rides and tweets in the NYC region. It can be seen that the Taxi rides are mostly centered around central New York whereas the tweets are well distributed over the entire city.

We further generated query datasets that consist of ranges (bounding boxes) in case of range query, query points and distances in case of distance query, and query points in case of *k*-NN query. For range queries and distance queries, we created seven different query datasets for seven different selectivities, ranging from 0.0001% to 1% (i.e., the query selects 0.0001% to 1% of the data). These query datasets consist of one million queries each. We evaluate various indexes in the libraries by issuing these queries sequentially. We chose to generate a large number of queries to minimize the effect of caching tree nodes from a previously issued query. Testing with many queries is especially important in cases with low selectivity where many indexes achieve a throughput of more than 100,000 queries per second. The benchmark frameworks that we use for evaluation run a benchmark multiple number of times until the result is statistically stable. It is thus necessary that we have sufficient queries that do not touch the same nodes in the index structures, but rather exercises several paths in the indexes. To generate these datasets, we uniformly generated points within the New York City bounding box and continuously expanded the range or the distance, depending on which query dataset is being generated, to meet the selectivity requirements. For the *k*-NN query dataset, we uniformly generated points within the NYC bounding box. For the point-in-polygon spatial join query, we use 289 polygons of neighborhood boundaries in NYC.

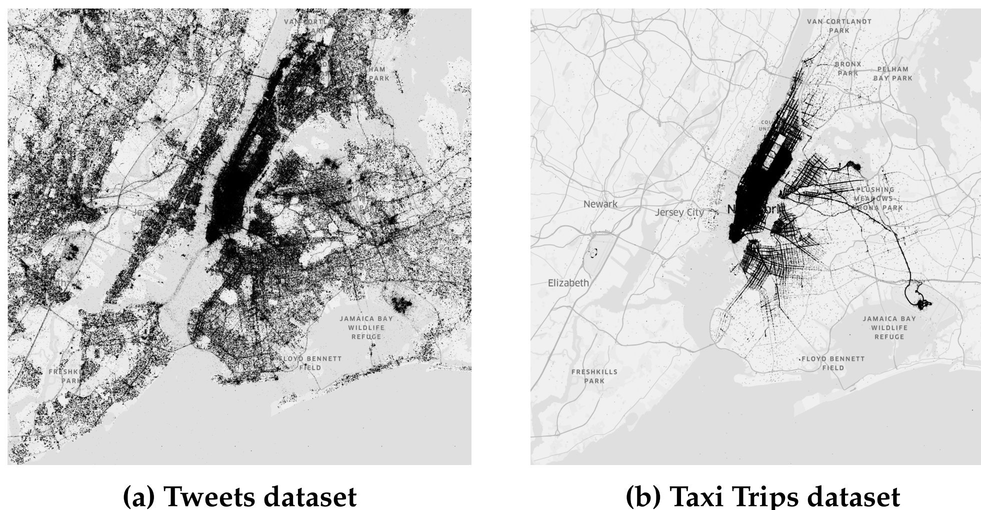
For planar geometry libraries, we projected the datasets to EPSG:32118 using *ogr2ogr* tool in GDAL. We used the *ogr2ogr* tool in GDAL to transform the lat/long coordinates in the datasets.

4.6 Evaluation

All experiments were run single threaded on a two-socket Ubuntu 18.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz turbo)[17] and 256 GB DDR3 RAM. We use the *numactl* command to bind the thread and memory to one node to avoid NUMA effects. CPU scaling was also disabled during benchmarking using the *cpupower* command.

We have benchmarked libraries written both in Java and C++. Although we have used language specific framework and tools to measure the performance of libraries, there are inherently many differences between the languages. For e.g., depending on JVM implementation and C++ compiler

[17]CPU: `https://ark.intel.com/content/www/us/en/ark/products/75272/intel-xeon-processor-e5-2660-v2-25m-cache-2-20-ghz.html`



(a) Tweets dataset **(b) Taxi Trips dataset**

Figure 4.1: Datasets: NYC Taxi trips are clustered in central New York while Tweets are spread across the city

(among various factors), a type int Object in Java requires 16 bytes while a type int in C++ requires 4 bytes. We ask the readers to carefully take such differences between languages into account while comparing performance of libraries written in different languages.

To evaluate the queries, we perform two experiments for each query. In the first experiment, we fix the selectivity of the query to 0.1% (we fix k to 10 in case of k-NN query) and vary the cardinality of the points dataset from 100,000 records to the maximum size of the dataset (i.e., 83 M records for Twitter dataset and 305 M for the Taxi dataset). In the second experiment, we fix the number of points to the maximum size of the dataset and vary the selectivity of the query from 0.0001% to 1% (we vary k from 1 to 10,000 in case of k-NN query). For all these experiments, we measure the throughput for each library in queries/s. In case of spatial join query, we report the join time in seconds. All query implementations are covered under the respective section. If a particular index does not support a query natively, the query is implemented using the *filter and refine* [127] approach.

4.6.1 Indexing Costs

ESRI Quadtree and JSI R-tree accept the rectangular range to index, and an identifier for the rectangular range, whereas other index structures are more liberal and allow users to put any user data along with the rectangular range. To be fair to all index structures, we only store the rectangular range to index and an identifier in every case and measure the size of these indexes in memory.

It is important at this point to categorize indexes in the libraries to better understand their behavior. Indexes in the libraries can be classified as: Point Access Methods (PAMs) and Spatial Access Methods (SAMs) [104]. PAMs are indexing methods that index point data, whereas SAMs index extended spatial objects such as rectangles, polygons etc. S2PointIndex, k-d tree and

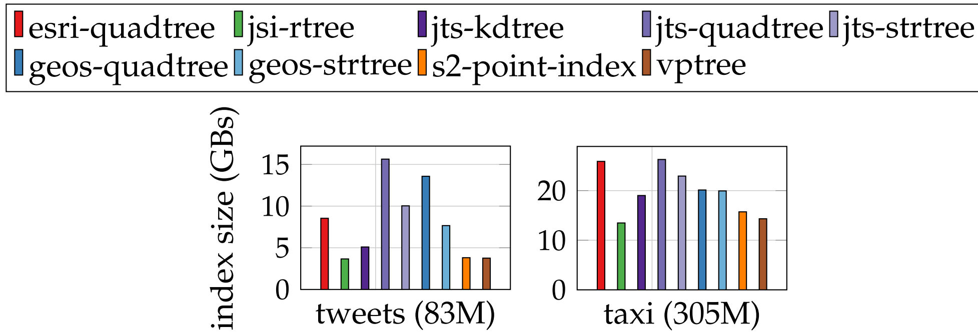


Figure 4.2: Index sizes for the two datasets

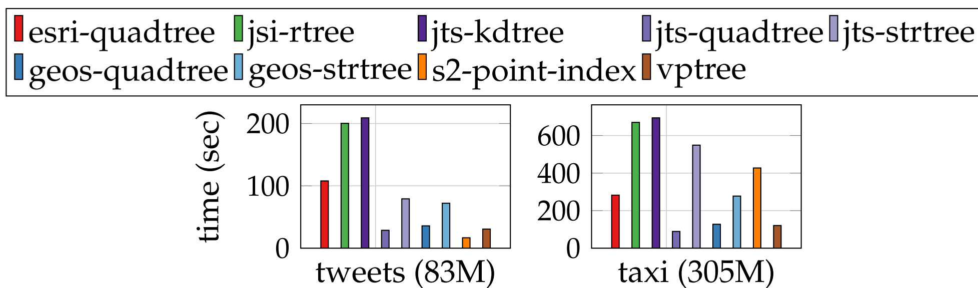


Figure 4.3: Index building times for the two datasets

vptree are PAMs and the rest are SAMs. The indexes can also be categorized as space-driven (follow the embedding space hierarchy), or data-driven (follow the data space hierarchy). k-d tree and Quadtrees are space-driven structures and the rest of the indexes are data-driven.

Figure 4.2 shows the sizes of indexes in various libraries and Figure 4.3 the time it takes to construct them. S2PointIndex, and vptree are PAMs which stores only points (at least two doubles) and hence the memory consumption is minimal. S2PointIndex is a B-tree that stores 64-bit integers (cell ids), and the overhead in inner nodes is minimal. jvptree only stores a vantage point, and a radius at every node, hence the intermediate nodes consume minimal memory. The rest of the indexes are SAMs and store rectangles and consume more memory than PAMs. This is expected, as the trees store rectangles[18], each of which require storage of at least four doubles. Figure 4.2 also shows that the R-tree in JSI consumes very little memory even though it stores rectangles. JSI heavily relies on trove4j[19] collections, which are generally faster to access, and consumes much less memory than Java's Util collections. There are two reasons for low memory consumption. First is that (any) primitive collections store data directly in an array of primitives (int, long, double, char), and thus only a single reference to an array of primitives is needed instead of an array of references to data objects. JSI also uses floating-point precision while the other index structure use double precision values. Second, each primitive data element consumes less memory than

[18]We store points from the datasets as degerate rectangles in SAMs
[19]`http://trove4j.sourceforge.net/html/benchmarks.shtml`

the Object (e.g., type int only requires 4 bytes instead of 16 bytes object Integer). The reason for better performance is that trove4j avoids boxing and unboxing elements every time a primitive value is queried to/from the collection. It can also be seen that the space-driven indexes, i.e., Quadtrees and k-d tree, consumes more memory compared to the other index structures. Since space-driven structures divide the space they index, more internal nodes are formed as they keep dividing the space until a certain threshold is not met for the leaf node size.

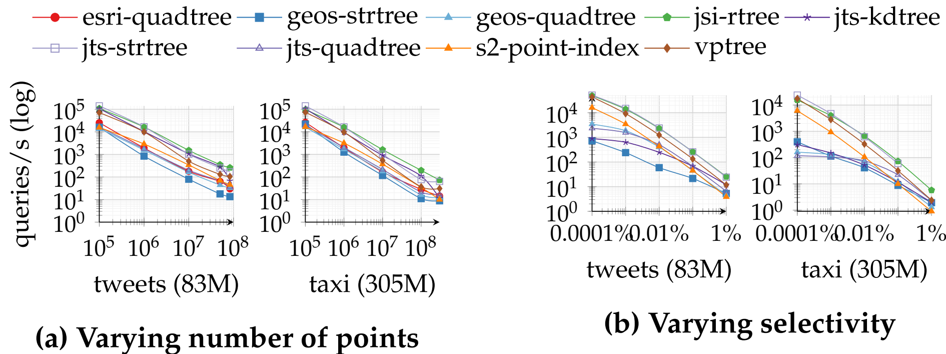
Index construction times have been measured using the benchmarking frameworks, and are averaged over several runs until the runtime is statistically stable. For both Taxi and Twitter datasets, jvptree is the fastest to construct, whereas k-d tree and STRtree in JTS, Quadtree in ESRI geometry API and R-tree in JSI are among the slowest to construct for all datasets.

4.6.2 Range Query

Implementation: All indexes, except for jvptree, natively provide an interface for range queries. To implement range queries in jvptree we first compute the centroid q of the query rectangle. Next, we determine the distance of the centroid q to one of the rectangle's corner vertices. The resulting circle (q, d) is always larger then the range query rectangle and can therefore be used as a *filter* to retrieve a list with qualifying points. This list is then *refined* to determine which points are actually contained in the range query rectangle. As mentioned earlier, k-d tree in JTS keeps a count of points, in case of duplicate points (up to a certain distance tolerance), rather than creating a new node for the duplicate points. We make sure that we materialize all such points for the range query, but we do use them as an optimization in distance and join query to reduce the refinement costs (i.e., skip refinement for duplicate points if one point qualifies the refinement check).

Another point to mention here is that Quadtree implementation in ESRI geometry API requires tuning. The initialization of the Quadtree expects a height parameter for the index. As mentioned in section 4.5, we generated range queries with varying selectivities from 0.0001% to 1%. We ran all these range queries from selectivity 0.0001% to 1% on both datasets, and varied the height of the Quadtree from 1 to 64 for both datasets and for each selectivity. We then ranked these heights based on the lowest query runtime for each query selectivity, and compute the aggregated rank of all heights across all selectivities. We then selected the height with the lowest rank for both datasets. We found that the Quadtree performed best with heights 18 and 9 for the Taxi and Tweets datasets respectively.

Analysis: Figure 4.4 shows the range query performance of various libraries on the Taxi and Twitter datasets. For both datasets, JSI R-tree show the best throughput numbers (259.87 and 72.779 queries per second, respectively, for Twitter and Taxi dataset for 0.1% selectivity). JSI R-tree is optimized for main memory usage for range queries and has the least height of all indexes (5 and 7 in the two datasets). Many of the tree nodes are cached and it suffers from the least number of cache misses as shown in Table 4.3.



(a) Varying number of points

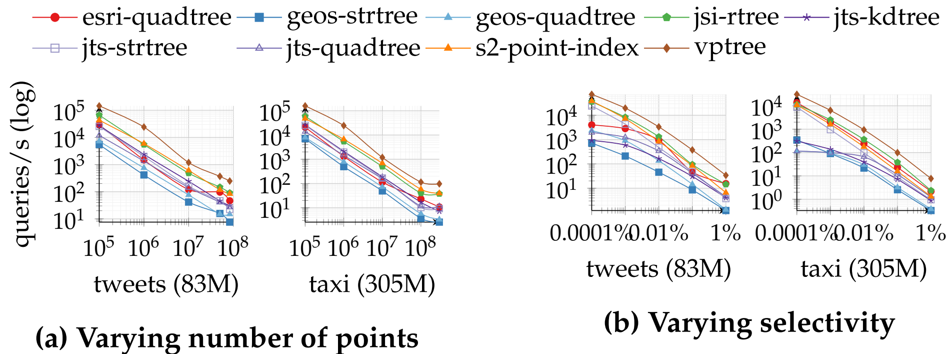
(b) Varying selectivity

Figure 4.4: Range query performance varying the number of points and selectivity of the query rectangle for NYC Taxi and Twitter Datasets

Table 4.3: CPU Counters - Range query datasize = 50M tweets, selectivity = 0.1 %, 1 thread, normalized by the number of range queries. All values are in millions except IPC.

	cycles	ipc	instr	L1 miss	LLC miss	branch miss
esri-quadtree	116	0.84	98	1.34	0.54	0.08
geos-quadtree	105	0.75	79	0.97	0.75	0.09
geos-strtree	236	0.37	88	4.04	**2.68**	0.51
geos-cfstrtree	91	0.87	80	1.21	**0.57**	0.46
jsi-rtree	8	1.25	10	0.13	0.06	0.03
jts-kdtree	8	1.12	9	0.14	0.02	0.04
jts-quadtree	68	1.17	80	0.82	0.27	0.19
jts-strtree	31	0.81	25	0.42	0.22	0.01
s2-pointindex	44	1.34	59	0.42	0.05	0.36
vptree	30	0.70	21	0.68	0.21	0.05

An interesting case in the results is the low query throughput of GEOS STRtree (17.8315 queries per second in the Tweets dataset for 50 M points and 0.1% selectivity). GEOS STRtree is much slower than the JTS STRtree. Upon investigation, we found that the reason for the low query throughput of STRtree in GEOS is an implementation artifact. It can be seen in Table 4.3 that GEOS STRtree suffers from a large number of LLC misses, 2.68 million in the Twitter dataset and 1.28 million in the Taxi dataset (not shown in table). R-trees in general store multiple rectangles at every node. When the tree is queried, the decision to explore the branches from each node in the tree is based on whether the query range overlaps any of these rectangles. In both cases, JTS and GEOS, every node in the STRtree contains a maximum of 10 such rectangles by default. GEOS STRtree stores a vector of **pointers** to these rectangles at every node. At every node, the algorithm in the range query iterates over these pointers, retrieves these rectangles from memory and checks whether there is any overlap with the query range and then based on the overlap explores the various branches from the node. Retrieving these rectangles from memory causes many cache misses in GEOS STRtree during the query execution. To validate this, we implemented a cache-friendly



(a) Varying number of points

(b) Varying selectivity

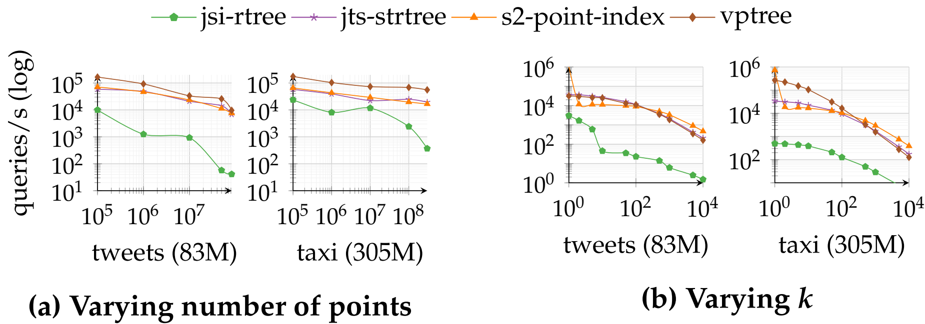
Figure 4.5: Distance query performance varying the number of points and selectivity of the query rectangle for NYC Taxi Dataset and Twitter Datasets

STRtree (designated as cfstrtree in the Table 4.3) in GEOS on top of the existing tree. We basically introduced another vector at every node in the tree, which stores the objects of these rectangles in contiguous memory. We replaced the logic to check for overlap to use these rectangle objects rather than the pointers to the rectangles. This reduces the number of LLC misses in the CFSTRtree relative to STRtree, by a large number as can be seen in Table 4.3.

