Efficient Query Processing when Spatial Data Meets Rdf Graph

Houssameddine Yousfi

THESE en cotutelle

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National – Arrêté du 25 mai 2016)

Ecole Doctorale : Mathématiques, informatique, matériaux, mécanique, énergétique (MIMME)

Secteur de Recherche : Informatique et Applications

Et

de l'Université de Tlemcen Abou Bekr Belkaid

Spécialité : Ingénierie des Systèmes d'Information et de connaissances et Aide à la décision

Présentée par :

Houssameddine YOUSFI

****

Efficient query processing when spatial data meets RDF graph

****

Directeurs de Thèse : **Allel HADJALI et Houcine MATALLAH**

Co-encadrant: **Amin MESMOUDI**

****

Soutenue le 07/12/2023

devant la Commission d'Examen

****

JURY

Rapporteurs :	**Mme Fatima DEBBAT**	Professeur, Université de Mascara M. STAMBOULI, Algérie
	M. Laurent D'ORAZIO	Professeur, Université Rennes 1, France
Membres du jury :	**M. Azeddine CHIKH**	Professeur, Université de Tlemcen, Algérie
	M. Mohand-Said HACID.	Professeur, Université Lyon 1, France
	M. Houcine MATALLAH	Maître de conférences - HDR, Université de Tlemcen, Algérie
	M. Amin MESMOUDI	Maître de conférences, Université de Poitiers, France
	M. Allel HADJALI	Professeur, ISAE-ENSMA, Poitiers, France

Acknowledgments

I would like to express my gratitude to the following individuals whose support and inspiration were instrumental in the completion of this thesis:

- First and foremost, I extend my heartfelt thanks to my supervisors Prof. Allel HADJALI, Dr. Amin MESMOUDI, and Dr. Houcine MATALLAH. Their guidance, patience, and unwavering support throughout my thesis were invaluable. The insightful discussions I had with each of them greatly influenced my ideas and helped me navigate my research. I am also grateful for their meticulous editing, which significantly contributed to the quality of this manuscript. I am truly grateful for the invaluable lessons in research they have imparted to me.

- I would like to express my sincere appreciation to Professors Fatima DEBBAT and Laurent D'ORAZIO for generously serving as the reviewers of my thesis. Their feedback and constructive criticism have been indispensable in enhancing the quality of my work. I am also thankful to Azeddine CHIKH Azeddine and Mohand-Saïd HACID for their involvement as members of the examination committee. Their thought-provoking questions have enriched my study from diverse perspectives.

- My gratitude extends to all the members of the LIAS laboratory who warmly welcomed me and with whom I shared numerous enjoyable moments throughout these three years. I am especially grateful to Pascal Richard for mentoring me during the initial stages of my teaching career. Additionally, I would like to express my sincere thanks to Bénédicte Boinot for her kindness, generosity, and invaluable administrative assistance in coordinating my thesis project.

- I am deeply thankful to my fellow Ph.D. colleagues and friends, both past and present. I am grateful for the fruitful discussions we had, where we exchanged ideas

on our academic endeavors. Your diverse conversations have been instrumental in broadening my perspective during challenging times.

- I want to express my heartfelt appreciation to my group of friends, who provided me with laughter and relief from the pressures of academic life. Your sense of humor has been a constant source of joy, brightening even the cloudiest days throughout this journey. Special thanks to Réda and Samir for their unwavering support throughout.

- Finally, I would like to convey my deep and sincere gratitude to my family for their unconditional love, unparalleled assistance, and unwavering support. To my parents, my sister Zakia, and my brother Mouhamed, thank you for always being there to uplift my spirits during moments of doubt. I am forever grateful for your unwavering support, as this journey would not have been possible without you.

Contents

Contents iii

List of Figures vii

List of Tables ix

Introduction 1

I State of the art 11

1 Preliminary 13

1.1 Introduction 15

1.2 Spatial data processing 15

1.2.1 Spatial Queries 18

1.3 RDF graph formalisation 20

1.4 Architectural overview of RDF_QDAG 22

1.4.1 Data storage 22

1.4.2 Scheduling layer 24

1.4.3 Engine layer 25

1.5 Conclusion 27

2 State of the art 29

2.1 Introduction 30

2.2 Taxonomy of spatial processing techniques 31

2.2.1 Storage medium 31

2.2.2 Indexing strategy 32

2.2.3 Parallel strategy 33

2.3 Centrelised spatial data processing . . . 33
2.3.1 Nested Loop . . . 33
2.3.2 Plane Sweep . . . 35
2.3.3 Index Nested Loop . . . 37
2.3.4 TOUCH . . . 40
2.3.5 Partition based spatial-merge join (PBSM) . . . 42
2.3.6 Dual index traversal (DIT) . . . 43
2.3.7 Discussion . . . 44
2.4 Spatial data processing at scale . . . 45
2.4.1 Distance-based join in Hadoop . . . 45
2.4.2 Spatial processing in Spark . . . 53
2.4.3 Discussion . . . 57
2.5 RDF data processing . . . 57
2.6 Spatial-RDF data processing . . . 59
2.7 Conclusion . . . 60

II Spatial and graph data processing 63

3 I/O Efficient R-tree utilisation 65
3.1 Introduction . . . 67
3.2 Problem Definition . . . 68
3.2.1 R-Tree Structure . . . 69
3.2.2 Problem statement . . . 69
3.3 FASTER approach . . . 71
3.3.1 Principle of FASTER . . . 71
3.3.2 Proof of correctness . . . 71
3.4 Experimental validation . . . 73
3.4.1 Experimental setup and data-sets . . . 74
3.4.2 Results discussion . . . 74
3.5 Conclusion . . . 75

4 Spatial RDF data querying 77
4.1 Introduction . . . 79

4.2 Query Evaluation strategies . . . 81
4.2.1 BGP-First strategy . . . 83
4.2.2 Spatial-First strategy . . . 85
4.3 Optimization techniques . . . 87
4.3.1 Query scheduling . . . 87
4.3.2 Spatial pruning . . . 91
4.4 Experimental evaluation . . . 92
4.4.1 Experimental setup and methodology . . . 93
4.4.2 Effect of evaluation strategies . . . 93
4.4.3 Effect of Scheduling . . . 94
4.4.4 Effect of Encoding . . . 96
4.4.5 Effect of spatial pruning . . . 97
4.4.6 Comparison against Virtuoso . . . 97
4.5 Conclusion . . . 98

Conclusion and Perspectives **101**

Bibliographie **107**

A **117**
A.1 Queries used for the experimental validation . . . 117
A.2 Results of estimation of each plan for the different queries considered . . . 119

List of Figures

1 Examples for RDF and SPARQL 3
2 Linked Open Data (LOD) evolution 4
3 Thesis outline. 8

1.1 Raster and vector representation. 16
1.2 Architectural overview of RDF_QDAG 23

2.1 Representation of the Revers Run Plane Sweep. 36
2.2 Representation of a kd-Tree. 38
2.3 Tree building and objects assignment in TOUCH. 41
2.4 Classification of scalable distance-based join techniques. 53

3.1 Overview of an R-tree's structure. 70
3.2 The effect of the number of the returned objects on the number of disk pages loaded. 73
3.3 Execution time (nanoseconds) of queries using SRT and FASTER. 75

4.1 The execution of an BGP-First plan 84
4.2 The structure of index pages and entries 86
4.3 Execution of Spatial-First strategy 86
4.4 Execution time (nanoseconds) of queries using both strategies BGP-first and Spatial-first. 93
4.5 Initial accuracy and the improved accuracy of the optimizer. 94
4.6 Execution time (ms) of queries using WKT and WKB. 96
4.7 Execution time (ms) of queries with and without Spatial Pruning. 97
4.8 Compression of execution time between Virtuoso and RDF_QDAG. 98

List of Tables

2.1 Summery of main contributions in spatial processing techniques 34
2.2 Summery of main contributions in spatial processing techniques (part 2) . 35
2.3 Types of spatial indices. 37
2.4 classification of join techniques. 45
2.5 Summery of main contributions in distance-based join processing in popular MPP frameworks (Hadoop and spark) 46
2.6 Summery of main contributions in distance-based join processing in popular MPP frameworks (Hadoop and spark), part 2. 47
2.7 Comparison of different storage strategies for RDF data, including examples of triplestores that utilize each strategy, their advantages and disadvantages. 58
2.8 Overview of different spatial extensions of RDF Triplestores. 60

3.1 Symbols and their meanings. 69

4.1 Example of RDF triples (dataset D1). 82
4.2 Execution time of queries on YAGO 94

A.1 Results of estimation of Q_1 120
A.2 Results of estimation of Q_2 120
A.3 Results of estimation of Q_3 120
A.4 Results of estimation of Q_4 120
A.5 Results of estimation of Q_5 121
A.6 Results of estimation of Q_6 121
A.7 Results of estimation of Q_7 121
A.8 Results of estimation of Q_8 121

Introduction

Context

The explosion in the amount of data generated and collected from various sources has led to the emergence of what is now known as the era of Big Data, where the management, analysis, and utilization of these vast volumes of data present significant challenges [FHL14]. According to estimates, we are witnessing an unprecedented production of spatial data. For example, space telescopes (e.g., Rubin Observatory) generate up to 20 terabytes (TB) of data per night[1] and hospitals produce spatial images (X-rays) at a rate of 50 petabytes (PB) per year[2]. Moreover, within the Twitter (now X) platform alone, there are 10 million geolocated tweets issued daily, accounting for approximately 2% of the total Twitter firehose [FMSHFM12]. This abundance of data and the increasing demand for large-scale processing characterize the era of Big Data. In the face of these challenges, it becomes essential to develop innovative methods for the efficient management of this data, leveraging technological advancements and proposing suitable approaches for storage, access, and processing of large scale data.

The rise of the Web has also played a significant role in the production and collection of large-scale data, giving rise to a wealth of structured and unstructured information. A key approach to organizing this knowledge and making it actionable is the creation of knowledge graphs, such as Google's well-known Knowledge Graph [3]. Knowledge graphs are labeled and directed data structures that encode information in the form of entities and relationships relevant to a specific domain or organization. These knowledge graphs play a crucial role in capturing and organizing a large amount of structured and multi-relational data, facilitating their exploration using query mechanisms. As powerful tools,

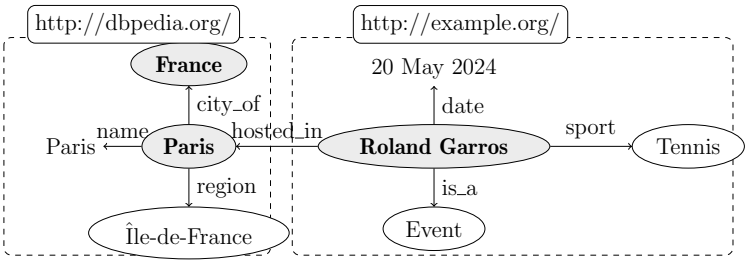
[1] `https://www.lsst.org/about/dm`
[2] https://www.tibco.com/blog/2021/02/26/
[3] `https://developers.google.com/knowledge-graph`

knowledge graphs become the backbone of the Web and existing information systems in various academic domains and industrial applications. Their power lies in their ability to extend existing knowledge without affecting previous knowledge, allowing for the continuous maintenance and enrichment of the knowledge base. This shift towards the creation of knowledge graphs further fuels the demand for standardized and efficient data representations, especially in the context of the semantic Web [BLHL01] with its vision of a globally accessible and linked internet of data.

The volume of knowledge graphs continues to grow exponentially, reflecting the increasing amount of structured information available on the Web. For example, Google's Knowledge Graph, which is one of the most widely known and used, contains billions of entities and relationships [4]. Additionally, initiatives such as DBpedia [5], which extracts structured data from Wikipedia, contain billions of facts representing knowledge in various domains. Another significant resource is the Wikidata project [6], which aims to create a free and collaborative knowledge base, also containing billions of facts. Furthermore, many organizations and institutions are creating their own knowledge graphs to represent and organize domain-specific information. The scale of these knowledge graphs is a testament to the growing importance of structured data and its potential to fuel artificial intelligence [TLH20], advanced information retrieval [LRZ+20], and numerous other knowledge-based applications.

In this context, the Linked Open Data (LOD) movement plays a central role in the aggregation and publication of large-scale knowledge graphs. LOD aims to make data available on the Web in a standardized and interconnected format, using the principles of the Semantic Web [Hit21]. Many organizations, universities, governments, and research projects actively contribute to LOD by publishing their structured datasets as publicly accessible knowledge graphs. These datasets cover a variety of domains such as geography [7], life sciences [8], culture [9], and more. For example, datasets such as DBpedia [10], Wikidata

[4] https://kennyallen.co.uk/exploring-googles-knowledge-graph/
[5] https://www.dbpedia.org/
[6] https://www.wikidata.org/wiki/Wikidata:Main_Page
[7] http://linkedgeodata.org/
[8] https://sparql.uniprot.org/
[9] https://data.bnf.fr/
[10] https://www.dbpedia.org/



(a) RDF Graph

```
@prefix db:<http://dbpedia.org/> .
@prefix ex:<http://example.org/> .

SELECT ?x ?y WHERE {
?x ex:hosted_in db:Paris .
?x ex:sport ?y .
}
```

(b) SPARQL query

Figure 1: Examples for RDF and SPARQL

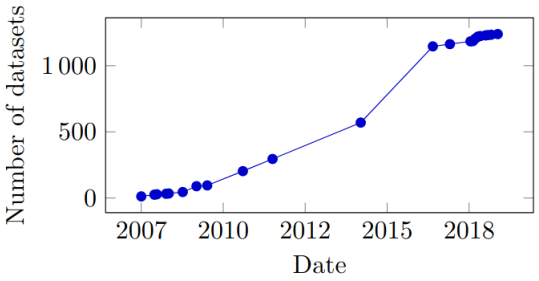
[11], GeoNames [12], and PubMed [13] provide rich and interconnected information on diverse subjects. By combining these datasets, semantic links can be established between different sources, allowing users to discover additional relations and knowledge. LOD is a valuable resource for research, data exploration, application development, and global knowledge enrichment.

RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language) play a fundamental role in leveraging knowledge graphs within the Linked Open Data (LOD) paradigm. RDF provides a standardized and extensible data model for representing information in the form of subject-predicate-object triplets, thereby structuring data in a format that is comprehensible by both humans and machines. This graph-based representation allows for capturing complex relationships between entities, enabling seamless data interconnection within the LOD.

> **Example 1** The statement *"Paris is a city of France"* can be represented by a triple as $\langle Paris, city\_of, France \rangle$[14]. This triple can be represented logically as a graph where two nodes (subject and object) are joined with a directed arc (predicate) as shown in Figure 1a.

SPARQL, on the other hand, is a query language specifically designed for querying RDF data. It provides powerful mechanisms for traversing knowledge graphs and extracting precise information from these interconnected datasets. SPARQL queries allow users to specify graph patterns, filters, and join operations to retrieve relevant information from the LOD. They also offer advanced features such as aggregation, sorting, and pagination of results.

[11] `https://www.wikidata.org/wiki/Wikidata:Main_Page`
[12] `https://www.geonames.org/`
[13] `https://pubmed.ncbi.nlm.nih.gov/`
[14] This triple is a part of DBpedia's Knowledge Graph.



Dataset	Number of triples (Millions)
DBLP	207
Yago	284
Freebase	2 000
DBPedia'14	3 000
Wikidata	12 000

Figure 2: Linked Open Data (LOD) evolution

Example 2 The query in Figure 1b asks for all the events hosted in Paris. The answer of this query is a subgraph of the queried graph(s) in which the variable terms are mapped to the values of the resulting subgraph. Processing a SPARQL query can be viewed as a subgraph matching problem. The results for the previous query are the mappings of $?x \rightarrow$ `<http://example.org/Roland Garros>` and $?y \rightarrow$ `<http:// example.org/Tennis>`.

The capabilities provided by RDF have led to the creation of large-scale databases. One notable initiative is the Linked Open Data (LOD) cloud, which allows for the referencing of available datasets on the Web. The number of datasets within the LOD cloud has grown rapidly, currently surpassing 1260 datasets. Many of these datasets consist of several billions of triples, as illustrated in Figure 2. Access to these datasets is facilitated through SPARQL endpoints, which are RESTful Web services that expose RDF data queried with SPARQL, or by downloading the data as data dumps.

RDF triples can be used to express spatial relationships by assigning geographic coordinates to entities such as places, points of interest, or geographic boundaries. This integration of spatial information in RDF enables the creation of interconnected geospatial knowledge graphs, opening up new possibilities for analyzing and discovering complex spatial relationships. However, to fully leverage this spatial information, it is necessary to incorporate spatial operators into SPARQL queries.

Spatial operators are features that allow specifying spatial conditions and relationships in SPARQL queries. For example, operators like "within", "intersects", or "near" can be used to filter geospatial entities based on their spatial relationships with other entities. These spatial operators provide powerful querying capabilities for analyzing and extracting geospatial information from RDF knowledge graphs.

Integrating spatial operators into SPARQL enables advanced queries that combine

both spatial conditions and graph criteria. This allows for sophisticated geospatial analysis and the discovery of rich relationships among geospatial entities. The combined use of RDF data representation and spatial operators in SPARQL provides a powerful means to explore and exploit geospatial information in knowledge graphs.

In the context of this thesis, our main objective is to address the challenge of efficient management of RDF data when incorporating spatial information, while ensuring an optimal balance between scalability and performance.

Spatial RDF: New Challenges for Data Management

RDF data processing represent a challenge that has been the focus of many studies. There are four main approaches to storing RDF data. The first approach is the single table strategy, which involves using a large relational table with three columns for subject, predicate, and object like Sesame [BKVH01]and 3-Store [HG03]. Another option is the binary table approach, where a binary table is created for each property, containing subject and object information like [AMMH09], C-store [WKB08]. This method is commonly used in scalable distributed systems. The third approach is the property table, where subjects with common properties are grouped and stored in a horizontal table with each column representing a property like Jena [WSK$^+$03] and DB2RDF [BDK$^+$13]. Finally, the fourth approach involves modeling and storing RDF data in its native graph form, treating subjects and objects as nodes and properties as labeled edges like Trinity [ZYW$^+$13] and RDF_QDAG [KMG$^+$21, ZMG$^+$21].

Spatial data presents a significant challenge due to its inherently multi-dimensional nature. Consequently, establishing a global order for the dataset becomes unfeasible. Traditional tree-based indexes rely on a global order, making them unsuitable for spatial data. Likewise, hash-based indexes are not well-suited since spatial operations primarily depend on proximity rather than exact matches. To tackle this challenge, researchers have proposed new indexing techniques. One approach taken by some researchers involves single-dimensional embedding. Works like Z-Curve and Hilbert-Curve aim to establish a global order by employing space-filling curves. Other researchers have opted to develop specialized indexes specifically designed for spatial data. These spatial indexes can be constructed using either object grouping techniques, such as R-tree and CR-tree [KCK01],

or space partitioning techniques, such as Xbr-Tree[RVL+15], Kdb-Tree [Rob81].

A second challenge rises not from spatial data itself but with the combination with the RDF data. Due to the difference between the two data representations, techniques that work for one representation do not work on the other. Many works have takled this challenge for example, Strabon [KKK12], which is an extension of Sesame [BKVH01], provides support for spatial data storage. It utilizes PostGIS to store the data and adopts a proprietary table-based approach.The query optimizer extension in Strabon is relatively simplistic, relying on heuristics to push down spatial filters. As Strabon is based on an older RDF store (Sesame), it lacks many of the optimization techniques employed in modern Triplestores. Brodt et al. [BNM10] also extended RDF-3X [NW08] to accommodate spatial data. However, their work only supports the range selection operation, offering a limited extension. Another spatial extension of RDF-3X is Geo-Store [WKC12], which employs a grid file for spatial data indexing. It is worth noting that many commercial systems, such as Oracle, Virtuoso [vir], and GraphDB [Gra], also support spatial RDF queries. However, specific details about their internal designs are not publicly accessible. Despite these efforts, efficient and large-scale management of spatial RDF data remains a challenging task that requires a complete revision of some components of RDF data management systems.

Approach and contributions

To accomplish our objective of achieving efficient Spatial RDF data management, we have made the decision to revise the core components of RDF_QDAG to incorporate Spatial operators. Numerous studies have demonstrated that RDF_QDAG offers a favorable balance between scalability and performance. This system relies on the physical fragmentation of RDF data and the exploration of the corresponding logical graph. Our proposals primarily focus on the storage, evaluation, and optimization layers.

The contributions made in this research can be summarized as follows. Firstly, we conduct an in-depth analysis of spatial data processing techniques in centralized environments, providing a comprehensive understanding of the existing approaches and techniques. Secondly, we propose a systematic classification of spatial data processing approaches at a large scale, specifically within popular frameworks. This classification

helps to categorize and compare different techniques based on their characteristics and capabilities.

Additionally, we introduce a novel exploration algorithm called FASTER, designed specifically for R-trees. This algorithm effectively reduces the number of input/output (I/O) operations, leading to improved performance in spatial data processing tasks. To ensure the reliability and accuracy of FASTER, we develop a formal proof of its correctness, validating its suitability for practical implementation.

Furthermore, we address the efficient processing of Spatial-RDF data within RDF_QDAG. We propose two evaluation approaches, namely Spatial-First and BGP-First, specifically tailored for spatial-RDF data. By adapting the FASTER indexing approach to be compatible with the graph exploration logic in RDF_QDAG, we enable efficient spatial data processing within the Spatial-First strategies.

To assess the performance of the proposed approaches, we conduct a comprehensive experimental evaluation using real-world datasets. We evaluate various measures and compare the results with a well-known commercial Triplestore, establishing the effectiveness and efficiency of the proposed techniques. Additionally, we study the impact of query evaluation strategies and optimization techniques on the performance of RDF_QDAG, providing insights into the factors influencing the system's efficiency.

Finally, we propose an optimizer capable of selecting the most suitable evaluation strategy based on the query characteristics and statistics about the RDF and spatial data. This optimizer enhances the overall performance of RDF_QDAG by dynamically choosing the optimal approach for query execution. The experimental evaluation and comparisons performed demonstrate the advantages of our proposed approaches in terms of efficiency and effectiveness, validating their applicability in real-world scenarios.

Thesis outline

The thesis is divided into two parts, as illustrated in figure 3. The first part consists of two chapters, which focus on fundamental concepts and the current state of the field. The second part, also comprising two chapters, is dedicated to presenting the contributions made by this thesis and describing the findings and results of the experimental validation.

Chapter 1 provides an introduction to background concepts related to RDF data and

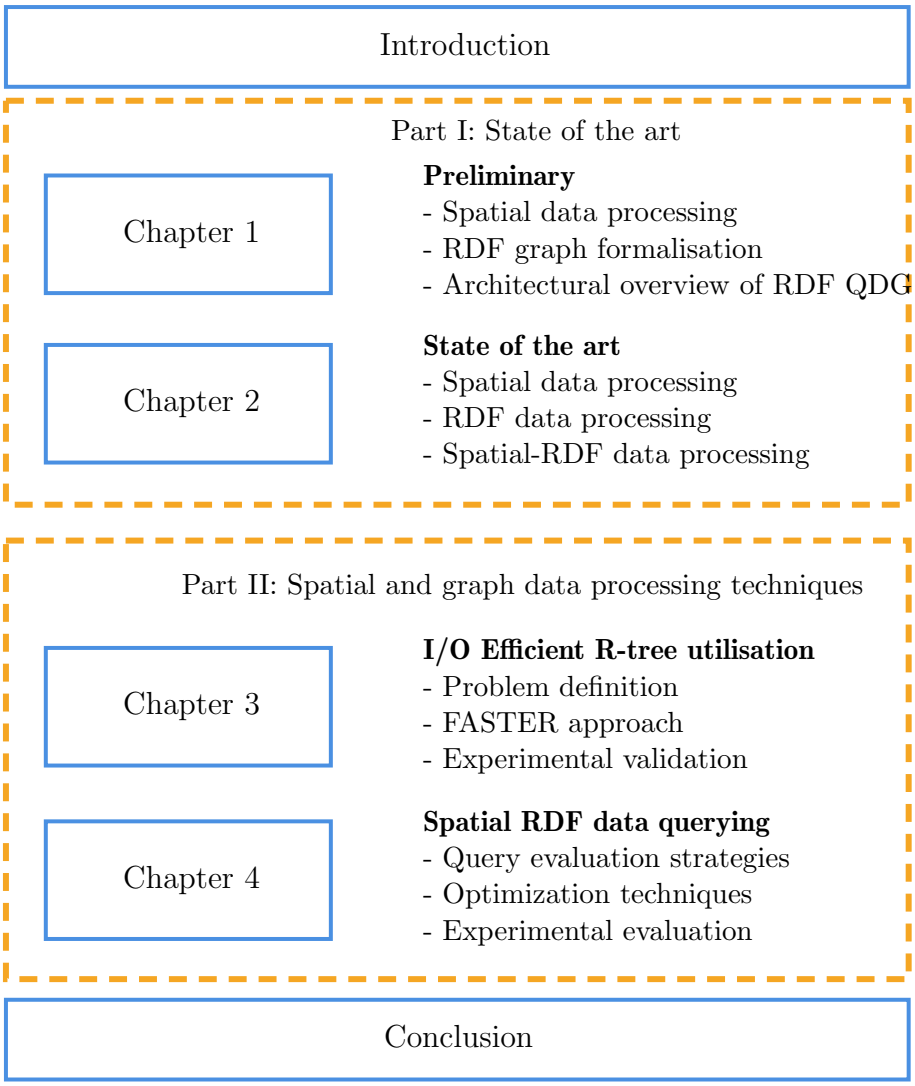


Figure 3: Thesis outline.

Spatial data. This involves formally defining various concepts and explaining the syntax of the querying language. The aim of this chapter is to provide the foundational elements that are crucial for comprehending the challenges addressed in this thesis, as well as the proposed solutions.

In Chapter 2, the state of the art in spatial data processing is examined. This chapter begins by presenting a taxonomy of spatial processing techniques, encompassing various aspects such as storage medium, indexing strategy, and parallel strategy. Additionally, centralized spatial data processing techniques are explored. The chapter further delves into spatial data processing at scale. The discussion also encompasses RDF data processing and the challenges it poses, as well as spatial-RDF data processing.

Chapter 3 focuses on the I/O efficient utilization of R-trees, a widely used spatial index structure. The chapter introduces the R-tree structure and the challenges associated with

its efficient utilization. The FASTER approach, which aims to improve I/O efficiency is proposed along with its underlying principles and proof of correctness. Experimental validation is conducted to assess the effectiveness of the proposed approach.

In Chapter 4, the attention shifts towards spatial RDF data processing. The chapter begins by exploring different query evaluation strategies, including BGP-First strategy and Spatial-First strategy, and their impact on performance. Optimization techniques, such as query scheduling and spatial pruning, are investigated to enhance query execution efficiency. Experimental evaluation is performed to evaluate the effectiveness of these techniques, considering various factors such as evaluation strategies, scheduling, encoding, and spatial pruning. Furthermore, a comparison against Virtuoso, a well-known RDF database system, is conducted to assess the performance improvements achieved.

Finally, the thesis concludes with a general conclusion and Perspectives, summarizing the key findings and contributions of the research. Future directions and potential areas for further exploration are discussed, highlighting the importance of ongoing advancements in spatial data processing and optimization techniques.

Publications

- Yousfi, H., Mesmoudi, A., Hadjali, A., Matallah, H., Benkabou, S.: SRDF QDAG: An Efficient End-to-End RDF Data Management when Graph Exploration Meets Spatial Processing. Computer Science and Information Systems, `https://doi.org/10.2298/CSIS230225046Y`.