STRtree implementation in JTS does not suffer from this. In both libraries, GEOS and JTS, the algorithm for constructing and traversing the trees are the same, but the difference in performance stems from how memory management works in the JVM. Every node in JTS STRtree stores the rectangle objects in a *List*. Lists in Java store the references to the objects, so logically it is similar to storing a vector of pointers in C++. But where this differs is that JVM makes a distinction between small and large objects during object allocation [184]. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between two and 128 kB. Small objects are allocated in thread local areas (TLAs). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. The size of the rectangle objects in JTS is 48 bytes each. This means that the rectangle objects qualify as small objects and are in contiguous memory. Only the access to the first rectangle causes a cache miss, and the other objects are most likely brought into memory as a side effect of that cache miss (speculative loading).

4.6.3 Distance Query

Implementation: S2PointIndex and jvptree provide native support for distance queries, so we directly issue the query point and the distance to these two indexes. The other indexes do not support distance query natively. To implement distance queries in these indexes, we again use the *filter and refine* paradigm. We first filter using a rectangle, whose corner vertices are at



(a) Varying number of points **(b) Varying k**

Figure 4.6: kNN query performance varying the number of points and k for NYC Taxi and Twitter Datasets

a distance of d from the query point q. We issue a range query to the various range based indexes using this rectangle. We then refine the resulting candidate set of points by using a *withinDistance* predicate (available in ESRI Geometry API, JTS, and GEOS). For JSI, we implemented our own predicate, which computes the Euclidean distance for all candidate points from the query point and checks if the candidate point is within distance $d * d$ rather than d from the query point. This helps in skipping the square root operation to calculate Euclidean distance.

Analysis: Figure 4.5 shows the distance query performance on Taxi and Twitter datasets. The performance for distance query is dominated by range query lookup for most indexes, apart from S2PointIndex and jvptree. These index support distance queries natively, i.e., have specialized tree traversal algorithms for distance query. For other indexes, we deploy the filter and refine paradigm. The performance of these indexes thus follows directly from the range query performance. JSI R-tree is slightly better than JTS k-d tree as we optimize the Euclidean distance computation by skipping the square root operation. We would also advise the readers to use this approach for refinement in GEOS as well. The *isWithinDistance* function in GEOS returns whether two geometries are within a certain distance from each other. By profiling the function we noticed that this function makes six *malloc()* calls, for every candidate point, which degrades the performance. By using our own predicate distance function, we were able to speed up distance query by up to 2× in GEOS. In many geometric operations, GEOS frequently allocates and deallocates memory, which is an overhead. This problem in memory management was also observed by [197], where authors use GEOS to introduce spatial processing in Impala.

4.6.4 k-NN Query

Implementation: Out of all the available indexes, only S2PointIndex, JTS STRtree, JSI R-tree, and jvptree support k-NN queries natively. We directly issue the query point to these indexes and measure their performance. We did not implement any tree traversal algorithms for any other available tree because we wanted to measure the performance of the libraries without making any changes to the library source code.

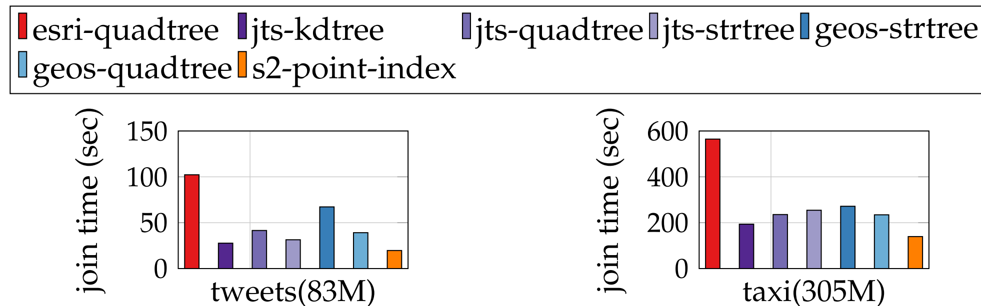


Figure 4.7: Join query performance for NYC Taxi and Twitter Datasets

Analysis: Figure 4.6 shows k-NN query performance of various indexes on the Taxi and Twitter datasets. jvptree again takes the crown as the best performing index for k-NN queries, with S2PointIndex close behind. It can be observed that for the Twitter dataset the performance of JSI R-tree fluctuates quite a bit. This can be explained by how the nearest-neighbor algorithm works in JSI R-tree (and also in JTS STRtree), which is known as branch-and-bound traversal. The algorithm starts with adding the root node to a priority queue of size k. The algorithm, then iterates over the tree continuously adding nodes until the priority queue is full. The algorithm then continues traversing the tree observing nodes, replacing the current farthest node in the queue with the current node being looked at, if it is closer. The JSI R-trees for different sized datasets are vastly different since JSI R-tree is a *dynamic* R-tree, the nodes are split at various times during insertion based on multiple factors. Thus, during the tree traversal for k-NN query, sometimes a large number of branches from a node can be dropped since they are not closer than the current farthest node in the priority queue and sometimes they cannot be dropped. This can lead to multiple search paths to be evaluated and hence the fluctuation in performance. JTS STRtree packed R-tree does not suffer from this because it is a type of *static* R-tree. It is built once after which no more elements can be added to it. STRtree is built by first sorting the leaf node in the x dimension, and then dividing the data in vertical slices, each containing an equal number of points. Within each slice, the data is sorted in the y dimension, and again divided into slices containing an equal number of points. The tree is then built on top of these slices by packing a pre-defined number of slices into nodes. The difference in tree node boundaries is still there in JTS STRtree but is more profound in the lower levels of the tree, rather than at various levels as in the case of JSI R-tree. Thus, JSI R-tree can sometime quickly discard branches at the top of the tree and other times it cannot, and this is reflected in the query throughput.

4.6.5 Point-In-Polygon Join Query

Implementation: In S2, we used the S2ShapeIndex, instead of S2PointIndex, which provides a native interface for the contains predicate. S2ShapeIndex[20]

[20] http://s2geometry.io/devguide/s2shapeindex.html

stores a map from S2CellId to the set of shapes that intersect that cell. The shapes are identified by a ShapeId. As shapes are added to the index, their cell ids are computed and added along with the shape id to the index. When a query point is issued against the index it retrieves the cells that contain the query point and identifies the shape(s) that this containing cell belongs to using the shape id. For other indexes, we again use the filter and refine approach. For GEOS and JTS we use PreparedGeometry[21] to index line segments of all individual polygons, which helps in accelerating the refinement check. In JTS, we also use k-d tree's points snapping technique to skip refinement for duplicate points in case one point qualifies or disqualifies the predicate check. In ESRI implementation, we use AcceleratedGeometry and set its *accelDegree* to *enumHot*[22] for the fastest containment performance.

Analysis: Figure 4.7 shows joins query performance on the Taxi and the Twitter datasets. Spatial join queries are notoriously expensive and this is reflected in the figure. For join queries S2ShapeIndex performs the best. As mentioned earlier, we skip the refinement check for duplicate points if one such point qualifies (or disqualifies) the refinement check and that is why it does slightly better than the other indexes. S2ShapeIndex natively supports the containment query and traverses the index appropriately and does not have to deal with refining many candidate set of points. The performance of other indexes follows from the range query performance. JTS/GEOS STRtree and Quadtree perform better than ESRI Quadtree because the refinement using PreparedGeometry is faster than AcceleratedGeometry in ESRI.

4.7 Discussion

In this section, we first discuss a research direction that we believe might not be getting the attention in the community that it should, before we outline how to use modern spatial libraries as building blocks for building distributed spatial systems.

4.7.1 Why Refinement Should Be Looked At?

As we learned in the past sections, the modern spatial libraries provide index structures which arrange spatial objects in a way that the access time to these geometric objects reduces. But we also learned that these index structures only support a limited set of native queries (range lookup and k-NN query in most cases). In other queries, such as distance query and spatial joins, these index structures primarily act as filters. The resulting candidate set of points (or geometries) after the filter phase needs to be further refined based on a spatial predicate. For distance query, the predicate is *withinDistance*, and for spatial joins, the predicate can be one of many predicates, such as

[21] `https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/prep/PreparedGeometry.html`

[22] `https://esri.github.io/geometry-api-java/javadoc/com/esri/core/geometry/Geometry.GeometryAccelerationDegree.html`

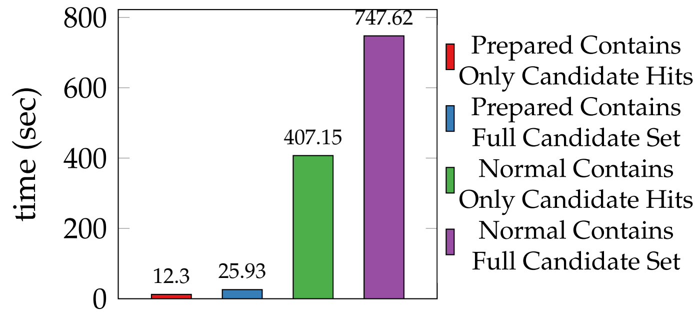


Figure 4.8: Refinement costs for Midtown Manhattan Polygon for NYC Taxi Dataset using various contains functions in JTS

contains, intersects, overlaps, etc. For these queries, we used the filter and refine paradigm. The set of geometry objects from the candidate set that do not qualify the predicate check are known as *false drops* and the ones that do are known as *candidate hits*. Generally, we can determine how good these indexes are for such queries by analyzing the ratio of number of false drops to the number of candidate hits. If the ratio is more than 1, it can be deduced that the amount of work being done for *false drops* is more than for *candidate hits*. This work done can be classified as an overhead, and the goal is to *minimize* this overhead.

In this study we also looked at an index structure, namely, Vantage Point Tree, which is specially designed to answer distance and *k*-NN queries. We saw in Section 4.6 that for distance queries an open-source implementation of VPTree, performs 2.48× better for the Taxi dataset (and 2.74× for the Twitter dataset) than its closest competitors S2PointIndex and JSI R-tree. Please note that in JSI R-tree we even skipped the overhead of square root operation in Euclidean distance computation. This is because jvptree reduces the overhead of *false drops* during the index lookup itself. In essence, the index structure completely skips the refinement phase for distance and *k*-NN queries and does not have to deal with *false drops*. This shows that if an index structure is built to answer certain queries, and no refinement is needed, the performance implications can be large.

Recent research acknowledges [180] [18] that there is potential in accelerating the refinement step for join queries. We consider the spatial point-in-polygon join query here, where filter and refinement is also required for some indexes. In point-in-polygon join, after the filter phase, the candidates set of points is typically refined using an algorithm known as ray tracing. In this algorithm, a line (ray) is drawn from the query point to a point known to be outside the polygon, and then the number of intersections of this line with all edges in the polygon is counted. This algorithm is linear with the number of edges in the polygon. So if the cardinality of the filtered candidate set of points after filtering from the index is n, then the work to be done is $\mathcal{O}(nk)$, where k is the number of edges in the polygon. If the n is large or if the polygons are complex, with a large number of edges then this has the potential to become a bottleneck. Midtown Manhattan is one of the neighborhoods in NYC that is highly skewed for the Taxi and the Twitter dataset alike. Using

the bounding box of Midtown Manhattan and querying any range-based index (i.e., which can be queried using a range, in this case MBR of Midtown Manhattan) as a filter with 305 million taxi rides, yields a candidate set with 78.35 million points. The final result after refinement has 42.55 million points (*candidate hits*), with **35.8 million** points being *false drops*.

Using Midtown Manhattan as a query polygon, we carried out an experiment to determine the costs of refinement using various contains functions in JTS and the results are shown in Figure 4.8. In PreparedGeometry, the individual geometry objects are indexed and the indexing scheme varies based on the geometry datatype. For example, for polygons, PreparedGeometry indexes the line segments of the polygons. If the refinement step can be skipped for *false drops*, there is gain of 2.10× in query performance (12.3 seconds without false drops vs. 25.93 seconds with false drops). The figure also shows the effect of indexing individual polygons. If line segments in polygons are not indexed, the polygon contains function takes 747.62 seconds compared to 2.93 seconds (28.83× improvement).

There are two potential research directions for improving point-in-polygon spatial join queries. As mentioned earlier, the potential work to be done in the refinement phase after filtering is $\mathcal{O}(nk)$. We can either try to reduce n or k (or both). Some of the recent research work [83, 84, 206, 82] tries to address the former and skip the refinement phase altogether. The latter is addressed to some extent in the libraries via PreparedGeometry (in JTS and GEOS) and AcceleratedGeomerty (in ESRI Geometry API). There is also a research [216] work that show that the refinement step can be improved by using interval trees to index the polygon line segments.

4.7.2 Distributed Spatial Analytics Systems

In the past few years, a number of big spatial analytics systems have emerged. While they differ in some architectural design aspects, many of the core fundamentals remain the same in terms of building a distributed spatial processing system. In this section, we briefly highlight these fundamentals and how a distributed spatial analytics system can be built from scratch using the libraries studied in this work. A cluster of commodity machines coordinating to complete a task generally have the following structure: a master node (the coordinator) and multiple worker nodes. Big spatial analytics systems today also deploy the same cluster setup since they are primarily built on big data infrastructures in the form of Hadoop, Apache Spark, Impala etc. There are three main components to designing a big spatial analytics system: (i) Partitioning Technique, (ii) Index Structures, and (iii) Supported Datatypes and Queries.

Index structures, as we saw in this work, are important for answering spatial queries. Spatial indexes allow access to the desired spatial objects in sub-linear time and thus accelerate spatial query processing. Index structures hence form an integral part of any spatial processing system, whether it be a relational database system, or a distributed spatial processing system.

Spatial partitioning is also an important part of distributed spatial processing system, which we will discuss in detail:

Spatial Partitioning

Spatially partitioning the input dataset(s) is an important aspect of distributed spatial processing system. Since there are multiple worker nodes in a cluster, an input dataset should be partitioned to fully utilize the parallel computing capability of the cluster. Also, since many of the spatial datasets are inherently spatially skewed, it is important to partition them spatially. A naive grid partitioning would introduce skew in some individual grid cells, and thus leads to the *straggling* nodes in the cluster, which would affect the overall query efficiency.

How is it done?: The usual practice today to build partitions is to sample the input dataset and to determine partitions based on the sample. Previous research [39] has shown that sampling 1% of the input dataset is sufficient to produce high-quality partitions. To further delve into detail, we will walk through an example, using an R-tree index. After sampling the input dataset, an R-tree is built on the sample. Sampling helps in capturing the density distribution of the input dataset, and indexing the samples in an R-tree spatially partitions the sample, thereby providing the *partitions boundaries* of the sample dataset. The minimum bounding rectangle (MBR) of the leaves of the R-tree are then used as the *partition boundaries*. Once the partition boundaries have been determined, the input dataset can then be loaded in parallel using these boundaries. Now since these partition boundaries were determined using only a sample of the dataset, and the input dataset may contain spatial objects that do not lie, or even overlap multiple partition boundaries. The common practice today is to expand the partition boundaries or duplicate the object in multiple partitions. We refer the readers to [39] to understand the trade-offs related to such decisions. Once the partitions have been built, the individual partition are indexed in an R-tree. The index does not necessarily have to be an R-tree but for the sake of continuity, we continue using the R-tree as an example. These index within individual partitions are called *local index* (i.e., local to a partition). Once these local indexes have been built, finally a *global index* is built using the spatial extent of these local indexes. We walked through an example with R-tree as the index to determine the partition boundaries, but R-tree may not be the best partitioning scheme in certain scenarios. We refer the readers to [39], which thoroughly compares and evaluates various spatial partitioning techniques.

Why is it done?: Spatial partitioning an input dataset helps in query processing. To better understand the importance of spatial partitioning, we will walk through an example. Consider a large input dataset, and a range query is issued to determine which spatial objects in the input dataset lie within the given range. The *Global index* is first used to determine which partitions the input range overlaps and then the overlapping partitions can be scanned with the given range. This saves unnecessary scans of the partitions that do not overlap the input range. This is a very simple example, but things get

more complex when join queries are considered. A join query can be processed as follows: the global indexes of the two datasets are first consulted to determine the partitions that overlap each other and then these partitions can be joined in parallel. This is again a very simple example of how the spatial partitions and indexes can be used to process a join query, and how to avoid joining partitions that do not spatially overlap. In reality, the systems deploy query optimizers that determine the best way to join the two datasets. For example, when the two global indexes are considered to determine which partitions overlap, it could very well be that a large number of partition pairs overlap (since they are two different datasets). A system may choose to repartition one dataset to minimize these overlapping partition pairs. These are design choices that these systems make, and they are based on various trade-offs. We refer the readers to the individual systems to better understand these design choices and trade-offs.

4.8 Related Work

To the best of our knowledge, no previous work in literature has evaluated the spatial libraries studied here empirically. One research work [73] compares indexing techniques for big spatial data, where the authors consider many big spatial data systems and one spatial library JSI, only to report the performance of each system/library on a standalone basis. [197] implement spatial query processing in Apache Spark, and Apache Impala using JTS and GEOS, respectively. They do observe some of the implementation differences between JTS and GEOS, but largely the work is about a comparative study of spatial processing in the Spark and Impala. [128] compares five Spark based spatial analytics systems, some of which use JTS library for spatial query processing. [54] shows how to efficiently implement distance join queries in distributed spatial data management systems.

4.9 Conclusions

In this work we empirically compared popular spatial libraries using four different queries: range query, distance query, k-NN query, and a point-in-polygon join query. We performed a thorough experimental evaluation of these libraries using two real-world points datasets. While we evaluated the libraries on the point dataype, there are other datatypes (such as linestring, polylines etc.) in the libraries that should also be evaluated. We leave evaluating the libraries on other geometric datatypes for future work.

Table 4.4 summarizes the strengths and weaknesses of the spatial libraries.

There is no clear winner for each of the considered queries, and this is mostly because all the indexes available in the libraries do not support all these queries natively (i.e., do not have specialized tree traversal algorithms for each query). ESRI geometry API and JTS/GEOS are complete planar geometry libraries, and are rich in features. They support multiple datatypes,

Table 4.4: Strengths/Weaknesses of the Libraries

Library	Strengths	Weaknesses
ESRI	(1) Active development and support	(1) Quadtree requires tuning
	(2) Full geometric types, refinements, and operations	
JSI	(1) R-tree performance as a filter	(1) No active development
		(2) No geometric refinements
GEOS and JTS	(1) Active development and support	(1) Memory management in GEOS requires improvement
	(2) Full geometric types, refinements, and operations	
jvptree	(1) Best distance and k-NN performance	(1) No geometric refinements
S2	(1) Best suited for geographic data	
	(2) Active development and support	
	(3) Many practical queries natively supported	

and have a variety of topological and geometry operations. They are also under active development and have a community for support. They do, however, come with some drawbacks. ESRI Quadtree has to be tuned for the dataset that it indexes, and memory management in GEOS could be improved. The k-d tree in JTS lacks support for the k-NN query. There are algorithms available to traverse the k-d tree efficiently in order to answer the k-NN queries and implementing the algorithm in the index would be a welcome addition. The R-tree in JSI exhibited the best performance for range lookups, however, JSI is very limited in features, and is also not under active development. Google S2 is a spherical geometry library and is best suited to work with geographic data. It is active under development and is used in many multimillion-dollar industries. It also has many practically used queries that are implemented natively on various indexes. Finally, jvptree, is a library that implements the Vantage Point Tree. It exhibits the best performance for distance and k-NN queries as it is specifically designed to answer these queries. The index can only be used as a filter for other queries, and users have to implement their own refinement operations for such queries.

We also identified areas of potential pitfalls in using the planar geometry libraries which can be critical from the perspective of actual users, either of these libraries or any system that is based on them. Particularly for distance computations, the differences can be significant when using planar geometry for processing GPS coordinates. Many important business decisions might be based on the outcome of such queries and there are potentially hundreds of

users and companies that are using software that is based on these state-of-the-art spatial libraries. While these libraries and systems correctly execute what they are designed to do, users should be aware of how to use them correctly.

Chapter 5

The Case For Learned Spatial Indexes

Excerpts of this chapter have been published in [130].

5.1 Introduction

With the increase in the amount of spatial data available today, the database community has devoted substantial attention to spatial data management. For e.g., NYC Taxi Rides open dataset [122] consists of pickup and drop-off locations of more than 2.7 billion rides taken in the city since 2009. This represents more than 650,000 taxi rides every day in one of the most densely populated cities in the world, but is only a sample of the location data that is captured by many applications today. Uber, a popular ride hailing service available via a mobile application, operates on a global scale and completed 10 billion rides in 2018 [183].

The unprecedented rise of location-based services has led to a considerable amount of research efforts that have been focused on four broad areas; (1) systems that scale out [2, 8, 40, 42, 61, 173, 175, 178, 192, 197, 203, 204, 202, 17], (2) support for spatial processing in databases [112, 106, 114, 126, 129], (3) improving spatial query processing [53, 82, 83, 84, 159, 180, 179, 181, 208, 146, 130, 201, 200], and (4) leveraging modern hardware and compiling techniques [36, 37, 168, 169, 206, 98], to handle the increasing demands of applications today.

Recently, Kraska et al. [91] proposed the idea of replacing traditional database indexes with learned models that predict the location of a key in a sorted dataset, and showed that learned models are generally faster than binary search. Kester et al. [79] showed that index scans are preferable over optimized sequential scans in main-memory analytical engines if a query selects a narrow portion of the data.

In this chapter, we build on top of these recent research results, and provide a thorough study for the effect of applying ideas from learned index structures (e.g., Flood [117]) to classical multi-dimensional indexes. In particular, we focus on five core spatial partitioning techniques, namely Fixed-grid [15], Adaptive-grid [119], Kd-tree [14], Quadtree [46] and STR [101]. Typically, query processing on top of these partitioning techniques include three phases; index lookup, refinement, and scanning (Details of these phases

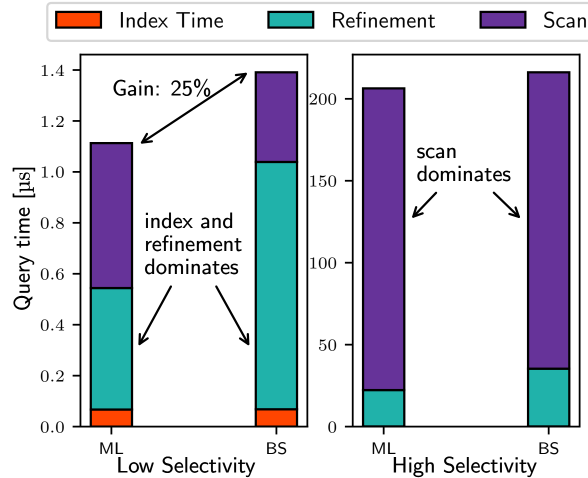
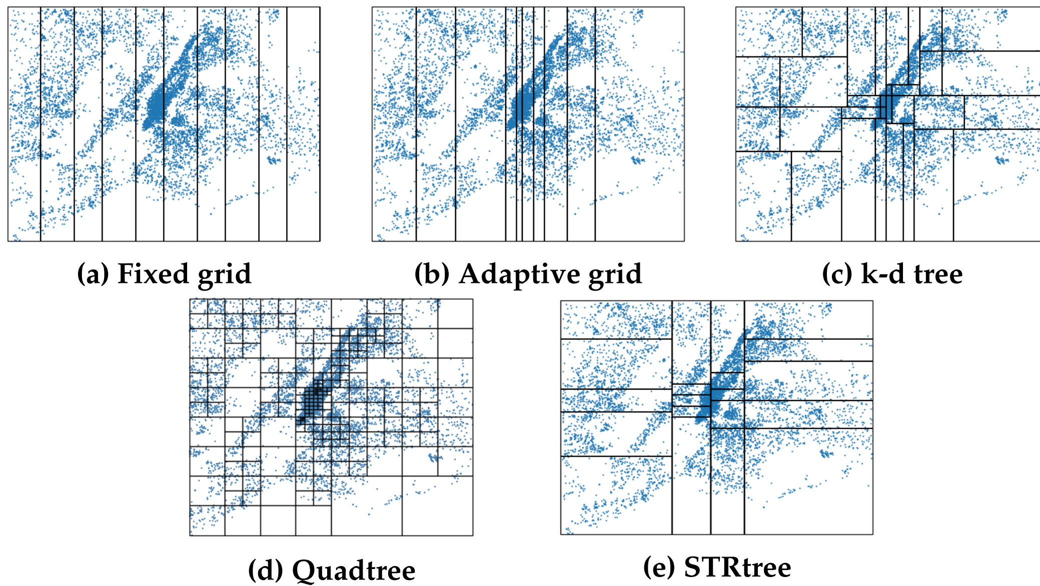


Figure 5.1: Machine Learning vs. Binary Search. For low selectivity (0.00001%), the index and refinement phases dominate, while for high selectivity (0.1%), the scan phase dominates (parameters are tuned to favor Binary Search)



(a) Fixed grid **(b) Adaptive grid** **(c) k-d tree**

(d) Quadtree **(e) STRtree**

Figure 5.2: An illustration of the different partitioning techniques

are in Section 5.2.3). We propose to replace the typical search techniques used in the refinement phase (e.g., binary search) with learned models (e.g., RadixSpline [86]).

Interestingly, we found that, by using a learned model as the search technique, we can gain a considerable speedup in the query run-time, especially for low selectivity range queries (Similar to the observation from Kester et al. [79]). Figure 5.1 shows the average running time of a range query using Adaptive-grid on a Tweets dataset, which consists of 83 million records (Section 5.3.1), with and without learning. It can be seen that for a low selectivity query (which selects 0.00001% of the data, i.e., 8 records) the index

and refinement times dominate the lookup, while for a high selectivity query (which selects 0.1% of the data, i.e., 83 thousand records) the scan time dominates. Another interesting finding from our study is that 1-dimensional grid partitioning techniques (e.g., Fixed-grid) can benefit from the learned models more than 2-dimensional techniques (e.g., Quadtree). Our study will assist researchers and practitioners in understanding the performance of different spatial indexing techniques when combined with learned models.

5.2 Approach

In this section, we first explain how a range query processing has been implemented. Then, we describe the spatial partitioning techniques that we have implemented in our work. We conclude the section by describing the search techniques used within the individual partitions.

5.2.1 Partitioning Techniques

Multidimensional access methods are generally categorized into Point Access Methods(PAMs) and Spatial Access Methods (SAMs) [49]. The Point Access Methods have been primarily designed to perform spatial searches on point data, which do not have any spatial extent. Spatial access methods, however, manage extended objects such as linestrings, polygons etc. In this work we primarily focus on PAMs.

Spatially partitioning a dataset into partitions (or cells), such that the objects within the partitions are also close in space, is known as spatial partitioning. Spatial partitioning techniques can be classified into space partitioning techniques (partitions the embedded space) or data partitioning techniques (partitions the data space). In this chapter, we employ Fixed-grid [15], Adaptive-grid [119], and Quadtree [46] as space partitioning techniques; and Sort-Tile-Recursive [101] and K-d tree [14] as data partitioning techniques. We will refer to the set of these five spatial partitioning techniques as α. Figure 5.2 illustrates these techniques on a sample of the Tweets dataset used in our experiments (details are in Section 5.3.1), where sample points and partition boundaries are shown as dots and grid axes respectively.

Fixed and Adaptive Grid

The grid (or cell) methods were primarily designed to optimize retrieval of records from disk and generally they share a similar structure. The grid family imposes a d-dimensional grid on the d-attribute space. Every cell in the grid corresponds to one data page (or bucket) and the data points that fall within a particular cell boundary resides in the data page of that cell. Every cell thus has to store a pointer to the data page it indexes. This mapping of grid cells to data pages is known as the grid directory. The Fixed-grid [15] method requires that the grid subdivision lines to be equidistant. The Grid File [119], or the Adaptive-grid, on the other hand relaxes this restriction. Since the grid subdivision lines are not equidistant in the case of

Grid File, it introduces an auxiliary data structure called linear scales, which are a set of d-dimensional arrays and define the partition boundaries of the d-dimensions. Flood [117] is a state-of–the–art learned multi-dimensional index for d-dimensional data, which partitions the data using a grid over d-1 dimensions and uses the last dimension as the sort dimension. In our implementation, the grid partitioning techniques use a similar approach where the space is divided in one dimension and the other dimension is used as the sort dimension.

Quadtree

Quadtree [46] along with its many variants is a tree data structure that also partitions the space like the k-d tree. The term quadtree is generally referred to the two-dimensional variant, but the basic idea can easily be generalized to d dimensions. Like the k-d tree, the quadtree decomposes the space using rectilinear hyperplanes. The important distinction is that quadtree is not a binary tree, and the interior nodes in the tree have 2^d children for d-dimensions. For d = 2, each interior node has four children, each corresponding to a rectangle. The search space is recursively decomposed into four quadrants until the number of objects in each quadrant is less than a predefined threshold (usually the page size). Quadtrees are generally not balanced as the tree goes deeper for the areas with higher densities.

K-d tree

K-d tree [14] is a binary search tree that recursively subdivides the space into equal subspaces by means of rectilinear (or iso-oriented) hyperplanes. The subdivision alternates between the k dimensions to be indexed. The splitting hyperplanes at every level are known as the discriminators. For k = 2, for example, the splitting hyperplanes are alternately perpendicular to the x-axis and the y-axis, and are called the x-discriminator and the y-discriminator respectively. The original K-d tree partitioned the *space* into equal partitions, for example if the input space consists of GPS co-ordinate system (-90.0, -180 to 90, 180) the space would be divided into equal halves (-45, -90 to 45, 90). K-d trees are thus not balanced if the data is skewed (most of which might only lie in one partition). K-d tree can be made data-aware by selecting a median point from the data and dividing the data into two halves. This ensures that both partitions in the binary tree are balanced. We have implemented the data-aware k-d tree in our work.

Sort-Tile-Recursive (STR) packed R-tree

An R-tree is a hierarchical data structure that is derived from the B-tree and is primarily designed for efficient execution of range queries. R-tree stores a collection of rectangles and any arbitrary geometric object can be stored by the approximation of the object in the form of a minimum bounding rectangle (MBR). At every level a node in the R-tree stores a maximum of N entries, each of which contains a rectangle R, and a pointer P. At the leaf level, the

pointer points to the actual object and R is the minimum bounding rectangle of the object, while in the internal nodes R represents the minimum bounding rectangle of the subtree pointed to by P.

Sort-Tile-Recursive [101] is a packing algorithm to fill R-tree [59] and aims to maximize space utilization. The main idea behind STR packing is to tile the data space into $S \times S$ grid. For example, consider the number of points in a dataset to be P and N be the capacity of a node. The data space can then be divided into $S \times S$ grid where $S = \sqrt{P/N}$. The points are first sorted on the x-dimension (in case of rectangles, the x-dimension of the centroid) and then divided into S vertical *slices*. Within each vertical slice, the points are sorted on the y-dimension, and packed into nodes by grouping them into runs of length N thus forming S horizontal slices. The process then continues recursively. Packing the R-tree in this way packs all the nodes completely, except the last node which may have fewer than N elements.

5.2.2 Building Index

In this section, we outline how a learned index can be used to index a given location dataset D. The location dataset D consists of locations available in latitude/longitude format. For ease of understanding, we will refer to them as x-dimension and y-dimension respectively. We first sort D on the x-dimension. Note that for Quadtree partitioning techniques this step is not required. Next, we partition the input dataset D using one of the techniques described in 5.2.1 (or in other words from α). The size of each partition is l (also called the leaf size or the partition size) locations. Once the input dataset has been partitioned, we loop over all the partitions of the dataset. For each partition, we sort all the points within the partition on the y-dimension and then we build a learned index on the y-dimension. The algorithm for building the index is outlined in Algorithm 1.

Algorithm 1 : A generic way of building indexes

Input : D: the input location dataset; l: the partition size
Output : D': the partitioned and indexed input dataset

```
1 Sort(D, x)
2 P ← Partition(some approach from α, l)
3 for p ∈ P do
4   | Sort(p, y)
5   | BuildLearnedIndex(p, y)
6 end
7 return D'
```

5.2.3 Range Query Processing

A given range query takes as input a query range q that has a lower and an upper bound in both dimensions. More specifically, q is represented by a

lower bound (q_{xl}, q_{yl}) and an upper bound (q_{xh}, q_{yh}). It also takes as input a set of input location dataset D, where each location point p is represented by the two dimensions, i.e., (p_x, p_y). The range query returns all objects in D that are contained in the query range q. Formally:

$$Range(q, D) = \{ \ p|p \in D: \ (q_{xl} \leq p_x) \wedge (q_{yl} \leq p_y) \wedge (q_{xh} \geq p_x) \wedge (q_{yh} \geq p_y) \ \}.$$

Algorithm 2 : Range Query Algorithm

Input : D': a partitioned and indexed input dataset; q: a query range

Output : RQ: a set of all points in D' within q

```
1  RQ ← {}
2  IP ← IndexLookup(D', q)            /* find intersected partitions (IP) */
3  for ip ∈ IP do
       /* if completely inside x-dimension range */
4      if q_xl <= ip_xl and ip_xh <= q_xh then
           /* if also completely inside y-dimension range, copy whole
              partition */
5          if q_yl <= ip_yl and ip_yh <= q_yh then
6              memcpy(RQ, ip.begin(), sizeof(ip))
7          else
8              lb ← EstimateFrom(ip, q_yl)                  /* lower bound */
9              LocalSearch(ip, lb, q_yl)          /* get exact lower bound */
10             ub ← EstimateTo(ip, q_yh)                    /* upper bound */
11             LocalSearch(ip, ub, q_yh)          /* get exact upper bound */
12             memcpy(RQ, ip + lb, ub − lb)
13         end
       /* Scan Phase */
14     else
15         lb ← EstimateFrom(ip, q_yl)                      /* lower bound */
16         LocalSearch(ip, lb, q_yl)
17         ub ← ip.end()
18         for i ∈ [lb, ub] and ip_yh <= q_yh do
19             p ← ip_i                     /* ith point in partition ip */
20             if p within q then
21                 RQ ← RQ ∪ {p}
22             end
23         end
24     end
25 end
26 return RQ
```

For faster query processing we use the partitioned and indexed input dataset D' from algorithm 1. Range query processing is shown in algorithm 2

and works in three phases:

- **Index Lookup**: In index lookup, we intersect a given range query using the grid directories (or trees) to find the partitions the query intersects with. These partitions are represented by IP (stands for intersected partitions and is reflected in line 2 of the algorithm 2). Note, that the index lookup is specific to each partitioning technique.
- **Refinement**: Once the partitions intersected have been determined from the index lookup phase, we use a search technique to find the lower bound of the query on the sorted dimension within the partition. There can be various cases on how a query intersects with the partition, and we only consult the search technique when it is actually needed to find the lower bound of the given query on the sorted dimension. For example, a partition could be fully inside the range query, and in such a case we simply copy all the points in the partition rather than use a search technique. This is one simple case and is reflected in line 6 of the algorithm 2.
- **Scan**: Once the lower bound in the sorted dimension has been determined in refinement, we scan the partition to find the qualifying points on both dimensions. We stop as soon as we reach the upper bound of the query on the sorted dimension, or we reach the end of the partition. Scan phase is reflected in the algorithm 2 from line 14 onwards.