- Yousfi, H. Spatial data processing meets RDF graph exploration: student research abstract. In : Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. 2022. p. 389-392. `https://doi.org/10.1145/3477314.3506964`.

- Yousfi, H., Mesmoudi, A., Hadjali, A., Matallah, H., Lahfa, F. (2022). Efficient R-Tree Exploration for Big Spatial Data. In: Kacprzyk, J., Balas, V.E., Ezziyyani, M. (eds) Advanced Intelligent Systems for Sustainable Development (AI2SD'2020). AI2SD 2020. Advances in Intelligent Systems and Computing, vol 1418. Springer, Cham. `https://doi.org/10.1007/978-3-030-90639-9_70`.

- Saidi, B., Yousfi, H., Mesmoudi, A., Benkabou, SE., Hadjali, A., Matallah, H.

(2022). RDF_QDAG in Action: Efficient RDF Data Querying at Scale. In: Chbeir, R., Huang, H., Silvestri, F., Manolopoulos, Y., Zhang, Y. (eds) Web Information Systems Engineering – WISE 2022. WISE 2022. Lecture Notes in Computer Science, vol 13724. Springer, Cham. https://doi.org/10.1007/978-3-031-20891-1_45

Part I

State of the art

Chapter 1

Preliminary

Contents

1.1 Introduction 15

1.2 Spatial data processing 15

1.2.1 Spatial Queries 18

1.3 RDF graph formalisation 20

1.4 Architectural overview of RDF_QDAG 22

1.4.1 Data storage 22

1.4.2 Scheduling layer 24

1.4.3 Engine layer 25

1.5 Conclusion 27

In this chapter, we will establish the fundamental concepts related to spatial data and RDF data, which will serve as the basis for the subsequent chapters. Our focus will be on introducing spatial data types and exploring spatial queries. Additionally, we will dive into the representation of graph data within the RDF framework and discuss querying techniques using SPARQL. Lastly, we will present the key concepts associated with RDF_QDAG, a triple store specifically designed to efficiently handle RDF data and provide effective answers to SPARQL queries. By covering these essential topics, our goal is to provide the necessary background knowledge for the upcoming chapters.

1.1 Introduction

The proposition of a novel approach for processing spatial graph data necessitates a profound comprehension of the techniques and models utilized in both the graph and spatial domains. As mentioned previously, the demand for efficient processing of spatial and graph data is on the rise, leading to the central problem addressed in this thesis. Prior to delving into the explanation of the contributions, it is imperative to establish the background information. This involves providing formal definitions of various concepts and elucidating the syntax of the querying language.

The aim of this chapter is to furnish the essential components required to comprehend the challenges tackled in this thesis, as well as the proposed solutions. The chapter will follow this structure: an initial section 1.2 will provide pertinent insights into spatial data storage and processing, followed by an presentation on the Resource Description Framework (RDF)(section 1.3). Finally, the architecture of RDF_QDAG, a specialized triple store tailored to effectively manage large scale RDF datasets and provide optimized answers to SPARQL queries, will be presented in section 1.4. This arrangement will enable readers to develop a comprehensive grasp of the indispensable concepts and technologies necessary for the subsequent discussions and analyses within this thesis.

1.2 Spatial data processing

Spatial Database Management Systems (SDBMS) are designed to integrate spatial data types into their models and query languages, enabling efficient processing of spatial operations [Güt94]. Spatial data types encompass any data object that includes coordinates within a multi-dimensional metric space. SDBMSs serve as the foundation for Geographic Information Systems (GIS) and various computer-assisted design systems (CADs).

There are two primary representations for spatial information: Raster and Vector [SKS+97]. Raster data is structured as an array or matrix, where each cell or pixel corresponds to a spatial unit. However, this study does not focus on spatial operations with raster data. On the other hand, vector data represents object features using a list of vertices, where curves and polygons are depicted through line segments. Figure 1.1 visually demonstrates the same spatial area mapped using both vector and raster representations.

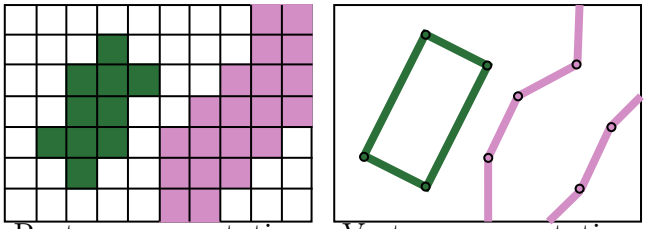


Figure 1.1: Raster and vector representation.

In addition to storing spatial data, Spatial Database Management Systems (SDBMS) are required to execute spatial operations in order to effectively respond to queries. Spatial queries can be classified into three categories: nearness queries, region queries, and join queries. Nearness queries involve retrieving objects that are in close proximity to a specified point, with a special case being the K nearest neighbor query. Region queries aim to identify objects that either fully or partially reside within a specified region. Lastly, join queries involve filtering the Cartesian product of two datasets based on a condition, typically expressed as a Boolean expression. When this condition involves a spatial operation, it is referred to as a spatial join. Further elaboration on the concept of spatial join, including a formal definition, will be provided in subsequent sections.

When considering a distance-based join, it refers to a join operation wherein the filtering condition requires the calculation of a distance. This distance is a function that assesses the separation between two objects within a metric space. The formal definition is provided below:

Definition 1.2.1 (**Distance function**)**.** given a set X, the distance d is a function

$$d : X \times X \to [0; +\infty[$$

In order to consider a function as a metric (i.e distance).For all x, y and $z \in X$ the following axioms must be satisfied:

$$\text{Non negativity axiom: } d(x; y) \geq 0 \tag{1.1}$$

$$\text{Identity of indiscernible: } d(x; y) = 0 \Leftrightarrow x = y \tag{1.2}$$

$$\text{symmetry: } d(x; y) = d(y; x) \tag{1.3}$$

$$\text{triangle inequality: } d(x; z) \leq d(x; y) + d(y; z) \tag{1.4}$$

Numerous distinct distance functions are utilized across various fields. Notable examples include the edit distance, applied in the realm of natural language processing [XWL08, WXLZ09, ZHOS10, WFL10, JDW+13], and the geodesic distance, commonly used in graph theory. However, among these measures, the Euclidean distance stands out as the most widely recognized and frequently employed. This distance metric is defined as the norm of the line segment connecting two points within Cartesian space. Formally, it can be expressed as follows:

Definition 1.2.2 (**Euclidean distance**)**.** in Cartesian space if $\mathbf{p} = (p_1, p_2, ..., p_n)$ and $\mathbf{q} = (q_1, q_2, ..., q_n)$ are two points in Euclidean n-space, then the distance (d) from $\mathbf{p}$ to $\mathbf{q}$ is:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Most of the approaches found in the existing literature are designed with a focus on the Euclidean distance in two or three-dimensional spaces. While these approaches could potentially be applied using different distance functions and in higher dimensions, the effects of these two parameters have not been thoroughly studied.

The second type of distance we will focus on throughout this work is the great-circle distance [Adm22]. It is defined as the length of the shortest arc between two points on the surface of a sphere. For any two points that are not opposite each other on a sphere, there exists a unique great circle that passes through the two points. This circle divides into two arcs, and the great-circle distance between the two points is determined by the length of the shorter arc. The great-circle distance is also referred to as the orthodromic distance or spherical distance. The formula for calculating the great-circle distance is as follows:

Definition 1.2.3 (**Great-circle distance**)**.** Let λ_1, ϕ_1 and λ_2, ϕ_2 be the geographical longitude and latitude of two points 1 and 2, $\Delta\lambda, \Delta\phi$ be their absolute differences; then $\Delta\sigma$, the central angle between them, is given by the spherical law of cosines if one of the poles is used as an auxiliary third point on the sphere:

$$\Delta\sigma = \arccos\Big(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos(\Delta\lambda)\Big).$$

1.2.1 Spatial Queries

Spatial queries are database queries that involve spatial data, such as maps, spatial databases, and geographic information systems. These queries allow users to search, analyze, and manipulate spatial data to extract meaningful insights and patterns. Spatial queries can be used to answer a wide range of questions, such as: What is the distance between two points? What features are located within a given area? Spatial queries can be categorized into three main types: region queries, nearness queries and join queries.

Nearness queries are a type of spatial query that focuses on identifying objects or features that are in close proximity to a specified location. These queries find widespread use in applications requiring geospatial analysis, including location-based services, environmental monitoring, and urban planning. For example, a nearness query can be employed to discover all restaurants within a specific distance from a given point. This type of query proves beneficial for both tourists and locals seeking dining options in a particular area. Likewise, a nearest-neighbor query can be executed to ascertain the object or feature nearest to a specified point, such as locating the closest gasoline station. This query type serves the needs of drivers who require refueling while on the road.

An important characteristic of nearness queries is their ability to be executed without requiring prior knowledge of the distance to the nearest object or feature. For instance, a nearest-neighbor query for a gasoline station can be conducted even when the user lacks information about the proximity of the nearest station. This inherent flexibility renders nearness queries highly advantageous, especially in scenarios where the user's familiarity with the surrounding area is limited.

The second type of spatial queries is region queries. They are a type of spatial query that focuses on identifying objects or features located within a specified spatial region. These queries find common use in geospatial analysis applications, including urban planning, environmental management, and natural resource management. A specific type of region query is the partial or complete containment query, which seeks objects located within a defined region on a map or within specific geographic boundaries. For example, a region query could be employed to identify all retail shops that are either partially or entirely situated within the geographic boundaries of a given town.

Such queries can serve a variety of purposes. For instance, urban planners may utilize region queries to pinpoint areas with a significant concentration of retail shops within a

specific town, aiding decisions regarding zoning and land use. Environmental managers may employ region queries to locate areas containing protected habitats, thus informing decisions concerning land conservation and wildlife management.

Furthermore, queries can also inquire about the union and intersection of regions. For example, a query might seek regions characterized by both low yearly rainfall and high population density, using details such as annual rainfall and population density.

The last category of queries involves distance-based joins. In our work, we focus on some of the most extensively studied ones, namely the k nearest neighbor join ($KNNJ$), the k closest pairs join ($KCPJ$) and the ϵ distance join (ϵDJ). It is worth noting that $KCPJ$ and ϵDJ share similarities, making it relatively straightforward to adapt algorithms from one to the other. There are other more specialized distance-based joins, such as the multi-way [VNB03] join and the iceberg join [LGWL17], but these are beyond the scope of this thesis.

Consider two datasets, D and Q. An ϵ-distance join requests objects from D that fall within a distance threshold of ϵ from objects in Q. The outcome is a set of pairs of objects that meet the specified distance criterion. For instance, an application could involve identifying all buildings that are within 10 km of a hospital. The formal definition of ϵDJ is provided below:

Definition 1.2.4 (ϵ **Distance Join (**ϵDJ**).** Let $P = \{p_0, p_1, ..., p_{n-1}\}$ and $Q = \{q_0, q_1, ..., q_{n-1}\}$ be two set of points in E^d, and a range of distance defined by $[\epsilon_1, \epsilon_2]$ such as that $\epsilon_1, \epsilon_2 \in \mathbb{R}^+$, and $\epsilon_1 \leq \epsilon_2$.The ϵ Distance Join (ϵDJ) of P and Q ($\epsilon DJ(P, Q, \epsilon_1, \epsilon_2) \subseteq P \times Q$) is a set which contains all the possible pairs of points (p_i, q_j) that can be formed by choosing one point $p_i \in P$ and one point $q_j \in Q$ such as: $\epsilon DJ(P, Q, \epsilon_1, \epsilon_2) = \{(p_i, q_j) \in P \times Q : dist(p_i, q_j) \in [\epsilon_1, \epsilon_2]\}$.

The K closest pair join differs slightly in that, for two datasets, D and Q, the result consists of the top k items from the list of pair combinations formed by D and Q, sorted by distance in ascending order. An illustrative real-life scenario involves identifying the nearest 10 hotels to a park. The formal definition is as follows:

Definition 1.2.5 (K **Closest Pair Query,** $KCPQ$**).** Let
$P = \{p_0, p_1, ..., p_{n-1}\}$ and $Q = \{q_0, q_1, ..., q_{n-1}\}$ be two set of points in E^d, let $K \in \mathbb{N}^+$, the K Closest Pairs Query ($KCPQ$) of P and Q ($KCPQ(P, Q, K) \subseteq P \times Q$) is set of

K different ordered pairs $KCPQ(P, Q, K) = \{(p_{Z1}, q_{L1}), (p_{Z2}, q_{L2}), ..., (p_{Z1k}, q_{Lk}))\}$ with $(p_{Zi}, q_{Li}) \neq (p_{Zj}, q_{Lj}), Z_i \neq Z_j \wedge L_i \neq L_j$ such that:

for any $(p, q) \in P \times Q\{(p_{Z1}, q_{L1}), (p_{Z2}, q_{L2}), ..., (p_{Z1k}, q_{Lk})\}$ we have $dist(p_{Z1}, q_{L1}) \leq dist(p_{Z2}, q_{L2}) \leq ... \leq dist(p_{Z1k}, q_{Lk}) \leq dist(p, q)$.

The case of the K nearest neighbor join is markedly distinct. Involving two datasets, D and Q, the $KNNJ$ query aims to locate, for each object in Q, the k nearest objects in D. For instance, one might seek the three closest taxis for each client. To formally define the $KNNJ$, we first introduce the following definition of "nearness":

Definition 1.2.6 (k **nearest neighbors,** KNN)**.** Given an object r, a dataset S and an integer k, the k nearest neighbors of r from S, denoted as $KNN(r, S)$, is a set of k objects from S that $\forall o \in KNN(r, S), \forall s \in S - KNN(r, S), |o, r| \leq |s, r|$.

Definition 1.2.7 (k **nearest neighbors join,** $KNNJ$)**.** Given two datasets R and S and an integer k, kNNjoin of R and S ($R \ltimes S$) is defined as:

$$R \ltimes_{kNN} S = \{(r, s) | \forall r \in R, \forall s \in KNN(r, S)\}$$

1.3 RDF graph formalisation

In the following sections of this study, we will introduce a system designed to handle spatial and graph data in RDF format. Consequently, this section is dedicated to presenting the fundamental concepts related to RDF. RDF serves as the cornerstone for semantic Web applications that can effectively manage extensive knowledge graphs [Hit21].

Data in RDF is expressed through triples known as SPO triples, which comprise a subject, property, and object. Subjects are uniquely identified using a Uniform Resource Identifier (URI). The property signifies the relationship between the subject and the object. The object in a triple can either serve as the subject of another triple or represent a simple data value known as a literal. By adhering to this format, data can be represented as a graph, where nodes correspond to subjects or objects, and edges symbolize the connecting properties. To aid comprehension in the subsequent sections, we provide the following formal definitions relevant to RDF graphs.

Definition 1.3.1. *(RDF graph)* An RDF graph is a four-tuple $G = \langle V, L_V, E, L_E \rangle$, where

1. V Is a collection of vertices that correspond to all subjects and objects in RDF data. The set V can be divided into V_l and V_e where V_l is the set of literal vertices and V_e is the set of entity vertices.

2. L_V is the set of vertex labels. The label of a vertex $u \in V_l$ is its literal value, and the label of a vertex $u \in V_e$ is its corresponding URI.

3. $E = \overrightarrow{u_1, u_2}$ is a collection of directed edges that connect the corresponding subject and objects.

4. L_E is a collection of edge labels. Given an edge $e \in E$, its edge label is its corresponding property.

For querying RDF data, the SPARQL query language is utilized [PAG09]. SPARQL formulates queries using a basic graph pattern (BGP) that involves variables. The query's outcome is a collection of variable mappings, wherein a subgraph within the data aligns with the query's graph pattern. Filters can be integrated into the query to define conditions on the graph elements. The forthcoming section offers a formal definition of a SPARQL query:

Definition 1.3.2. *(Query graph)* A query graph is a five-tuple $Q = \langle V^Q, L_V^Q, E^Q, L_E^Q, FL \rangle$, where

1. $V^Q = V_e^Q \cup V_l^Q \cup V_p^Q$ is a collection of vertices that correspond to all subjects and objects in a SPARQL query, where V_p^Q is a collection of parameter vertices, and V_e^Q and V_l^Q are collections of entity vertices and literal vertices in the query graph Q respectively.

2. L_V^Q is a collection of vertex labels in Q. A vertex $v \in V_p^Q$ has no label, while that of a vertex $v \in V_l^Q$ is its literal value and that of a vertex $v \in V_e^Q$ is its corresponding URI.

3. E^Q is a collection of edges that correspond to properties in a SPARQL query. L_E^Q are the edge labels in E^Q.

4. FL are constraint filters, such as a wildcard constraint or a spatial constraint.

When spatial data is stored in RDF format, the spatial information is commonly represented as literals. Consequently, to express spatial operations, spatial functions are employed within the filter section of a SPARQL query. Numerous extensions to the SPARQL language have been developed to support spatial filters. In this context, we rely on the GeoSPARQL standard [BK12] established by the Open Geospatial Consortium (OGC). GeoSPARQL extends both RDF and SPARQL to facilitate the representation and querying of spatial information. For a comparable range of functionalities, stSPARQL [KK10] can also be considered.

1.4 Architectural overview of RDF_QDAG

RDF_QDAG [KMG+21] is structured into multiple layers, with each layer comprising several components, as illustrated in Figure 1.2. This section provides an overview of the system's architecture and delves into the intricacies of the query evaluation process.

1.4.1 Data storage

The storage layer within RDF_QDAG plays a pivotal role in the efficient storage and retrieval of diverse data types, with a particular focus on Graph and Spatial data. The data within RDF_QDAG encompasses various native types, such as Strings, Integers, Doubles, and more. To ensure optimal querying across this spectrum of data, RDF_-QDAG relies predominantly on three access methods: B+tree, R-tree, and a Dictionary.

The B+tree serves as the central storage mechanism for graph data within RDF_-QDAG, enabling efficient indexing and retrieval of data based on key-value pairs. Through the utilization of B+trees, the storage layer ensures the organization and access of graph data in an ordered and optimized fashion.

To uphold the semantics of the graph, which hinges on the relationships among the graph elements (predicates in the context of RDF), the graph is divided into graph fragments. This partitioning accounts for the connectivity between these fragments. An optimal graph partitioning strategy seeks to maximize inter-partition connectivity while minimizing intra-partition connectivity. This strategic partitioning not only maintains the integrity of the graph but also amplifies the performance and efficiency of queries conducted on the graph data.

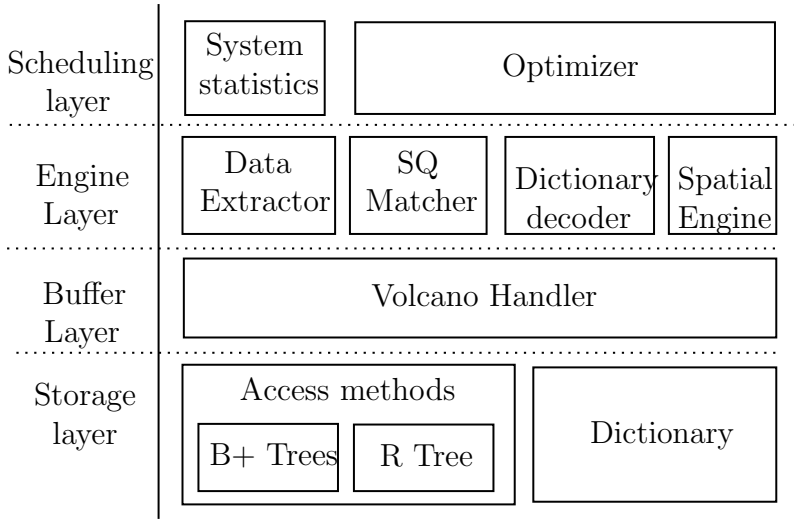


Figure 1.2: Architectural overview of RDF_QDAG

Collectively, the synergy between B+trees as the primary storage structure and a proficient graph partitioning strategy empowers RDF_QDAG to adeptly store and access graph data, while upholding its semantics and facilitating high-performance querying.

Every graph fragment comprises a collection of Data Stars. Data Stars expand upon the concept of a tuple in the relational model. Formally, we define Data Stars as follows:

Definition 1.4.1. *(Data Star)* Given a node x (named data star head) in a RDF graph G, a Data Star $DS(x)$ is the set of either triples sharing the same subject x, or the same object x. We name *Forward Data Star* and *Backward Data Star* the sets $\overrightarrow{DS}(x) = \{(x,p,o)|\exists_{p,o} : (x,p,o) \in G\}$ and $\overleftarrow{DS}(x) = \{(s,p,x)|\exists_{s,p} : (s,p,x) \in G\}$ respectively.

When we contrast the concept of a Data Star with that of a tuple, the primary key of a tuple aligns with the head x of a Data Star $DS(x)$. RDF_QDAG organizes comparable Data Stars into sets referred to as *Graph Fragments*, utilizing characteristic sets [NM11].

Each subject s in the graph G possesses a characteristic set defined as $\overrightarrow{cs}(s) = p|\exists_o : (s,p,o) \in G$. Similarly, for objects, $\overleftarrow{cs}(o) = p|\exists_s : (s,p,o) \in G$. Forward graph fragments, denoted as $\overrightarrow{Gf}$, group forward Data Stars with the same characteristic set. Backward graph fragments, represented as $\overleftarrow{Gf}$, are formed in a similar manner. The formal definition of this concept is provided in Definition 1.4.2.

Definition 1.4.2. *(Graph Fragment)* A Graph Fragment is a set of Data Stars. It is named a Forward Graph Fragment $\overrightarrow{Gf}$ if it groups Forward Data Stars such that:

$$\overrightarrow{Gf} = \{\overrightarrow{DS}(x)|\forall_{i \neq j} \overrightarrow{cs}(x_i) = \overrightarrow{cs}(x_j)\}.$$

Likewise, a Backward Graph Fragment $\overleftarrow{Gf}$ is defined as

$$\overleftarrow{Gf} = \{\overleftarrow{DS}(x) | \forall_{i \neq j} \overleftarrow{cs}(x_i) = \overleftarrow{cs}(x_j)\}.$$

After partitioning the graph into graph fragments, each fragment is loaded into an index. In the context of RDF_QDAG, a B+tree is employed as the index. The efficacy of this indexing approach for graph data is extensively explored [KMG+21]. Additionally, compression techniques are employed to optimize storage space utilization and reduce the number of pages loaded into buffers during query evaluation [NW08].

As part of the effort to optimize space usage, Subjects and Objects within the graph, if of the type String or URI, are substituted with an ID. This step is essential to manage the potential substantial size of the fragments, particularly when subject/object values recur frequently within the fragments. However, this approach mandates the creation of a dictionary to store $< value, ID >$ pairs. Furthermore, an additional encoding and decoding process becomes necessary for evaluating each query.

The third and final access method is a spatial access method, specifically the R-tree. We introduced this extension to RDF_QDAG within the context of this work to facilitate support for spatial queries. Further details regarding spatial indexing are provided in Section 4.2.

1.4.2 Scheduling layer

The main component of the scheduling layer is the optimizer. The optimizer has the role of selecting the best execution plan for a given query. This process is divided into two steps: (i) plan enumeration and (ii) cost estimation. The plan with the lowest estimated cost is the one chosen by the optimizer for evaluation.

The nature of the plan is contingent upon the system design. In conventional systems, a plan can be viewed as a sequence of join operations on triple patterns. However, in the context of RDF_QDAG, the concept of a data star is introduced as an analog of a tuple in the relational model. Similarly, the notion of a star query is proposed, wherein triple patterns sharing the same Subject or Object are grouped as either a forward or backward data star.

Definition 1.4.3. *(Query Star)* Let Q be the SPARQL query graph. A Forward Query

Star $\overrightarrow{QS}(x)$ is the set of triple patterns such that $\overrightarrow{QS}(x) = \{(x, p, o)|\exists_{p,o} : (x, p, o) \in Q\}$, x is named the *head* of the Query Star. Likewise, a Backward Query Star $\overleftarrow{QS}(x)$ is $\overleftarrow{QS}(x) = \{(s, p, x)|\exists_{s,p} : (s, p, x) \in Q\}$. We use $\overrightarrow{QS}, \overleftarrow{QS}$ to denote the set of forward and backward query stars and qs to denote indistinctly a forward and backward query star.

An execution plan is an ordering function applied to a set of Query Stars and Filter Units. This function determines the sequence in which the mappings for each Query Star will be identified and the order in which the filter units will be evaluated.

Definition 1.4.4. *(Execution Plan)* . We denote by $\mathcal{P} = [QS_1, QS_2, Fu_1(p_1, p_2), ..., QS_n]$ the plan formed by executing QS_1, then QS_2, then evaluating the filter unit $Fu_1(p_1, p_2)$ which requires the mappings of p_1 and p_2 parameters.

1.4.3 Engine layer

The engine layer is assigned the responsibility of assessing the query, with a specific emphasis on executing the most advantageous plan as furnished by the optimizer.

The evaluation of a Query Star involves identifying matches between the variables of the Query Star and the nodes within the data graph. For each triple within the star, we aim to identify the set of mappings that fulfill its conditions. Subsequently, we combine the mappings associated with the individual triples to construct the matches for the Query Star.

Definition 1.4.5. (Star Query Evaluation) The evaluation of a Query Star $QS(x)$ against the graph G is formally defined as follows:

$$[\![QS(x)]\!]_G := \{[\![tp_1]\!]_G \bowtie [\![tp_2]\!]_G \bowtie ... \bowtie [\![tp_n]\!]_G | n = card(QS(x))\}$$

where:

$$[\![tp_i]\!]_G \bowtie [\![tp_j]\!]_G = \{\mu_l \cup \mu_r | \mu_l \in [\![tp_i]\!]_G \text{ and } \mu_r \in [\![tp_j]\!]_G, \mu_l \sim \mu_r \text{ and } \mu_l(tp_i) \neq \mu_r(tp_j)\}$$

We denote that a mapping μ is a function $V_p^Q \rightarrow V^G$. Given two mappings μ_1 and μ_2, $\mu_1 \sim \mu_2 \Rightarrow \mu_1(?x) = \mu_2(?x)$.

Based on the previous definitions, we can determine the evaluation of a query using the set of query stars, as follows:

Definition 1.4.6. (Query Evaluation) Given a set of stars, $\{qs_1, qs_2,...,qs_n\}$, that cover the query, $Triples_q(qs_1) \cup Triples_q(qs_2) \cup ... \cup Triples_q(qs_n) = Triplets(q)$, the evaluation of the BGP part of the query q using the set of query stars is defined as follows:

$[\![q]\!]_G = \{\mu : \forall \mu \in [\![qs_1]\!]_G \bowtie [\![qs_2]\!]_G \bowtie ... \bowtie [\![qs_n]\!]_G \}$

We can also set the query BGP evaluation based on fragments, as follows:

$[\![q_g]\!]_G = \{\mu : \forall \mu \in \bigcup_{Gf \models qs_1} [\![qs_1]\!]_{Gf} \bowtie \bigcup_{Gf \models qs_2} [\![qs_2]\!]_{Gf} \bowtie ... \bowtie \bigcup_{Gf \models qs_n} [\![qs_n]\!]_{Gf}\}$

Where $Gf \models qs$ iff $cs(qs) \subset cs(Gf)$

The full evaluation of the query is the evaluation of the BGP part and the filters FL and it is defined as follows

$[\![q_g]\!]_G = \{\mu : \forall \mu \in [\![qs_1]\!]_G \bowtie [\![qs_2]\!]_G \bowtie ... \bowtie [\![qs_n]\!]_G | \mu \models FL\}$

Listing 1.1: Example of simple RDF query (Q_1)

```
SELECT ?p
WHERE {
?p <hasArea> ?a .
?p <isLocatedIn> ?l .
?l <hasGeometry> ?g .
};
```

An execution plan $\mathcal{P}$ is called an Acceptable Execution Plan if it fulfills the following conditions:

1. *Coverage:* All nodes and predicates of the given query are *covered* by the set of Query Stars of the plan. In the case of Query Q_1 the execution plan $[\overleftarrow{?l}, \overrightarrow{?p}]$ is not an acceptable plan since it does not cover the edge $< hasGeometry >$ and the variable $?g$.