Search Within Partition

As mentioned earlier, we use a search technique on the sorted dimension (y-dimension) to look for the lower bound (and in some cases upper bound) of the query range q. A search technique can either be a learned index or binary search (hereby search technique). In all our experiments, we have sorted on the longitude value of the location.

We use a RadixSpline [86, 85] over the sorted dimension which consists of two components: 1) a set of spline points, and 2) a radix table to quickly determine the spline points to examine for a lookup key (in our case the dimension over which the data is sorted). At lookup time, first the radix table is consulted to determine the range of spline points to examine. In the next step, these spline points are searched over to determine the spline points surrounding the lookup key. In the last step, linear interpolation is used to predict the position of the lookup key in the sorted array. Unlike the RMI [91], the RadixSpline only requires one pass over the data to build the index, while retaining competitive lookup times. The RadixSpline and the RMI, at the time of writing, only work on integer values, and we adapted the open-source implementation of RadixSpline to work with floating-point values (spatial datasets generally contain floating point values). In our implementation, we have set the spline error to 32 in all experiments.

It is important to make a distinction between how we use RadixSpline and binary search for refinement. In case of binary search, we do a lookup

for the lower bound of the query on the sorted dimension. As learned indexes come with an error, usually a local search (signified by $LocalSearch()$ in algorithm 2) is done to find the lookup point (in our case the query lower bound). For range scans, as we do, there can be two cases. The first case is that the estimated value from the spline is lower than the actual lower bound on the sorted dimension. In this case, we scan up until we reach the lower bound on the sorted dimension. In the second case, the estimated value is higher than the actual lower bound, hence, we first scan down to the lower bound, materialize all the points in our way until we reach this bound, and after that we scan up until the query upper bound (or the partition end). In case the estimated value is lower than the upper bound of the query (i.e. the estimated value is within both query bounds), the second case incurs zero cost for local search as we can scan in both directions until we reach the query bounds within the partition.

5.2.4 Distance Query Processing

A distance query takes a query point q_p, a distance d, and a set of geometric objects D. It returns all objects in D that lie within the distance d of query point q_p. Formally:

$$Distance(q_p, d, D) = \{\ p | p \in D\ \wedge\ \text{dist}(q_p, p) \leq d\}.$$

As in the case of Range query processing(Section 5.2.3), we use the partitioned and indexed input dataset D' from algorithm 1 for faster query processing. Distance query has been implemented using the *filter and refine* [127] approach, which is also used in popular database system Oracle Spatial [76].

Algorithm 3 : Distance Query Algorithm

Input : D': partitioned and indexed input dataset; q_p: a query point; d: distance
Output : DQ: a set of all points in D' within distance d of q_p

```
   /* Filter using MBR */
1  MBR ← GetMBR(q, d)                    /* generate mbr using q and d */
2  RQ ← RangeQuery(D, mbr)
3  DQ ← {}
   /* Refine */
4  for p ∈ RQ do
5      if WithinDistance(p, q_p, d) then
6          DQ ← DQ ∪ {p}
7      end
8  end
9  return DQ
```

Algorithm 3 shows the algorithm for distance query processing. We first filter using a rectangle (reflected in line 1 of algorithm 3), whose corner vertices are at a distance of d from the query point q. We issue a range query

using this rectangle, and then refine the resulting candidate set of points by using a *withinDistance* predicate. Please note that we are using GPS coordinates (or Geographic coordinate system). Special care needs to be taken if either of the poles or the 180th meridian are within the query distance d. We compute the coordinates of the minimum bounding rectangle by moving along the geodesic arc as described in [21], and then handle the edge cases of the poles and the 180th meridian. At the time of writing, we only utilize one bounding box, but this is not the optimal way and would lead to materializing a large number of points in case the 180th meridian is in the query circle. One way to avoid such a case is to break the bounding box into two parts, one at either side of the 180th meridian. We leave this case of optimization for future work. Once we have materialized all the points in the MBR of q and d, we refine these candidate set of points. We compute the Haversine distance between the query point q and these candidate set of points. If the distance between them is less than d, then we add such points to the final result.

5.2.5 Join Query Processing

A spatial join takes two input sets of spatial records R and S and a join predicate θ (e.g., overlap, intersect, contains, within, or withindistance) and returns a set of all pairs (r, s) where $r \in R$, $s \in S$, and the join predicate θ is fulfilled. Formally:

$$R \bowtie_{\theta} S = \{ \ (r, s) \mid r \in R, \ s \in S, \ \theta(r, s) \text{ holds } \}.$$

We implemented a join query between a set of polygons, and the partitioned and indexed input location dataset D'. The join algorithm is outlined in algorithm 4. The join query has also been implemented using the *filter and refine* [127] approach, where we use the minimum bounding rectangle of each polygon, and run a range query using this rectangle. We then refine these candidate set of points with the predicate θ as contains. The query thus returns all points contained in each polygon. The contains predicate has been implemented using the ray-casting algorithm, where a ray is casted from the candidate point to a point outside the polygon, and then number of intersections with polygon edges is counted. Some polygons could potentially contains hundreds or thousands of edges. To facilitate quick lookup of the edges intersected with the ray, we index the polygon edges in an interval tree. Interval trees allows us to quickly lookup the edges that the ray would intersects with, and the interval tree is implemented using a binary search tree.

Algorithm 4 : Join Query Algorithm

Input : D': partitioned and indexed input dataset; *POLYGONS*: a set of polygons
Output : *JQ*: a set of sets, a set of points within each polygon in *POLYGONS*

```
1  JQ ← {}
2  for polygon ∈ POLYGONS do
       /* Filter using MBR */
3      MBR ← GetMBR(polygon)              /* get mbr of the polygon */
4      RQ ← RangeQuery(D, mbr)
5      contained ← {}
       /* Refine */
6      for p ∈ RQ do
7          if Contains(polygon, p) then
8              contained ← contained ∪ {p}
9          end
10     end
11     JQ ← JQ ∪ {contained}
12 end
13 return JQ
```

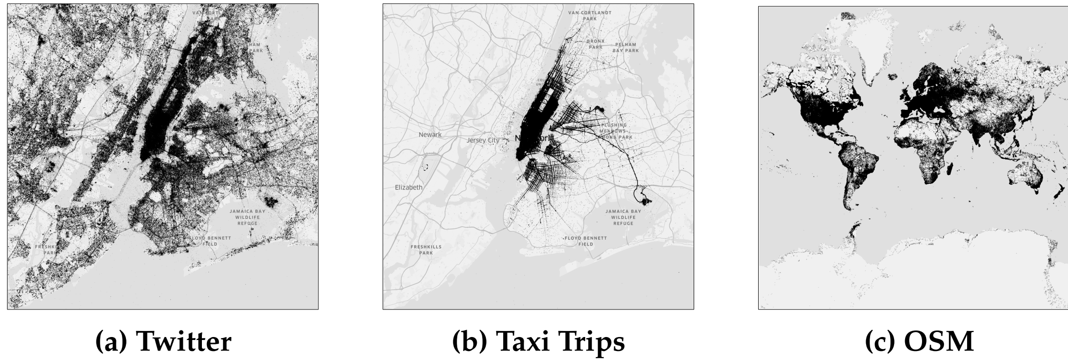
5.3 Evaluation

All experiments were run single threaded on an Ubuntu 18.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 10 cores, 3.00 GHz turbo)[1] and 256 GB DDR3 RAM. We use the *numactl* command to bind the thread and memory to one node to avoid NUMA effects. CPU scaling was also disabled during benchmarking using the *cpupower* command.

5.3.1 Datasets

For evaluation, we used three datasets, the New York City Taxi Rides dataset [122] (NYC Taxi Rides), geo-tagged tweets in the New York City area (NYC Tweets), and Open Streets Maps (OSM). NYC Taxi Rides contains 305 million taxi rides from the years 2014 and 2015. NYC Tweets data was collected using Twitter's Developer API [182] and contains 83 million tweets. The OSM dataset has been taken from [128] and contains 200M records from the All Nodes (Points) dataset. Figure 5.3 shows the spatial distribution of the three datasets. We further generated two types of query workloads for each of the three datasets: skewed queries (which follows the distribution of the underlying data) and uniform queries. For each type of query workload, we generated six different workloads ranging from 0.00001% to 1.0% selectivity. For example, in the case of Taxi Rides dataset (305M records), these queries

[1]CPU: `https://ark.intel.com/content/www/us/en/ark/products/75272/intel-xeon-processor-e5-2660-v2-25m-cache-2-20-ghz.html`



(a) Twitter **(b) Taxi Trips** **(c) OSM**

Figure 5.3: Datasets: (a) Tweets are spread across New York, (b) NYC Taxi trips are clustered in central New York, and (c) All Nodes dataset from OSM

would materialize 30 records to 3 million records. These query workloads consist of one million queries each. To generate skewed queries, we select a record from the data, and expand its boundaries (using a random ratio in both dimensions) until the selectivity requirement of the query is met. For uniform queries, we generated points uniformly in the embedding space of the dataset and expand the boundaries similarly until the selectivity requirement of the query is met. The query selectivity and the type of query are mostly application dependent. For example, consider the application Google Maps, and a user issues a query to find the popular pizzeria near the user. The expected output for this query should be a handful of records, i.e. a low selectivity query (a list of 20-30 restaurants near the user). On the other hand a query on an analytical system, would materialize many more records (e.g. find average cost of all taxi rides originating in Manhattan).

Firstly, we compare the performance of learned indexes and binary search as search techniques within a partition. Furthermore, we compare our implementation of the learned indexes with two best performing indexes from Chapter 4. More specifically, we compare our implementations with STRtree implementation from Java Topology Suite (JTS), and S2PointIndex from Google S2 for the range and the distance queries. For the join query, we make use of the S2ShapeIndex available in Google S2.

5.3.2 Range Query Performance

In this section, we first explore tuning the partition sizes, and why tuning is crucial to obtain optimal performance. Next, we present the total query runtime, when the partition sizes for each index is tuned for optimal performance. Here, we also compare the performance of the learned indexes with the two state-of-the-art spatial indexes mentioned earlier.

Tuning Partitioning Techniques

Recent work in learned multi-dimensional and spatial indexes have focused on learning from the data and the query workload. The essential idea behind

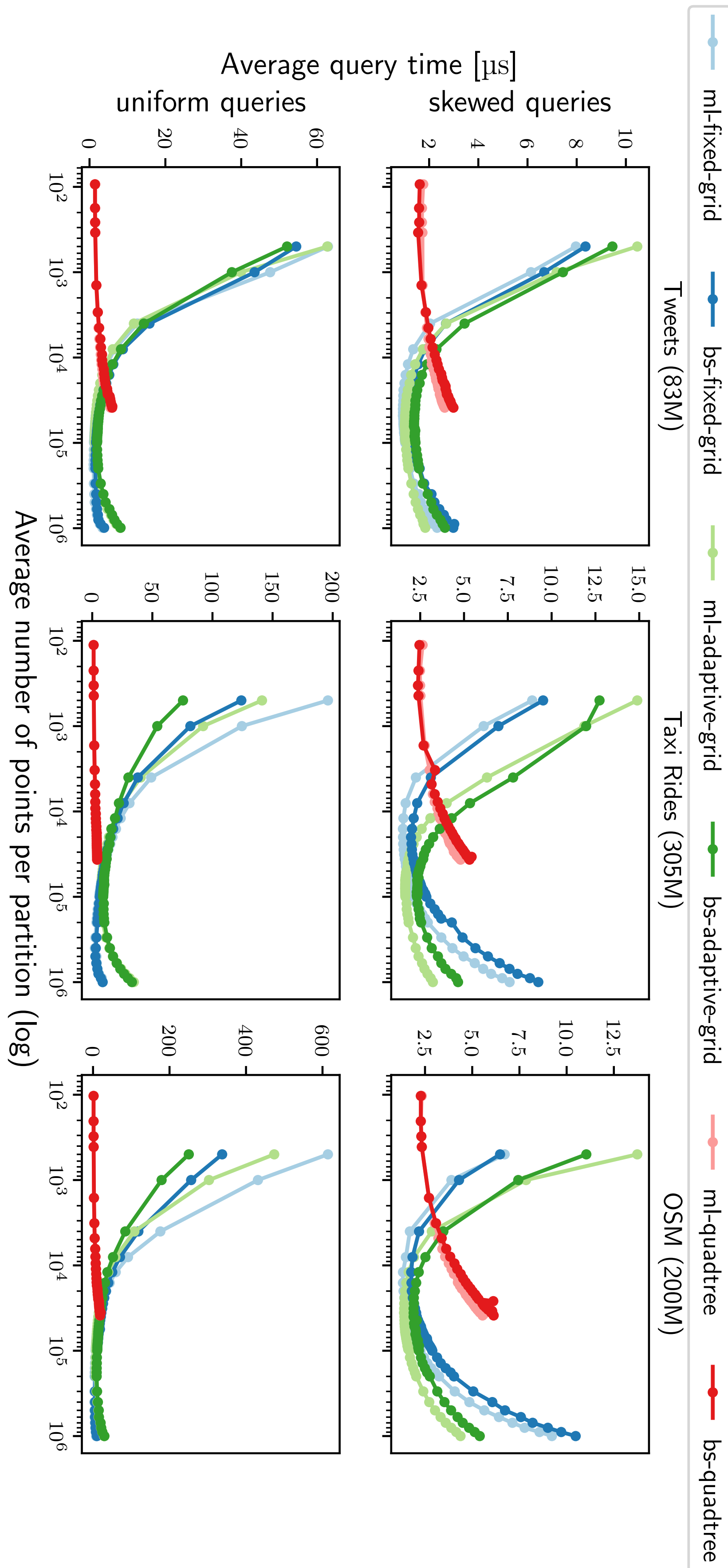


Figure 5.4: Range query configuration - ML vs. BS for low selectivity (0.00001%)

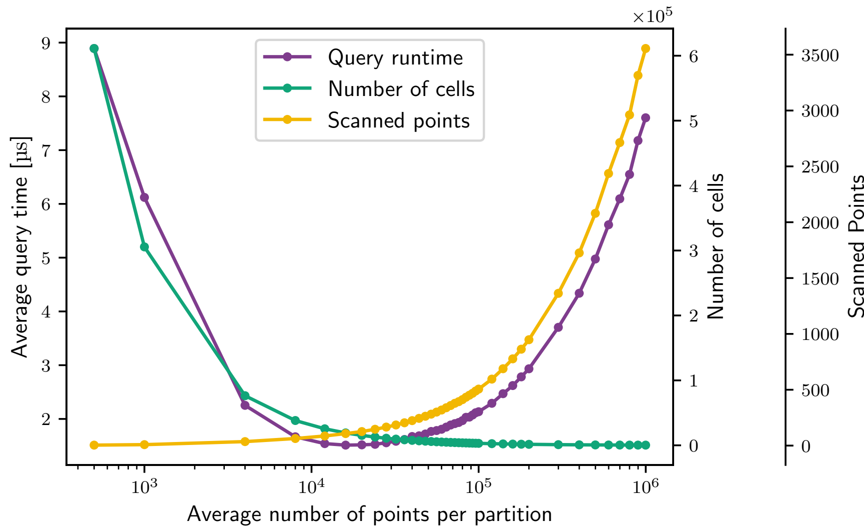


Figure 5.5: Effect of number of cells and number of points scanned for Fixed-grid on Taxi Trip dataset for skewed queries (0.00001% selectivity)

learning from both data and query workload is that a particular usecase can be instance-optimized. To study this effect, we conducted multiple experiments on the three datasets by varying the sizes of the partitions, tuning them on two workloads with different selectivities (to cover a broad spectrum we tune the indexes on queries with low and high selectivity) for both skewed and uniform queries.

Figure 5.4 shows the effect of tuning when the indexes are tuned for the lowest selectivity workload for the two query types. It can be seen in the figure that it is essential to tune the grid partitioning techniques for a particular workload. Firstly, they are susceptible to the size of the partition. As the size of the partition increases, we notice an improvement in the performance until a particular partition size is reached which corresponds to the optimal performance. After this point, increasing the size of the partitions only degrades performance. It can be seen that, usually, for grid (single-dimension) partitioning techniques the partition sizes are much larger compared to partitioning techniques which filter on both dimensions (only Quadtree is shown in the figure but the same holds for the other partitioning techniques we have covered in this work, we do not show the other trees because the curve is similar for them). Due to the large partition sizes in grid partitioning techniques, we notice a large increase in performance while using a learned index compared to binary search. This is especially evident for skewed queries (which follow the underlying data distribution). We encountered a speedup from 11.79% up to 39.51% compared to binary search. Even when we tuned a learned index to a partition size which corresponds to the optimal performance for binary search, we found that in multiple cases learned index frequently outperformed binary search. Learned indexes do not help much for partitioning techniques which filter on both dimensions, instead the performance of Quadtree (and STRtree) dropped in many cases, see Table 5.1. The reason is that the optimal partition sizes for these techniques is very low (less

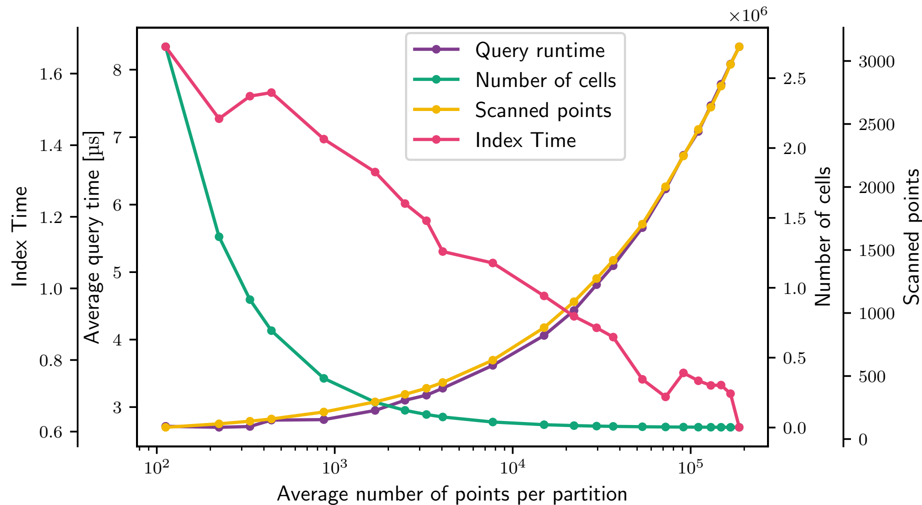


Figure 5.6: Effect of number of cells and number of points scanned for Quadtree on Taxi Trip dataset for skewed queries (0.00001% selectivity)

than 1,000 points per partition for most configurations). The refinement cost for learned indexes is an overhead in such cases. K-d tree on the other hand, contains more points per partition (from 1200 to 7400) for the optimal configuration for Taxi Trips and OSM datasets and thus learned indexes perform faster by 2.43% to 9.17% than binary search. For Twitter dataset, the optimal configuration contains less than 1200 points per partition, and we observed a similar drop in performance using learned indexes.

Figure 5.5 shows the effect of number of cells and number of points that are scanned in each partition on query runtime for Fixed-grid on Taxi Trips dataset for lowest selectivity. As the number of points per partitions increases (i.e. fewer number of partitions), the number of cells decreases. At the same time, the number of points that need to be scanned for the query increases. The point where these curves meet is the optimal configuration for the workload which corresponds to the lowest query runtime. For tree structures, the effect is different. Figure 5.6 shows that the structures that filter on both dimensions do most of the pruning in the index lookup. The dominating cost in these structures is the number of points scanned within the partition and the query runtime is directly proportional to this number. To minimize the number of points scanned, they do most of the pruning during index lookup which require more partitions (i.e. less number of points per partition), but then they pay more for index lookup.

Query Performance

Figure 5.7 shows the query runtime for all learned index structures. It can be seen that Fixed-grid along with Adaptive-grid performs (1D schemes) perform the best for all the cases except uniform queries on Taxi and OSM datasets. For skewed queries, Fixed-grid is 1.23× to 1.83× faster than the closest competitor, Quadtree (2D), across all datasets and selectivity. The

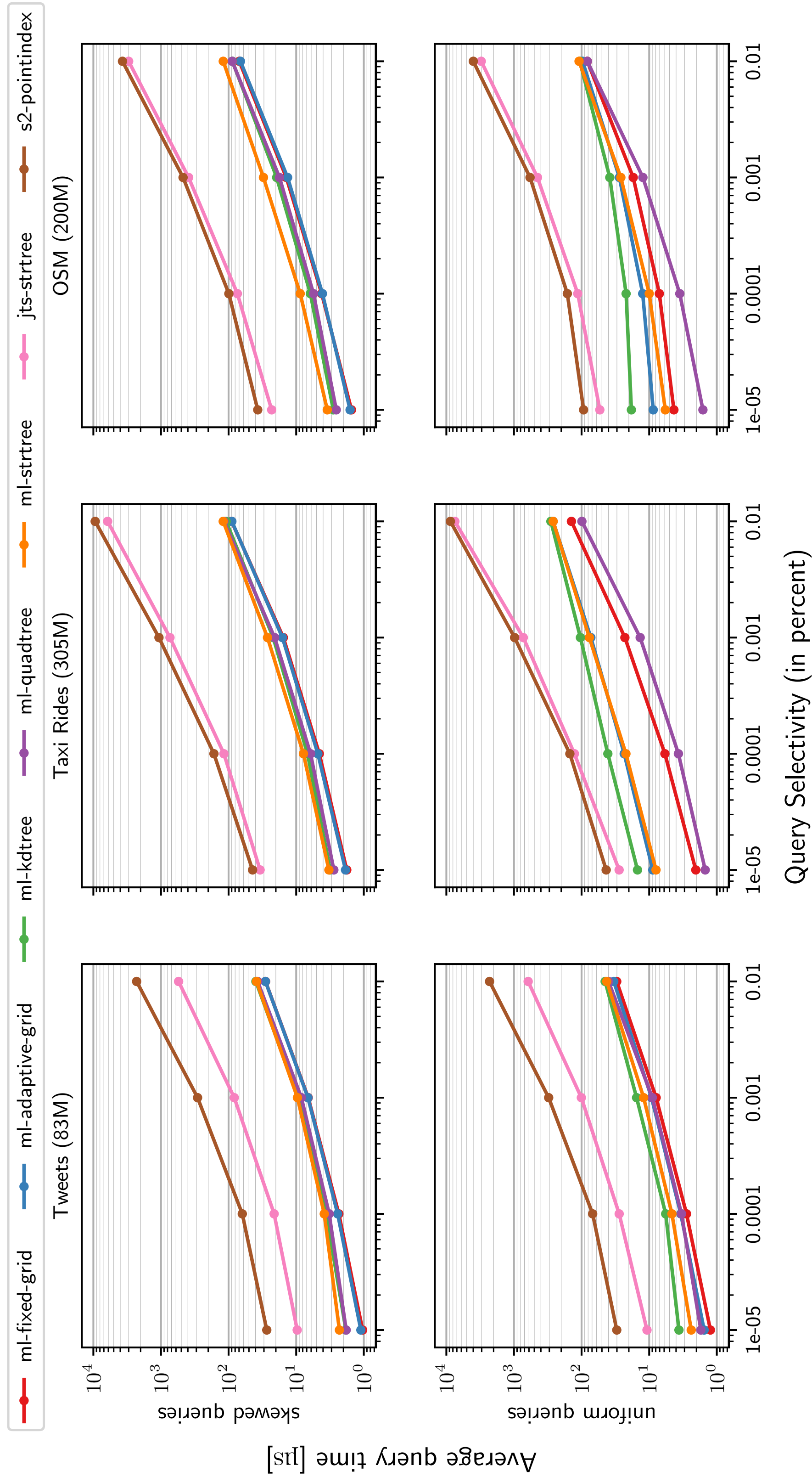


Figure 5.7: Total range query runtime with parameters tuned on selectivity 0.00001%

slight difference in performance between Fixed-grid and Adaptive-grid comes from the index lookup. For Adaptive-grid, we use binary search on the linear scales to find the first partition the query intersects with. For Fixed-grid, the index lookup is almost negligible as only an offset computation is needed to find first intersecting partition. It can also be seen in the figure that the Quadtree is significantly better for uniform queries in case of Taxi Rides dataset (1.37×) and OSM dataset (2.68×) than the closest competitor Fixed-grid. There are two reasons for this, firstly the Quadtree intersects with fewer number of partitions than the other index structures, see Table 5.2. Secondly, for uniform queries, the Quadtree is more likely to traverse the sparse and low-depth region of the index. This is in conformance with an earlier research [80], where the authors report similar findings while comparing the Quadtree to the R*-tree and the Pyramid-Technique.

In Figure 5.7, we can also see the performance of learned indexes compared to JTS STRtree and S2PointIndex. Fixed-grid is 8.67× to upto 43.27× faster than JTS STRtree. Fixed-grid is also 24.34× to upto 53.34× faster than S2PointIndex. Quadtree, on the other hand, is 6.26× to upto 33.99× faster than JTS STRtree, and by 17.53× to upto 41.91× faster than S2PointIndex. Note, that the index structures in the libraries are not tuned, and are taken as is out of the box with default values. The poor performance of S2PointIndex is because it is not optimized for range queries. S2PointIndex, is a B-tree on the S2CellId's (64-bit integers), and the cell id's are a result of the Hilbert curve enumeration of a Quadtree like space decomposition. Hilbert curve suffers from skewed cases where the range query rectangle covers the whole curve. To avoid such a case S2PointIndex, decomposes the query rectangle into four parts so as to avoid hitting every portion of the curve. It still ends up scanning many superfluous points.

5.3.3 Distance Query Performance

Similar to range query processing, we first carry out experiments to determine the best partition size for each index. The results are similar to range query performance. Next we compare the performance of the learned indexes with JTS STRtree and S2PointIndex. As mentioned earlier in Section 5.2.4, we implemented the distance query using the filter and refine [127] approach, which is a norm in spatial databases such as Oracle Spatial [76], and PostGIS [136]. Note that we use GPS coordinates to index the points and use the Harvesine distance in the refinement phase.

Tuning Partitioning Techniques

Figure 5.8 shows the effect of tuning when the indexes are tuned for the lowest selectivity workload for the two query types. Similar to range query, the 1D partitioning techniques again are highly susceptible to the size of partition, while the 2D schemes do not seem to be affected unless the partition size becomes large. For 1D partitioning techniques, as the size of the partition increases, we observe improvement in performance until a particular partition

Table 5.1: Total range query runtime (in microseconds) for both RadixSpline (ML) and binary search (BS) for Taxi Rides dataset on skewed and uniform query workloads (parameters are tuned for selectivity 0.00001%)

	Taxi Trips (Skewed Queries)						Taxi Trips (Uniform Queries)					
	Fixed		Adaptive		Quadtree		Fixed		Adaptive		Quadtree	
Selectivity (%)	ML	BS	ML	BS	ML	BS	ML	BS	ML	BS	ML	BS
0.00001	1.78	2.35	1.86	2.40	2.77	2.51	2.02	2.58	81.4	10.54	1.48	1.31
0.0001	4.54	5.82	4.67	6.12	6.12	5.82	5.85	6.91	228.1	27.69	3.69	3.42
0.001	14.97	18.83	15.32	19.49	20.84	19.47	22.87	24.34	708.8	87.49	13.59	12.98
0.01	90.13	97.04	89.48	95.96	117.01	104.37	141.24	151.47	2634.4	309.62	98.85	112.77
0.1	678.12	698.39	675.14	696.49	922.67	793.96	988.35	922.96	9609.9	1174.79	891.24	1101.95
1.0	8333.94	8408.15	8301.56	8399.69	10678.04	9512.29	8843.71	8753.68	8574.84	8836.28	10647.97	12377.14

Table 5.2: Average number of partitions intersected for each partitioning scheme for selectivity 0.00001% on Taxi Rides and OSM datasets

	Taxi Rides		OSM	
Partitioning	Skewed	Uniform	Skewed	Uniform
Fixed	1.97	7.98	1.72	23.73
Adaptive	1.74	31.57	1.51	24.80
k-d tree	1.70	21.62	1.56	30.95
Quadtree	1.79	2.12	1.37	7.96
STR	2.60	47.03	1.90	11.05

size is reached which coincides with the optimal performance, after which the performance starts degrading. We again observe that for optimal performance the partition size of the 1D techniques is much larger than the corresponding 2D techniques. These results are a direct consequence of the filter and refine approach, since we utilize the range query to answer the distance query. Another thing to notice is that the difference between learned indexes and binary search reduces for distance queries. This is because after the filter phase, we refine using the Haversine distance, which is computationally expensive. Haversine distance requires multiple additions, multiplications, and division as well as three trigonometric function calls. Although, we only use Harvesine distance on a subset of points from the filter phase, it still is expensive to compute. Morever, there are cases where a query contains the 180th meridian. As mentioned earlier in Section 5.2.4, a way to mitigate this problem is to divide the range into two part, one on each side of the 180th meridian. We leave these improvements for future work.

Query Performance

Figure 5.9 shows the distance query runtime for all partitioning techniques as well as the two spatial indexes, S2PointIndex, and JTS STRtree. In the figure we can make two important observations. First is that the difference in performance between the learned indexes diminishes quickly as we increase the selectivity of the query. Grid based index perform the best for lower selectivities (0.00001% and 0.0001%), except for uniform queries for Taxi Rides and OSM datasets where Quadtree is better (similar to range query). But after that, the dominant cost is Haversine distance, as more points qualify the filter phase, and thus all partitioning schemes mostly converge in their respective performances. Second observation is that S2PointIndex outperforms most of the indexes for uniform queries on the OSM dataset. The reason for this is that after the filter phase, many points need refinement for uniform queries for the OSM dataset. For example for the OSM dataset,

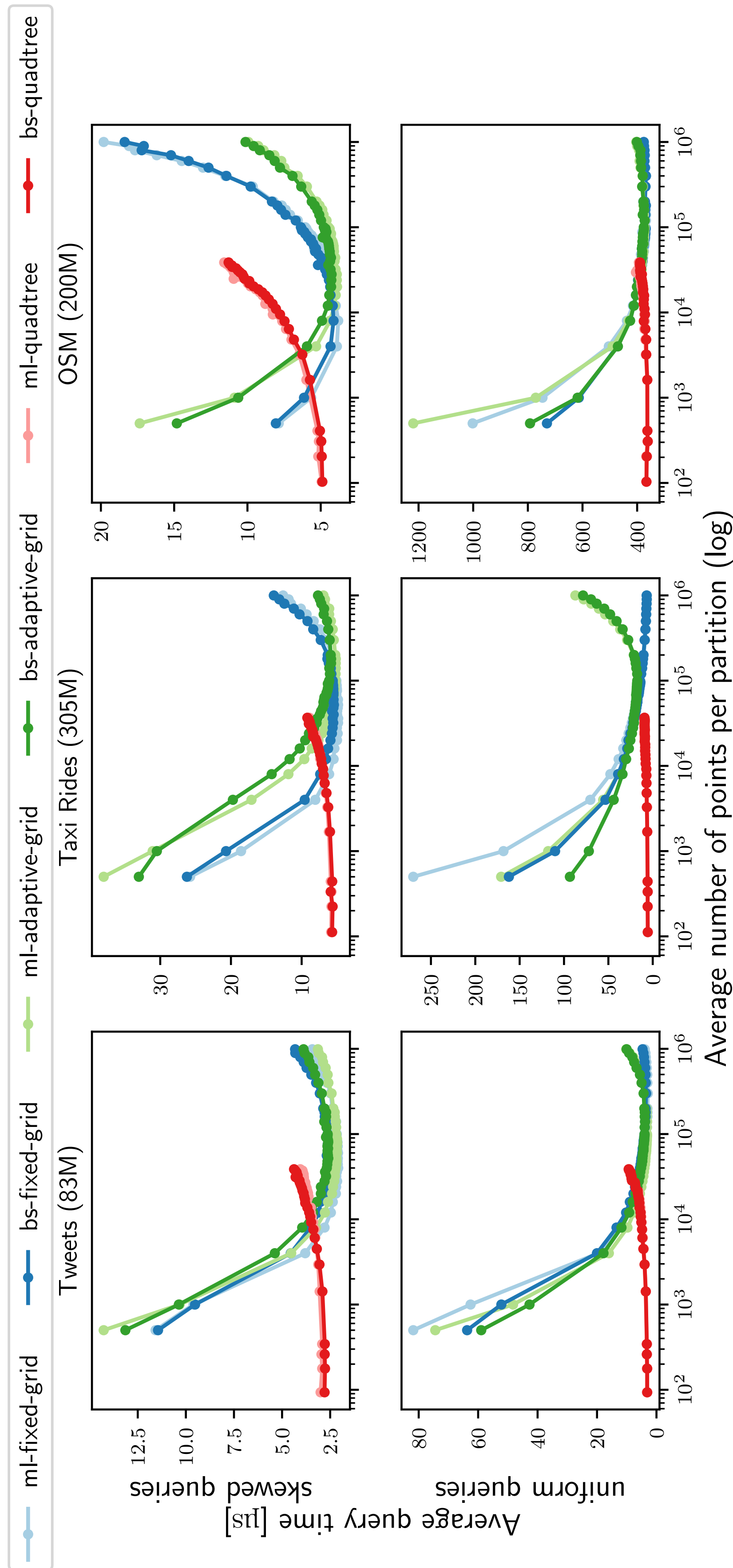


Figure 5.8: Distance query configuration - ML vs. BS for low selectivity (0.00001%)

the average number of points that need refinement after the filter phase for skewed queries are 25, 257, and 2561 for the selectivities 0.00001%, 0.0001% and 0.001%. For uniform queries, the average number of points that need refinement after the filter phase are 4257, 7263, 17612 for the OSM dataset. The dominant cost again is the Haversine distance computation, and thus we also do not observe much difference in performance between the learned indexes. S2PointIndex has some optimization for distance queries, where it carefully increases the radius of the a data structure called S2Cap (a circular disc with a center and a radius). Distance query with S2PointIndex for uniform queries on OSM dataset is $1.91\times$ to upto $7.75\times$ faster than the learned indexes. The comparison is a bit fairer with JTS STRtree since the learned indexes and JTS STRtree both deploy the *filter and refine* approach to answer distance queries. Fixed-grid is $1.33\times$ to upto $11.92\times$ faster that JTS STRtree.