2. *Instantiated head:* This condition guarantees that for a plan $\mathcal{P} = [QS_1, ... , QS_n]$, $\forall_{i>1} QS$, the head of the QS_i must be already instantiated. We use this condition to avoid to a Cartesian product when mappings are exchanged between two star queries. For example, in the case of Query Q_1 the execution plan $[\overleftarrow{?l}, \overleftarrow{?g}, \overrightarrow{?p}]$ is not an acceptable plan since the mapping of $?g$ is not yet available for the second $\overleftarrow{?g}$ to be evaluated. In this case the instantiated head condition is not satisfied.

The formal definition of an Acceptable Plan is given in Proposition 1.4.7.

Definition 1.4.7. *(Acceptable Plan)* $\mathcal{AP}$ Let us consider Q as a given query, $\overrightarrow{QS}$ and $\overleftarrow{QS}$ as the sets of forward and backward graph star queries respectively, T has the set of triple patterns and the following functions:

- Tr: $Q \cup \overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow T$ It returns the set triple patterns of a query star or a query.
- Nd: $\overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow V$ It returns the nodes of a query star (subject or object).
- Head: $\overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow V$ a function that returns the head of a query star.

An **acceptable plan** $\mathcal{AP}$ is a tuple $< X, f >$ where $X \subset \overrightarrow{QS} \cup \overleftarrow{QS}$ and $f : X \rightarrow \{1...|X|\}$ is the query stars order function such that:

1. $\bigcup_{QS \in X} Tr(QS) = Tr(Q)$
2. $\forall i \in \{2...|X|\}, Head(f^{-1}(i)) \in \bigcup_{j=1}^{i-1} Nd(f^{-1}(j))$

1.5 Conclusion

In conclusion, this chapter has laid the groundwork for understanding the subsequent chapter by introducing fundamental concepts related to spatial data representation and processing. The discussion also encompassed the representation of graph data within the RDF framework. Moreover, the main architecture of RDF_QDAG, a triple store specifically designed for storing RDF data and executing SPARQL queries, was presented.

It is important to emphasize that comprehending the architecture of RDF_QDAG is essential for grasping the subsequent exploration of the integration of spatial features, as discussed in Chapter 4. The insights gained from this chapter serve as a solid foundation for delving into the spatial aspects of RDF_QDAG in the following chapters.

Chapter 2

State of the art

Contents

2.1 Introduction 30

2.2 Taxonomy of spatial processing techniques 31

2.2.1 Storage medium 31

2.2.2 Indexing strategy 32

2.2.3 Parallel strategy 33

2.3 Centrelised spatial data processing 33

2.3.1 Nested Loop 33

2.3.2 Plane Sweep 35

2.3.3 Index Nested Loop 37

2.3.4 TOUCH 40

2.3.5 Partition based spatial-merge join (PBSM) 42

2.3.6 Dual index traversal (DIT) 43

2.3.7 Discussion 44

2.4 Spatial data processing at scale 45

2.4.1 Distance-based join in Hadoop 45

2.4.2 Spatial processing in Spark 53

2.4.3 Discussion 57

2.5 RDF data processing 57

2.6 Spatial-RDF data processing 59

2.7 Conclusion 60

This chapter presents a comprehensive exploration of the contemporary landscape in spatial data processing, RDF data processing, and the intersection of spatial-RDF data processing. It commences by introducing a taxonomy of spatial processing techniques, encompassing considerations of storage medium, indexing strategy, and parallelization approach. Subsequently, the chapter delves into centralized spatial data processing techniques, offering insights into methods such as nested loops, Plane Sweep, and partition-based spatial-merge join. It then delves into the realm of scalable spatial data processing, focusing on distance-based joins within Hadoop and spatial processing using Spark. The chapter also addresses RDF data processing, encompassing RDF triple stores and SPARQL query processing. Finally, it explores the emergent field of spatial-RDF data processing, shedding light on the integration of spatial and RDF data. By laying this multifaceted groundwork, the chapter establishes a solid platform for the ensuing chapters while providing valuable insights into the advancements and challenges across these domains.

2.1 Introduction

This chapter provides an in-depth exploration of the state of the art in spatial data processing, RDF data processing, and spatial-RDF data processing. It begins by presenting a taxonomy of spatial processing techniques, considering factors such as storage medium, indexing strategy, and parallel strategy. This taxonomy serves as a foundation for understanding the different approaches and algorithms discussed throughout the chapter.

The chapter then delves into the centralized spatial data processing techniques. It explores various methods, including nested loop[ME92], Plane Sweep[CB17], TOUCH[NTH+13], partition-based spatial-merge join (PBSM)[PD96], and dual index traversal (DIT)[BKS93]. Each technique is analyzed in terms of its strengths, limitations, and applicability to different scenarios.

Next, the chapter focuses on spatial data processing at scale, considering the challenges posed by large-scale datasets. It discusses distance-based join in Hadoop, which enables efficient spatial processing in a distributed computing environment. Additionally, it explores spatial processing in Spark, a popular framework for big data processing, highlighting its capabilities and performance characteristics.

Subsequently, the chapter shifts its attention to RDF data processing. It examines the unique characteristics of RDF data and presents various approaches for storage, querying, and analysis. The discussion encompasses RDF triple stores, SPARQL query processing, and RDF data integration techniques.

Finally, the chapter addresses the emerging field of spatial-RDF data processing. It explores the integration of spatial and RDF data, highlighting the potential benefits and challenges. The chapter concludes with a summary of the discussed techniques, their implications, and potential future directions for research and development in the field.

2.2 Taxonomy of spatial processing techniques

Before introducing existing techniques from the literature, we initiate by describing our proposed taxonomy and elucidating the rationale behind our classification criteria. Indeed, numerous approaches exist for classifying distance-based join techniques. Our chosen criteria for this taxonomy include: firstly, the consideration of the storage medium; secondly, the reliance on the indexing strategy; and lastly, the classification based on the parallel strategy.

2.2.1 Storage medium

In terms of the storage medium utilized, we can categorize existing work into two distinct groups: those that employ internal memory and those that utilize external memory (*i.e.*,disk).

Memory-based approaches have gained significant popularity in recent years due to the declining cost of memory acquisition and the continuous increase in main memory size. This approach offers numerous advantages. Firstly, it leads to performance improvements by eliminating disk overhead (i.e., no seek time is required). Moreover, managing data in internal memory is facilitated by the random access capabilities of this medium. it is worth noting that many in-disk approaches ultimately involve in-memory execution at some stage. Consequently, one potential implication is that in-memory techniques can be extended to leverage available disk space.

Despite the advantages presented by memory-based approaches, they are not without their challenges. For instance, the dataset size is constrained by the available memory

size. Although spilling to secondary disk is feasible, it contradicts the core objective of the approach. Another notable issue is the lack of fault tolerance for in-memory data, owing to the volatile nature of this storage medium. To address this concern, various in-memory replication techniques spanning multiple workers have been proposed.

In contrast, a typical in-disk workflow involves several steps. The first step involves splitting the data into partitions, a process known as data partitioning. These generated partitions are then loaded into memory and processed independently. The process of loading and processing all partitions of the dataset is referred to as a run. Depending on the process, multiple runs may be performed on the data to answer the query. Between two runs, intermediary results can be generated and stored on disk. The final step involves aggregating these intermediary results to obtain the final outcome.

The primary advantage of disk-based approaches lies in their capability to process a virtually unlimited amount of data, owing to the availability of secondary storage. However, such propositions are often constrained by I/O costs. Indeed, a well-optimized algorithm strives to avoid loading unnecessary data into memory, which may require intricate access methods and buffer management strategies. Additionally, the unique challenge of spatial data being situated in a continuous multi-dimensional space further complicates the partitioning process necessary for in-disk approaches. Often, the need to consider adjacent partitions arises when query points are located close to the edge of a given partition.

2.2.2 Indexing strategy

The presence of indices is a prerequisite for several approaches, albeit the process of building them can be resource-intensive. A conventional query optimizer may prioritize certain plans over others based on the presence of indices. It is in this context that we have chosen to categorize existing techniques based on the number of indices needed for processing the join. This classification can be divided into three cases: neither of the two sets is indexed, both sets are indexed, or only one of the sets is indexed.

Various indexing techniques have been proposed for spatial data. Some employ one-dimensional embedding to leverage existing mono-dimensional structures like B-trees. Others are purpose-built for spatial data. The simplest structure utilizes grid files [LN97], where the space is partitioned into adjacent rectangular cells. More sophisticated tree-

based structures have also been introduced. Among the most well-known are Kd-trees[Ben75] and quad-trees [FB74]. For preserving the concept of nearness more effectively, other tree-based structures have been proposed. The most prevalent ones in the literature include R-trees, which rely on Minimum Bounding Boxes (MBRs), and M-trees, which rely on minimum bounding hyperspheres [BKS93].

2.2.3 Parallel strategy

In the era of large-scale data processing, it becomes imperative to facilitate substantial parallel execution for a given approach. From this perspective, we differentiate between two cases: parallel algorithms and local algorithms. In the context of local algorithms, we refer to sequential single-threaded algorithms.

For parallel evaluation of the join operator, data partitioning across multiple workers is essential. As mentioned earlier, partitioning spatial data while preserving nearness poses a challenge. Any partitioning undertaken may incur an additional cost associated with inter-worker communication during the join process. Consequently, a parallel approach must strike a balance between performance and scalability.

2.3 Centrelised spatial data processing

In this section, we outline the primary contributions concerning the processing of distance-based join. We commence by introducing the naive algorithm, which serves as a baseline for processing such joins. Subsequently, we delve into various other approaches, providing an analysis of their strengths and trade-offs. Finally, we conclude with a summarizing table categorizing the presented techniques. Additionally, a comprehensive discussion of these approaches is provided. The surveyed work in this section are listed in Table 2.1.

2.3.1 Nested Loop

The nested loop algorithm (NL) [ME92] is often regarded as the naive join algorithm. It exhibits the worst computational complexity, with an order of $O(n^2)$.

The NL algorithm compares each object in the first dataset with every object from the second dataset using the joining condition. In other words, it relies on filtering the Cartesian product of the two datasets.

Table 2.1: Summery of main contributions in spatial processing techniques

Author	Approach name	Compared with	Evaluation metrics	Evaluation dataset
Klaus Hinrichs et al	PS		Theoretical evaluation	
Jon Louis Bentley	kd-Tree		Theoretical evaluation	
John T. Robinson	Kdb-Tree		Storage utilisation, pages accessed, query efficiency.	Synthetic data
R. A. Finkel *et al.*	Quad-Trees		Pages accessed	Synthetic data
George Roumelis et al (2014)	RRPS	Plane Sweep	Response time, distance computations, X-axis distance computations	6 Real world datasets
George Roumelis et al (2016)	FCCPS SCCPS FRCPS SRCPS	Compare the proposed approaches mutually	Response time, distance computations, X-axis distance computations, The number of disk accesses	6 Real world dataset
George Roumelis et al (2011)	Xbr-tree (using BF and DF algorithems)	Compare BF and DF mutually	Disk accesses and execution time	6 Real world dataset
Kihong Kim et al	CR-tree	R-tree	Operations execution time, Nbr of node accesses, False hit ratio, Nbr of caches misses	Synthetic data

Table 2.2: Summery of main contributions in spatial processing techniques (part 2)

Author	Approach name	Compared with	Evaluation metrics	Evaluation dataset
Sadegh Nobari et al	TOUCH	NL, Nes R-tree, PS, Dual R-tree PBSM, S3	Nbr of comparisons, Execution time, Memory usage, Nbr of object filtered	Sythetic data, Private neuro simulation data
Jignesh M Patel et al	PBSM	Nes R-tree, Dual R-tree	Execution time, Replication overhead	TIGER data, Hydrogra-phy, Rail

The nested loop algorithm was initially proposed as an in-memory solution. Despite its simplicity, adapting it for on-disk execution is straightforward since it requires the presence of only two objects in memory at any given time. The on-disk version is referred to as the "external nested loop join" (ENL). This approach divides both datasets into smaller partitions, allowing two subsets to fit into memory simultaneously. The ENL then applies a classic NL operation on each possible combination of subset pairs.

2.3.2 Plane Sweep

The Plane Sweep (PS) algorithm was originally proposed in [HNS88] as an approach that does not rely on any index yet offers improved performance compared to NL by reducing the search space. This method was initially used in computational geometry to perform interval joins and find line intersections [CB17]. However, it can also be adapted for spatial data processing. The steps to execute the Plane Sweep algorithm are as follows:

- The first step is to sort both datasets along one axis (the same axis for both datasets), let say the X axis for the sake of simplicity.

- The second step consists on passing a sweeping plane along this axis (let say from left to right). The idea of the sweeping plane is to compare the most left point from both datasets(called the reference point) to the left points of the second dataset by calculating Δx with δx being the distance on the x axis. If $\Delta x < \epsilon$ then the true distance is calculated. If $\Delta x > \epsilon$, there is no more object to be joined with the current object on the second dataset since it's sorted by X. In this case the

algorithm pass to the next most left object from any of the two datasets.

An example of Plane Sweep execution is shown in figure 2.1

Various improvements can be applied to enhance the classical Plane Sweep by introducing additional filtering steps between the Δx evaluation and the actual distance calculation. For instance, the Sliding Window Plane Sweep (SWPS) [RVCM14] incorporates a Δy evaluation. On the other hand, the Sliding Semi-circle Plane Sweep (SSPS) [RVCM14] introduces further comparisons involving the squared distance, which is the computationally intensive part of the Euclidean distance calculation. This evaluation can be visualized geometrically as a sliding semicircle centered on the query point. Figure 2.1 depicts the geometric interpretation of the window and semicircle evaluations.

The Reverse Run Plane Sweep (RRPS) is an advancement over the PS that achieves even more effective filtering by altering the evaluation order [RVCM14]. It introduces the concept of ”runs” defined as sequences of points ordered by x from one dataset that remain uninterrupted by any object from the other dataset, as illustrated in Figure 2.1. By comparing each pair of consecutive runs starting from the closest pair and extending outward, the algorithm can filter objects from both sides, resulting in a further reduction of the search space.

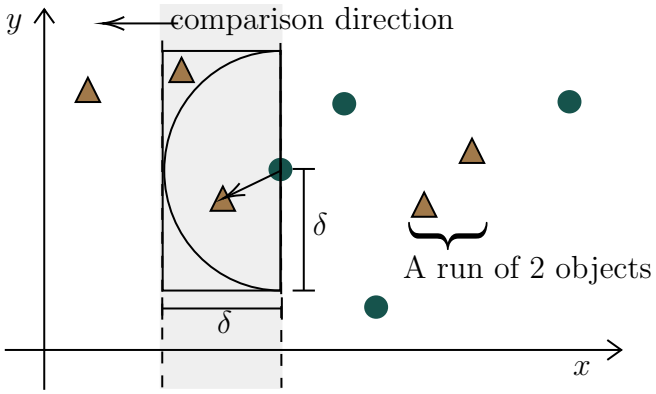


Figure 2.1: Representation of the Revers Run Plane Sweep.

The PS and its variants were proposed originally as an in-memory approaches. However further work adapt them for secondary storage by combining them with partitioning mechanisms. In [RCVM16] the authors propose and evaluate four variants of disk-based Plane Sweep witch are FCCPS, FRRPS, SCCPS and SRRPS. The four variants are based on different plane sweep techniques combined with partitioning techniques. The results of the experiments show that FRRPS outperforms the others in most of the cases.

2.3.3 Index Nested Loop

Index Nested Loop (INL) is a variation of the NL algorithm that incorporates an index lookup, resulting in a reduced search space. In the INL approach, one of the datasets (typically the outer dataset) is traversed, and potential join candidates from the second dataset are selected using index lookups. This algorithm can be tailored to support various types of indices. The effectiveness of the algorithm hinges on the pruning capabilities of the chosen index.

Spatial indices can be categorized into two main groups: space-based partitioning and object-based grouping. Space-based partitioning indices involve dividing the space into distinct non-overlapping regions. This approach can result in data duplication when objects intersect multiple regions, leading to their insertion into multiple partitions. Conversely, object-based partitioning indices permit partition intersection while preventing object fragmentation. If an object falls within multiple partitions, it is inserted into just one of them. Table 2.3 provides a classification of various existing indices, which will be elaborated further in the subsequent sections.

Table 2.3: Types of spatial indices.

Space partitioning based indices	Object partitioning based indices
Grid files	R-tree
Kd-tree	CR-tree
Q-tree	
Xbr-tree	
R*-tree	

Grid files

Grid files represent one of the simplest multidimensional data indices that can outperform single-dimensional embedding. This technique involves partitioning the space into stripes along each dimension.

The intersection of n stripes from n distinct dimensions is referred to as a cell. Each cell is associated with a data bucket, typically containing data bounded by its left and lower boundaries, while excluding the upper and right boundaries. The width of a stripe can vary, and the number of stripes can differ for each dimension.

Grid files can exhibit good performance in specific applications [ŠŠC+09], but they also suffer from significant drawbacks. For instance, as the dimensionality of the data

increases, performance tends to deteriorate, particularly for nearness queries where processing surrounding cells becomes necessary. In n dimensions, the number of surrounding cells is $(3^n) - 1$, and checking all of these cells can substantially degrade performance. Another limitation pertains to the cost of balancing. Splitting or resizing a single cell requires evaluating all cells in the same stripe and, in some cases, adjacent stripes as well. This limitation makes grid files ill-suited for highly skewed data.

Kd-trees

Kd-trees are a tree-based structure that stores data entries in the leaves [Ben75]. They rely on the fact that each internal node splits the objects into two sets along one dimension. This operation is repeated by cycling through dimensions until a fixed point condition is reached. The fixed point condition is related, for example, to the limit on the number of objects in each cell or, in some other cases, to the cell coverage (such as covered area, for example).

Figure 2.2 provides a structural representation of a kd-tree and its associated geometric interpretation. In a similar context, the kdb-tree structure was introduced to support secondary storage. This variant of the kd-tree permits a larger fan-out[Rob81]. Constructing a kd-tree is efficient, with a time complexity of $O(n \log n)$ [WH06]. Kd-trees excel in efficient nearness search, offering a time complexity of $O(\log n)$. However, balancing the tree proves challenging. Traditional tree rotation techniques are ineffective due to the changing split direction in each level. To address this, Adaptive kd-trees[Rob81] tackle the balancing issue during construction by selecting the optimal dimension for splitting at each level. While this approach yields well-balanced trees for static data, it struggles to maintain this balance with frequently updated dynamic data.

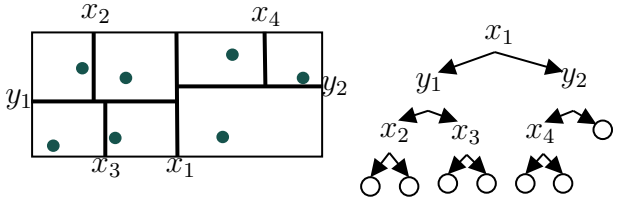


Figure 2.2: Representation of a kd-Tree.

Quad-trees

In the context of a two-dimensional plane, Quad-trees enable each internal node to partition its designated space into four quadrants [FB74]. This concept can be extended to higher dimensions. For instance, the oct-tree was introduced to handle three-dimensional space, while the hyper-quad-tree generalizes the idea to more than three dimensions.

Quad-trees operate by recursively dividing their space until each bucket (leaf node) contains fewer objects than a specified maximum capacity. During update operations, when a bucket surpasses this capacity threshold, a split operation is triggered. Quad-trees demonstrate strong performance in nearness queries and KNN join scenarios [TYA+16]. However, they do face challenges with regard to balancing since they rigidly divide the space into four quadrants. Furthermore, they tend to store numerous empty nodes, and at times, they exhibit long mono-child chains that could be shortened. Notably, Quad-trees possess a low fan-out (equal to 4 in 2D space), making them less suitable for secondary storage.

Many contributions have been made in the literature to overcome this limitation and adapt this structure for secondary storage. One such effort is the External Balanced Regular (x-BR) Trees [RVC11]. XBR-trees similarly divide space using the same process adopted by quad trees, but each internal node has a dynamic number of children (a multi-way tree). To achieve this, each node stores pointers for each child and an address to describe the zone covered by it. The size of the address is also dynamic and limited only by the page size of the given node.

Further improvements of XBR-trees were proposed within the framework of XBR$^+$trees [RVL+15], aimed at mitigating the dynamic size of children addresses in an internal node. Another effort, known as PBL+, introduced an efficient algorithm for bulk loading an xbr+ tree [RVCM18].

R-trees

R-trees are another tree-based access method that relies on objects grouping through a bottom-up approach, as opposed to the previously mentioned structures that use a top-down approach based on the subdivision of space [Gut84].

In the case of R-trees, objects grouping is based on constructing Minimum Bounding Rectangles (MBRs), also known as minimum bounding boxes, from the data objects.

Each MBR is designed to satisfy either a maximum space criterion or a maximum filling criterion in terms of the number of objects.

To achieve a tree-like structure, we recursively group MBRs inside each other to construct higher levels, continuing this process until we obtain a single root for the tree.

MBRs can overlap with each other, but a single object is inserted into only one of them. Consequently, R-trees have a lighter memory footprint compared to the previously mentioned indices. However, to answer a query, multiple MBRs need to be checked. The intersection of MBRs significantly impacts the performance of operations on R-trees. Finding the optimal clustering of MBRs is a NP-Hard problem [Gut84]. Different implementations address this issue in the splitting procedure. In addition to the exhaustive algorithm, the original R-tree paper proposes two splitting strategies: Quadratic split and Linear split [Gut84].

To execute the quadratic split, the algorithm selects two objects to be the first inserted in two separate nodes. These chosen objects are the two that would result in the maximum wasted area if inserted in the same node. For each of the remaining objects, the algorithm identifies the object that leads to the minimum increase in area after insertion. This process is repeated until all objects are inserted.

The linear split is much simpler; it involves selecting the two objects that are most widely separated to be placed in separate nodes. Then, each of the remaining objects is inserted into the appropriate node based on the increase in area, without taking into consideration the other objects that need to be inserted.

Further work have improved upon the original R-trees. For instance, in [KCK01], the authors propose the CR-tree, which is a cache-conscious version of the R-tree. The rationale behind this approach involves compressing the MBR keys of children to achieve a higher fan-out within a fixed page size.

Other variants, such as the R*-trees, adopt a space partitioning approach by preventing partitions from intersecting at a given level. Objects that intersect multiple partitions are assigned to each of them [BKSS90].

2.3.4 TOUCH

Touch is a novel spatial join algorithm that relies on a hierarchical tree-based index, similar to an R-tree. However, it distinguishes itself from a regular R-tree by including

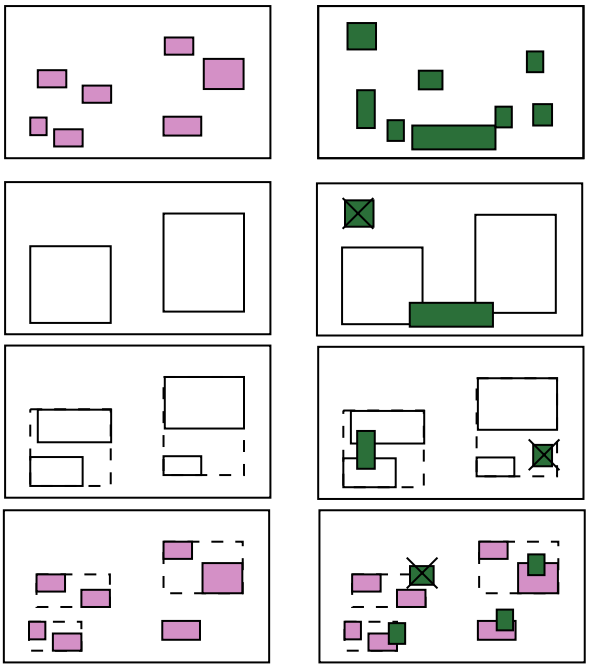


Figure 2.3: Tree building and objects assignment in TOUCH.

data entries in both intermediary nodes and leaves. Additionally, the index incorporates entries from both datasets involved in the join. The join operation performed by TOUCH is executed in three steps [NTH+13].

1. **Tree building.** An R-tree is constructed based on one of the datasets. The original implementation uses STR for efficient bulk loading. At this step, all the data entries reside in the leaf nodes.

2. **Objects assigning.** In this step, we assign objects from the other dataset to the existing tree. Notice that this is not an update operation since the structure of the index does not change. The objects are inserted into the intermediary nodes while adhering to the following rules:

   - If the object does not intersect any MBR in a given level, it gets filtered.
   - If the object intersects with one and only one MBR in a given level, we distinguish between two cases. In the case where this MBR has only leaf children, we insert the object to the node with the associated MBR. In the other case (i.e., MBR with intermediate children), the algorithm checks its overlap with children MBRs and decides recursively where insert the object.

- If the object intersects multiple MBR in a given level, the object gets inserted in the lowest common parent node.

Both Tree building and assignment steps are represented in figure 2.3.

3. **Join processing.** In this step, each object in the intermediate node is compared with its children using Plane Sweep as a local join algorithm. A proof of the correctness and completeness of the approach is provided in [NTH+13].

Touch is memory-efficient as it utilizes a single index structure for both datasets. Furthermore, it mitigates data-object duplication by employing object-based partitioning rather than space-based partitioning. Additionally, Touch prevents query object duplication through its hierarchical structure, which involves assigning objects to intermediary nodes.

The efficiency of TOUCH is primarily influenced by the fan-out parameter, which should be low (preferably 2). A low fan-out parameter leads to a high tree structure. As each intermediary node is connected to both its direct and indirect children, a high tree can effectively reduce the search space. While a low fan-out parameter is suitable for in-memory processing, it becomes less ideal for on-disk processing due to the increased loading of numerous disk pages.

Another limitation of TOUCH is its sensitivity to the order of joins, as demonstrated in [NQJ17]. The study shows that TOUCH performs better when the index is constructed using the larger dataset to maximize the filtering effect. In cases where data statistics are unavailable, it is advisable to use the smaller dataset to build the index.

2.3.5 Partition based spatial-merge join (PBSM)

This approach was introduced by Patel et al. in [PD96] as a comprehensive solution that covers both the filtering and refinement steps. In the initial filtering step, only Minimum Bounding Rectangles (MBRs) are taken into consideration. The filtering step is preceded by a partitioning process, which operates as follows:

- Since the algorithm is implemented within the context of a Spatial Database Management System (SDBMS), it estimates the universe of the input from a catalog.

- Given the two input sets R and S, along with their cardinalities $||R||$ and $||S||$ respectively, the number of partitions (P) is calculated using the following function:

$$P = \lceil \frac{(||R|| + ||S||) \cdot E}{M} \rceil$$

  Here, E represents the size of an entry (object ID + MBR), and M is the size of the main memory.