5.3.4 Join Query Performance

For join queries, we utilized the *filter and refine* approach for the learned indexes and JTS STRtree. We use the bounding box of the polygon objects, and issue a range query on the indexed points, while in S2, we utilize the S2ShapeIndex which is especially built to test for containment of points in polygonal objects. As mentioned in Section 5.2.5, we index the polygon objects using an interval tree in case of the learned indexes. For JTS STRtree, we utilize the PreparedGeometry[2] abstraction, to index line segments of all individual polygons, which helps in accelerating the refinement check.

We utilized, three polygonal datasets for the location datasets that are in the NYC area (i.e., Tweets and Taxi Rides datasets). We used the Boroughs, Neighborhood, and the Census block boundaries (polygons) for the join query. Boroughs consists of five polygons, Neighborhoods dataset consists of 290 polygons, and the Census blocks dataset consists of approximately 40 thousand polygons. For the OSM dataset, we join it with the Countries dataset which consists of 255 country boundaries. Similar to range and distance queries, we first find the optimal partition size for each of the learned index for each dataset.

Figure 5.10 shows the join query performance. It can be observed in the figure that most of the learned indexes are similar in join query performance. The reason behind this is that *Filter* phase is not expensive for the join query, and the *Refinement* phase is the dominant cost. This reasoning is similar to what we have already discussed in Section 4.7.1. Although, we use an interval tree to index the edges of the polygons for quickly determining which edges to intersect the ray casted from the candidate point, this phase is still expensive. For future work, we plan to investigate the performance using the main-memory index for polygon objects proposed in [82].

It can also be seen in the figure, that the learned indexes are considerably faster than JTS STRtree, and S2ShapeIndex for the join query. Fixed-grid, for example, is $1.81\times$ to $2.69\times$ faster than S2ShapeIndex, and $2.7\times$ to

[2] `https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/prep/PreparedGeometry.html`

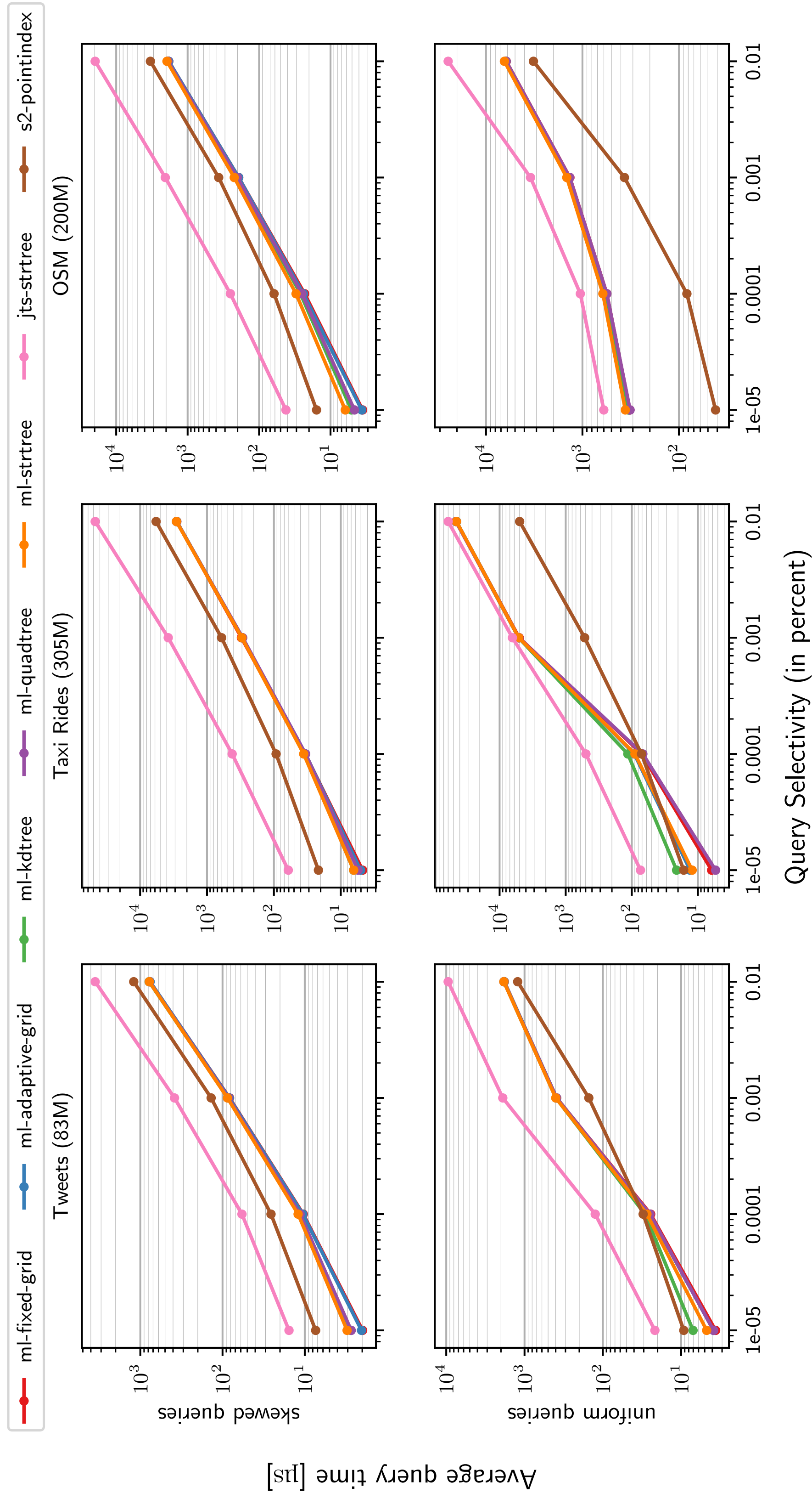


Figure 5.9: Total distance query runtime with parameters tuned on selectivity 0.00001%

3.44× faster than JTS STRtree for the Tweets dataset across all three polygonal datasets. Similarly, for Taxi Rides dataset Fixed-grid is 2.39× to 4.96× faster than S2ShapeIndex, and 3.017× to 4.49× faster than JTS STRtree. Finally, for the OSM dataset, it is 2.89× faster than S2ShapeIndex, and 7.311× faster than JTS STRtree for the join query.

5.3.5 Indexing Costs

Figure 5.11 shows that Fixed-grid and Adaptive-grid are faster to build than the tree based learned indexes. Fixed-grid is 2.11×, 2.05×, and 1.90× faster to build than closest competitor STRtree. Quadtree is the slowest to build because it generates a large number of cells for optimal configuration. Not all partitions in Quadtree contain an equal number of points as it divides space rather than data, thus leading to an imbalanced number of points per partition. Fixed-grid and Adaptive grid do not generate large number of partitions, as the partitions are quite large for optimal configuration. They are lower in size for similar reasons. The index size in Figure 5.11 also includes the size of data being indexed.

In the figure, we can also see that the learned indexes are faster to built and consume less memory that S2PointIndex and JTS STRtree. Fixed-grid, for example, is 2.34× to upto 15.36× faster to build than S2PointIndex, and 11.09× to upto 19.74× faster to build than JTS STRtree. It also consumes less memory than S2PointIndex (3.04× to upto 3.4×), and JTS STRtree (4.96× to upto 8.024×). The comparison on index size with JTS STRtree is not completely fair. As mentioned earlier in Section 4.6.1, JTS STRtree is a SAM (spatial access method), where it stores four coordinates for each point (since the points have been stored as degenerate rectangles). The learned indexes implemented in this work are PAMs (point access method), where we only store two coordinates for each data point.

5.4 Related Work

Recent work by Kraska et al. [91] proposed the idea of replacing traditional database indexes with learned models that predict the location of a key in a dataset. Their learned index, called the Recursive Model Index (RMI), only handles one-dimensional keys. Since then, there has been a corpus of work on extending the ideas of the learned index to spatial and multi-dimensional data.

Flood [117] is an in-memory read-optimized multi-dimensional index that organizes the physical layout of d-dimensional data by dividing each dimension into some number of partitions, which forms a grid over d-dimensional space. Points that fall into the same grid cell are stored together. Flood adapts to the data and workload in two ways: first, it automatically learns the best number of partitions in each dimension by using a cost model. Second, it spaces the partitions in each dimension so that an equal number of points fall in each partition.

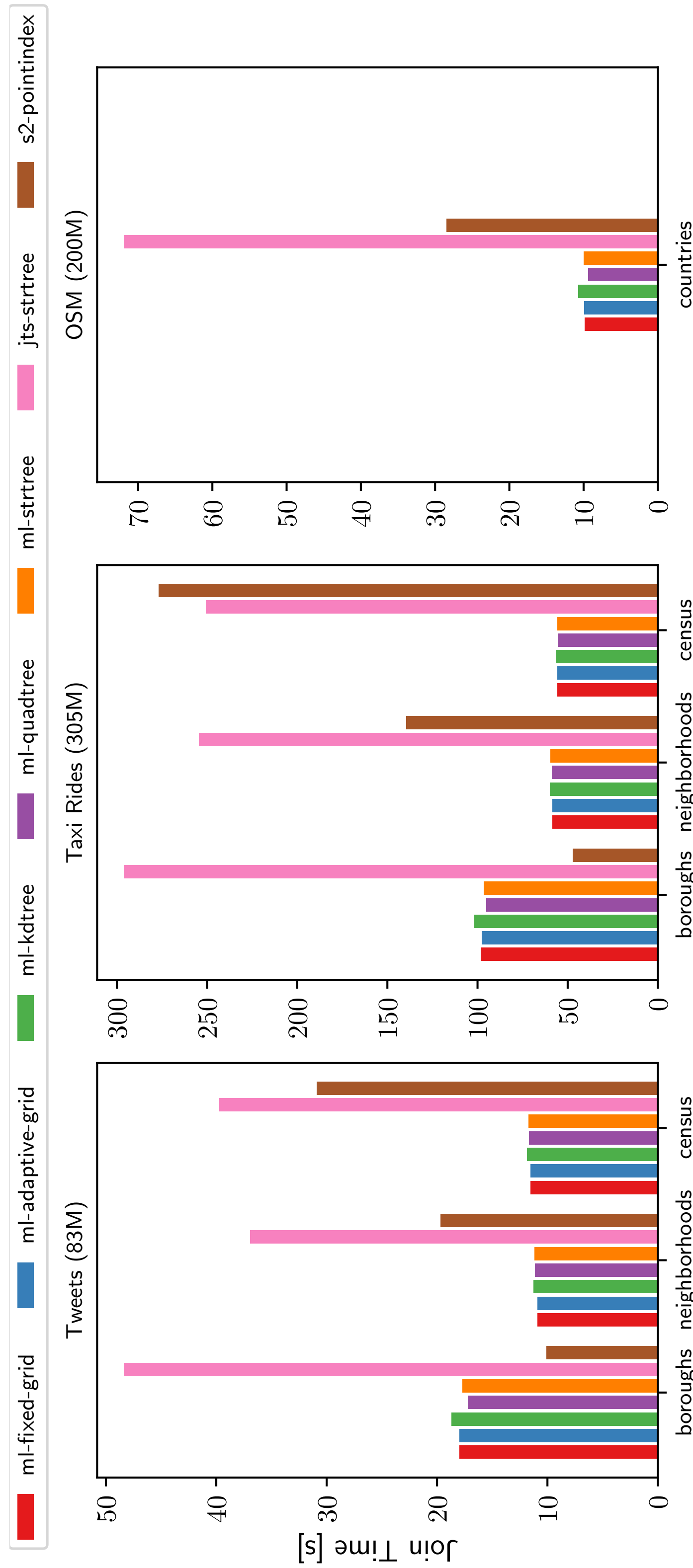


Figure 5.10: Join query performance for the three datasets

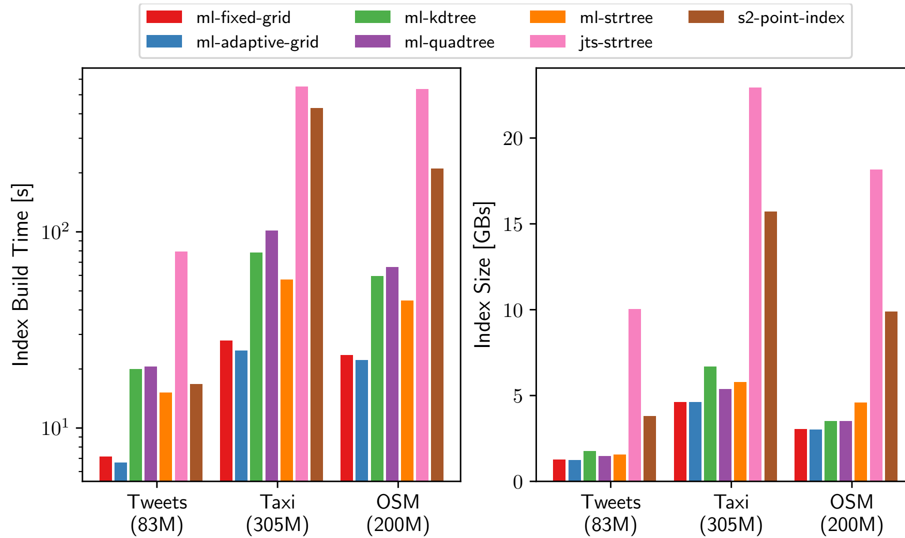


Figure 5.11: Index build times and sizes for the three datasets

Learning has also been applied to the challenge of reducing I/O cost for disk-based multi-dimensional indexes. Qd-tree [194] uses reinforcement learning to construct a partitioning strategy that minimizes the number of disk-based blocks accessed by a query. LISA [103] is a disk-based learned spatial index that achieves low storage consumption and I/O cost while supporting range queries, nearest neighbor queries, and insertions and deletions.

Past work has also aimed to improve traditional indexing techniques by learning the data distribution. The ZM-index [188] combines the standard Z-order space-filling curve with the RMI from [91] by mapping multi-dimensional values into a single-dimensional space, which can then be learned using models. The ML-index [32] combines the ideas of iDistance [72] and the RMI to support range and KNN queries. [60] augment existing indexes with light-weight models to accelerate range and point queries.

All of these works share the idea that a multi-dimensional index can be instance-optimized for a particular use case by learning from the dataset and query workload. In our work, we apply these same ideas of learning from the data and workload to improve traditional multi-dimensional indexes for spatial data.

5.5 Conclusions and Future Work

In this work, we implemented techniques proposed in a state-of-the-art multi-dimensional index, namely, Flood [117], which indexes points using a variant of the Grid-file and applied them to five classical spatial indexes. We have shown that replacing binary search with learned indexes within each partition can improve overall query runtime by 11.79% to 39.51%. As expected, the effect of using a more efficient search within a partition is more pronounced for queries with low selectivity. With increasing selectivity, the effect of a fast search diminishes. Likewise, the effect of using a learned index

is larger for (1D) grid partitioning techniques (e.g., Fixed-grid) than for (2D) tree structures (e.g., Quadtree). The reason is that the partitions (cells) are less representative of the points they contain in the 1D case than in the 2D case. Hence, 1D partitioning requires more refinement within each cell.

In contrary, finding the qualifying partitions is more efficient with 1D than with 2D partitioning, thus contributing to lower overall query runtime (1.23x to 1.83x times faster). Currently, we are using textbook implementations for Quadtree and K-d tree. Future work could study replacing these tree structures with learned counterparts. For example, we could linearize Quadtree cells (e.g., using a Hilbert or Z-order curve) and store the resulting cell identifiers in a learned index.

So far we have only studied the case where indexes and data fit into RAM. For on-disk use cases, performance will likely be dominated by I/O and the search within partitions will be of less importance. We expect partition sizes to be performance-optimal when aligned with the physical page size. To reduce I/O, it will be crucial for partitions to not contain any unnecessary points. Hence, we expect 2D partitioning to be the method of choice in this case. We refer to LISA [103] for further discussions on this topic.

We also compared the performance of the learned indexes with two state-of-the-art indexes, namely, S2PointIndex, and JTS STRtree (also evaluated in Chapter 4). These indexes are used in many applications as well as systems. We observed that learned indexes are faster than the aforementioned indexes in most cases. We also discussed, throughout this chapter, many optimizations that can still be applied to the learned indexes, and we plan to address them in future work.

Chapter 6

Future Work

In this thesis, we have made multiple contributions to the research area of spatial analytical database systems. In Chapter 2, we implement spatial datatypes, and spatial processing in a main-memory database system, HyPer. We show that, even by building an index on-the-fly, HyPerSpace achieves a much lower latency than other related systems. While we did study the performance of HyPerSpace by computing the *SuperCovering* on-the-fly, it would be interesting to see the improvement in latency when the *SuperCovering* can be persisted. We expect a 4× performance improvement for subsequent calls to spatial functions, once the *SuperCovering* or *CellIds* are persisted as an index.

In Chapter 3, we studied various modern big spatial analytics systems. Out of all the systems studied, GeoSpark comes close to a complete spatial analytics systems because of data types and queries supported and the control user has while writing applications. It also exhibits the best performance in most cases. While, the systems are promising as they can efficiently scale out with the increase in data, we believe a performance improvement is needed for them to be truly *real-time*. Luckily, one of the systems, namely GeoSpark, is actively under development and is now incubating as an Apache Project under the name Apache Sedona. We believe that with hundreds of software engineers contributing to the system, this will soon be addressed and Sedona will be the state-of-art system to measure against in the upcoming years.

In Chapter 4, we carried out an experimental evaluation of the modern spatial libraries. These libraries are used by hundreds of systems, applications, and in various research areas today. This work will facilitate researchers, and practitioners alike in choosing a library that best suits their needs. We believe that improvements in the libraries are needed, especially in the implementation of the index structures, which are widely used for spatial partitioning, and indexing spatial data. GEOS is one very popular library which is used in hundreds of libraries, including PostGIS and Shapely (a popular python geospatial library, used by more than 12 thousand projects in GitHub). We showed that the STRtree implementation in GEOS, at every node, retrieves the bounding boxes of every child node from the memory to check if it intersects with the query range, and thus suffers from a large number of cache misses. We showed that one way to mitigate this problem is by storing the bounds of the children node in contiguous memory, and only retrieve (and visit) the children which actually intersect with the query. We

believe the performance of GEOS STRtree will then be at par with JTS counterpart. We plan to address this problem, by contributing to the open-source project.

In Chapter 5, we proposed an approach to apply learned indexes to five classical spatial indexes in order to improve spatial query processing on location-data. We showed that learned index outperform binary search for searching within a spatial partition and that spatial index structures require tuning for various datasets and query workloads for optimal performance. While we manually tuned every index structure for optimal partition size based on the query workload, one potential improvement can be using machine learning to learn the optimal partition size based on the dataset and the query workload. [117] use a synopsis/sketch from the dataset and query workload, where they essentially sample from the dataset and query workload, and decide the optimal data layout based on the synopsis. This technique can directly be applied to learn the optimal layout for the learned spatial indexes as well. We also show that the effect of using a learned index is larger for (1D) grid partitioning techniques (e.g., Fixed-grid) than for (2D) tree structures (e.g., Quadtree). The reason is that the partitions (cells) are less representative of the points they contain in the 1D case than in the 2D case. Hence, 1D partitioning requires more refinement within each cell. In contrary, finding the qualifying partitions is more efficient with 1D than with 2D partitioning, thus contributing to lower overall query runtime (1.23x to 1.83x times faster). At the moment, we implement a textbook implementation of 2D partitioning techniques, and in future work we plan to linearize the tree structure for the 2D schemes (since they generate hundred of thousands of partitions for optimal configuration), apply learned indexes on them, and observe the effect. Also, in Chapter 5 we have studied the learned indexes based on location-data (spatial data with zero extent). Another potential future work is incorporating spatial data with non-zero extents, using a technique known as query expansion [214, 167]. With learned spatial indexes, we studied the case where all data fits into memory. Another potential research area is to utilize byte-addressable non-volatile memory (NVM). NVM has been leveraged in database systems in previous works [143, 145, 144]. It would be interesting to see learned indexes for spatial data and spatial data in general, can be adapted to NVM. We also compared the performance of the learned indexes with two state-of-the-art indexes, namely, S2PointIndex, and JTS STRtree (both evaluated in Chapter 4). These indexes are used in many applications as well as systems. We observed that learned indexes are faster than the aforementioned indexes in most cases. The learned indexes can act as drop in replacement for spatial indexes used in big spatial analytics systems for location-data to improve spatial query processing. We also discussed, throughout the Chapter 5, multiple micro-optimizations that can still be applied to the learned indexes, and we plan to address them in future work.

Acknowledgments

This work has been partially supported by the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

Bibliography

In compliance with § 6 Abs. 6 Satz 3 Promotionsordnung der Technischen Universität München, publications by the author of this thesis are marked with an asterisk (*).

[1] Jemal H. Abawajy. “Comprehensive analysis of big data variety landscape”. In: *Int. J. Parallel Emergent Distributed Syst.* 30.1 (2015), pp. 5–14.

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. “Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce”. In: *PVLDB* 6.11 (2013), pp. 1009–1020.

[3] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. “Automatic Database Management System Tuning Through Large-scale Machine Learning”. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, pp. 1009–1024.

[4] *Amazon Elastic Block Store: Easy to use, high performance block storage at any scale*. `https://aws.amazon.com/ebs/`.

[5] *Amazon Elastic File System (Amazon EFS)*. `https://aws.amazon.com/efs`.

[6] *Amazon Relational Database Service (RDS): Set up, operate, and scale a relational database in the cloud with just a few clicks*. `https://aws.amazon.com/rds/`.

[7] *Amazon S3: Object storage built to store and retrieve any amount of data from anywhere*. `https://aws.amazon.com/s3/`.

[8] Koichiro Amemiya and Akihiro Nakao. “Layer-Integrated Edge Distributed Data Store for Real-time and Stateful Services”. In: *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, pp. 1–9.

[9] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB - The Definitive Guide: Time to Relax*. O’Reilly, 2010. ISBN: 978-0-596-15589-6.

[10] *Apache Hadoop*. `https://hadoop.apache.org`.

[11] *Azure Disk Storage: High-performance, highly durable block storage for Azure Virtual Machines*. `https://azure.microsoft.com/en-us/services/storage/disks/`.

[12] *Azure SQL Database: Build apps that scale with the pace of your business with managed and intelligent SQL in the cloud*. https://azure.microsoft.com/en-us/services/sql-database/.

[13] Jason Barkes, Marcelo R Barrios, Francis Cougard, Paul G Crumley, Didac Marin, Hari Reddy, and Theeraphong Thitayanun. "GPFS: a parallel file system". In: *IBM International Technical Support Organization* (1998).

[14] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (1975), pp. 509–517.

[15] Jon Louis Bentley and Jerome H. Friedman. "Data Structures for Range Searching". In: *ACM Comput. Surv.* 11.4 (1979), pp. 397–409.

[16] *Blob storage: Massively scalable and secure object storage for cloud-native workloads, archives, data lakes, high-performance computing, and machine learning*. https://azure.microsoft.com/en-us/services/storage/blobs/.

[17] Nemanja Boric, Hinnerk Gildhoff, Menelaos Karavelas, Ippokratis Pandis, and Ioanna Tsalouchidou. "Unified Spatial Analytics from Heterogeneous Sources with Amazon Redshift". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 2781–2784.

[18] Panagiotis Bouros and Nikos Mamoulis. "Spatial joins: what's next?" In: *SIGSPATIAL Special* 11.1 (2019), pp. 13–21.

[19] Mohamed Ben Brahim, Wassim Drira, Fethi Filali, and Noureddine Hamdi. "Spatial data extension for Cassandra NoSQL database". In: *Journal of Big Data* 3.1 (2016), pp. 1–16.

[20] Eric A. Brewer. "Towards robust distributed systems". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. ACM, p. 7.

[21] Ilja N Bronshtein and Konstantin A Semendyayev. *Handbook of mathematics*. Springer Science & Business Media, 2013.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[23] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26.

[25] Jinchuan Chen, Yueguo Chen, Xiaoyong Du, Cuiping Li, Jiaheng Lu, Suyun Zhao, and Xuan Zhou. "Big data challenge: a data management perspective". In: *Frontiers Comput. Sci.* 7.2 (2013), pp. 157–164.

[26] Zhida Chen, Gao Cong, and Walid G. Aref. “STAR: A Distributed Stream Warehouse System for Spatial Data”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 2761–2764.

[27] Christopher Cherry. *The Value of Location Intelligence in the Communications Industry*. `https://www.pb.com/docs/US/pdf/Products-Services/Software/Articles/Identify-Market-Expansion-Opportunities/Value-of-Location-Intelligence-in-the-Telecommunications-Industry-WhitePaper.pdf`. 2012.

[28] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010. ISBN: 978-1-449-38156-1.

[29] *CockroachDB: The most highly evolved database on the planet.* `https://www.cockroachlabs.com/product/`.

[30] James C. Corbett et al. “Spanner: Google's Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (2013), 8:1–8:22.

[31] Benoît Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, pp. 215–226.

[32] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. “The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries”. In: *2020 Conference on Extending Database Technology (EDBT)*.

[33] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, pp. 137–150.

[34] GlusterFS Developers. *Gluster File System 3.3. 0 Administration Guide*. 2006.

[35] Jialin Ding et al. “ALEX: An Updatable Adaptive Learned Index”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 969–984.

[36] Harish Doraiswamy and Juliana Freire. “A GPU-friendly Geometric Data Model and Algebra for Spatial Queries”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 1875–1885.

[37] Harish Doraiswamy and Juliana Freire. “A GPU-friendly Geometric Data Model and Algebra for Spatial Queries: Extended Version”. In: *CoRR* abs/2004.03630 (2020). `https://arxiv.org/abs/2004.03630`.

[38] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. "Selectivity Estimation for Range Predicates using Lightweight Models". In: *Proc. VLDB Endow.* 12.9 (2019), pp. 1044–1057.

[39] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. "Spatial partitioning techniques in SpatialHadoop". In: *PVLDB* 8.12 (2015), pp. 1602–1605.

[40] Ahmed Eldawy and Mohamed F. Mokbel. "SpatialHadoop: A MapReduce framework for spatial data". In: *ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, pp. 1352–1363.

[41] Ahmed Eldawy and Mohamed F Mokbel. "The era of big spatial data". In: *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*. IEEE, pp. 42–49.

[42] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. "Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data". In: *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings*, pp. 65–83.

[43] *Engineering with a Global Dataset*. `https://labs.strava.com/`.

[44] *EPSG:32118: New York Long Island*. `https://spatialreference.org/ref/epsg/32118/`.

[45] Nivan Ferreira Ferreira, Jorge Poco, Huy T. Vo, Juliana Freire, and Clüdio T Silva. "Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips". In: *IEEE Transactions on Visualization and Computer Graphics*, pp. 2149–2158.

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

[47] Reuben Fischer-Baum and Carl Bialik. *Uber Is Taking Millions Of Manhattan Rides Away From Taxis*. `http://fivethirtyeight.com/features/uber-is-taking-millions-of-manhattan-rides-away-from-taxis/`.

[48] *Foursquare*. `https://foursquare.com/about`.

[49] Volker Gaede and Oliver Günther. "Multidimensional Access Methods". In: *ACM Comput. Surv.* 30.2 (1998), pp. 170–231.

[50] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "FITing-Tree: A Data-aware Index Structure". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, pp. 1189–1206.

[51] Enrico Gallinucci and Matteo Golfarelli. “SparkTune: tuning Spark SQL through query cost modeling”. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, pp. 546–549.

[52] Amir Gandomi and Murtaza Haider. “Beyond the hype: Big data concepts, methods, and analytics”. In: *Int. J. Inf. Manag.* 35.2 (2015), pp. 137–144.

[53] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. “Improving Distance-Join Query processing with Voronoi-Diagram based partitioning in SpatialHadoop”. In: *Future Gener. Comput. Syst.* 111 (2020), pp. 723–740.

[54] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. “Efficient distance join query processing in distributed spatial data management systems”. In: *Inf. Sci.* 512 (2020), pp. 985–1008.

[55] Karl Friedrich Gauss and Peter Pesic. *General investigations of curved surfaces*. Courier Corporation, 2005.

[56] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. ACM, pp. 29–43.

[57] *Google Zeitgeist: 1.2 trillion searches*. `https://archive.google.com/zeitgeist/2012/#the-world`. 2012.

[58] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, pp. 1917–1923.

[59] Antonin Guttman. “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *SIGMOD’84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. ACM Press, pp. 47–57.

[60] Ali Hadian, Ankit Kumar, and Thomas Heinis. “Hands-off Model Integration in Spatial Index Structures”. In: *CoRR* abs/2006.16411 (2020). `https://arxiv.org/abs/2006.16411`.

[61] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. “The STARK Framework for Spatio-Temporal Data Analytics on Spark”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme” (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pp. 123–142.

[62] Michael Hausenblas and Jacques Nadeau. “Apache Drill: Interactive Ad-Hoc Analysis at Scale”. In: *Big Data* 1.2 (2013), pp. 100–104.

[63] Alex Heath. *Snap confirms that it paid $213 million to buy Zenly and $135 million for Placed*. https://www.businessinsider.com/snapchat-paid-213-million-for-zenly-and-135-million-for-placed-2017-8/. 2017.

[64] Jeffrey Heer and Sean Kandel. "Interactive analysis of big data". In: *XRDS* 19.1 (2012), pp. 50–54.

[65] Mark Hung. "Leading the iot, gartner insights on how to lead in a connected world". In: *Gartner Research* (2017), pp. 1–29.

[66] *IBM Cloud Object Storage: Flexible, cost-effective and scalable cloud storage for unstructured data*. https://www.ibm.com/cloud/object-storage.

[67] Stratos Idreos and Tim Kraska. "From Auto-tuning One Size Fits All to Self-designed and Learned Data-intensive Systems". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, pp. 2054–2059.

[68] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. "Overview of Data Exploration Techniques". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, pp. 277–281.

[69] Stratos Idreos et al. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[70] *Introduction to BigQuery GIS*. https://cloud.google.com/bigquery/docs/gis-intro.

[71] Hojjat Jafarpour. "Quantcast File System (QFS)". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org.

[72] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. "IDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search". In: *ACM Trans. Database Syst.* 30.2 (2005), 364–397. ISSN: 0362-5915.

[73] Abdul Jhummarwala, Mazin Alkathiri, Miren Karamta, and M. B. Potdar. "Comparative Evaluation of Various Indexing Techniques of Geospatial Vector Data for Processing in Distributed Computing Environment". In: *Proceedings of the 9th Annual ACM India Conference, Gandhinagar, India, October 21-23, 2016*, pp. 167–172.

[74] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. "Selecting Subexpressions to Materialize at Datacenter Scale". In: *Proc. VLDB Endow.* 11.7 (2018), pp. 800–812.

[75] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. "Computation Reuse in Analytics Job Service at Microsoft". In: (2018), pp. 191–203.

[76] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002*. ACM, pp. 546–557.

[77] Avita Katal, Mohammad Wazid, and R. H. Goudar. "Big data: Issues, challenges, tools and Good practices". In: *Sixth International Conference on Contemporary Computing, IC3 2013, Noida, India, August 8-10, 2013*. IEEE, pp. 404–409.

[78] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots". In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pp. 195–206.

[79] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. "Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?" In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, pp. 715–730.

[80] You Jung Kim and Jignesh M. Patel. "Rethinking Choices for Multidimensional Point Indexing: Making the Case for the Often Ignored Quadtree". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, pp. 281–291.

[81] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. "Learned Cardinalities: Estimating Correlated Joins with Deep Learning". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[82] * Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, pp. 347–358.

[83] * Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Geospatial Joins for Modern Hardware". In: *CoRR* abs/1802.09488 (2018).

[84] * Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Approximate Geospatial Joins with Precision Guarantees". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 1360–1363.

[85] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. "RadixSpline: a single-pass learned index". In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*. ACM, 5:1–5:5.

[86] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. "RadixSpline: A Single-Pass Learned Index". In: *CoRR* abs/2004.14541 (2020). `https://arxiv.org/abs/2004.14541`.

[87] * Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* Pp. 49–60.

[88] * Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: *ACM Trans. Database Syst.* 44.1 (2019), 1:1–1:35.

[89] Marcel Kornacker et al. "Impala: A Modern, Open-Source SQL Engine for Hadoop". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org.

[90] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. "SageDB: A Learned Database System". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[91] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, pp. 489–504.

[92] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning". In: *CoRR* abs/1808.03196 (2018).

[93] John Krumm, Nigel Davies, and Chandra Narayanaswami. "User-Generated Content". In: *IEEE Pervasive Comput.* 7.4 (2008), pp. 10–11.

[94] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, pp. 239–250.

[95] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. “Skew-resistant parallel processing of feature-extracting scientific user-defined functions”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, pp. 75–86.

[96] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40.

[97] Doug Laney. “3D data management: Controlling data volume, velocity and variety”. In: *META group research note* 6.70 (2001), p. 1.

[98] Harald Lang, Andreas Kipf, Linnea Passing, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines”. In: *Proceedings of the 14th International Workshop on Data Management on New Hardware, 2018*. ACM, 5:1–5:8.