- To perform the partitioning, the universe is initially divided into T tiles, where $T \gg P$. Subsequently, each tile is assigned to a partition using either a round-robin approach or a hash function based on the tile number.

- Every object is assigned to the partitions of the tiles it intersects with. If an object intersects with tiles from different partitions, it will be duplicated across those partitions.

In the filtering step, a Plane Sweep algorithm is executed on each partition. In the subsequent refinement step, the complete shapes of the objects to be evaluated are retrieved from disk. To mitigate random seeks, the objects are sorted before being fetched. Drawing inspiration from [Val87], a duplicate elimination process is integrated into the sorting algorithm.

This approach is designed as an external algorithm, and its partitioning nature makes it suitable for implementation in a parallel environment. However, applying PBSM to real-world data presents several challenges. Finding the optimal tile size is crucial: a large tile size might result in imbalanced partitions for skewed datasets, while an excessively small tile size could introduce more duplication overhead, thereby degrading algorithm performance.

2.3.6 Dual index traversal (DIT)

Dual index traversal was originally introduced as a dual R*-tree traversal [BKS93], although it can be adapted to support various tree indices. For the sake of simplicity, we will elucidate the underlying concept of this algorithm using the case of a dual R-tree traversal.

The first property utilized by this approach is based on the observation that if two intermediary nodes from the two indexes do not intersect, the objects contained within them do not intersect either. The algorithm can iteratively examine the intersection of intermediary nodes, eventually reaching the intersecting data stored at the leaf nodes.

Further improvement can be made to reduce the search space using the following property. Given two data nodes E_s and E_r and there respective Mbrs M_s and M_r:

$$(o_r, o_s) \in (E_s \bowtie E_r) \implies o_r \cap (M_s \cap M_r) \wedge o_s \cap (M_s \cap M_r)$$

Using this property, we can limit the search to only the intersection space between the two MBRs.

A well-implemented DIT algorithm enables a sorted intersection test, which reduces the number of intersection checks and takes advantage of data locality in the buffer. Furthermore, the local order facilitates the utilization of a Plane Sweep technique as a local join algorithm, enhancing the pruning capabilities even further.

Enhanced performance can be achieved by tailoring the buffer manager strategy to align with this approach. The original authors suggest pinning frequently accessed pages in the buffer. This is determined by a degree score assigned to each page during the pinning phase. At a given point in execution, a page's score corresponds to the count of its intersections with pages from the other dataset that have not been processed yet. When page replacement is necessary, the pages with the lowest degree scores are selected for eviction. However, a potential drawback of this approach arises from the concurrent use of two indices, which can result in increased space overhead. Moreover, for non-selective queries, this approach might incur high disk costs.

2.3.7 Discussion

Based on the previous analysis of existing techniques in the literature, it is evident that none of the proposed approaches can be considered a comprehensive solution. Each approach has its own set of advantages and trade-offs. The various contributions have been summarized in Table 2.4, and we have classified them according to the taxonomy outlined in Section 2.2.

Observing Table 2.4, it becomes evident that there is a notable absence of approaches

Table 2.4: classification of join techniques.

	In memory approaches	On disk approaches	
0 Index	NL [ME92], PS [HNS88], SSPS, RRPS [RVCM14].	FCCPS, FRRPS, FRRPS [RCVM16].	Local
		PBSM [PD96].	Parallel
1 Index	Touch [NTH+13].	Nes Grid [ŠŠC+09], Nes CR-tree [KCK01], Nes Kd-tree [Rob81], Nes Q-tree [FB74], Nes Xbr+ tree [RVL+15].	Local
			Parallel
2 Indices		Dual R-tree [BKS93].	Local
			Parallel

that are inherently designed for parallel execution. A majority of the existing approaches that enable large-scale processing of distance-based joins rely on Big Data frameworks such as Hadoop and Spark. However, within these frameworks, users often lack control over data locality. This disparity is the reason we dedicate a distinct section to the processing of distance-based joins for large-scale datasets.

2.4 Spatial data processing at scale

So far this chapter has focused on spatial processing within centralized environments. We presented several techniques and we proposed a taxonomy of them. The subsequent section will delve into join processing within a distributed environment. The reviewed work is outlined in Table 2.5, accompanied by pertinent details about the validation approach and datasets employed for validation in each case. Our focus centers on the processing of distance-based joins utilizing the MPP (Massively Parallel Processing) frameworks, namely Hadoop and Spark, which are renowned as the most fitting tools for scalable data processing.

2.4.1 Distance-based join in Hadoop

Hadoop, a free and open-source framework, is meticulously designed to facilitate highly parallel data processing by leveraging a cluster of machines within a shared-nothing architecture. It operationalizes the Map-Reduce parallel programming paradigm [DG08] as its core processing model. Since its inception, Hadoop has swiftly emerged as the preferred

Table 2.5: Summery of main contributions in distance-based join processing in popular MPP frameworks (Hadoop and spark)

Author	Approach name	Compared with	Evaluation metrics	Evaluation dataset
Shubin Zhang *et al.*[ZHL+09]	SJMR	PPBSM	Execution time	TIGER/Line
Chi Zhang *et al.*[ZLJ12]	H-BRJ	H-BNLJ	Execution time	OpenStreetMap, Synthetic Data
Wei Lu *et al.*[LSCO12]	PGBJ	H-BRJ	Partition size, Execution time, Selectivity, Shuffling cost	OpenStreetMap Forest cover Expanded forest cover
Ablimit Aji *et al.*[AWV+13]	Hadoop-GIS	Comertial SDBMS	Execution time	OpenStreetMap Pathology imaging
Ahmed Eldawy *et al.*[EM15]	SpatialHadoop	Hadoop	Execution time, Throughput(Jobs/Minute)	TIGER, OpenStreetMap, NASA, Synthetic data
Simin You *et al.*[YZG15]	SpatialSpark , ISP-Mc	Proposed approaches, mutually	Execution time	NYC taxi trip data, NYC street network, GBIF species occurrence
Jia Yu *et al.*[YWS15]	GeoSpark	SpatialHadoop	Execution time	TIGER
F García-García *et al.*(2016) [GGCI+16]	KCPQ in S-Hadoop	Compare the proposed, approaches mutually	Execution time, Distance computations	OpenStreetMap

Table 2.6: Summery of main contributions in distance-based join processing in popular MPP frameworks (Hadoop and spark), part 2.

Author	Approach name	Compared with	Evaluation metrics	Evaluation dataset
Dong Xie *et al.*[XLY+16]	SIMBA	GeoSpark, SpatialSpark, SpatialHadoop, Hadoop GIS, Geomesa, comercial SDBMS	Execution time, hroughput(jobs/min)	OpenStreetMap, GDELT, Synthetic data
Mingjie Tang *et al.*[TYM+16]	LocationSpark	PGBJ, GeoSpark, SpatialSpark, Simba	Execution time, Nbr of shuffled records	OpenStreetMap, Collected Tweets
F García-García *et al.*(2018) [GGCI+18]	Improvement on S-Hadoop	KCPQ and ϵDJ in S-Hadoop	Nbr of considered cells, Execution time	OpenStreetMap

choice for extensive data processing. With the burgeoning requirement for scaling spatial data operations, Hadoop has been identified as a pivotal solution for executing spatial tasks.

Within this section, we showcase endeavors aimed at implementing distance-based join within the Hadoop framework. Moreover, we delve into various initiatives that strive to enhance Hadoop's core functionality to provide more robust support for spatial operations and spatial data types.

Spatial processing in Hadoop

We begin by introducing the SJMR (Spatial Join with MapReduce) technique [ZHL+09]. This approach draws inspiration from PBSM and employs a comparable tile-based partitioning method. The spatial join is executed through a single map/reduce job.

Regarding the map phase, a space partitioning function (SPF) is executed to divide the universe into N_t uniform tiles. Notably, the number of tiles N_t is significantly higher than the desired number of partitions P. it is important to note that P also signifies the number of reducers, as each reducer is responsible for performing a local join on a specific partition. To uniquely identify each tile, an identifier is generated using a coding

technique, such as Z-curve or Hilbert-curve.

The algorithm then iterates through the tiles and assigns each one to a specific partition. The spatial partitioning function is defined by three parameters: the tile number, the tile coding method (either Z-curve or Hilbert-curve), and the tile-to-partition mapping scheme (Round Robin or hashing). Based on experiments conducted by the designers of SJMR, the optimal choice appears to be using the Z-curve as the tile coding technique and Round Robin as the mapping scheme. Similar to the partitioning algorithm utilized in PBSM, this approach handles objects that overlap with multiple tiles through a duplication avoidance technique.

On the reduce side, each reducer carries out the join over one partition following a two-step strategy: filtering and refinement. Filtering is accomplished using a variant of the Plane Sweep technique referred to as "strip Plane Sweep," as coined by the authors. The concept underlying strip Plane Sweep involves dividing a single partition into a number of strips that are parallel to the sweeping axis. The conventional Plane Sweep technique is then executed for each strip, enabling the pruning of many objects along the axis that isn't being swept. This approach significantly enhances the performance of the Plane Sweep, with the performance improving as the number of strips increases, up to a certain point. Through experimentation, the authors determined that utilizing 8 strips yielded the best results for their validation data. Similar to the partitioning phase, objects that are associated with multiple strips are managed using the same duplication avoidance technique.

In the refinement step, the actual geometric shapes of the objects are taken into consideration. To avoid random seeking, the objects are sorted based on their IDs, and the chosen ID order closely follows the storage order.

When it comes to handling duplicates, SJMR prioritizes duplicate avoidance over duplicate elimination, which is implemented at two levels: inter-partitions and inter-stripes. To achieve this, the authors introduce the "duplication avoidance technology." The underlying idea of this technology is that if two intersecting objects span multiple tiles (or stripes), they are reported only by the partition that contains the smallest tile common to both objects.

SJMR marked a significant advancement in the realm of efficient parallel processing for spatial join. Following its lead, subsequent research endeavors aimed to implement various

types of distance-based join queries within the Hadoop framework. In the subsequent section, we delve into a selection of studies that pertain to the implementation of parallel KNN-join algorithms using Hadoop.

Knn join processing in Hadoop

KNN-join presents a relatively higher processing complexity compared to ϵ-DJ or $KCPJ$.

Designing a parallel KNN join algorithm is more challenging than designing parallel ϵ-DJ or $KCPJ$ algorithms. Traditionally, processing KNN join within Hadoop involves partitioning the larger dataset across reducers and sending the entirety of the smaller dataset to all reducers. However, this approach is computationally expensive and cannot be effectively used for datasets of equal size, as the data volume can exceed reducer memory limits. A more optimized strategy, proposed in [ZLJ12], is called H-BRJ. This strategy involves dividing both datasets into $\sqrt{N}$ partitions, where N represents the number of reducers. Each reducer performs a local join on a combination of R_i and S_i. While H-BRJ addresses the memory overflow issue, it still faces performance challenges due to high shuffle costs.

Further improvements are proposed in [LSCO12] to minimize the shuffle cost. The proposed solution is based on the Voronoi diagram and is executed in the following three steps:

1. **Pivot Point Selection**. In this step, a set of points is selected as pivots for the Voronoi diagram. The selection is done by sampling and applying some pivot selection strategy on the sample. The proposed strategies are the following: random selection, furthest selection, and k-means selection. From experiments done by the authors, farthest selection and k-means selection are harder to perform yet they don't provide better partitioning than the standard random selection.

2. **First partitioning**. This task is executed as a map job. It allows partitioning both datasets based on the distance of each object to the selected pivots. In addition to partitioning, the map function outputs a summary table for each dataset to derive a distance boundary which will serve to re-partition the second dataset.

3. **Re-partitioning of the second dataset**. This task is executed as a map job. Its purpose is to re-partition the second dataset S so that each partition S_i contains

the k nearest neighbors (k-NN) of each object in partition R_i. This is achieved by computing a distance threshold to determine the adherence of each object O_s to a specific partition S_i.

4. **Local join**. This step is executed as a reduce task. Each reducer performs the k-NN join on their local sets. No further processing is required, as the completeness of the result is ensured by the previously calculated distance boundary.

Despite the enhancements in distance-based join processing presented in the previous work, performing spatial operations in Hadoop remains inefficient and challenging due to the lack of support for spatial data types. The following section will delve into work that aim to enhance Hadoop's spatial awareness, with a specific focus on Hadoop-GIS and Spatial-Hadoop. Throughout the discussion, we will emphasize the join aspect of each proposal.

Hadoop-GIS

[AWV+13, ASV+13] Hadoop-GIS is a data warehousing system designed for executing large-scale spatial queries on Hadoop. It is developed as a package for HBase, a distributed and non-relational database solution within the Hadoop ecosystem.

Hadoop-GIS is structured into three layers on the Hadoop system. In the language layer, it offers an implementation of the ISO SQL/MM Spatial standard [Sto03]. In the query translation layer, Hadoop-GIS supplies a parser and query optimizer. Lastly, in the engine layer, the authors introduce RESQUE, a Real-time Spatial Query Engine capable of constructing and querying spatial indexes.

Hadoop-GIS partitions data using a uniform grid, with the option for further recursive cell splitting. To address data skew, Hadoop-GIS splits high-density cells along the best axis. Each partition is identified by a UID and an MBR (Minimum Bounding Rectangle). Objects that span multiple cells are accommodated through duplication. The resultant tiles are consolidated and stored in HDFS as a single large file.

When it comes to handling spatial joins, Hadoop-GIS employs an algorithm similar to the standard relational join. However, it invokes the RESQUE engine when executing spatial operations. The join process is outlined as follows:

1. A map function executes the "WHERE" clause of the query to eliminate unnecessary

objects and then outputs the remaining objects using the UID as the key. If the query is a self-join, only one scan will be performed to generate multiple Key/Value pairs.

2. Hadoop will handle the entire shuffle phase, grouping objects with the same UID together.

3. In the reduce function, two temporary files are initialized to hold records from the datasets to be joined. Then, the RESQUE engine is invoked to execute the join.

4. The RESQUE engine builds an R*-tree from the temporary files with a page utilization ratio of 100% (since the indices will not be re-used or updated). Then the engine performs a Dual tree traversal algorithm to join the datasets.

While Hadoop-GIS provides better support for spatial data, it still suffers from serious drawbacks:

1. Since it uses Hadoop as a black box, it inherits the same bottlenecks and performance issues as other Hadoop-based approaches.

2. It supports only a uniform grid as a global partitioning and indexing strategy.

3. It does not allow the re-use of the constructed indices.

4. It does not support other distance-based join operations (e.g., kNNJ and KCPQ).

5. It relies on additional software components other than Hadoop (i.e., Hbase).

In the next section, we present Spatial-Hadoop that allows to mitigate some limitations imposed by Hadoop-GIS.

Spatial-Hadoop [EM15, EM13]

Spatial-Hadoop offers spatial capabilities within Hadoop itself, instead of adding a layer on top of it. Spatial-Hadoop modifies the core components of Hadoop at four different levels:

- **Language Level:** Spatial-Hadoop enables the use of various spatial data types (e.g., lines and polygons) and spatial operations (e.g., overlap) within the Hadoop environment.

- **Storage Level:** Originally designed to support three types of spatial indices (Grid, R-trees, and R+-trees), Spatial-Hadoop has later incorporated additional index support, including Quadtree, k-d tree, Z-Curve, and Hilbert-Curve.
- **Map/Reduce Level:** Spatial-Hadoop introduces a spatial files splitter and spatial record reader to take advantage of the spatial indices.
- **Operations Level:** Spatial-Hadoop extends Hadoop with a variety of spatial operations, such as range queries, k-nearest neighbor queries, and spatial joins.

The creators of Spatial-Hadoop implemented an indexing logic within HDFS to leverage existing spatial index structures. They introduced a two-level index system: global and local.

- **Local Index:** The local index is designed to fit within a single HDFS block, which has a default size of 64 MB (Now the default size is 256). This design choice ensures that the default HDFS load balancer can handle the index as a single unit while distributing blocks across the cluster.
- **Global Index:** The global index is stored in the master node's memory. This in-memory index provides a comprehensive overview of the spatial data distribution across the entire dataset.

Spatial-Hadoop proposes a distributed spatial join algorithm, which consists of four steps. While two of these steps are mandatory, the other two are executed only if necessary:

1. **Pre-processing Step:** In this initial stage, the smaller dataset is partitioned to align with the partitions of the larger dataset. The decision to perform pre-processing involves estimating the costs of the join with and without re-partitioning. This step aims to optimize the subsequent join operation.
2. **Global Join:** Leveraging the global index, overlapping partitions are paired together. Meanwhile, partition pruning is conducted for non-overlapping partitions, as they will not contain any overlapping objects.
3. **Local Join:** Utilizing the local index, Spatial-Hadoop efficiently processes the local join using a dual index traversal technique.

4. **Duplicate Elimination:** This step is optional and is executed only when the employed indices allow for duplicates. This is the case with the Grid index and R+-tree, for example.

Francisco García-García *et al.*[GGCI+16] propose further improvement by adding KCPQ join to Spatial-Hadoop.

The implementation of KCPQ follows the same principles as the distributed spatial join initially proposed by Spatial-Hadoop. It involves executing a Plane Sweep algorithm at the inter-block level, followed by another Plane Sweep at the intra-block level.

In the same work, an improvement has also been proposed to eliminate certain partitions and objects. A pre-processing step is triggered to derive an upper bound distance δ using a data sample. This distance is subsequently utilized in both the global and local joins to prune combinations of partitions or objects that are not promising.

Finally, another improvement is presented in [GGCI+18], demonstrating that a more accurate estimation of the δ value through local sampling can lead to significant performance improvements.

2.4.2 Spatial processing in Spark

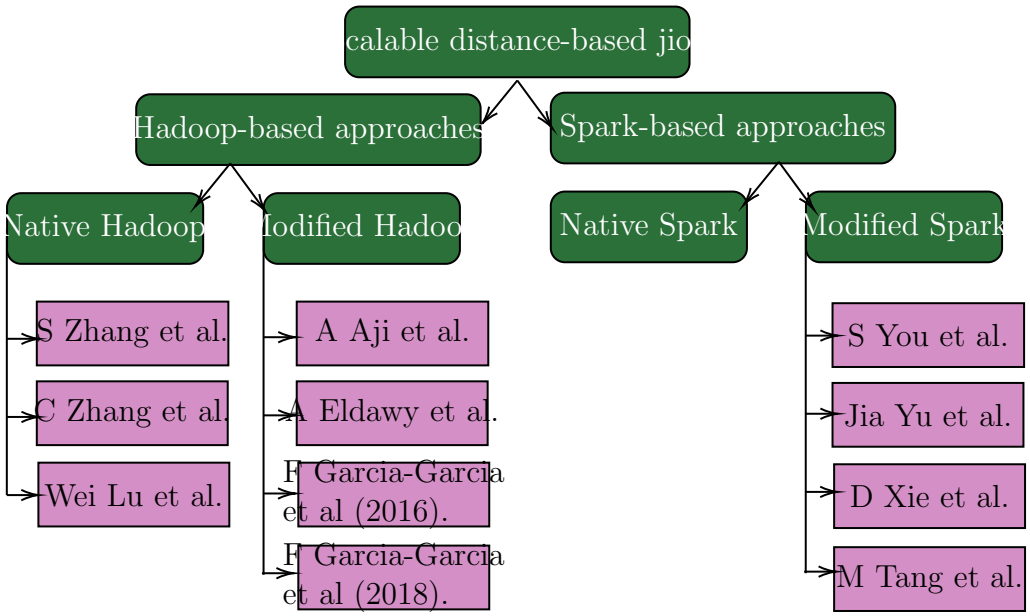


Figure 2.4: Classification of scalable distance-based join techniques.

Spark is a free and open-source framework for in-memory parallel data processing over commodity shared-nothing clusters [ZCF+10]. It introduces the concept of RDDs

(Resilient Distributed Datasets), which are an abstraction of a collection of objects partitioned across several machines [ZCD+12]. Spark can outperform Hadoop by up to 10x in iterative operations that reuse the same set of data across multiple parallel operations, while also providing similar fault tolerance and load balancing features.

Several Spark-based spatial data processing frameworks have emerged over the years, such as SIMBA [XLY+16], LocationSpark [TYM+16], GeoSpark [YWS15], SpatialSpark [YZG15], and GeoMesa [HAE+15]. To keep this chapter concise, we focus on two important frameworks: LocationSpark and SIMBA. These two frameworks were chosen due to their efficient processing of distance-based joins and their representation of the techniques commonly used by other frameworks for distance-based join processing.

LocationSpark [TYM+16]

LocationSpark is a parallel spatial data processing framework built on top of Spark. It extends Spark by incorporating spatial indices, query scheduling, and a query executor. Moreover, the designers of LocationSpark have incorporated innovative techniques to improve performance, such as a spatial Bloom filter and adaptive data caching. This caching strategy enables less frequently accessed objects to be stored on disk, reducing memory pressure.

LocationSpark is capable of performing both spatial join and kNN join. In the following sections, we will delve into how LocationSpark manages these types of operations.

To execute spatial queries, LocationSpark employs a global execution plan that orchestrates the data (re)partitioning and local plan execution. Each worker node selects a local plan based on the available spatial indices. The chosen local plan corresponds to the specific local join algorithm utilized.

LocationSpark offers two algorithms for spatial join: indexed nested-loop and dual index traversal. The indexed nested-loop algorithm employs one of the available indices locally. LocationSpark supports three types of indices: R-trees, Grids, and Quad-trees. On the other hand, the dual index traversal algorithm conducts a parallel depth-first search over the two indices, as explained in section 2.3.6. Typically, LocationSpark's planner prioritizes nested Quad-trees for point data and dual R-tree traversal for more intricate geometric shapes.

In terms of the kNN join, LocationSpark incorporates two nested index loop algorithms

using R-trees and Quad-trees. The creators of LocationSpark also evaluated three existing block-based approaches: Gorder [XLOH04], PGBJ [LSCO12], and Spitfire [CCZY+15]. Ultimately, they settled on an indexed nested-loop approach based on a quadtree index due to its superior performance compared to other methods.

Simba (Spatial In-Memory Big data Analytics) [XLY+16]

Simba is an extension of the Spark SQL engine [AXL+15], designed to incorporate support for various spatial data types and spatial queries. It stands as the pioneering spatial extension for Spark that combines the capabilities of both SQL and DataFrame APIs. Simba enhances the overall performance of generic Spark operations by introducing spatial indexes over RDDs and a spatial-aware query optimizer, thereby achieving low latency and high throughput.

From an architectural perspective, Simba introduces modifications to Spark SQL without directly altering the core Spark engine. This loose coupling ensures that developers can readily adapt Simba to accommodate future Spark releases. Before delving into how distance-based join is managed in Simba, it is important to discuss the indexing and partitioning strategies that are implemented within this framework.

Simba introduces indexing capabilities through the introduction of a new RDD type known as Index RDD. The indexing is implemented across two levels: a local level within each RDD partition, and a global level within the driver program stored in the master node. At the local index level, data is embedded in an array to ensure efficient random access, followed by the construction of an R-tree on top of this array. The R-tree is stored within the same partition, and the leaf nodes of the R-tree contain the array indexes. This arrangement facilitates the creation of a local index without significantly compromising scan performance.

Regarding the data partitioning task, Simba introduces a new strategy that offers improved data locality characteristics. This new partitioner utilizes a set of random samples and employs the STR algorithm to establish the boundaries of partitions. These boundaries are subsequently extended to encompass the entire data space.

When it comes to processing ϵDJ, Simba follows a three-step process, which is outlined as follows:

1. **Data Partitioning:** This step is omitted if both datasets are already indexed. Oth-

erwise, the same STR partitioning strategy is applied. The only modification is that the partitioner ensures that two partitions (one from each dataset) can fit into the main memory simultaneously.

2. **Global Join:** Pairs of partitions are chosen where the distance between them is below the ϵ threshold.

3. **Local Join:** If no local index exists for the data in one of the two partitions, it is indexed locally. Otherwise, an indexed nested-loop algorithm is executed.

Unlike the ϵDJ query, processing the kNN join in Simba is not straightforward. Instead, a novel join algorithm called RKJSpark (R-tree kNN join in Spark) is introduced. This new approach involves partitioning a dataset S into n partitions so that for each object $r \in R$, $kNN(r, S) \subseteq S_i$. To achieve this, the following process is proposed:

1. R **Partitioning:** If not already done, the dataset R is partitioned using the same STR strategy as used in the ϵDJ. The center of each partition cr_i, and the distance from this center to the furthest object in the partition u_i are then sent to the master node.

2. S **Pre-processing:** A sample S' is selected from S, and an R-tree is built on top of it. The R-tree is then sent to the master node for the next step.

3. **Distance Bound Calculation:** In this step, the master collects the results of the two previous steps and then calculates $knn(cr_i, S')$ for each partition center. The distance δ_i is defined as the distance between cr_i and the furthest $knn(cr_i, S')$. The following maximum distance bound is derived:

$$\gamma_i = 2u_i + \delta_i \text{ for each partition } R_i$$

The proof that for any partition R_i, we have $\forall r \in R_i, knn(r, S) \subset \{s | s \in S, |cr_i, s| \leq \gamma_i\}$ is detailed in [XLY+16]. The distance bound γ_i will serve to partition S.

4. **Parallel** $kNNJ$ **Processing:** After partitioning both R and S, an indexed nested-loop algorithm based on R-tree is executed on every partition pair (R_i, S_i). In this case, S_i is used as the inner table (The R-tree is built over S_i).

2.4.3 Discussion

In this section, we propose a classification of scalable distance-based join techniques, which we illustrate in Figure 2.4. We observe in this classification the lack of approaches that use the native Spark. We explain this with the success of spatial-enhanced Hadoop frameworks over traditional map/reduce-only approaches. This has made spatial-enhanced Spark a promising research proposition.

One major downside when using an enhanced version of Spark (or even Hadoop) over the standard approach is the effort needed to support newer versions of the framework. Developers need to adapt their approaches to the updated programming interfaces, which is time-consuming.

From the previous analysis of the join process in spatial data-parallel processing frameworks, we can observe some patterns that emerge. The first common technique used by most of the approaches is to reduce the search space as early as possible in the join process. This is generally done by sampling the data to prune as many partitions as possible. The second common technique that we noticed in processing kNN join is the aim to eliminate the intermediary shuffle by better partitioning the data. This is generally achieved by deriving a maximum distance bound for each R_i partition and using it to find the objects that will be part of the S_i partition.

2.5 RDF data processing

One can summarize the approaches dedicated to RDF data processing with respect to their storage strategies for data. Four families of approaches can be distinguished (see Table 2.7 for a comparison):

1. The most intuitive way to store RDF data is by using a single big relational table that contains three columns corresponding to the subject, predicate, and object. This strategy is known as the single table strategy.

2. A second alternative storage option is the binary table. In this approach, for each property, the system stores a binary table containing the subject and the object. This approach is widely used for scalable distributed systems [CSPG20].

3. The third approach is called the ”Property table”. In this approach, subjects with

Table 2.7: Comparison of different storage strategies for RDF data, including examples of triplestores that utilize each strategy, their advantages and disadvantages.

Storage Strategy	**Triplestore Examples**	**Advantages**	**Disadvantages**
Single Table	Oracle, Sesame [BKVH01], 3-Store [HG03]	Intuitive	Large number of self joins needed for queries
Binary Table	SW-Store [AMMH09], C-store [WKB08]	Suitable for distributed systems	Loss in performance with multiple properties, many tables needed for updates
Property Table	Jena [WSK+03], DB2RDF [BDK+13], 4store [HLS+09]	Efficient for queries with star patterns	Difficulties with chain queries, storage overhead due to null values, does not allow multiple values for the same property
Native Graph Form	Trinity [ZYW+13], gStore, RDF_QDAG [KMG+21, ZMG+21]	Stores RDF data in its native form	N/A

common properties are grouped and stored in a large horizontal table. Each column in the table corresponds to a property.

4. In the fourth approach, RDF data is modeled and stored in its native graph form. Subjects and objects are considered as nodes, while properties are considered as labeled edges.

The most intuitive way to store RDF data is by using a single big relational table that contains three columns corresponding to the subject, predicate, and object. This strategy is known as the single table strategy. Examples of stores that use this strategy are Oracle, Sesame [BKVH01], and 3-Store [HG03]. The problem with this approach is the large number of self-joins that need to be executed to answer the generated SQL query. Various studies have attempted to address this issue using heavy indexing, such as RDF-3x [NW08] and Hexastore [WKB08].

A second alternative storage option is the binary table. In this approach, for each property, the system stores a binary table containing the subject and the object. This approach is widely used for scalable distributed systems [CSPG20]. However, this can result in a loss of performance when a query requires many properties, leading to numerous

join operations. Another limitation of this approach is that many tables need to be accessed in the case of an update. This makes it more suitable for analytical workloads rather than transactional ones.

The third approach is called the "Property table". In this approach, subjects with common properties are grouped and stored in a large horizontal table. Each column in the table corresponds to a property. Some examples of this design are Jena [WSK+03], DB2RDF [BDK+13], and 4store [HLS+09]. This approach is very efficient for queries with star patterns. However, it faces difficulties when processing chain queries. Additionally, the property table can have many null values, which can increase the storage overhead. This standard approach does not allow multiple values for the same property as well.

In the last approach, RDF data is modeled and stored in its native graph form. Subjects and objects are considered as nodes, while properties are considered as labeled edges. As examples of this design, one can cite Trinity [ZYW+13], gStore, and RDF_QDAG [KMG+21, ZMG+21]. As our work is based on RDF_QDAG, we provide below more details about this Triplestore.

As for the Triplestore RDF_QDAG [KMG+21], it stores RDF data in a graph form and it answers queries using graph exploration. For efficient exploration, RDF_QDAG uses a combination of data partitioning and indexing techniques. First, the graph is partitioned into many fragments called Graph Fragments ($\mathcal{GF}$ for short). Each $\mathcal{GF}$ is then indexed using a clustered B+Tree. Similar to some existing work, RDF_QDAG keeps a separate dictionary of string values. The indices store only IDs rather than the strings. For more efficiency, RDF_QDAG makes use of two different orders: SPO and OPS, to store indices.

2.6 Spatial-RDF data processing

In order to represent geographical linked data for the semantic Web, the Open Geospatial Consortium has proposed GeoSPARQL [BK12] as a standard that extends classic SPARQL. Many Triplestores have subsequently been extended to support the processing of this new standard. The spatial extension of RDF stores extensively depends on the storage model and the query evaluation engine.

For instance, Strabon [KKK12], an extension of Sesame [BKVH01], supports spatial data. It stores data in PostGIS and implements a property table approach, where each

Table 2.8: Overview of different spatial extensions of RDF Triplestores.

Extension	**Underlying system**	**RDF Storage**	**Spatial storage**
Strabon [KKK12]	Sesame [BKVH01]	Triple table in PostgreSQL	R-tree
Brodt et al. [BNM10]	RDF-3X[NW08]	Heavy indexing	R-Tree
Geo-Store [WKC12]	RDF-3X[NW08]	Heavy indexing	Grid file
Virtuoso [vir]	RDBMS	N/A	N/A
Oracle	N/A	N/A	N/A
GraphDB [Gra]	N/A	N/A	N/A

table is indexed using SO and OS indices. Spatial data are saved in a separate relational table, which is indexed using an R-tree [Gut84]. The query optimizer extension of Strabon is simple and relies on heuristics to push down spatial filters. However, since Strabon is based on an older RDF store (i.e., Sesame), it lacks many optimization techniques used in modern Triplestores.

Brodt et al. [BNM10] extended RDF-3X [NW08] to support spatial data. In this work, only the range selection operation is supported, leading to a very limited extension. Moreover, spatial filtering is also limited to either the beginning or the end of query evaluation.

Geo-Store [WKC12] is another spatial extension of RDF-3X. Geo-Store relies on a grid file to index the spatial data, using the Hilbert space-filling curve to establish a global order for each cell on the grid. Each spatial object is paired in the order of the cell it resides in. An additional triple is added to the data graph in the form of $< o, \text{hasPosition}, \text{gridPosition} >$, which adds an extra join step to query processing.

it is worth noting that many commercial systems also support spatial RDF queries, such as Oracle, Virtuoso [vir], and GraphDB [Gra]. However, details about their internal design are not easily accessible.

2.7 Conclusion

In conclusion, this chapter has provided a comprehensive overview of the current state of spatial data processing, RDF data processing, and the intersection of the two in spatial-RDF data processing. We have delved into a multitude of techniques and methodologies

employed in each domain. Starting from centralized spatial data processing techniques to scalable approaches utilizing platforms like Hadoop and Spark, we have examined diverse strategies for managing spatial data across varying scales. Furthermore, we have explored RDF data processing, encompassing storage and query processing methodologies. Lastly, the emerging field of spatial-RDF data processing has been introduced, emphasizing the fusion of spatial and RDF data. This chapter lays the groundwork for subsequent sections, providing the necessary foundation for further exploration and research in the realms of spatial and RDF data processing. The insights gleaned from this chapter will contribute significantly to the development of efficient and effective solutions for the manipulation and analysis of spatial and RDF data within the context of large-scale applications.

Part II

Spatial and graph data processing

Chapter 3

I/O Efficient R-tree utilisation

Contents

3.1 Introduction 67

3.2 Problem Definition 68

3.2.1 R-Tree Structure 69

3.2.2 Problem statement 69

3.3 FASTER approach 71

3.3.1 Principle of FASTER 71

3.3.2 Proof of correctness 71

3.4 Experimental validation 73

3.4.1 Experimental setup and data-sets 74

3.4.2 Results discussion 74

3.5 Conclusion 75

In this chapter, we delve into our first contribution, which concerns a novel strategy for exploring R-Trees. We thoroughly examine the problem at hand and present the FASTER (FASter r-Tree ExploRation) approach as a solution. We introduce the R-Tree structure and outline the problem statement, emphasizing the need to minimize disk usage and I/O operations in spatial data exploration. The FASTER approach is presented, highlighting its core principle and providing the proof of its correctness. To validate our approach, we conduct experimental validation using various datasets and discuss the results in detail. The chapter concludes with a summary of the findings, emphasizing the effectiveness of the FASTER approach in reducing I/O operations and enhancing overall performance in

spatial data processing. These insights lay the groundwork for the subsequent chapters, where we integrate FASTER into a more complex system for processing spatial-RDF data.

3.1 Introduction

The rise of the internet has triggered a transformative revolution in data management. Individuals, businesses, and devices have become prolific data producers, generating an overwhelming volume of information daily. Yet, effectively handling such immense data volumes poses a significant challenge, hindering the optimization and utilization of the diverse datasets collected. This challenge is particularly crucial for applications in domains such as Smart cities, location-based services, and the Internet of Things (IoT), where meticulous attention is needed to ensure the efficient utilization of collected data. These applications specifically require considering the spatial nature of the data to make informed decisions. Moreover, domains like medical imagery processing [WKC+11] and molecular dynamics simulation [Ver67] also have the potential to benefit from spatial data processing capabilities.

As data management has progressed from centralized processing to Massively Parallel Processing (MPP), novel approaches are essential to manage spatial data efficiently. Within this context, established and widely adopted solutions are constructed upon frameworks like Hadoop-MapReduce or Spark. Notable examples include Spatial-Hadoop [EM15] and Location-Spark [TYM+16]. These solutions augment traditional frameworks by infusing spatial awareness into different layers and components of the system.

At the processing layer, support for various spatial operators, such as range queries, nearness queries, and spatial joins, has been introduced. This enables the efficient execution of spatial operations on large-scale datasets. On the storage and access method layer, the Hadoop Distributed File System (HDFS) is enhanced with spatial-aware partitioning techniques, optimized spatial indexes, and refined data structures. These enhancements collectively enhance the storage and retrieval performance of spatial data within the distributed file system.

To ensure efficient access to spatial data, solutions like Spatial-Hadoop and Location-Spark adopt a two-level indexing structure. This structure comprises a local index within each block or machine, alongside a global index typically stored on the name node. It is important to highlight that the R-Tree structure is the prevalent choice for indexing spatial data. Moreover, the exploration methods utilized with this index have a direct influence on performance.

Nonetheless, conventional browsing methods of R-trees exhibit a notable drawback.

Retrieving objects that meet a specific query demands a considerable number of page loads. This limitation can potentially impede the overall efficiency of spatial data retrieval.

In this context, we present FASTER (FASter r-Tree ExploRation), an advanced index browsing algorithm specifically designed for R-tree family indices. FASTER is engineered to simultaneously minimize disk usage and reduce the count of I/O operations, while retaining the existing index structure without any modifications. We will offer a formal proof to establish the correctness of this algorithm. Furthermore, a series of experiments will be conducted using real-world datasets to illustrate the impact of our solution on the count of I/O operations and execution time. These experiments serve to evaluate and highlight the effectiveness of FASTER in enhancing the performance of spatial data retrieval.

The remaining part of this chapter is organized as follows:

- **Problem Definition:** We begin by outlining the specific problem we intend to address in this research.
- **Enhanced R-tree Spatial Indexing:** We introduce our enhanced R-tree spatial indexing method, highlighting its key features and advantages. Furthermore, we provide a formal proof to establish its correctness.
- **Experimental Evaluation:** To underscore the significance of our proposal, we present an extensive series of experiments conducted on real-world datasets. These experiments are meticulously described and analyzed.
- **Conclusion:** We wrap up this chapter by summarizing our findings and contributions.

By following this organization, we aim to present a clear and logical progression of our research, addressing the problem, introducing our solution, validating it through experiments, and concluding with insights and future prospects.

3.2 Problem Definition

In this section, we will revisit fundamental concepts associated with the R-Tree structure. Subsequently, we will provide an outline of the specific problem that has captured our interest.

To begin, we will introduce the symbols and notations that will be used throughout the chapter. These are summarized in Table 3.1.

Table 3.1: Symbols and their meanings.

Symbol	Meaning
$o \in E^d$	A spatial object in d-dimensional Euclidean space (E)
$S \subset E^d$	A set of spatial objects
$s_i \in S$	An object form the set S
$U \subset E^d$	The universe, defined as the set of all objects in the spatial database
$Q \subset E^d$	The set of objects that answer the query at hand
Qb	The Minimum Bounding Rectangle (MBR) that contains all objects of the set Q.
P_i	The disk page with the id i
Pb_i	The MBR that contains all objects of the page P_i
$\|a, b\|$	The distance between two objects a and b

3.2.1 R-Tree Structure

In the context of an R-tree structure, it is important to understand the roles of various elements. An R-tree consists of internal nodes and leaf nodes, each potentially containing multiple children. Leaf nodes store either entire objects or their corresponding identifiers. Within each node, the concept of a Minimum Bounding Rectangle (MBR) is pivotal. The MBR, also referred to as the Minimum Bounding Box, defines the smallest rectangle aligned with the axes (applicable to both X and Y dimensions and extendable to higher dimensions) that encloses all objects encompassed by the node. The coordinates of the MBR's upper-left and lower-right corners are essential in determining the key for locating a specific node within its parent node. For a visual representation of the R-tree structure, please refer to Figure 3.1.

3.2.2 Problem statement

To illustrate the Standard R-Tree exploration (SRT) algorithm, we will walk through a practical example. In this case, we will use a basic window query to demonstrate the execution process. The goal of this example is to retrieve all objects that intersect with a specified rectangle.

In Figure 3.1, we present the R-tree that is being utilized in this particular example. The structure comprises two levels of intermediary nodes and one level of leaf nodes. Each intermediary node directs to five children, and we have displayed only the relevant

children for the query at hand. The leaf nodes, on the other hand, hold five spatial objects, with only the objects that intersect the query being shown. To visually differentiate the objects and pointers that are relevant to the query, we use a green color. It is clear that the exploration process is based on Property 3.2.1, which allows us to assess only the pages where the intersection between Pb_i and Qb occurs.

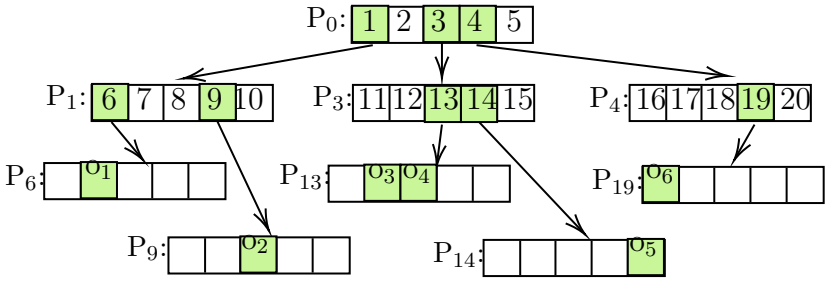


Figure 3.1: Overview of an R-tree's structure.

Property 3.2.1. Let $o \in E^d$, $S = \{s_1, s_2, .., s_n\} \subset E^d$ and Sb the MBR that bound all elements of S.

$$|o, Sb| > \epsilon \implies \forall s_i \in S, |o, s| > \epsilon \tag{3.1}$$

When processing the example from Figure 3.1, and after evaluating the leaf page number 6, we need to load the parent page (page number 1) in order to get the next leaf page. This backtrack operation will be repeated many times during the evaluation of the query. As a result, many unnecessary input operations will be performed. The loading sequence, in this case, will contain fifteen (15) Input operations that are listed as follows:

$$P_0, P_1, P_6, P_1, P_9, P_1, P_0, P_3, P_{13}, P_3, P_{14}, P_3, P_0, P_4, P_{19}$$

assuming that no buffer is associated to the R-Tree exploration algorithm. Otherwise, the number of pages loaded depends on the strategy (e.g., LRU, FIFO) adopted by the buffer, which avoids loading some pages.
One can observe that the serious drawback of STR-based approaches resides in the excessive need of I/O operations when it comes to doing a back tracking while browsing the R-Tree.

The problem we are interested in this chapter is how to reduce the number of I/O operations related to backtracking when managing R-Tree indexing in the spatial data context.

3.3 FASTER approach

In this section, we delve into the concept underlying FASTER and subsequently provide a proof of its correctness.

3.3.1 Principle of FASTER

To minimize the number of loaded pages by the R-tree, we introduce an enhancement using an additional queue. Algorithm 1 outlines this improvement, incorporating a queue named L. The management of this queue is separate and does not impact the structure of the R-tree. Each loaded page is examined to determine whether it is an internal or leaf node. For internal nodes, only pages that satisfy Property 3.2.1 are placed into the queue L (line 14). The process is then recursively repeated until the queue is empty. It is important to note that the root page of the index needs to be loaded into L prior to the first invocation of the function.

By incorporating an additional queue to store upcoming pages for loading, we effectively minimize unnecessary I/O operations. When applying the algorithm to the scenario depicted in Figure 3.1, the resulting loading sequence involves just nine (9) input operations:

$$P_0, P_1, P_6, P_9, P_3, P_{13}, P_{14}, P_4, P_{19}$$

3.3.2 Proof of correctness

The correctness of Algorithm 1 is proven by Theorem 3.3.4. The proof is based on the following propriety that we derive directly from Propriety 3.2.1:

Property 3.3.1. Let $o \in E^d$, $S = \{s_1, s_2, .., s_n\} \subset E^d$, Sb and Qb the MBRs that bound all elements of S and Q respectively.

$$Qb \cap Sb = \text{‰} \implies \forall s_i \in S, s_i \cap Qb = \text{‰} \tag{3.2}$$

Proof. From Propriety 3.2.1 we have:

$$|Qb, Sb| > \epsilon \implies \forall s_i \in S, |s_i, Qb| > \epsilon \tag{3.3}$$

Algorithm 1: Window Query Using R-tree (L,Qb,Q)

Result: Q: The set of object that satisfy the query

```
1  if L is empty then
2  |  return Q;
3  else
4  |  P ← dequeue(L);
5  |  load P into memory;
6  |  if P is a leaf page then
7  |  |  for o_i ∈ objectsOf(P) do
8  |  |  |  if ob_i ∩ Qb then
9  |  |  |  |  add o_i to Q
10 |  |  end
11 |  else
12 |  |  for p_i ∈ referencedPagesIn(P) do
13 |  |  |  if pb_i ∩ Qb then
14 |  |  |  |  add p_i to L
15 |  |  end
16 |  end
17 |  Window Query Using R-tree(L,Qb,Q)
18 end
```

for $\epsilon = 0$

$$|Qb, Sb| > 0 \implies \forall s_i \in S, |s_i, Qb| > 0 \tag{3.4}$$

$$Qb \cap Sb = \text{‰} \implies \forall s_i \in S, s_i \cap Qb = \text{‰} \tag{3.5}$$

□

Lemma 3.3.2 (Completeness)**.** The algorithm does not miss any object that satisfies the query.

Proof. In line 12, we iterate throw all keys in current page and we prune pages that do not have any results using Propriety 3.3.1. We suppose that the result set is not empty:

From (2) we have:

$$Qb \cap Pb = \text{‰} \implies \forall p_i \in P, p_i \cap Qb = \text{‰} \tag{3.6}$$

$$\implies Qb \subseteq (Ub - Pb) \tag{3.7}$$

$$\implies Qb \subseteq L^* \tag{3.8}$$

L^* denotes the space covered by all pages in L. Since we call the algorithm recursively with L in line 17, every value in L is evaluated. Thus, the algorithm does not miss any object $o \in Q$. □

Lemma 3.3.3 (Soundness)**.** The set of objects returned by the algorithm is a subset of objects that intersect with the query window.

Proof. In line 7 of Algorithm 1, all objects inserted into Q are tested for intersection with Qb. □

Theorem 3.3.4 (Correctness)**.** The algorithm finds all and only objects that intersect with the query window.

Proof. Since the algorithm is complete (Lemma 1) and sound (Lemma 2) then the algorithm is correct. □

3.4 Experimental validation

In this section, we present the outcomes of a series of experiments that we conducted to assess the performance of the novel FASTER algorithm in comparison to the STR algorithm. Our evaluation is based on two key performance metrics: the overall execution time and the count of input operations, which corresponds to the number of pages loaded into memory. We start by providing specifics about the experimental configuration and the datasets employed for validation purposes.

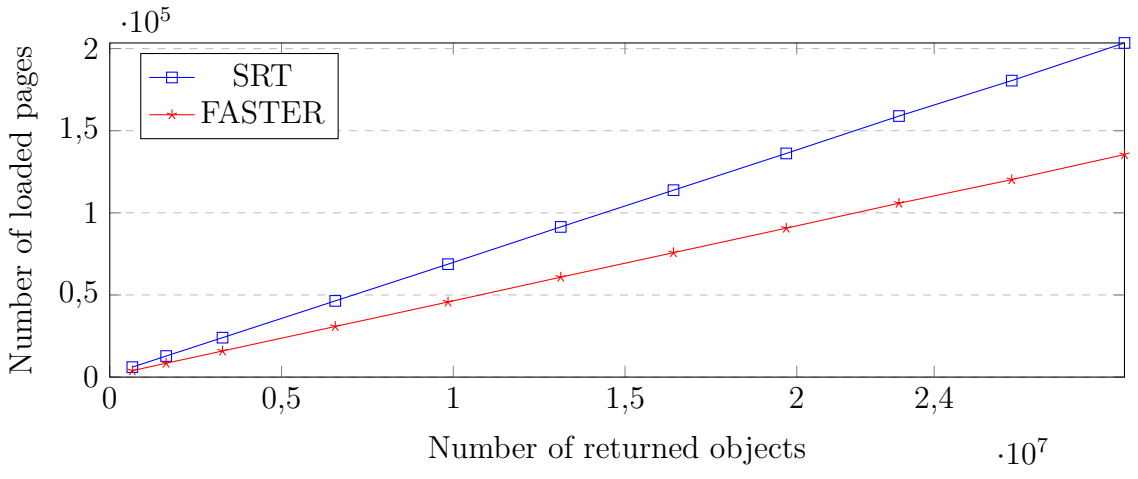


Figure 3.2: The effect of the number of the returned objects on the number of disk pages loaded.

3.4.1 Experimental setup and data-sets

The experiments have been performed on a Linux machine running Ubuntu server 18.04 LTS with kernel version x86_64 Linux 4.15.0-72-generic, equipped with an Intel(R) Xeon(R) CPU E5-2630 v3 clocked at 2.40GHz and 4 GB RAM. For a fair comparison, all experiments are performed on the same pre-loaded index. We used STR [LLE97] to bulk-load the index. Both approaches are implemented in C++ and compiled using GCC version 7.5.0.

We used the TIGER dataset [1] , which is a real-world dataset extracted from US Census Bureau TIGER files. This dataset contains 70 million polygons with a total volume of 25 GB. As for the queries we used randomly generated region queries and intersection queries.

3.4.2 Results discussion

As shown in Figure 3.2, FASTER requires fewer input operations to get the results and provides a significant improvement of 30% on average. This improvement in I/O directly translates to an improvement in the execution time (as shown in Figure 3.3). When it comes to execution time, SRT varies significantly depending on the number of backtracking operations performed and the availability of pages (managed by the OS buffer) in main memory.

For SRT, the best-case scenario is when no backtracking operations are performed or all requested backtracked pages are buffered by the OS. In this case, SRT performs as well as FASTER. However, from our experiments shown in Figure 3.3, this scenario rarely occurs (for example, in query Q_6). Otherwise, FASTER performs better, providing up to a 50% speedup (as in the case of query Q_4).

[1] `https://www.census.gov/programs-surveys/geography/technical-documentation/complete-technical-documentation/tiger-geo-line.html`

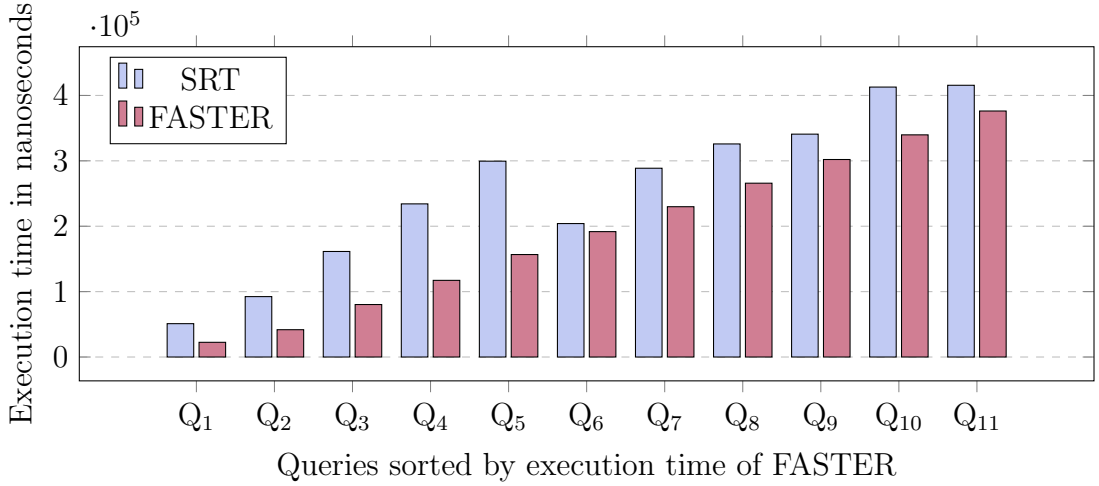


Figure 3.3: Execution time (nanoseconds) of queries using SRT and FASTER.

3.5 Conclusion

Modern spatial data processing applications are expected to efficiently process significant amounts of data using a distributed architecture. Traditional indexing structures and access methods can no longer keep up with the increasing volume and velocity of spatial data. In this work, we discussed the limitations of traditional indexing techniques. Specifically, we highlighted the unnecessary I/O operations performed by the standard R-tree exploration method.

To cope with the increasing performance demands of modern applications, we proposed FASTER, a novel R-tree exploration algorithm that minimizes the number of I/O operations required to answer a query. We demonstrated the correctness of FASTER and conducted a series of experiments to study its impact on total execution time and the number of disk pages loaded into main memory. The experiments revealed that FASTER consistently requires 30% fewer I/O operations compared to traditional exploration methods. Furthermore, it achieves up to a 50% reduction in execution time.

As a future perspective of this work, we intend to explore more complex spatial queries, such as spatial joins and k-nearest neighbor (k-n-n) joins. We believe that FASTER could be readily adapted to handle these query types as well. Additionally, we plan to integrate FASTER with spatial data partitioning techniques to minimize not only disk costs but also the network costs associated with parallel processing of spatial queries.

Chapter 4

Spatial RDF data querying

Contents

4.1 Introduction 79

4.2 Query Evaluation strategies 81

4.2.1 BGP-First strategy 83

4.2.2 Spatial-First strategy 85

4.3 Optimization techniques 87

4.3.1 Query scheduling 87

4.3.2 Spatial pruning 91

4.4 Experimental evaluation 92

4.4.1 Experimental setup and methodology 93

4.4.2 Effect of evaluation strategies 93

4.4.3 Effect of Scheduling 94

4.4.4 Effect of Encoding 96

4.4.5 Effect of spatial pruning 97

4.4.6 Comparison against Virtuoso 97

4.5 Conclusion 98

In this chapter, we delved into query evaluation strategies within the context of Spatial RDF data processing. We examined two primary strategies: BGP-First and Spatial-First, elucidating their distinct characteristics and advantages. Furthermore, we delved into optimization techniques, with a specific focus on query scheduling and spatial pruning. To empirically validate these approaches, we conducted an extensive experimental evaluation. We provided insights into our experimental setup and methodology, and presented

the resultant evaluation outcomes, encompassing the influence of evaluation strategies, scheduling, encoding, and spatial pruning. To offer a comprehensive perspective, we also compared our approach against Virtuoso, a prominent RDF data management system. In conclusion, the chapter summarized the accumulated insights, underscoring the advantages of the proposed query evaluation strategies and optimization techniques in augmenting the efficiency of Spatial RDF data processing.

4.1 Introduction

Ever since Google popularized the term "Knowledge Graph" to describe the body of knowledge employed by its search engine, the proliferation of datasets of this nature has been relentless. Knowledge Graphs (KGs) can be defined as labeled and directed multi-graphs that encapsulate information in the form of entities and relationships pertinent to a specific domain or organization. These KGs stand as potent tools for capturing and structuring substantial volumes of data that are multi-relational in nature, thereby offering the ability to explore them through query mechanisms. Given these attributes, KGs have evolved into foundational elements within the fabric of the Web and existing information systems across diverse academic domains and industrial applications. Their potency emanates from their capability to seamlessly expand existing knowledge while safeguarding the integrity of prior information.

With the surging popularity of knowledge graphs, the necessity for a standardized data representation format has become increasingly evident. This need is particularly pronounced in the context of the semantic Web, which envisions a landscape of globally accessible and interconnected data on the internet. To fulfill this requirement, the Resource Description Framework (RDF) has emerged as the primary standard for the semantic Web. Within the RDF format, data are logically depicted through a graph-based structure. The notable advantage of this representation lies in its schema-less nature, imparting flexibility and adaptability across various application domains. Furthermore, this inherent flexibility renders RDF particularly suitable for swiftly evolving data scenarios, where the constraints of normalization may not be feasible or practical due to frequent alterations in the underlying schema.