[99] Kisung Lee, Raghu K. Ganti, Mudhakar Srivatsa, and Ling Liu. “Efficient spatial query processing for big data”. In: *Proceedings of the 22nd ACM SIGSPATIAL, 2014*.

[100] Kisung Lee, Ling Liu, Raghu K. Ganti, Mudhakar Srivatsa, Qi Zhang, Yang Zhou, and Qingyang Wang. “Lightweight Indexing and Querying Services for Big Spatial Data”. In: *IEEE Trans. Services Computing* 12.3 (2019), pp. 343–355.

[101] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. “STR: A Simple and Efficient Algorithm for R-Tree Packing”. In: *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*. IEEE Computer Society, pp. 497–506.

[102] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. “QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning”. In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2118–2130.

[103] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. “LISA: A Learned Index Structure for Spatial Data”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 2119–2133.

[104] Ling Liu and M. Tamer Özsu, eds. *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[105] *Making the most detailed tweet map ever*. `https://blog.mapbox.com/making-the-most-detailed-tweet-map-ever-b54da237c5ac`.

[106] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. “Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data”. In: *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019*. Vol. 2322. CEUR Workshop Proceedings. CEUR-WS.org.

[107] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. "Evaluating Geospatial Geometry and Proximity Queries Using Distributed Hash Tables". In: *Computing in Science and Engineering* 16.4 (2014), pp. 53–61.

[108] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. "Polygon-Based Query Evaluation over Geospatial Data Using Distributed Hash Tables". In: *IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013, Dresden, Germany, December 9-12, 2013*.

[109] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*. ACM, 3:1–3:4.

[110] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1705–1718.

[111] Chris A. Mattmann. "Computing: A vision for data science". In: *Nat.* 493.7433 (2013), pp. 473–475.

[112] *MemSQL Geospatial*. `http://www.memsql.com/content/geospatial/`.

[113] João Miranda. *Uber Unveils its Realtime Market Platform*. `http://www.infoq.com/news/2015/03/uber-realtime-market-platform/`.

[114] *MongoDB Releases - New Geo Features in MongoDB 2.4*. `https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24/`. 2013.

[115] Laurence Moore. "Transverse Mercator Projections and US Geological Survey Digital Products". In: *US Geological Survey, Professional Paper* (1997).

[116] *NASA OpenNEX*. `https://nex.nasa.gov/nex/static/htdocs/site/extra/opennex/`.

[117] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. "Learning Multi-Dimensional Indexes". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 985–1000.

[118] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 677–689.

[119] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure". In: *ACM Trans. Database Syst.* 9.1 (1984), pp. 38–71.

[120] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1634–1645.

[121] *Number of smartphones sold to end users worldwide from 2007 to 2020*. `https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/`.

[122] *NYC Taxi and Limousine Commission (TLC) - TLC Trip Record Data*. `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`. 2019.

[123] *One Million Rides a Day*. `https://blog.lyft.com/posts/one-million-rides-a-day`.

[124] Peter van Oosterom, Oscar Martinez-Rubi, Milena Ivanova, Mike Hörhammer, Daniel Geringer, Siva Ravada, Theo Tijssen, Martin Kodde, and Romulo Goncalves. "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark". In: *Comput. Graph.* 49 (2015), pp. 92–125.

[125] *OPTIMIZE LOCAL AND GLOBAL DECISIONS WITH SNOWFLAKE'S GEOSPATIAL SUPPORT*. `https://www.snowflake.com/blog/optimize-local-and-global-decisions-with-snowflakes-geospatial-support/`.

[126] *Oracle Spatial and Graph Spatial Features*. `https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/spatialfeatures-1902020.html/`. 2019.

[127] Jack A. Orenstein. "Redundancy in Spatial Databases". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*.

[128] * Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "How Good Are Modern Spatial Analytics Systems?" In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1661–1673.

[129] * Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. "High-Performance Geospatial Analytics in HyPerSpace". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 2145–2148.

[130] * Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. "The Case for Learned Spatial Indexes". In: *AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan*.

[131] * Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. “An Evaluation Of Modern Spatial Libraries”. In: *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 21-24, 2020, Proceedings, Part II*. Vol. 12113. Lecture Notes in Computer Science. Springer, pp. 157–174.

[132] * Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. “How Good Are Modern Spatial Libraries?” In: *Data Sci. Eng.* 6.2 (2021), pp. 192–208.

[133] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. “QuickSel: Quick Selectivity Learning with Mixture Models”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 1017–1033.

[134] Wendel Góes Pedrozo, Júlio César Nievola, and Deborah Carvalho Ribeiro. “An Adaptive Approach for Index Tuning with Learning Classifier Systems on Hybrid Storage Environments”. In: *Hybrid Artificial Intelligent Systems - 13th International Conference, HAIS 2018, Oviedo, Spain, June 20-22, 2018, Proceedings*. Vol. 10870. Lecture Notes in Computer Science. Springer, pp. 716–729.

[135] *Persistent Disk: Reliable, high-performance block storage for virtual machine instances*. https://cloud.google.com/persistent-disk.

[136] *PostGIS*. http://postgis.net/.

[137] *Project voldemort: A distributed database*. https://www.project-voldemort.com/voldemort/. 2010.

[138] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. “Effectively Learning Spatial Indices”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2341–2354.

[139] Matt Ranney. *Scaling Uber's Real-time Market Platform*. https://www.infoq.com/presentations/uber-market-platform/. 2015.

[140] T. Ramalingeswara Rao, Pabitra Mitra, Ravindara Bhatt, and A. Goswami. “The big data system, components, tools, and technologies: a survey”. In: *Knowl. Inf. Syst.* 60.3 (2019), pp. 1165–1245.

[141] David Reinsel, John Gantz, and John Rydning. “The digitization of the world from edge to core”. In: *Framingham: International Data Corporation* (2018). https://storecloud.org/media/idc-seagate-dataage-whitepaper.pdf.

[142] Frank Ren, Xiaohu Li, Devin Thomson, and Daniel Geng. *Geosharded Recommendations Part 1: Sharding Approach*. https://tech.gotinder.com/geosharded-recommendations-part-1-sharding-approach-2/. 2018.

[143] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. "Managing Non-Volatile Memory in Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, pp. 1541–1555.

[144] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Building blocks for persistent memory". In: *The VLDB Journal* (2020), pp. 1–19.

[145] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Persistent Memory I/O Primitives". In: *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*. ACM, 12:1–12:7.

[146] Keven Richly. "Optimized Spatio-Temporal Data Structures for Hybrid Transactional and Analytical Workloads on Columnar In-Memory Databases". In: *Proceedings of the VLDB 2019 PhD Workshop, co-located with the 45th International Conference on Very Large Databases (VLDB 2019), Los Angeles, California, USA, August 26-30, 2019*. Vol. 2399. CEUR Workshop Proceedings. CEUR-WS.org.

[147] Victoria Rubin and Tatiana Lukoianova. "Veracity roadmap: Is big data objective, truthful and credible?" In: *Advances in Classification Research Online* 24.1 (2013), p. 4.

[148] *S2 cells and Pokémon GO*. https://pokemongohub.net/post/wiki/s2-cells-pokemon-go/. 2018.

[149] *S2Geometry Overview - Spherical Geometry*. https://s2geometry.io/about/overview/.

[150] Zahra Sadri, Le Gruenwald, and Eleazar Leal. "Online Index Selection Using Deep Reinforcement Learning for a Cluster Database". In: *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, pp. 158–161.

[151] Shubham Saxena. *Appreciating the geo/S2 library*. https://blog.gojekengineering.com/fe-f0e4a909d56f. 2017.

[152] AMM Scaife. "Big telescope, big data: towards exascale with the Square Kilometre Array". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190060.

[153] Todd Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance*. http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.

[154] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. "Geodetic Distance Queries on R-Trees for Indexing Geographic Data". In: *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings*, pp. 146–164.

[155] Raghav Sethi et al. “Presto: SQL on Everything”. In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, pp. 1802–1813.

[156] Salman Ahmed Shaikh, Komal Mariam, Hiroyuki Kitagawa, and Kyoung-Sook Kim. “GeoFlink: A Distributed and Scalable Framework for the Real-time Processing of Spatial Streams”. In: *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. ACM, pp. 3149–3156.

[157] *Shipments of Wearable Devices Leap to 125 Million Units, Up 35.1% in the Third Quarter, According to IDC*. https://www.idc.com/getdoc.jsp?containerId=prUS47067820. 2020.

[158] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. “The Hadoop Distributed File System”. In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. IEEE Computer Society, pp. 1–10.

[159] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. “Improving Spatial Data Processing by Clipping Minimum Bounding Boxes”. In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, pp. 425–436.

[160] Antoine Sinton. *Geospatial indexing on Hilbert curves*. https://blog.zen.ly/geospatial-indexing-on-hilbert-curves-2379b929addc/. 2018.

[161] Uthayasankar Sivarajah, Zahir Irani, and Vishanth Weerakkody. “Evaluating the use and impact of Web 2.0 technologies in local government”. In: *Gov. Inf. Q.* 32.4 (2015), pp. 473–487.

[162] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, and Vishanth Weerakkody. “Critical analysis of Big Data challenges and analytical methods”. In: *Journal of Business Research* 70 (2017), pp. 263–286.

[163] Swaminathan Sivasubramanian. “Amazon dynamoDB: a seamlessly scalable non-relational database service”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, pp. 729–730.

[164] *Spatial Reference*. https://spatialreference.org/.

[165] *SR-ORG:6864 | EPSG:3857*. https://spatialreference.org/ref/sr-org/6864/.

[166] Ram Sriharsha. *Magellan: Spark as a Geospatial Analytics Engine*. https://databricks.com/session/magellan-spark-as-a-geospatial-analytics-engine/. 2019.

[167] Emmanuel Stefanakis, Yannis Theodoridis, Timos K. Sellis, and Yuk-Cheung Lee. "Point Representation of Spatial Objects and Query Window Extension: A New Technique for Spatial Access Methods". In: *Int. J. Geogr. Inf. Sci.* 11.6 (1997), pp. 529–554.

[168] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. "How to Architect a Query Compiler, Revisited". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, pp. 307–322.

[169] Ruby Y. Tahboub and Tiark Rompf. "Architecting a Query Compiler for Spatial Workloads". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 2103–2118.

[170] Ruby Y. Tahboub and Tiark Rompf. "On supporting compilation in spatial query engines: (vision paper)". In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*. ACM, 9:1–9:4.

[171] Javid Taheri, Albert Y. Zomaya, Howard Jay Siegel, and Zahir Tari. "Pareto frontier for job execution and data transfer time in hybrid clouds". In: *Future Gener. Comput. Syst.* 37 (2014), pp. 321–334.

[172] MingJie Tang, Ruby Y. Tahboub, Walid G. Aref, Mikhail J. Atallah, Qutaibah M. Malluhi, Mourad Ouzzani, and Yasin N. Silva. "Similarity Group-by Operators for Multi-Dimensional Relational Data". In: *IEEE Trans. Knowl. Data Eng.* (2016).

[173] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. "LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data". In: *PVLDB* 9.13 (2016), pp. 1565–1568.

[174] *The size of the World Wide Web (The Internet)*. `www.worldwidewebsize.com/`.

[175] Konstantinos Theocharidis, John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. "SRX: efficient management of spatial RDF data". In: *VLDB J.* 28.5 (2019), pp. 703–733.

[176] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. "Hive - a petabyte scale data warehouse using Hadoop". In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE Computer Society, pp. 996–1005.

[177] John Paul Titlow. *How Foursquare Is Building A Humane Map Framework To Rival Google*. `https://www.fastcompany.com/3007394/how-foursquare-building-humane-map-framework-rival-googles/`. 2013.

[178] Theodoros Toliopoulos, Nikodimos Nikolaidis, Anna-Valentini Michailidou, Andreas Seitaridis, Anastasios Gounaris, Nick Bassiliades, Apostolos Georgiadis, and Fotis Liotopoulos. "Developing a Real-Time Traffic Reporting and Forecasting Back-End System". In: *Research Challenges in Information Science - 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23-25, 2020, Proceedings*. Vol. 385. Lecture Notes in Business Information Processing. Springer, pp. 58–75.

[179] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. "Parallel In-Memory Evaluation of Spatial Joins". In: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*. ACM, pp. 516–519.

[180] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. "Parallel In-Memory Evaluation of Spatial Joins". In: *CoRR* abs/1908.11740 (2019).

[181] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. "A Two-level Spatial In-Memory Index". In: *CoRR* abs/2005.08600 (2020). https://arxiv.org/abs/2005.08600.

[182] *Tutorials: Filtering Tweets by location*. https://developer.twitter.com/en/docs/tutorials/filtering-tweets-by-location. 2020.

[183] Uber. *Uber Newsroom: 10 Billion*. https://www.uber.com/newsroom/10-billion/. 2018.

[184] *Understanding Memory Management | Oracle*. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html/.

[185] Jeffrey S Vitter. "Random sampling with a reservoir". In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[186] Hoang Vo, Ablimit Aji, and Fusheng Wang. "Sato: A spatial data partitioning framework for scalable query processing". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, pp. 545–548.

[187] Mehul Nalin Vora. "Hadoop-HBase for large-scale data". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1. IEEE, pp. 601–605.

[188] H. Wang, X. Fu, J. Xu, and H. Lu. "Learned Index for Spatial Queries". In: *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pp. 569–574.

[189] Eric W Weisstein. "Great circle". In: (2002).

[190] *Wing: Air delivery when you need it.* https://wing.com/.

[191] Chris Wong. *FOILing NYC's Taxi Trip Data*. http://chriswhong.com/open-data/foil_nyc_taxi/.

[192] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. “Simba: Efficient In-Memory Spatial Analytics”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 1071–1085.

[193] Qihe Yang, John Snyder, and Waldo Tobler. *Map projection transformation: principles and applications*. CRC Press, 1999.

[194] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. “Qd-tree: Learning Data Layouts for Big Data Analytics”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, pp. 193–208.

[195] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. “Selectivity Estimation with Deep Likelihood Models”. In: *CoRR* abs/1905.04278 (2019).

[196] Peter N. Yianilos. “Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”. In: *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA.* Pp. 311–321.

[197] Simin You, Jianting Zhang, and Le Gruenwald. “Large-scale spatial join query processing in Cloud”. In: *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pp. 34–41.

[198] Simin You, Jianting Zhang, and Le Gruenwald. “Spatial join query processing in cloud: Analyzing design choices and performance comparisons”. In: *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on*. IEEE, pp. 90–97.

[199] *YouTube for Press*. `https://blog.youtube/press/`.

[200] Jia Yu, Raha Moraffah, and Mohamed Sarwat. “Hippo in Action: Scalable Indexing of a Billion New York City Taxi Trips and Beyond”. In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, pp. 1413–1414.

[201] Jia Yu and Mohamed Sarwat. “Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems”. In: *Proc. VLDB Endow.* 10.4 (2016), pp. 385–396.

[202] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. “A demonstration of GeoSpark: A cluster computing framework for processing big spatial data”. In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, pp. 1410–1413.

[203] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. "GeoSpark: a cluster computing framework for processing large-scale spatial data". In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, 70:1–70:4.

[204] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. "Spatial data management in apache spark: the GeoSpark perspective and beyond". In: *GeoInformatica* 23.1 (2019), pp. 37–78.

[205] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. "Reinforcement Learning with Tree-LSTM for Join Order Selection". In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, pp. 1297–1308.

[206] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. "GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons". In: *Proc. VLDB Endow.* 11.3 (2017), pp. 352–365.

[207] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. "The Case for Distance-Bounded Spatial Approximations". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org.

[208] Eleni Tzirita Zacharatou, Darius Sidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. "Efficient Bundled Spatial Range Queries". In: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*. ACM, pp. 139–148.

[209] Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[210] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, pp. 423–438.

[211] Zeroghan. *A Comprehensive Guide to S2 Cells and Pokémon GO*. `https://pokemongohub.net/post/article/comprehensive-guide-s2-cells-pokemon-go/`. 2019.

[212] Feng Zhang, Ye Zheng, Dengping Xu, Zhenhong Du, Yingzhi Wang, Renyi Liu, and Xinyue Ye. "Real-Time Spatial Queries for Moving Objects Using Storm Topology". In: *ISPRS Int. J. Geo Inf.* 5.10 (2016), p. 178.

[213] Ji Zhang et al. "An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, pp. 415–432.

[214] Rui Zhang, Jianzhong Qi, Martin Stradling, and Jin Huang. "Towards a Painless Index for Spatial Objects". In: *ACM Trans. Database Syst.* 39.3 (2014), 19:1–19:42.

[215] Zheng Zhao, Ruiwen Zhang, James Cox, David Duling, and Warren Sarle. "Massively parallel feature selection: an approach based on variance preservation". In: *Mach. Learn.* 92.1 (2013), pp. 195–220.

[216] Tianyu Zhou, Hong Wei, Heng Zhang, Yin Wang, Yanmin Zhu, Haibing Guan, and Haibo Chen. "Point-polygon topological relationship query using hierarchical indices". In: *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*, pp. 562–565.

[217] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. "Database Meets Artificial Intelligence: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* (2020).

[218] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. "BestConfig: tapping the performance potential of systems via automatic configuration tuning". In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, pp. 338–350.