RDF data is organized using the concept of triples in the form of $< subject, predicate, object >$. In this structure, the object can either be a literal with predetermined types (such as string or double) or it can serve as the subject of another triple, thus creating a graph-like structure.

In the realm of RDF data, SPARQL has emerged as a prominent query language. Given that RDF represents data in a graph structure, a SPARQL query predominantly consists of a sub-graph wherein certain subjects, predicates, or objects are substituted with variables. This sub-graph is termed a Basic Graph Pattern (BGP). The task of answering a SPARQL query entails locating sub-graphs that match the specified query

pattern. Beyond BGP matching, filters can be applied to variables, encompassing Boolean expressions and regular expressions. The standard W3C specification for SPARQL lacks inherent support for spatial filters. Nevertheless, several extensions have been proposed to enhance SPARQL's expressiveness, enabling the inclusion of spatial filters. Notable extensions include GeoSPARQL [BK12] and stSPARQL [KK10].

The implementation of OGC GeoSPARQL [BK12] has been the focus of numerous endeavors within the community. This implementation poses a formidable challenge due to the necessity for modifications across various facets of the triple store, encompassing storage, indexing, evaluation engines, and optimizers. The nature of these changes is also contingent upon the type and architecture of the Triplestore. For instance, strategies that prove effective for a Triplestore built upon a single table approach (e.g., 3-Store [HG03]) may not be applicable to one built upon a property table approach (e.g., Jena [WSK$^+$03]).

Several existing Triplestores possess varying capabilities in responding to Spatial-RDF queries. Many of these are rooted in the relational model, while others are founded upon single table or fact strategies. However, a common drawback across these approaches is the prevalence of a high number of joins, leading to notable issues in terms of performance and scalability. Notably, a recent advancement in the field [KMG$^+$21] introduced the RDF_QDAG Triplestore, leveraging graph fragmentation and exploration to strike a more favorable balance between performance and scalability. Building upon this foundation, our work extends the system's capabilities to accommodate spatial data without compromising this trade-off. To the best of our knowledge, the proposed extensions represent the first instance of providing spatial-RDF data processing within a graph exploration framework.

In this chapter, we discuss the extension of RDF_QDAG in order to add the support of spatial operators and filters proposed in GeoSPARQL. The contributions presented in this chapter can be summarized follows:

- Two evaluation approaches for spatial-RDF data (Spatial-First and BGP-First) are proposed.
- An existing spatial indexing approach [LLE97] is adapted to be compatible with the graph exploration logic in RDF_QDAG triplestore. This indexing approach is used in the Spatial First strategies.
- The effect of the query evaluation strategies and the optimization techniques on the

performance of RDF_QDAG, is studied in depth.

- An optimizer capable of selecting the best available evaluation strategy based on the query and statistics about the RDF and spatial data, is developed.

- Finally, we conduct a comprehensive experimental evaluation of the proposed approaches and compare them against a widely recognized and utilized commercial Triplestore.

4.2 Query Evaluation strategies

In this section, we introduce two evaluation strategies for Geo-SPARQL queries, both implemented within RDF_QDAG. To better illustrate these strategies, we walk through the processing of an example query, Q1, on the dataset D1.

Listing 4.1: Example of spatial selection query (Q_1)

```
PREFIX gv: <http://geovocab.org/geometry#>
PREFIX ogis: <http://www.opengis.net/ont/geosparql#>

select ?g
where {
    ?o type "cultural".
    ?o gv:geometry ?p.
    ?p ogis:asWKT ?g.
    FILTER( bif:st_intersects(
        bif:st_geomfromtext( "POLYGON((7 43, 8 43,
        8 44, 7 44, 7 43))" ), ?g ) )
};
```

It is worth noticing that the existing formal framework for query plan and query evaluation do not take the filters into consideration. Previous contributions have focused on the graph matching aspect of the query evaluation. The filters were considered as an implementation detail. However, to introduce support for spatial filter, the existing formal definitions need to be extended to consider the spatial operators used in the filter clause of the query.

Table 4.1: Example of RDF triples (dataset D1).

Subject	**Predicate**	**Object**
Tennis Championship	hostedIn	Paris
Tennis Championship	type	Sports
Tennis Championship	geometry	G1
G1	asWKT	Point(2.34 48.85)
Festival of Lights	hostedIn	Lyon
Festival of Lights	type	Cultural
Festival of Lights	geometry	G2
G2	asWKT	Poit(4.846 45.75)
Film Festival	hostedIn	Cannes
Film Festival	Type	Cultural
Film Festival	geometry	G3
G3	asWKT	Point(7.012 43.55)

As we mentioned in definition 1.3.2, the Filter function FL is a truth function. We then express this function in a conjunctive normal form. We also introduce the concept of filter units as the operands of the mentioned conjunction.

Definition 4.2.1. *(Filter Unit)*

Let P be a subset of query parameters $P \in \mathcal{P}(V_p^Q)$. and qs_p the parameters of the star query qs. A filter is a truth function $FL : [\![q_g]\!]_G \rightarrow \{0, 1\}$. Filter function can be expressed as a conjunction of operands. We name each operand a filter unit Fu.

$$FL = Fu_1 \wedge Fu_2 \wedge ... \wedge Fu_n$$

Using this concept of filter units Fu, one can see that the definition of an execution plan is extended. In the previous definition, the execution plan is a sequence of star query evaluation. While this is sufficient to perform the graph matching, it is not enough to consider the filters. In the new definition of the plan, we consider two types of operators: the classical star query evaluation and the new filter unit evaluation.

Definition 4.2.2. *(Execution Plan - extended definition)* .Let $\mathcal{P}$ be a tuple $< X, f >$ where $X \subset \overrightarrow{QS} \cup \overleftarrow{QS} \cup FL$ and $f : X \rightarrow \{1...|X|\}$ is the query stars order function.

We denote by $\mathcal{P} = [QS_1, QS_2, Fu_1(p_1, p_2), ..., QS_n]$ the plan formed by executing QS_1, then QS_2, then evaluating the filter unit $Fu_1(p_1, p_2)$ which requires the mappings parameters p_1 and p_2.

As mentioned before, to ensure a graph exploration logic, not all plans are acceptable.

An execution plan is considered acceptable if, starting from the second star query, the head of the star is already instantiated. In a similar fashion, the position of the filter unit is critical. We can execute a filter unit only if the mappings for the parameters of the filter units are already available. On this principle, we extend the definition of an acceptable plan using the following condition:

Definition 4.2.3. *(Instantiated filter unit parameter)* Let consider the function Param: $FU \rightarrow V_p$ a function that returns the parameters of a Filter Unit. An acceptable plan $\mathcal{AP}$ is a tuple $< X, f >$ where $X \subset \overrightarrow{QS} \cup \overleftarrow{QS} \cup FL$ and $f : X \rightarrow \{1...|X|\}$ is the query stars order function such as $\forall i \in \{2...|X|\}, Param(f^{-1}(i)) \in \bigcup_{j=1}^{i-1} Nd(f^{-1}(j))$.

To explain better the concept of an acceptable plan, let us return to the query Q1 wich contain the following Star Queries in the BGP :$\overleftarrow{?g}$, $\overrightarrow{?p}$, $\overleftarrow{?p}$ $\overrightarrow{?o}$ and $\overleftarrow{?o}$. The filter function consists of one filter unit $Fu(?g) = ?g \neg DC"POLYGON((-100...20))"$. One can see that many execution plans can be built. However not all of them are acceptable. For example, the plan $[\overrightarrow{?o}, \overleftarrow{?g}, Fu(?g)]$ is not an acceptable plan due to the instantiated head condition, neither the plan $[\overrightarrow{?o}, Fu(?g), \overrightarrow{?p}]$ due to the instantiated filter condition.

An example of an acceptable plan of the query $Q1$ is $[\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$ or $[\overleftarrow{?g}, Fu(?g), \overleftarrow{?p}]$. To evaluate acceptable plans, two strategies are discussed: BGP-First strategy and Spatial-First strategy.

4.2.1 BGP-First strategy

This strategy consists of finding matches for the graph pattern first, before proceeding to run the filter on the results of the matching process. An example of a plan where this strategy can be considered is the following $AP_1 = [\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$.

The sequence of star queries and filter units listed in the logical execution plan does not consider implementation details. Therefore, we illustrate the full execution in figure 4.1. First, the graph matching part of the query is evaluated. Appropriate graph fragments are considered for evaluating each star query. Data in each fragment is stored in a B+tree in order to efficiently retrieve it from the disk. Once the information needed is retrieved, it is placed in a buffer, named SQ-buffer, so it can be used by the following operator in the plan.

The same logic is applied to spatial values. The true objects shapes can be significantly large depending on the geometry of the object (values describing Polygons are larger than

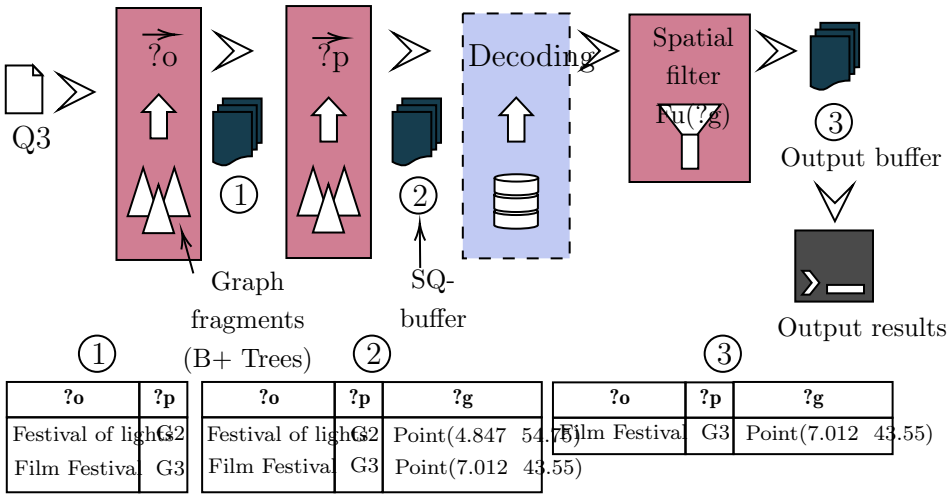


Figure 4.1: The execution of an BGP-First plan

values describing points for example) and on the resolution used to represent the object. To keep the size of the database low, and to maintain system performance, true shapes are stored in the dictionary.

Algorithm 2: Intersection Filter (L, Qb, Q)

Data: M: List of mappings;
s: Spatial object;
use_true_shape: flag to use true shape;
Result: Q: The set of mappings that intersect s

```
1  Q ← ∅;
2  for m ∈ M do
3  |  (MBR(m), m) ← decode(m);
4  |  if MBR(m)¬DCMBR(s) then
5  |  |  add m to Q;
6  |  |  continue;
7  |  if m is a point then
8  |  |  continue;
9  |  if use_true_shape = false then
10 |  |  continue ;
11 |  GEO_m ← parseGeometry(m);
12 |  if GEO_m¬DCs then
13 |  |  add m to Q;
14 end
15 return Q;
```

Once the shapes are retrieved from the dictionary, the filter function FL is evaluated. In the case of $Q1$, the filter function is composed of a single filter unit $Fu(?g)$. This latter is evaluated in two steps (filter and refine). The Algorithm 2 is an example of an intersection filter without any loss of generality to other region connection calculus operations. In the

filter step, only MBRs of the shapes are considered (line 4) to significantly reduce the search space. The refining step considers the full geometry (line 11 and 12) hence, it is computationally expensive. However, it is necessary to eliminate false positives from the previous step.

4.2.2 Spatial-First strategy

The BGP-First strategy presented above can answer spatial-RDF queries and can be easily integrated into the execution model of RDF_QDAG. However, it has some limitations that we discuss in this section. In this section, we introduce the second proposed strategy Spatial-First.

When we consider the same example query $Q1$ with the same dataset $D1$, one can observe that multiple valid plans can be run to answer the query. We can list a few of them as an example: $[\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$, $[\overleftarrow{cultural}, \overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$, $[\overleftarrow{?p}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?o}]$. All the listed plans have a common problem. Since the filter unit relies on the execution of the previous query stars, values of the geometry need to be obtained from the dictionary. As a result, it is impossible to use any spatial access method to speed up the spatial filter evaluation.

In the Spatial-First strategy, we try to take advantage of a spatial access method. To do so, we can only consider execution plans that start with the spatial filter. In the case of query $Q1$, the plan we consider is the following $[Fu(?g), \overleftarrow{?g}, \overleftarrow{?p}, \overrightarrow{?o}]$. As before, the spatial filter is run using two steps. However, this time, the filtering step can benefit from the spatial index.

The structure of the spatial index we use is an R-tree with some modifications for better integration with RDF_QDAG. The R-tree stores only object approximations in the form of MBRs with the necessary information to continue the graph exploration. This ensures the efficiency of the first step of the spatial filter by minimizing the number of pages. The page size in the index is 16 Kb. The structure of the pages is demonstrated in the figure 4.2.

For inner pages, we save 24 bytes as page header. The rest is filled with inner entries where each entry is composed of an MBR (4 X 8 bytes) as a key and pointer to the appropriate page (4 bytes). An inner page can have up to 454 entries.

As for the leaf pages, we keep two types of entries: Points and MBRs. The MBRs are

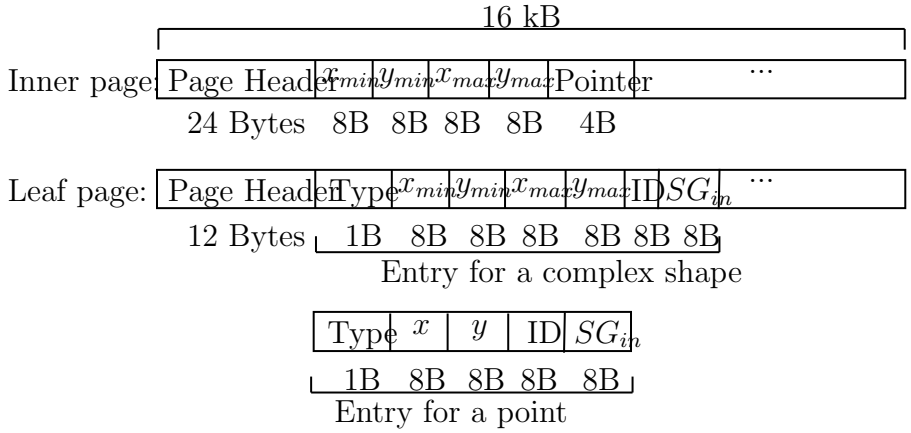


Figure 4.2: The structure of index pages and entries

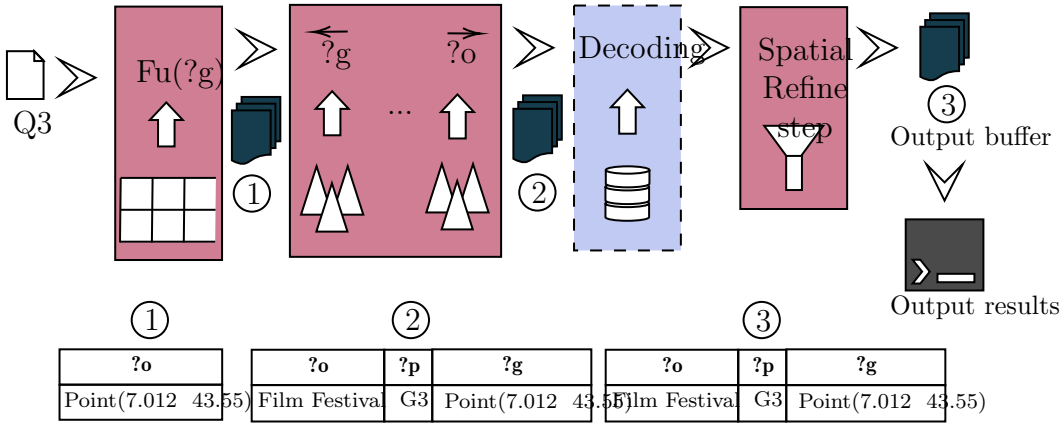


Figure 4.3: Execution of Spatial-First strategy

generally approximations of complex geometries. A point is represented by two coordinates (x, y) and an MBR is represented by four $(x_{min}, y_{min}, x_{max}, y_{max})$. On the leaf page, we save 12 bytes as page header, the rest is filled with leaf entries. For each entry, we store the object type in 1 byte, then we store the key, which is a point/MBR in 2*8/4*8 bytes, respectively, the object id in 8 bytes and the inward pointing fragment ID also in 8 bytes. The fragment ID is used to continue with the graph exploration. A leaf page can hold from 334 to 496 entries depending on the object types.

In the example shown in figure 4.3, only geometries $?g$ where $MBR(?g)\ \neg DC\ MBR(q)$ are returned after the exploration of the index. The next operator in the plan is standard graph exploration matchings.

At the end of the evaluation, the decoding operation is performed to replace object IDs with the true value. The same is applied to spatial data where MBR approximation is replaced using the true geometries. Once the full shapes are available (true geometries) the refining step can be performed in the same way as in the BGP-First strategy.

4.3 Optimization techniques

In this section, we present details about some optimization techniques that we propose to further improve execution time for both proposed strategies.

4.3.1 Query scheduling

A typical DBMS can answer the same query using different execution plans. All the plans provide the same results, however, the cost of execution for each plan is different. The same logic applies for RDF_QDAG. In the case of the latter, an execution plan is a sequence of SQ and Filter units. Since the execution time can vary significantly from a plan to another, it is important to choose the best execution plan for a given query.

A traditional approach to select the best plan is to use two steps: plan enumeration and cost estimation. In the plan enumeration step, we list all the possible execution plans. However the number of execution plans can be very significant, so enumerating all the plans is either not possible or not efficient. Many DBMS use a heuristic approach to enumerate only the most promising plans. In the cost estimation step, we estimate the cost of executing each plan to select the plan with the lowest possible cost. This is generally done using dynamic programming since many plans share some parts between each other and it is not reasonable to recalculate the cost of the same plan segment multiple times.

RDF_QDAG uses the GOFast approach for the optimization [ZMG+21]. In this approach, both the enumeration and estimation are performed in parallel. In order to do so, authors rely on a branch and bound algorithm. They start by constructing a tree where each node represents the accumulated cost of all previous operations and the edges represent plan operations. Naturally, the cost in the root is 0. The algorithm starts by estimating the cost of all possible first operations, then it expands on the operation with the lowest cost. GoFast continues on expanding the branch with the least cost until it gets a full execution plan.

Estimation in GoFast is based on the statistics collected for each graph fragment. The statistics also make it possible to reflect the interaction

The existing GOFast optimization do not takes into account the filters since the majority of the cost is caused by the graph exploration. However, this is not the case for

the spatial filters since the cost of comparing complex shapes is high. On top of that, the use of an additional access method (R-tree) must be accounted for in calculating the cost. Consequently, we extend the existing logic in order to take into account the cost of filter units. The cost of a plan is mainly the sum of cost of all star queries (both normal ans spatial ones):

$$Cost(\mathcal{P}) = \sum_{qs \in \mathcal{P}} Cost(qs) \tag{4.1}$$

The estimation of the cost of star query is already part of RDF_QDAG system, however we changed it to be calculated in terms of triples not in terms of data stars. We opted for this change since the number of data star did not show (see appendix) a correlation with the choice of the best plan in the case of spatial queries contrary to the number of triples.

To estimate the full cost of the plan, for each part of the execution plan, two estimations needs to be done: estimation of the input and estimation of the number of results. This is necessary since the estimation of the cost of part of the plan depends on the number of results of the previous parts.

Estimation of the number of spatial objects.

The estimation of the cost of a star query is already detailed in previous work [ZMG+21], we will detail only the cost of the filter unit. In the case of an BGP-First plan, no spatial access method is used. In the case of Spatial-First plan, the cost of fu is the number of spatial objects that needs to be retrieved from the index:

$$Cost(fu) = SOC(Q) \tag{4.2}$$

$SOC(Q)$ is the number of spatial objects estimated using the spatial index. We can do this by taking advantage of the shallow depth of an R-tree. Indeed, since the fan-out is high, the depth is low (generally 4 to 5 layers maximum). In the estimation phase, we scan only the top layers of the R-Tree without loading the leaf layer. Naturally, we count only pointers where the attached key satisfies the filter. To calculate the number of objects ($SOC(Q)$) we simply multiply the number of leaf pages that satisfy the query by the average number of objects in a page. This assumption is based on the fact that most of the pages are close to 100% fill rate since the index is loaded using STR [LLE97] and

no updates are performed later. The only limitation of this estimation is the fact that not all objects in the leaf pages satisfy the query.

Estimation of spatial filter results

.

The estimation of the number of results after the filter is necessary for the rest of the process. The number of objects $SOC(Q)$ can be considered as an estimation of the number of results since it is an estimation of objects where the MBR satisfies the spatial filter. However, to be able to continue calculating the cost with the GOFast approach for the rest of the plan, the total number of objects is insufficient. We need to calculate an estimation of the number of objects for each fragment.

The cost of a plan $\mathcal{P}$ is calculated in terms of the number of triples that need to be retrieved from the disk since the disk cost is the most important cost of the query. The cost of a particular plan is the sum of the cost of all star queries sq_i that compose the plan (equation 4.1). The cost of a star query is the number of triples retrieved from the relevant fragments fg_j:

$$Cost(qs) = \sum_{fg_j \in sq} Input\_Tr(fg_j, sq) \tag{4.3}$$

In the case of the first star query, no previous input is needed. As a consequence, the number of triples retrieved from a particular fragment fg_j is simply the number of triples in the fragment that satisfy the predicates of the star query:

$$Input\_Tr(fg_j, sq_1) = \#triples(fg_i, prd(sq_1)) \tag{4.4}$$

However for the rest of the star queries, the number of triples retrieved from a particular fragment fg_j is calculated using:

- $\#tripls(fg_j, pred(sq_i))$: the number of triples that satisfy the predicates of the star query sq_i
- $Input\_Ds(fg_j, sq_i)$: the number of data stars considered as in input
- $dist(fg_j)$: the number of data stars in the fragment fg_j

The formula for calculating the number of triples retrieved in case $i > 1$ is the following:

$$Input\_Tr(fg_i, sq_i) = \frac{\#triples(fg_i, pred(sqi)) * Input\_Ds(fg_)}{dist(fg_j)} \tag{4.5}$$

Detailed calculation of $Input_Ds(fg_j, sq_i)$ and $dist(fg_j)$ is found in Zouaghi et al[ZMG+21] since we did not change it. As for the number of triples retrieved from a particular fragment it is the sum of all triples in the fragment where the predicate is the same as one of the star query predicates:

$$\#triples(fg_i, prd(sq_1)) = \sum_{P_j \in Pred(sq_i)} count(pj, fg) \tag{4.6}$$

In the case of Spatial-First plan, the number of triples is identical to the number of spatial objects estimated for each fragment:

$$Input\_tr(fg_i, SQ_1) = \#releventObject(fg_j/Q) \tag{4.7}$$

The number of spatial objects estimated for each fragment is estimated based on the selectivity of the spatial query as follows:

$$\#releventObject(fgj/Q) = size\_of(fg_j) * S\_select \tag{4.8}$$

Where $size\_of(fg_j)$ is the total number of triples in the fragmentfg_j and the spatial selectivity (S_select) is calculated as follows:

$$S\_select = \frac{SOC(Q)}{total\_spatial} \tag{4.9}$$

Where *total_spatial* is the total number of spatial objects stored in the index.

On top of the estimation of the number of relevant triples to read from disk, GoFast optimizer also relies on the number of results produced by each star query $outpu\_DS_{sqi}$ defined in [ZMG+21] as follows:

$$output\_DS_{qs_i} = \{(Gf_j, p_i, k'') | p_i \in edges(qs_i) \wedge k'' = NDS_{p_i}\}$$

Where NDS_{p_i} is the number of data stars heads relevant to the predicate pi (4.10)

However, we had to change the calculation of NDS_{p_i} to take into account the spatial filters. The new formula is the following:

$$NDS_{p_i} = \begin{cases} 1 & , if\ e.node\ is\ const \\ \frac{k'}{dist(Gf_j)} * dist\_NE(p_i, Gf_j) * S\_select & , otherwise \end{cases} \tag{4.11}$$

Where $dist\_NE(p_i, Gf_j)$ is the number of distinct nodes linked to the data star head in fg_j with respect to the predicate p_i.

With both estimations of the number of spatial objects and the spatial filter results, the GoFast optimizer can choose the best execution plan for Spatial-RDF queries.

4.3.2 Spatial pruning

Earlier, we proposed two execution strategies, "BGP-First" and "Spatial-First", of which only the latter can benefit from a spatial access method. The "BGP-first" strategy lacks spatial awareness at the beginning of the process, which means that it misses opportunities to reduce the search space based on spatial constraints. To address this issue, we propose a new optimization technique called "Spatial pruning".

As discussed in section 1.4, the initial RDF graph is partitioned into graph fragments $\mathcal{GF}$ for indexing and storage. When evaluating a query, only the necessary fragments are considered based on the characteristic sets of each fragment. However, when a query contains a spatial filter, many fragments that are considered due to their characteristic sets do not contribute to the final results. This is because the spatial filter in the query eliminates all the graph patterns produced by these fragments since they are connected to spatial objects that do not satisfy the filter.

To eliminate fragments that do not contribute to the results earlier in the process, we associate each graph fragment to an MBR such as all spatial objects connected to the fragment are situated inside this MBR. When processing the query, the optimizer do not choose fragment based on the graph part only, but also based on the spatial filter. If the MBR of a fragment ($MBR(Fg)$) satisfies the filter, it can contain the results. However, if the MBR does not satisfy the filter, it is immediately pruned and not considered while evaluating the query.

The proposed algorithm 3 operates on a set of star queries specified in a query plan.

The algorithm iterates through each star query in the plan (line 3). For each star query, the relevant fragments are obtained based on the characteristic set (line 4). These fragments are then linked to the fragments of the previous star query using the function LinkToPreviousFragments() (line 5). If the current star query does not contain a spatial filter (line 6), the algorithm proceeds to the next star query (line 7). However, if the current star query contains a spatial filter (line 6), the algorithm loops through each fragment while testing the intersection of the fragment's Minimum Bounding Rectangle (MBR) with the query (line 9). If there is no intersection between the fragment's MBR and the query (line 10), the fragment and all fragments linked to it are removed from further consideration (line 11).

Algorithm 3: Spatial pruning

Data: $\mathcal{P}$: Execution Plan
$\mathcal{GF}$: Set of graph fragments
$SF : Gf_s \rightarrow MBR(Gf_s)$: List of spatial fragments MBRs
Result: $Gfs : \text{qs} \rightarrow \mathcal{GF}_{Sq} | \mathcal{GF}_{Sq} \subseteq \mathcal{GF}$:Set of fragment for each Sq

```
1  Gfs ← [];
2  QS ← getQSList(P);
3  for sq_i ∈ QS do
4  |   CurrentFGs ← getCurrentFragments(sq_i);
5  |   LinkToPreviousFragments(Gfs, CurrentGfs);
6  |   if isSpatialFilter(qs_i) = false then
7  |   |   continue;
8  |   for fg_i ∈ CurrentFGs do
9  |   |   MBR_fgi ← SF.getMBR(fg_i);
10 |   |   if MBR_fgi ¬DCs then
11 |   |   |   Gfs.removeAllFGsConnectedTo(fg_i);
12 |   end
13 end
14 return Gfs;
```

4.4 Experimental evaluation

In this section we discuss several experimental results on the various approaches and optimisation techniques mentioned in the previous section. We also compare our proposed solution with a well-known commercial Triplestore Virtuoso.

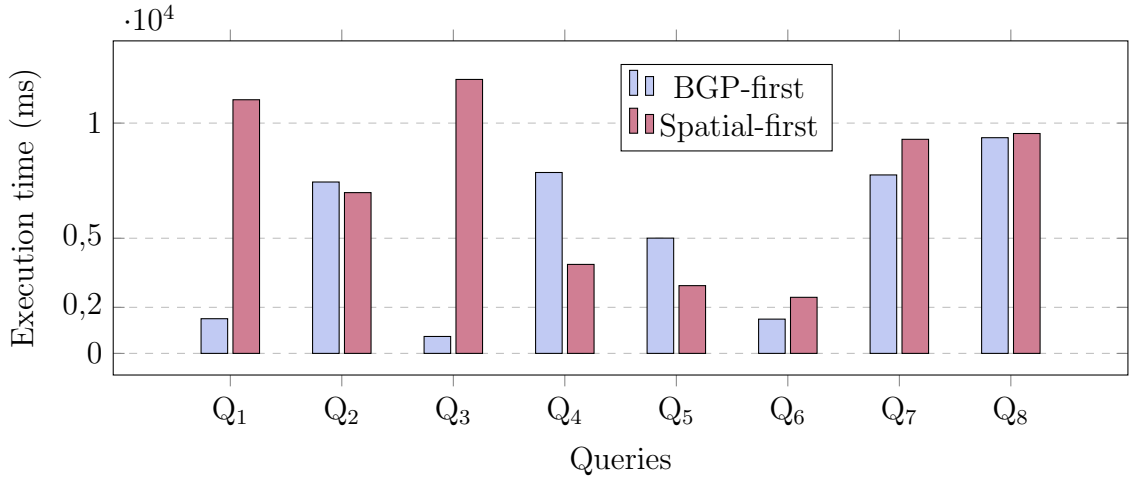


Figure 4.4: Execution time (nanoseconds) of queries using both strategies BGP-first and Spatial-first.

4.4.1 Experimental setup and methodology

We perform several experiments on RDF_QDAG after integrating the approach and techniques proposed in this chapter. RDF_QDAG is a project developed using Java and C++. The storage and access methods are developed using C++ and compiled using GCC version 7.5.0. The engine and optimizer are implemented using Java 11 and built using maven 3.8.6. For the run environment, we used Open JDK version 11.0.16.

All experiments were run on a machine equipped with Intel Xeon (Skylake, IBRS) @ 10x 2.295GHz and 64 GB of RAM and an SSD running Ubuntu 18.04 bionic with linux kernel x86_64 Linux 4.15.0-194-generic.

For the evaluation, we used the YAGO knowledge base. YAGO is a real world data-set that contains more than 234 million facts on witch 4 million are spatial objects.

All experiments are performed on a fresh install of the operating system. We clear page cache, dentry and inode cache before each query. Execution time is calculated from the submission of the query to the end of writing the results into an output file.

4.4.2 Effect of evaluation strategies

To study the effect of evaluation strategies on the execution time, we ran several queries on the YAGO data-set.

The results of the execution time for queries using the BGP-First and Spatial-first strategies are shown in figure 4.4. Neither approach consistently outperforms the other, as demonstrated by the varying performance in queries Q_4, Q_2, and Q_5, where the Spatial-

Table 4.2: Execution time of queries on YAGO

Query	Best BGP-First plan			Best Spatial-First plan		
	Plan ID	Exec time (ms)	# Triples	Plan ID	Exec Time (ms)	# Triples
Q1	1	1506	**32503**	5	11014	463841
Q2	4	7442	238414	6	6981	**144671**
Q3	4	735	**4377**	5	11896	699942
Q4	3	7855	217526	2	3864	**91326**
Q5	4	5005	205310	6	2942	**60374**
Q6	1	1488	**10639**	3	2435	38092
Q7	3	7749	217526	2	9296	**56272**
Q8	1	9366	**369054**	3	9548	438063

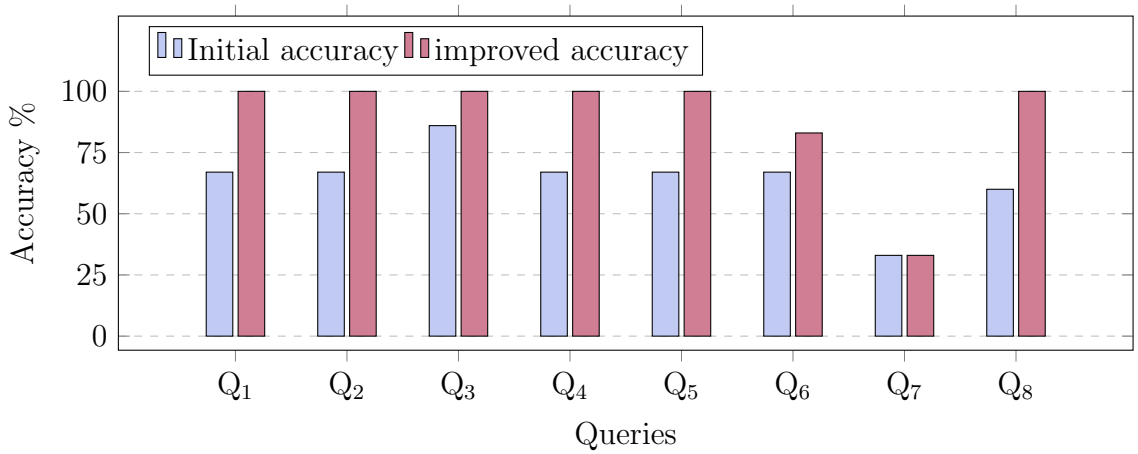


Figure 4.5: Initial accuracy and the improved accuracy of the optimizer.

first approach is superior, and the remaining queries, in which BGP-first performs better.

To further investigate the factors contributing to the varying performance of each approach, we analyzed intermediary results in both the spatial and graph parts of the queries to extract the total number of triples loaded from the disk. The total number of triples is displayed in table 4.2. The results in the table show a clear correlation between the choice of the best execution strategy and the number of triples fetched from the disk. In each query, the strategy with the lower number of triples is the best-performing one. This observation has motivated the improvements of the optimizer and the cost model proposed in section 4.3.

4.4.3 Effect of Scheduling

To select the best execution plan and execution strategy, we extended the GoFast optimizer to be able to estimate the cost and number of results of spatial filters. As far as experimental validation goes, we propose to compare the improved version of GoFast

with the existing one. For that, we use the accuracy of the best plan prediction as a performance metric. The accuracy of the optimizer for a given query is calculated as follows:

$$A = \frac{\#plans - Rank\_plan}{\#plans - 1} \tag{4.12}$$

The primary function of an optimizer is to select the optimal execution plan for a given query. To accomplish this, the optimizer assigns a rank to each candidate plan based on an estimation of its cost. The accuracy of the optimizer is measured in terms of the rank of the true best plan. Specifically, the accuracy is calculated as the proportion of the true best plan's rank among all the candidate plans. A higher rank for the true best plan corresponds to a higher accuracy, with an accuracy of 100% indicating that the optimizer has successfully identified the true best plan as the top-ranked plan. Conversely, an accuracy of 0% would indicate that the optimizer ranked the true best plan as the worst among the candidates.

The figure 4.5 shows the initial accuracy of Go-Fast and the improved accuracy. As we can notice, the optimizer after the proposed improvements provides a better prediction of the best execution plan. It can find the actual best execution plan for the all of the test queries except $Q6$ and $Q7$. Moreover, even for the latter queries, it provides the same or better accuracy than the original optimizer. This is due to a better estimation of the cost of the spatial filters.

The accuracy of both approaches is plotted in the figure 4.5. However, more detailed results are in the Appendix where we list the results of estimation of each plan compared to the true cost. We will refer to values form the detailed tables to better explain the results. The accuracy on queries $Q6$ and $Q7$ demonstrates that there is still room for improvement for the optimizer. In $Q6$, the improved optimizer chooses the second best execution plan performing better then the old approach, which choose the third best plan. This is due to the error of estimation. The best plan for $Q6$ is the plan $\mathcal{P}1$ with a real cost of 10639, followed by the plan $\mathcal{P}7$ with a real cost of 7338. The results of the estimation proposed a cost of 9894 for $P1$ and 7338 for $P7$ leading to the choice of $P7$ as the best plan.

We can notice the same problem with the query $Q7$ where the cost of $P3$ is 217526 however it is estimated to be 194145. The gap between the real cost and the estimation

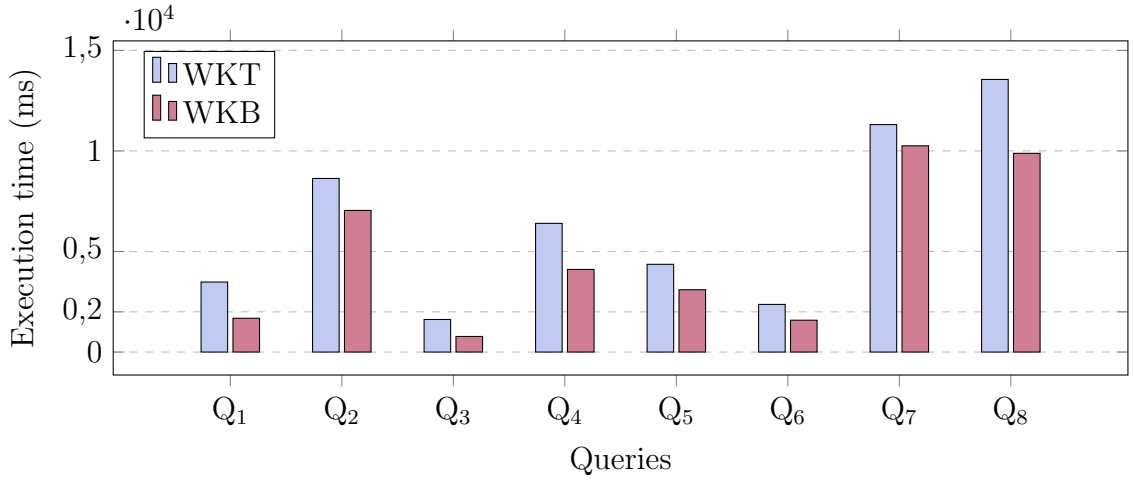


Figure 4.6: Execution time (ms) of queries using WKT and WKB.

is due to the number of objects eliminated with the refinement step in the spatial filter. In the refinement step true shapes are considered and in the case of $Q7$ many objects do not satisfy the spatial filter despite that there MBR approximations do satisfy the latter. On top of that, the number of acceptable plans is very low for $Q7$ (only four acceptable plans, meaning that each error is amplified when using the accuracy metric leading to 33% accuracy.

4.4.4 Effect of Encoding

As we mentioned in chapter 1, RDF_QDAG stores data in three types of files: spatial index, graph fragments and dictionary files. The description of a spatial object in a vector format can be long, for example the map of a state or a river. For efficiency, we store the full resolution shape definition in the dictionary. The full value will be replaced by an ID in the graph fragments and with an approximation (MBR) in the spatial index.

For the storage of the spatial object, we have mainly two options: The Well Known Text format (WKT) and the Well Known Binary format (WKB). RDF_QDAG is capable of outputting both representations, however, for the storage format, we experimented with both representations to determine the best encoding format for the system.

In Figure 4.6, we show the effect of the encoding format on the performance of the queries. We can clearly notice that the WKB encoding outperforms the WKT one for all queries. This is due to the different sizes of the two encoding formats. WKB is generally more compact than WKT, which leads to less I/O cost. On top of that, deserializing the WKB format is more efficient than parsing the WKT format. For RDF_QDAG system,

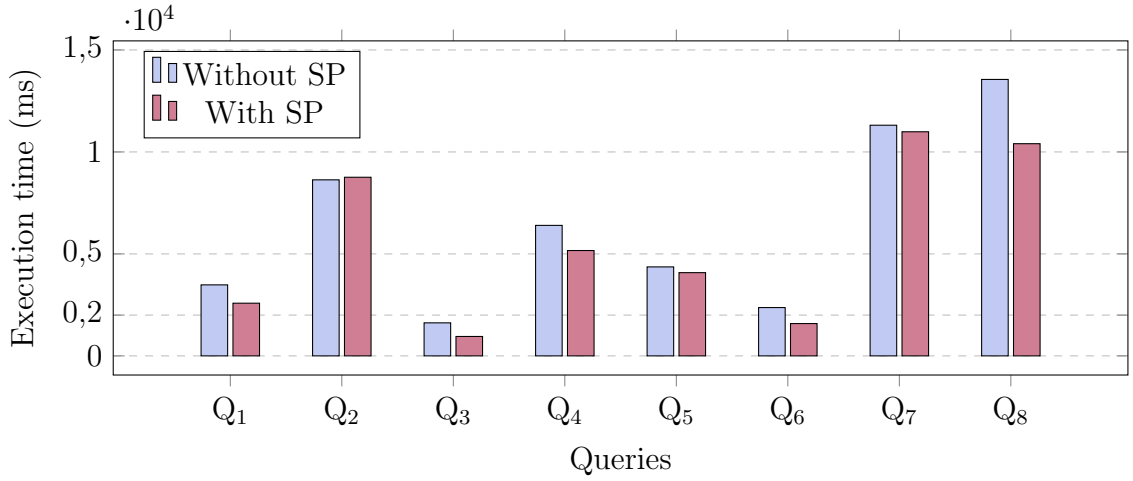


Figure 4.7: Execution time (ms) of queries with and without Spatial Pruning.

if the user requests an output of the WKT format, it is more efficient to deserialize the WKB stored and convert it to WKT than to parse the WKT format.

4.4.5 Effect of spatial pruning

In figure 4.7, we compare the execution time of queries with and without spatial pruning. As demonstrated in the figure, the spatial pruning improves performance for most of the queries. This is due to the decreasing size of the search space. However, this is not the case of all queries, since the number of pruned fragments depends on the query and can vary form one to another. This is the case of query Q_2 where no fragment is pruned. Furthermore, the incurred overhead associated with evaluating fragments for pruning purposes can be deemed negligible, as exemplified by the examination of query. (Q_2).

4.4.6 Comparison against Virtuoso

After the optimization techniques applied to improve the performance of RDF_QDAG, we compare it with a commercial Triplestore Virtuoso. We choose Virtuoso since it is a stable and wildly used Triplestore. On top of that it is one of the few Triplestores capable of answering spatial-rdf queries since it support the GeoSPARQL norm proposed by the Open Geospatial Consortium. As for the other solutions (e.g., GraphDB and Strabon) we where unable to load the dataset due to stability issues in the mentioned systems.

The figure 4.8 depicts the execution times of queries run on both Virtuoso and RDF_QDAG. For RDF_QDAG, we plot the execution time of two different runs, one without

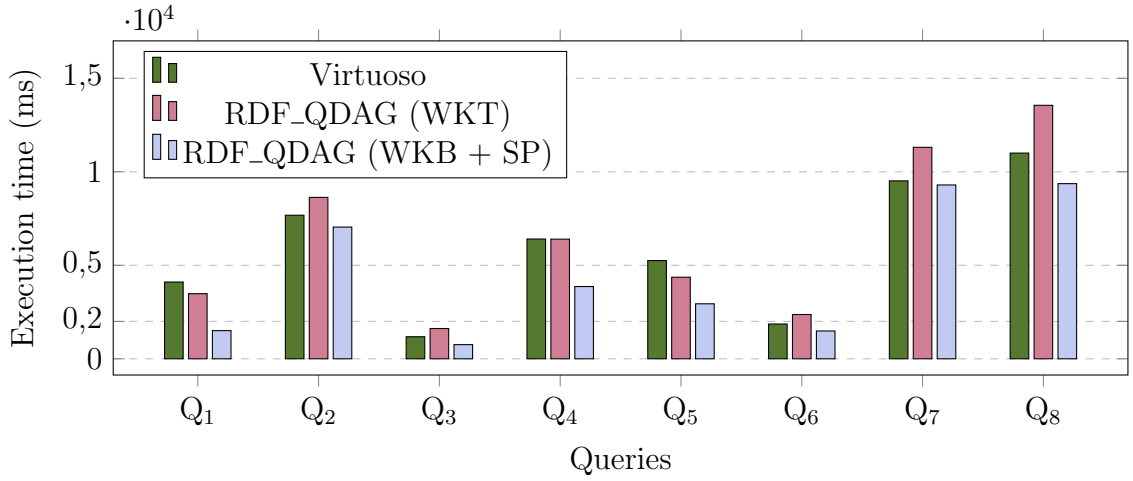


Figure 4.8: Compression of execution time between Virtuoso and RDF_QDAG.

any optimization technique used (WKT) the other one with the optimization techniques proposed and studied in previous sections (WKB+SP). We can notice that the WKT approach outperforms Virtuoso in some queries like Q_1 and Q_5. However, on most of the queries,Virtuoso still had better performace leading to a better total execution time of 47 seconds for Virtuoso compared to 52 seconds for WKT. On the other hand, after applying the proposed optimization techniques (WKT+SP), RDF_QDAG outperforms Virtuoso on all of the test queries without exception and has a better total execution time.

4.5 Conclusion

In this chapter, we addressed the evaluation of spatial RDF queries issue in the setting of a graph exploration-based system, known as RDF_QDAG. To enhance the system's capability to answer such queries, we proposed an extension that integrates spatial awareness into the system's storage layer, evaluation engine and optimization process. More specifically, we proposed the use of an R-tree data structure, which is adapted to better fit the system, as well as the integration of the evaluation of spatial filters into the execution plans. Additionally, we introduced two evaluation strategies, namely, BGP-First and Spatial-First, for the execution engine. In terms of optimization, we presented a cost model that considers the cost of spatial operations in order to optimize the selection of execution plans. Furthermore, we proposed a spatial pruning technique to further improve performance by reducing the search space.

On the other hand, we validated our proposed extension to RDF_QDAG through an

experimental setup using a real-world dataset (i.e., YAGO). Our results indicated that the use of optimization techniques such as WKB encoding and spatial pruning improve the performance of the system. We also evaluated the proposed execution strategies of BGP-First and Spatial-First, and found that each strategy had advantages and limitations depending on the query being executed. To address this, we developed a cost model to determine the most suitable strategy for each query. Our results also indicated that the proposed cost model enables the system to better predict the best execution plan compared to the existing one.

Conclusion and Perspectives

Conclusion

This thesis delves into the intricacies of spatial and RDF data processing, with a specific focus on spatial data storage, the integration of spatial data processing into triplestores, and optimization techniques applicable to hybrid spatial and RDF datasets. A thorough exploration of spatial data processing techniques utilized in spatial database management systems was undertaken to evaluate diverse trends and approaches. Furthermore, an examination of triplestores capable of housing RDF data and handling SPARQL queries was conducted. Through this study, numerous challenges affecting the performance of spatial-RDF data processing were identified. These challenges include:

- Efficient Processing of Disk-Resident Spatial Data: As the magnitude of spatial data handled by contemporary applications continues to grow, leveraging secondary storage media becomes imperative. Yet, the access time associated with such media presents a substantial bottleneck. The selection of a suitable indexing structure and access method becomes pivotal to achieving peak performance. The intrinsic characteristics of spatial data, including the absence of a universal order, further complicate this endeavor, given that many access methods depend on such an order.

- Processing Spatial Data in the Context of Graph Databases: The central challenge pertains to processing spatial data within graph databases, with a special emphasis on triplestores. Within graph structures, nodes might encapsulate spatial details through coordinates or shapes. Queries can encompass both graph-related constraints (e.g., connectivity to specific nodes) and spatial constraints (e.g., intersecting with a region or proximity to an object). These queries introduce distinctive challenges due to the contrasting attributes of the two data types and the varied

processing methods employed. The development of a hybrid processing strategy, adept at effectively managing both data types, becomes pivotal for attaining optimal performance.

- Optimizing Execution Plans for Hybrid Spatial-RDF Data: Within the realm of triplestores, multiple execution plans can arise as equivalents from a single query. Nonetheless, the costs and execution times of these plans may exhibit substantial discrepancies. The selection of the most suitable execution plan plays a pivotal role in achieving optimal performance. This decision hinges upon cost estimation, a task further complicated by the disparities in data types and processing methodologies employed.

In response to the initial challenge of efficient processing of spatial data stored on disk, we undertook an exhaustive examination of established indexing methodologies for spatial data. We elucidated the merits and demerits associated with each indexing technique. Through our exploration, a consistent pattern emerged within tree-based indexes, notably in R-tree-based variants, wherein backtracking operations are employed to reassess upper levels of the index tree. Nevertheless, these backtracking procedures frequently entail supplementary disk expenses when reloading the necessary pages.

To mitigate these disk costs and improve efficiency, we proposed a novel indexing technique called FASTER. FASTER is an R-tree traversal algorithm designed to minimize disk costs by reducing the number of backtracks performed. The key concept behind FASTER is to evaluate all entries within a specific index node and store the entries with potential answers further down the tree in a highly available queue stored in the main memory.

To evaluate the performance of FASTER, we compared it with the Standard R-Tree Traversal algorithm (SRT) using real-world datasets. The results of our evaluation clearly demonstrate that FASTER outperforms SRT in almost all scenarios. It provides a significant speedup, reaching up to 50% improvement in favorable scenarios, while incurring only minimal or negligible overhead in worst-case scenarios. These findings highlight the effectiveness of FASTER in reducing disk costs and enhancing the efficiency of spatial data processing.

To tackle the second challenge of processing spatial data in the context of graph databases, we selected RDF_QDAG as our triplestore and extended it to incorporate spa-

tial data processing capabilities. This decision was motivated by RDF_QDAG's efficiency and its ability to strike a balance between performance and scalability. In order to integrate spatial data processing, several modifications were made to different aspects of the system, including storage, access methods, evaluation engine, parsing and optimizer.

We proposed two evaluation strategies: BGP-first strategy and Spatial-First strategy. The BGP-first strategy involves evaluating the graph constraints before the spatial constraints. However, this approach does not fully leverage the benefits of spatial access methods since they typically require early utilization. To address this limitation, we introduced the Spatial-First strategy, which takes advantage of the FASTER access method.

Both of these strategies were empirically evaluated using real-world datasets and compared against a commercial triplestore, specifically Virtuoso. The results demonstrated significant variations in execution time across different queries and execution plans. Each strategy exhibited the potential to outperform Virtuoso and deliver superior performance in favorable scenarios. However, in worst-case scenarios, their performance fell short. Consequently, the need arose to determine the best execution strategy for each query, considering the specific characteristics of the query and the available execution plans.

The third challenge revolves around optimizing execution plans for hybrid Spatial-RDF data. The selection of the most suitable execution plan relies on accurately estimating their costs. RDF_QDAG employs the GoFast optimizer for this purpose. However, GoFast lacks spatial awareness as it does not gather information pertaining to the spatial distribution of the data. To overcome this limitation, we proposed a novel cost model that incorporates spatial data into the cost evaluation process at various stages of the execution plan, leading to more precise estimations. The new cost model leverages the previously described Index as a valuable source of statistics regarding spatial data.

Next, we compared the performance of the new cost model with that of the existing one. To assess the accuracy of the optimizer, we employed a ranking system where a higher rank assigned to the true best plan indicates a greater level of accuracy. The enhanced optimizer demonstrated superior prediction of the best execution plan, surpassing the original optimizer for the majority of the tested queries. The accuracy of the optimizer was significantly improved, enabling it to identify the true best plan in most cases, with only a few queries exhibiting room for further enhancement.

Perspectives

The research presented in this manuscript not only provides valuable insights into the integration of spatial data in triplestores but also paves the way for numerous prospective directions. Some avenues are currently under investigation, while others are envisaged for the medium term.

Expanding the work to cover spatio-temporal data. While the primary emphasis of this study revolves around the integration of spatial data, it's imperative to recognize that the temporal dimension holds immense importance in numerous real-world contexts. Therefore, broadening the horizons of this work to encompass the integration of spatio-temporal data presents a highly promising avenue for exploration.

Incorporating the temporal aspect into the existing framework poses intriguing challenges and opportunities. Users often require temporal constraints in their queries, such as retrieving results that occurred before a specific event or within a certain time frame. Introducing the temporal dimension introduces the necessity to handle and process moving objects, thereby necessitating the adoption of alternative storage, indexing, and processing techniques. Consequently, future investigations should delve into the intricate interplay between spatial and temporal data, exploring novel approaches for effectively managing and querying spatio-temporal information within triplestores.

The use of machine learning techniques for spatial-RDF data. The increasing popularity and adoption of machine learning techniques within the realm of database management systems have engendered a multitude of notable contributions, such as learned indexes and machine learning-based optimizers. Specifically, in the case of RDF_QDAG, the system is equipped with a wealth of valuable statistics pertaining to the databases, as well as extensive query logs and execution plans, all of which serve optimization purposes. Leveraging this abundance of information presents an opportunity to train a model that can estimate the cost associated with executing different execution plans, thus facilitating the selection of the optimal plan.

Within the domain of spatial data processing, machine learning models can be effectively trained to discern patterns and understand the distribution of spatial objects throughout the data universe. By leveraging these models, it becomes possible to enhance the prediction accuracy of spatial object quantities and the associated costs of performing spatial operations. Furthermore, the introduction of learned indexes offers a

promising alternative to the traditional spatial indexes employed by the system, enabling more efficient spatial data retrieval and query processing.

The incorporation of machine learning techniques within the optimization framework and spatial data processing modules of RDF_QDAG introduces novel avenues for exploration and research. The exploitation of learned models for cost estimation and the adoption of learned indexes for spatial data retrieval hold significant potential for advancing query optimization and bolstering the overall performance of hybrid Spatial-RDF databases.

A Parallel Evolution of Spatial Queries. Recently, RDF_QDAG has been equipped with a parallel query evaluation strategy. This strategy is based on the Bulk Synchronous Parallel (BSP) model, which aligns well with the exploration-based evaluation of the logical graph associated with the data. Supporting spatial operators presents certain challenges. Indeed, our proposals in this thesis need to be adapted to preserve the scalability-performance trade-off. Furthermore, an in-depth study of the data partitioning strategy is required. In addition to the graph partitioning proposed in RDF_QDAG, spatial object-based partitioning based on spatial coordinates can be envisioned. The evaluation engine will also be impacted by this evolution. It is essential to account for the distributed nature of the data, especially for distance-based join processing. Lastly, the cost associated with network transfers related to query evaluation must be incorporated into the cost model integrated within the RDF_QDAG optimizer.

Bibliographie

[Adm22] Great Britain Admiralty. Admiralty manual of navigation. *(No Title)*, 1922. (Cited in page 17)

[AMMH09] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009. (Cited in page 5), (Cited in page 58)

[ASV+13] Ablimit Aji, Xiling Sun, Hoang Vo, Qioaling Liu, Rubao Lee, Xiaodong Zhang, Joel Saltz, and Fusheng Wang. Demonstration of hadoop-gis: a spatial data warehousing system over mapreduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 528–531. ACM, 2013. (Cited in page 50)

[AWV+13] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013. (Cited in page 46), (Cited in page 50)

[AXL+15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015. (Cited in page 55)

[BDK+13] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132, 2013. (Cited in page 5), (Cited in page 58), (Cited in page 59)

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. (Cited in page 33), (Cited in page 38)

[BK12] Robert Battle and Dave Kolas. Enabling the geospatial semantic web with parliament and geosparql. *Semantic Web*, 3(4):355–370, 2012. (Cited in page 22), (Cited in page 59), (Cited in page 80)

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. *Efficient processing of spatial joins using R-trees*, volume 22. ACM, 1993. (Cited in page 30), (Cited in page 33), (Cited in page 43), (Cited in page 45)

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990. (Cited in page 40)

[BKVH01] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information, 2001. (Cited in page 5), (Cited in page 6), (Cited in page 58), (Cited in page 59), (Cited in page 60)

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001. (Cited in page 2)

[BNM10] Andreas Brodt, Daniela Nicklas, and Bernhard Mitschang. Deep integration of spatial query processing into native rdf triple stores. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 33–42, 2010. (Cited in page 6), (Cited in page 60)

[CB17] Francesco Cafagna and Michael H Böhlen. Disjoint interval partitioning. *The VLDB Journal—The International Journal on Very Large Data Bases*, 26(3):447–466, 2017. (Cited in page 30), (Cited in page 35)

[CCZY+15] Georgios Chatzimilioudis, Constantinos Costa, Demetrios Zeinalipour-Yazti, Wang-Chien Lee, and Evaggelia Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *IEEE Transactions on Knowledge and Data Engineering*, 28(4):925–938, 2015. (Cited in page 55)

[CSPG20] Tanvi Chawla, Girdhari Singh, Emmanuel S Pilli, and Mahesh Chandra Govil. Storage, partitioning, indexing and retrieval in big rdf frameworks:

A survey. *Computer Science Review*, 38:100309, 2020. (Cited in page 57), (Cited in page 58)

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (Cited in page 45)

[EM13] Ahmed Eldawy and Mohamed F Mokbel. A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data. *Proceedings of the VLDB Endowment*, 6(12):1230–1233, 2013. (Cited in page 51)

[EM15] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*, pages 1352–1363. IEEE, 2015. (Cited in page 46), (Cited in page 51), (Cited in page 67)

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974. (Cited in page 33), (Cited in page 39), (Cited in page 45)

[FHL14] Jianqing Fan, Fang Han, and Han Liu. Challenges of big data analysis. *National science review*, 1(2):293–314, 2014. (Cited in page 1)

[FMSHFM12] Vanessa Frias-Martinez, Victor Soto, Heath Hohwald, and Enrique Frias-Martinez. Characterizing urban landscapes using geolocated tweets. In *2012 International conference on privacy, security, risk and trust and 2012 international confernece on social computing*, pages 239–248. IEEE, 2012. (Cited in page 1)

[GGCI+16] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Enhancing spatialhadoop with closest pair queries. In *East European Conference on Advances in Databases and Information Systems*, pages 212–225. Springer, 2016. (Cited in page 46), (Cited in page 53)

[GGCI+18] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Efficient large-scale distance-based join queries in spatialhadoop. *GeoInformatica*, 22(2):171–209, 2018. (Cited in page 47), (Cited in page 53)

[Gra] Graphdb. `https://graphdb.ontotext.com/`. Accessed: 2021-10-18. (Cited in page 6), (Cited in page 60)

[Gut84] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984. (Cited in page 39), (Cited in page 40), (Cited in page 60)

[Güt94] Ralf Hartmut Güting. An introduction to spatial database systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, 3(4):357–399, 1994. (Cited in page 15)

[HAE+15] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, volume 9473, page 94730F. International Society for Optics and Photonics, 2015. (Cited in page 54)

[HG03] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. *1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), Sanibel Island, Florida*, pages 1–15, 2003. (Cited in page 5), (Cited in page 58), (Cited in page 80)

[Hit21] Pascal Hitzler. A review of the semantic web field. *Communications of the ACM*, 64(2):76–83, 2021. (Cited in page 2), (Cited in page 20)

[HLS+09] Steve Harris, Nick Lamb, Nigel Shadbolt, et al. 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, volume 94, 2009. (Cited in page 58), (Cited in page 59)

[HNS88] Klaus Hinrichs, Jurg Nievergelt, and Peter Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26(5):255–261, 1988. (Cited in page 35), (Cited in page 45)

[JDW+13] Yu Jiang, Dong Deng, Jiannan Wang, Guoliang Li, and Jianhua Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 341–348. ACM, 2013. (Cited in page 17)

[KCK01] Kihong Kim, Sang K Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. In *ACM SIGMOD Record*, volume 30, pages 139–150. ACM, 2001. (Cited in page 5), (Cited in page 40), (Cited in page 45)

[KK10] Manolis Koubarakis and Kostis Kyzirakos. Modeling and querying metadata in the semantic sensor web: The model strdf and the query language stsparql. In *Extended Semantic Web Conference*, pages 425–439. Springer, 2010. (Cited in page 22), (Cited in page 80)

[KKK12] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial dbms. In *International Semantic Web Conference*, pages 295–311. Springer, 2012. (Cited in page 6), (Cited in page 59), (Cited in page 60)

[KMG+21] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, Mohand-Saïd Hacid, and Emmanuel Coquery. Combining graph exploration and fragmentation for scalable rdf query processing. *Information Systems Frontiers*, 23(1):165–183, 2021. (Cited in page 5), (Cited in page 22), (Cited in page 24), (Cited in page 58), (Cited in page 59), (Cited in page 80)

[LGWL17] Yongxuan Lai, Xing Gao, Tian Wang, and Ziyu Lin. Efficient iceberg join processing in wireless sensor networks. *International Journal of Embedded Systems*, 9(4):365–378, 2017. (Cited in page 19)

[LLE97] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. Str: A simple and efficient algorithm for r-tree packing. In *Proceedings 13th International Conference on Data Engineering*, pages 497–506. IEEE, 1997. (Cited in page 74), (Cited in page 80), (Cited in page 88)

[LN97] Scott T Leutenegger and David M Nicol. Efficient bulk-loading of gridfiles. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):410–420, 1997. (Cited in page 32)

[LRZ+20] Jiaying Liu, Jing Ren, Wenqing Zheng, Lianhua Chi, Ivan Lee, and Feng Xia. Web of scholars: A scholar knowledge graph. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2153–2156, 2020. (Cited in page 2)

[LSCO12] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012. (Cited in page 46), (Cited in page 49), (Cited in page 55)

[ME92] Priti Mishra and Margaret H Eich. Join processing in relational databases.

ACM Computing Surveys (CSUR), 24(1):63–113, 1992. (Cited in page 30), (Cited in page 33), (Cited in page 45)

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994. IEEE, 2011. (Cited in page 23)

[NQJ17] Sadegh Nobari, Qiang Qu, and Christian S Jensen. In-memory spatial join: The data matters! In *EDBT*, pages 462–465, 2017. (Cited in page 42)

[NTH+13] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. Touch: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 701–712. ACM, 2013. (Cited in page 30), (Cited in page 41), (Cited in page 42), (Cited in page 45)

[NW08] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008. (Cited in page 6), (Cited in page 24), (Cited in page 58), (Cited in page 60)

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009. (Cited in page 21)

[PD96] Jignesh M Patel and David J DeWitt. Partition based spatial-merge join. In *ACM Sigmod Record*, volume 25, pages 259–270. ACM, 1996. (Cited in page 30), (Cited in page 42), (Cited in page 45)

[RCVM16] George Roumelis, Antonio Corral, Michael Vassilakopoulos, and Yannis Manolopoulos. New plane-sweep algorithms for distance-based join queries in spatial databases. *GeoInformatica*, 20(4):571–628, 2016. (Cited in page 36), (Cited in page 45)

[Rob81] John T Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981. (Cited in page 6), (Cited in page 38), (Cited in page 45)

[RVC11] George Roumelis, Michael Vassilakopoulos, and Antonio Corral. Nearest neighbor algorithms using xbr-trees. In *2011 15th Panhellenic Conference on Informatics*, pages 51–55. IEEE, 2011. (Cited in page 39)

[RVCM14] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. A new plane-sweep algorithm for the k-closest-pairs query. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 478–490. Springer, 2014. (Cited in page 36), (Cited in page 45)

[RVCM18] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. An efficient algorithm for bulk-loading xbr+-trees. *Computer Standards & Interfaces*, 57:83–100, 2018. (Cited in page 39)

[RVL+15] George Roumelis, Michael Vassilakopoulos, Thanasis Loukopoulos, Antonio Corral, and Yannis Manolopoulos. The xbr+-tree: An efficient access method for points. In *International Conference on Database and Expert Systems Applications*, pages 43–58. Springer, 2015. (Cited in page 6), (Cited in page 39), (Cited in page 45)

[SKS+97] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997. (Cited in page 15)

[ŠŠC+09] Darius Šidlauskas, Simonas Šaltenis, Christian W Christiansen, Jan M Johansen, and Donatas Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009. (Cited in page 37), (Cited in page 45)

[Sto03] Knut Stolze. Sql/mm spatial-the standard to manage spatial data in a relational database system. In *BTW*, volume 2003, pages 247–264, 2003. (Cited in page 50)

[TLH20] Ilaria Tiddi, Freddy Lécué, and Pascal Hitzler. Knowledge graphs for explainable artificial intelligence: Foundations, applications and challenges. 2020. (Cited in page 2)

[TYA+16] M Tang, Y Yu, WG Aref, AR Mahmood, QM Malluhi, and M Ouzzani. In-memory distributed spatial query processing and optimization. Technical report, Purdue technical report, 2016. (Cited in page 39)

[TYM+16] 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. *Proceedings of the VLDB Endowment*,

9(13):1565–1568, 2016. (Cited in page 47), (Cited in page 54), (Cited in page 67)

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, 1987. (Cited in page 43)

[Ver67] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967. (Cited in page 67)

[vir] Virtuoso. `https://virtuoso.openlinksw.com/`. Accessed: 2021-10-18. (Cited in page 6), (Cited in page 60)

[VNB03] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings 2003 VLDB Conference*, pages 285–296. Elsevier, 2003. (Cited in page 19)

[WFL10] Jiannan Wang, Jianhua Feng, and Guoliang Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proceedings of the VLDB Endowment*, 3(1-2):1219–1230, 2010. (Cited in page 17)

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o (n log n). In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69. IEEE, 2006. (Cited in page 38)

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008. (Cited in page 5), (Cited in page 58)

[WKC+11] Fusheng Wang, Jun Kong, Lee Cooper, Tony Pan, Tahsin Kurc, Wenjin Chen, Ashish Sharma, Cristobal Niedermayr, Tae W Oh, Daniel Brat, et al. A data model and database for high-resolution pathology analytical image informatics. *Journal of pathology informatics*, 2, 2011. (Cited in page 67)

[WKC12] Chih-Jye Wang, Wei-Shinn Ku, and Haiquan Chen. Geo-store: a spatially-augmented sparql query evaluation system. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 562–565, 2012. (Cited in page 6), (Cited in page 60)

[WSK+03] Kevin Wilkinson, Craig Sayers, Harumi A Kuno, Dave Reynolds, et al. Efficient rdf storage and retrieval in jena2. In *SWDB*, volume 3, pages

131–150. Citeseer, 2003. (Cited in page 5), (Cited in page 58), (Cited in page 59), (Cited in page 80)

[WXLZ09] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 759–770. ACM, 2009. (Cited in page 17)

[XLOH04] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. Gorder: an efficient method for knn join processing. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 756–767. VLDB Endowment, 2004. (Cited in page 55)

[XLY+16] 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*, pages 1071–1085. ACM, 2016. (Cited in page 47), (Cited in page 54), (Cited in page 55), (Cited in page 56)

[XWL08] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944, 2008. (Cited in page 17)

[YWS15] 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*, page 70. ACM, 2015. (Cited in page 46), (Cited in page 54)

[YZG15] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 34–41. IEEE, 2015. (Cited in page 46), (Cited in page 54)

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012. (Cited in page 54)

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010. (Cited in page 53)

[ZHL+09] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. IEEE, 2009. (Cited in page 46), (Cited in page 47)

[ZHOS10] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 915–926. ACM, 2010. (Cited in page 17)

[ZLJ12] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th international conference on extending database technology*, pages 38–49. ACM, 2012. (Cited in page 46), (Cited in page 49)

[ZMG+21] Ishaq Zouaghi, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, and Taoufik Aguili. Gofast: Graph-based optimization for efficient and scalable query evaluation. *Information Systems*, 99:101738, 2021. (Cited in page 5), (Cited in page 58), (Cited in page 59), (Cited in page 87), (Cited in page 88), (Cited in page 90)

[ZYW+13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013. (Cited in page 5), (Cited in page 58), (Cited in page 59)

Appendix A

A.1 Queries used for the experimental validation

$\mathbf{Q}_1$.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/hasGivenName> ?gn .
?p <http://yago-knowledge.org/resource/hasFamilyName> ?fn .
?p <http://yago-knowledge.org/resource/hasWonPrize> ?pr .
?p <http://yago-knowledge.org/resource/diedIn> ?c .
?c <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-100 20, -80 20, -80 40, -100 40, -100 20))" ), bif:st_geomfromtext(?g) ) )
};
```

$\mathbf{Q}_2$.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/hasGivenName> ?gn .
?p <http://yago-knowledge.org/resource/hasFamilyName> ?fn .
?p <http://yago-knowledge.org/resource/wasBornIn> ?c .
?c <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-95 40, -90 40, -90 45, -95 45, -95 40))" ), bif:st_geomfromtext(?g) ) )
};
```

$\mathbf{Q}_3$.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
```

```
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/hasAcademicAdvisor> ?a .
?a <http://yago-knowledge.org/resource/worksAt> ?w .
?w <http://yago-knowledge.org/resource/isLocatedIn> ?l .
?l  <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-160 -50, -150 -50, -150 -40, -160 -40, -160 -50))" ), bif:st_geomfromtext(?g) ) )
};
```

Q_4.

```
SPARQL
SELECT (COUNT(?e) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?e <http://yago-knowledge.org/resource/happenedIn> ?l .
?l <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-130 40, -120 40, -120 50, -130 50, -130 40))" ), bif:st_geomfromtext(?g) ) )
};
```

Q_5.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/hasGivenName> ?gn .
?p <http://yago-knowledge.org/resource/hasFamilyName> ?fn .
?p <http://yago-knowledge.org/resource/wasBornIn> ?c .
?c <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-105 45, -100 45, -100 50, -105 50, -105 45))" ), bif:st_geomfromtext(?g) ) )
};
```

Q_6.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/graduatedFrom> ?u .
?p <http://yago-knowledge.org/resource/worksAt> ?w .
?u <http://yago-knowledge.org/resource/isLocatedIn> ?l .
?l <http://yago-knowledge.org/resource/hasGeometry> ?g .
```

```
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-110 50, -100 50, -100 60, -110 60, -110 50))" ), bif:st_geomfromtext(?g) ) )
};
```

Q_7.

```
SPARQL
SELECT (COUNT(?e) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?e <http://yago-knowledge.org/resource/happenedIn> ?l .
?l <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-90 30, -80 30, -80 40, -90 40, -90 30))" ), bif:st_geomfromtext(?g) ) )
};
```

Q_8.

```
SPARQL
SELECT (COUNT(?p) AS ?cnt)
FROM <http://YAGO_2S.com>
WHERE
{
?p <http://yago-knowledge.org/resource/hasArea> ?a .
?p <http://yago-knowledge.org/resource/isLocatedIn> ?l .
?l <http://yago-knowledge.org/resource/hasGeometry> ?g .
FILTER( bif:st_intersects( bif:st_geomfromtext(
    "POLYGON((-100 30, -90 30, -90 40, -100 40, -100 30))" ), bif:st_geomfromtext(?g) ) )
};
```

A.2 Results of estimation of each plan for the different queries considered

For all of the following tables, the best execution plan is highlighted in bold.

Table A.1: Results of estimation of Q_1

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c}, Fu(?g), \overleftarrow{?c}, \overrightarrow{?p}]$	4774913	4775175	7	7
1	$[\overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	**5943.0**	**29657.0**	**3**	**1**
2	$[\overleftarrow{?c}, \overrightarrow{?c}, Fu(?g), \overrightarrow{?p}]$	5595	54502	1	2
3	$[\overleftarrow{?c}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	5720	54627	2	3
4	$[\overleftarrow{?f}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	286682	859297	5	4
5	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$	437395	446318	6	5
6	$[\overleftarrow{?n}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	83750	857355	4	6

Table A.2: Results of estimation of Q_2

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c}, Fu(?g), \overleftarrow{?c}, \overrightarrow{?p}]$	4775880.0	4777965.0	7	7
1	$[\overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	165958.0	493909.0	4	4
2	$[\overleftarrow{?a}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	243796.0	1321696.0	5	6
3	$[\overleftarrow{?b}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	389065.0	1154313.0	6	5
4	$[\overleftarrow{?c}, \overrightarrow{?c}, Fu(?g), \overrightarrow{?p}]$	14421.0	192320.0	1	2
5	$[\overleftarrow{?c}, \overrightarrow{?p}, \overrightarrow{?c}, Fu(?g)]$	16412.0	194311.0	2	3
6	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$	**104914.0**	**129598.0**	**3**	**1**

Table A.3: Results of estimation of Q_3

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?a}, \overleftarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l}, Fu(?g)]$	7231.0	7388.0	4	4
1	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}, \overleftarrow{?w}, \overleftarrow{?a}]$	4774850.0	4774858.0	8	8
2	$[\overrightarrow{?p}, \overrightarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l}, Fu(?g)]$	5447.0	5676.0	3	2
3	$[\overrightarrow{?w}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?w}, \overleftarrow{?a}]$	669919.0	1252615.0	7	7
4	$[\overleftarrow{?a}, \overrightarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l}, Fu(?g)]$	**4231.0**	**5492.0**	**2**	**1**
5	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}, \overleftarrow{?w}, \overleftarrow{?a}]$	441052.0	702586.0	6	5
6	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?w}, \overleftarrow{?a}]$	59251.0	1250718.0	5	6
7	$[\overleftarrow{?w}, \overleftarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l}, Fu(?g)]$	3245.0	7045.0	1	3

Table A.4: Results of estimation of Q_4

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?e}, \overrightarrow{?l}, Fu(?g)]$	201726.0	208424.0	3	3
1	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}]$	4774844.0	4774844.0	4	4
2	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}]$	**74903.0**	**98141.0**	**2**	**1**
3	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g)]$	17716.0	194145.0	1	2

Table A.5: Results of estimation of Q_5

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c}, \overleftarrow{?c}, \overrightarrow{?p}, Fu(?g)]$	4775880.0	4777965.0	7	7
1	$[\overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	165958.0	493909.0	4	4
2	$[\overleftarrow{?a}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	243796.0	1321696.0	5	5
3	$[\overleftarrow{?b}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	389065.0	1154313.0	6	6
4	$[\overleftarrow{?c}, \overrightarrow{?c}, \overrightarrow{?p}, Fu(?g)]$	14421.0	192320.0	1	2
5	$[\overleftarrow{?c}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	16412.0	194311.0	2	3
6	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$	**51616.0**	**75701.0**	**3**	**1**

Table A.6: Results of estimation of Q_6

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}, \overleftarrow{?u}, \overrightarrow{?p}]$	4774894.0	4774981.0	7	7
1	$[\overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$	**5901.0**	**9894.0**	**3**	**2**
2	$[\overrightarrow{?u}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?u}, \overrightarrow{?p}]$	669962.0	1252701.0	6	6
3	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}, \overleftarrow{?u}, \overrightarrow{?p}]$	30958.0	250625.0	5	4
4	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?u}, \overrightarrow{?p}]$	59295.0	1250841.0	4	5
5	$[\overleftarrow{?u}, \overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$	5192.0	32596.0	2	3
6	$[\overleftarrow{?w}, \overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$	3394.0	7338.0	1	1

Table A.7: Results of estimation of Q_7

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?e}, \overrightarrow{?l}, Fu(?g)]$	201726.0	208424.0	2	2
1	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}]$	4774844.0	4774844.0	4	3
2	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}]$	270889.0	297661.0	3	4
3	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g)]$	**17716.0**	**194145.0**	**1**	**1**

Table A.8: Results of estimation of Q_8

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}, \overrightarrow{?p}]$	4775053.0	4775546.0	6	6
1	$[\overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$	**105462.0**	**339654.0**	**3**	**1**
2	$[\overleftarrow{?a}, \overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$	134597.0	469582.0	4	3
3	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}, \overrightarrow{?p}]$	186351.0	420618.0	5	2
4	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g), \overrightarrow{?p}]$	59454.0	1251406.0	1	4
5	$[\overleftarrow{?l}, \overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$	59857.0	1251809.0	2	5

Résumé

Depuis l'apparition du modèle relationnel, les systèmes de gestion de données relationnelles ont dominé les autres systèmes en raison de la simplicité liée à la représentation des données et de leur capacité à répondre aux requêtes déclaratives. Cependant, le modèle relationnel souffre de plusieurs limitations qui le rendent indésirable pour de nombreux cas d'utilisation. En effet, le modèle relationnel ne convient pas à certains types de données comme les données graphes (souvent utilisées dans la manipulation des graphes de connaissances) et les données spatiales (souvent utilisées dans les systèmes d'information géographique). Cette limitation a conduit à l'introduction de bases de données spatiales et les systèmes de stockage des triplets pour les données spatiales et les données graphes respectivement.

Dans cette thèse, nous considérons les deux types de données : Graphe et Spatial. Cependant, nous nous concentrons davantage sur les données spatiales et les défis imposés par les données hybrides (contenant des objets provenant des deux représentations : spatiale et graphe). Le premier problème est le coût élevé de l'évaluation des opérateurs spatiaux. Nous essayons d'améliorer les performances des opérateurs spatiaux sur de grands jeux de données spatiales stockées sur disque. Le deuxième problème abordé est le traitement des jeux de données hybrides, puisqu'ils donnent lieu à plusieurs problèmes à plusieurs niveaux tels que le stockage, l'indexation, l'interrogation et l'optimisation.

Afin d'améliorer les performances des opérateurs spatiaux, nous proposons une nouvelle technique pour explorer les indexes spatiaux tout en minimisant le nombre d'opérations d'E/S vers/depuis le disque. Nous comparons l'approche proposée avec l'état de l'art en utilisant des jeux de données réels. En plus de, et afin de résoudre les problèmes engendrés par les données hybrides, nous proposons une extension (Spatial-Qdag) d'un triplestore existant (RDF_QDAG) qui couvre plusieurs couches du système : stockage, évaluation et optimisation. Nous comparons l'extension spatiale (Spatial-Qdag) avec des triplestores commerciaux enutilisant des jeux de données réels.

Les résultats des expérimentations menées démontrent une amélioration significative des performances des opérateurs spatiaux pour la plupart des requêtes en utilisant l'approche proposée. Ce qui signifie une supériorité de l'extension proposée (Spatial-Qdag) par rapport aux systèmes comparés.

Mots-clés : Big Data, RDF, SPARQL, Données spatiales, Performance, Passage à l'échelle

Thèse préparée dans le cadre d'une cotutelle entre l'Université Abu Bekr Belkaid Tlemcen et l'École Nationale Supérieure de Mécanique et d'Aérotechnique

Abstract

Since the appearance of the relational model, relational data management systems have dominated the other systems due to simplicity related to data representation and their ability to answer declarative queries. However, the relational model suffers from several problems that make it undesirable for many use cases. For instance, the relational model is not suitable for some data types such as Graph data that is often used in knowledge graphs and Spatial data that is often used in Geographical information systems. The later limitation has led to the introduction of spatial databases and triple stores for Spatial data and graph data respectively.

In this thesis, we consider both types of data Graph and Spatial. However, we focus more on spatial data and the challenges imposed by hybrid data (the data-set contains objects from both representations: spatial and graph). The first problem is the high cost of evaluating spatial operators. We try to improve performance of spatial operators on large spatial data sets stored on disk. The second problem is the processing of hybrid data sets, since they give rise to several challenges many levels such as storage, indexing, querying and optimisation.

In order to improve the performance of spatial operators. We propose a novel technique to explore spatial indices while minimizing the number of I/O operations to/from the disk. We compare our approach with the existing state of art using real world dataset. On top of that, to solve problems imposed by hybrid data, we propose an extension (Spatial-Qdag) of an existing triple store (RDF_-QDAG) that covers every layer: storage, evaluation and optimisation. We compare the spatial extension (Spatial-Qdag) with commercial triple stores using real world datasets.

The experimental results demonstrate a significant improvement in spatial operators' performance in most queries while using the proposed approach. On top of that, the results demonstrate the superiority of the proposed extension (Spatial-Qdag) compared to the competition.

Keywords : Big Data, RDF, SPARQL, Spatial Data, Performance, Scalability

---

Thèse préparée dans le cadre d'une cotutelle entre l'Université Abu Bekr Belkaid Tlemcen et l'École Nationale Supérieure de Mécanique et d'Aérotechnique

ملخص

منذ ظهور النموذج العلائقي ، سيطرت أنظمة إدارة البيانات العلائقية الأنظمة الأخرى بسبب البساطة المتعلقة بتمثيل البيانات وقدرتها على الإجابة على الاستفسارات التوضيحية. ومع ذلك ، فإن النموذج العلائقي يعاني من العديد من المشاكل التي تجعله غير مرغوب فيه العديد من حالات الاستخدام. على سبيل المثال ، النموذج العلائقي غير مناسب لبعض أنواع البيانات مثل الرسم البياني البيانات التي تُستخدم غالبًا في الرسوم البيانية المعرفية والبيانات المكانية التي تُستخدم غالبًا في الجغرافيا نظم المعلومات. وقد أدى القيد اللاحق إلى إدخال قواعد البيانات المكانية والثلاثية يخزن للبيانات المكانية وبيانات الرسم البياني على التوالي. في هذه الأطروحة ، نعتبر كلا نوعي البيانات الرسم البياني والمكاني. ومع ذلك ، فإننا نركز أكثر على البيانات المكانية والتحديات التي تفرضها البيانات الهجينة (تحتوي مجموعة البيانات على كائنات من كليهما تمثيلات: مكانية ورسم بياني). المشكلة الأولى هي التكلفة العالية لتقييم العوامل المكانية. نحاول تحسين أداء المشغلين المكانيين على مجموعات البيانات المكانية الكبيرة المخزنة على القرص. ال المشكلة الثانية هي معالجة مجموعات البيانات المختلطة ، لأنها تؤدي إلى العديد من التحديات مستويات مثل التخزين والفهرسة والاستعلام والتحسين. من أجل تحسين أداء المشغلين المكانيين. نقترح تقنية جديدة للاستكشاف المؤشرات المكانية مع تقليل عدد عمليات الإدخال / الإخراج إلى / من القرص. نقارن لدينا نهج مع حالة الفن الحالية باستخدام مجموعة بيانات العالم الحقيقي. علاوة على ذلك ، لحل المشاكل التي تفرضها البيانات المختلطة ، التي تغطي كل طبقة: التخزين والتقييم والتحسين. نقارن (RDF - QDAG) لمتجر ثلاثي موجود (Spatial-Qdag) نقترح امتدادًا مع متاجر ثلاثية تجارية باستخدام مجموعات بيانات حقيقية. تظهر النتائج التجريبية تحسنًا كبيرًا في أداء (Spatial-Qdag) المكاني المشغلين المكانيين في معظم الاستفسارات أثناء استخدام النهج المقترح. علاوة على ذلك ، تظهر النتائج أن تفوق الامتداد المقترح مقارنة بالمنافسة (Spatial-Qdag).

الكلمات الرئيسية:

البيانات المكانية ، الأداء ، قابلية التوسع ، SPARQL ، RDF ، البيانات الضخمة

THÈSE PRÉPARÉE DANS LE CADRE D'UNE COTUTELLE ENTRE L'UNIVERSIT E ABU BEKR BELKAID TLEMCEN ET L´ECOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE