Spatial Query Optimization and Distributed Data Server - Application in the Management of Big Astronomical Surveys

Mariem Brahem

université PARIS-SACLAY

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

NNT : 2019SACLV009

Thèse de doctorat

Optimisation de requêtes spatiales et serveur de données distribué - Application à la gestion de masses de données en astronomie

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université de Versailles-Saint-Quentin-en-Yvelines

Ecole doctorale n°580 sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Versailles, le 31 Janvier 2019, par

MARIEM BRAHEM

Composition du Jury :

Bernd AMANN Professeur, Sorbonne Université	Président
Farouk TOUMANI Professeur, Université Clermont Auvergne	Rapporteur
Laurent D'ORAZIO Professeur, Université de Rennes	Rapporteur
Bruno DEFUDE Professeur, Telecom SudParis	Examinateur
Karine ZEITOUNI Professeur, UVSQ	Directrice de thèse
Laurent YEH Maître de Conférences, UVSQ	Co-encadrant de thèse
Véronique VALETTE Chef de Projet, CNES	Invitée

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Optimisation de requêtes spatiales et serveur de données distribué - Application à la gestion de masses de données en astronomie

Mots clés : Bases de données astronomiques, Big Data, Optimisation de requêtes, Systèmes distribués, Partitionnement, Spark

Résumé : Les masses de données scientifiques générées par les moyens d'observation modernes, dont l'observation spatiale, soulèvent des problèmes de performances récurrents, et ce malgré les avancées des systèmes distribués de gestion de données. Ceci est souvent lié à la complexité des systèmes et des paramètres qui impactent les performances et la difficulté d'adapter les méthodes d'accès au flot de données et de traitement. Cette thèse propose de nouvelles techniques d'optimisations logiques et physiques pour optimiser les plans d'exécution des requêtes astronomiques en utilisant des règles d'optimisation. Ces méthodes sont intégrées dans ASTROIDE, un système distribué pour le traitement de données astronomiques à grande échelle. ASTROIDE allie la scalabilité et l'efficacité en combinant les avantages du traitement distribué en utilisant Spark avec la pertinence d'un optimiseur de requêtes astronomiques. Il permet l'accès aux données à l'aide du langage de requêtes ADQL, couramment utilisé. Il implémente des algorithmes de requêtes astronomiques (cone search, kNN search, cross-match, et kNN join) en exploitant l'organisation physique des données proposée. En effet, ASTROIDE propose une méthode de partitionnement des données permettant un traitement efficace de ces requêtes grâce à l'équilibrage de la répartition des données et à l'élimination des partitions non pertinentes. Ce partitionnement utilise une technique d'indexation adaptée aux données astronomiques, afin de réduire le temps de traitement des requêtes.

Title : Spatial Query Optimization and Distributed Data Server - Application in the Management of Big Astronomical Surveys

Keywords : Astronomical Databases, Big Data, Query optimization, Distributed systems, Data partitioning, Spark

Abstract : The big scientific data generated by modern observation telescopes, raises recurring problems of performances, in spite of the advances in distributed data management systems. The main reasons are the complexity of the systems and the difficulty to adapt the access methods to the data. This thesis proposes new physical and logical optimizations to optimize execution plans of astronomical queries using transformation rules. These methods are integrated in ASTROIDE, a distributed system for large-scale astronomical data processing. ASTROIDE achieves scalability and efficiency by combining the benefits of distributed processing using Spark with the relevance of an astronomical query optimizer. It supports the data access using the query language ADQL that is commonly used. It implements astronomical query algorithms (cone search, kNN search, cross-match, and kNN join) tailored to the proposed physical data organization. Indeed, ASTROIDE offers a data partitioning technique that allows efficient processing of these queries by ensuring load balancing and eliminating irrelevant partitions. This partitioning uses an indexing technique adapted to astronomical data, in order to reduce query processing time.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Contents

List of Figures

List of Tables

List of Queries

Acknowledgments

First, I would like to express my sincere gratitude to my advisor Prof. Karine Zeitouni for her motivation, enthusiasm and immense knowledge. I have been extremely lucky to have such a tremendous mentor. On the academic level, she helped me to grow as a scientific researcher in the database systems area. Without her advice, intelligent ideas and solid experience, I may not have ever pursued this challenging work. On the personal level, she inspired me by her determination and hard-working. Since my master internship, she believed in me and encouraged me to strive for excellence. I am indebted to her for her confidence and assistance.

My special words of thanks should also go to my co-advisor Associate Prof. Laurent Yeh for always being so helpful and motivating. His constant guidance and cooperation have always kept me going ahead.

I would also like to acknowledge all my thesis committee members (Prof. Bernd Amann, Prof. Bruno Defude, Prof. Laurent D'Orazio, Prof. Farouk Toumani) for their guidance. I am thankful to Prof. Laurent D'Orazio and Prof. Farouk Toumani for reviewing my dissertation and the time they gave me to build constructive feed-backs. I greatly acknowledge Prof. Bruno Defude for examining my work. I am quite appreciative of Prof. Bernd Amann for agreeing to be part of my dissertation committee on such a short notice as a replacement for Prof. Christine Collet. Prof. Christine is a brilliant teacher and a pioneer in the database systems area, who unfortunately passed away a few days before the dissertation defense.

I express my heartfelt gratitude to Mme Véronique Vallette who was honorably invited to participate in my thesis committee. I appreciate her friendly nature and constant motivation. She offered me a great opportunity to collaborate with the Centre National d'Etudes Spatiales (CNES), a major player in Europe's space endeavours who co-funded my thesis.

My acknowledgement will never be complete without the special mention of all the members of the DAVID Lab, and in particular the ADAM team. My deep appreciation goes to Associate Prof. Stéphane Lopez for his invaluable feedback on my research and for always being so supportive of my work. I am also thankful to Associate Prof. Zoubida Kedad for her endless support and motivation. I am also thankful to Associate Prof. Yehia Taher and Associate Prof. Nicoleta Preda for their encouragement and advice. I would like to thank my lab mates (Livia, Redouane, Souheir, Jingwei, Ahmed, Mohammed, Alaa, Alexandros, Hafsa) for always being there and for supporting me. I am proud to say that my experience in the David lab was exciting and fun, and has energized me to continue in academic research.

Last but not least, I want to thank my precious family: my parents who encouraged me to have confidence in my abilities and for their constant support through the ups and downs, my sister Sarra and my brother Salmen for their endless motivation and emotional support. My most heartfelt thanks go to my dear husband for his unconditional love. And Finally, I dedicate this thesis to my children Essia and Elyes, my champions, who blessed me with a life of joy and happiness, who inspired me and have made me stronger.

Mariem Brahem

CHAPTER 1

Introduction

Contents

> Do not look at stars as bright spots only. Try to take in the vastness of the universe.
>
> Maria Mitchell

Astronomy is undergoing a large and unprecedented growth of astronomical data in both of volume and complexity. Advances in new instruments and extremely large sky surveys are creating massive datasets including billions of objects and Terabytes of data. This will enable new discoveries by identifying rare or new astronomical objects, exploring the universe and thus empowering astronomers. Analyzing these vast amounts of data, making new scientific discoveries and extracting knowledge from this data effectively pose a considerable challenge. Traditional astronomical analysis techniques are no longer adequate, not only because of data explosion, but also because of computational complexity of astronomical queries. In this chapter, we briefly present the nature of astronomical data and the complexity of queries. We also discuss the main research challenges in this dissertation and provide an overview of our contributions.

1.1 Motivation

Humans throughout History have looked up to the sky to answer fundamental questions of where we came from, navigate vast oceans, measure time and mark seasons. The sky inspired ancient civilizations and reshaped their view of the world. Early cultures believed that gods dictated their motions with celestial objects and tried to unveil these messages. Today, our understanding of the universe has progressed. New telescopes and detectors have led to ambitious and well-organized surveys of star positions. Advances in new instruments have offered more powerful telescopes than ever to astronomers. Thus, these surveys are moving into a petascale regime and creating massive datasets including billions of objects.

The Sloan Digital Sky Survey (SDSS) [1], the Large Synoptic Survey Telescope (LSST) [2] and the GAIA mission [3] by the European Space Agency (ESA) are the largest and most accurate three-dimensional maps of our Galaxy ever obtained in the history of astronomy.

The SDSS project was designed in the 1990s by James Gunn and his colleagues, it produces each night about 200 GB of data. The scientific impact of SDSS has been remarkable, it has dramatically enhanced our image of the milky way and has led to discoveries that revolutionized astronomy. Another important ongoing space observatory is the ESA mission called GAIA that observed the positions, distances and movements of more than 1 billion stars with unprecedented

precision. The mission is expected to last until December 2020 and has already produced several data releases.

Another promising project is the future LSST project, it is expected to produce a massive photometric and astrometric dataset of about 37 billion stars and galaxies. LSST will produce about 15 TB per night, leading to a total catalog size of 15 PB. The objective of this enormous data archive is to provide super-deeps views and discoveries of the universe. Such powerful project will produce data sizes that have never been handled by astronomers.

The National Academy of Sciences discussed the New Worlds and New Horizons in Astronomy [4] and recognized the growing scale of astronomical data. It defined Cyber-Discovery as an important aspect for giving meaning to the data. It remembered that many of the most far-reaching and revolutionary discoveries in astronomy were not solely the direct result of observations with telescopes. But, they also depend on the ways about how to analyze and process the data, and make testable predictions. *How will astronomy archives survive the data tsunami*, a question raised by Berriman et al. [5] and recognized the needs to engage and partner with computer scientists to support distributed processing of data and use optimization techniques.

Meanwhile, the analysis of such surveys is the basis of subsequent astronomical discoveries. Astronomers will be able to understand much more about the structure, properties and evolution of our Galaxy. Analyzing large amounts of astronomical data has been high on the list of priorities for astronomy development, it forms a vital constituent in the future success of any telescope program. The data generated from these projects need to be analyzed so that interesting phenomena can be identified for further scientific studies. Large volumes of data from highly productive space missions have to be efficiently stored and analyzed in such a way that astronomers maximize their scientific return from these missions. For example, cross-matching is a fundamental operation in astronomical data processing. Cross-matching allows correlating two catalogs, matching newly extracted set of objects with an existing catalog, or even tracking transients from a series of observations. This query, in a nutshell, is a join on a distance condition. It enables astronomers to identify and correlate objects belonging to different observations in order to make new scientific achievements by studying the temporal evolution of the sources or combining physical properties (such as the brightness of the stars). Astronomers will observe the same sky in other wavelengths and

combine the available observations. This would immensely help in making new explorations faster and easier and formulate new theories to plan further observations. Moreover, the competitiveness of these surveys strongly depends on the quality of the survey data management.

1.2 Characteristics of Astronomical Applications

1.2.1 Large Sky Surveys

A data avalanche is occurring in astronomy with datasets measured in Terabytes [6]. The sky is being surveyed with billions of stars giving their positions, magnitudes, and other properties. The large sky surveys presented above have become the principal data source in astronomy, leading to a new golden era of astronomical discoveries. The important increase in data volume is based on the great progress in technology including advanced telescopes which follow an exponential growth and fulfill the Moore's law in their data-generation. Most sky surveys are generated in digital form and make either spectroscopic, astrometric or photometric observations.

1.2.2 Compute Intensive Queries

Astronomical queries are potentially computational intensive. They involve costly geometric computation. For example, kNN join queries (which combine each object in a dataset R with the k closest objects in another dataset S) need to compute the distance between each pair of objects from R and S, this usually lead to a cross product between billion-scale catalogs of complexity of $O(|R|.|S|)$. The cross product and the spatial refinement operation to check distance between pairs of stars are very expensive to process, due to the pairwise distance computation costs, along with the communication and I/O costs incurred. Thus, the design of an efficient and scalable querying system is needed for astronomical missions.

1.2.3 Complex Astronomical Queries

Astronomical queries are hard to express in current DBMSs. It is highly desirable that a given astronomical data management system provides an interface

language that integrates both astronomical and relational queries. At present, the Astronomical Data Query Language (ADQL) [7] is defined by the International Virtual Observatory Alliance (IVOA) as a standard to query astronomical data. ADQL can express cross-matching between tables and complex search criteria with geometric functions (such as `DISTANCE`, `CONTAINS` ...) and geometry data types (such as `POINT`, `CIRCLE` ...). ADQL supports both geometric and conventional non-spatial predicates. Therefore, it is needed to provide an efficient query processing engine with a simple, clear and unified interface for accessing astronomical data using ADQL.

1.2.4 Use of Spherical Coordinates

There are different kinds of spherical coordinate systems that are used in astronomy to uniquely determine the position of the stars on the sky including horizon system, celestial system, ecliptic system, galactic system and super galactic system. However, the most used one is the celestial equatorial system. The current standard celestial reference system adopted by the International Astronomical Union (IAU) is the International Celestial Reference System (ICRS) where the sky is projected on a celestial sphere with the earth in the center (see Figure 1.1). Using ICRS, the position of a star is recorded as two-dimensional coordinates, namely the right ascension `ra` and the declination `dec`. The ranges of `ra` and dec are $[0°, 360°]$ and $[-90°, 90°]$ respectively. `ra` is the celestial equivalent of terrestrial longitude. It measures the angular distance eastward along the celestial equator. While `dec` is the latitude-like coordinate, it measures the angular distance of a celestial object north or south of the celestial equator. Spatial queries are typically optimized using spatial indices in a Cartesian space. But these indices do not cope well with the spherical coordinates.

1.2.5 Use of Spherical Distance

Common astronomical queries use spherical distance as a spatial filtering step. Some astronomical queries involve spatial joins according to a spherical distance, which can be a very expensive and complex operation to process.

The spherical distance (see Definition 3.2.2) reflects the distance of the shortest path between two points along the surface of the sphere. It uses trigonometric

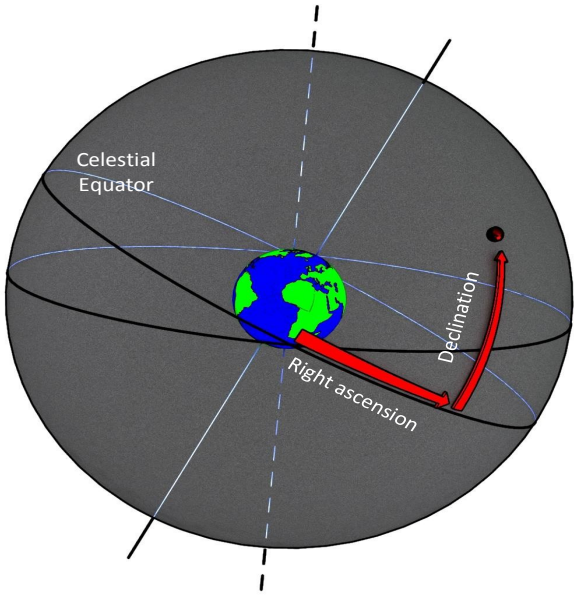


Figure 1.1 Spherical Coordinates.

functions for calculating the distance between two celestial objects. This subroutine consumes more time when compared with the euclidean distance that uses multiplications and additions.

1.3 Problem Statement

The growing scale of astronomical data and the increased accuracy of observation tools bring a change of paradigm for data processing. Also, most astronomical operations are very expensive to process because of their compute-intensive nature [6] especially for complex and costly operations. The cross-match of astronomical catalogs is a typical example commonly used by astronomers to correlate objects belonging to different observations. Through this operation, they can integrate catalogs from several instruments observed at different points in time, combine physical properties, or study the temporal evolution of the sources.

In this respect, the growing scale of observed surveys coupled with the compute-intensive nature of astronomical operations require a scalable solution that provides full-fledged spatial data exploration.

However, the specific context of astronomical data handling brings up many questions:

- What is the state-of-the-art on the handling of astronomical data ?
- How to exploit big data technologies to pursue astronomical data challenges ?
- How to build a scalable architecture for astronomical data analytics ?
- How to interact with petabytes of astronomical data?
- How to reduce the computational complexity of astronomical operations ?
- How to provide the astronomers with an efficient support of astronomical queries, i.e. combining several operations, without worrying about query optimization ?

  To resolve some of the difficulties mentioned above, a common admitted solution is to apply data partitioning to parallelize query computation by distributing the data on different worker nodes. Astronomical data partitioning is particularly challenging due data skewness. It also necessitates the use of techniques that support the multi-dimensional nature of astronomical data. This raises new questions:

- How to efficiently partition astronomical data ?
- Which data indexing technique should be utilized ?

Traditional relational database systems are no longer adequate, not only because of data explosion, but also because of computational complexity of astronomical queries. They do not have the power to process or even store these large datasets.

Recently, the shared-nothing [8] type of parallel architecture, which uses commodity hardware, is becoming a de facto standard in massive data handling. In this context, the distributed in-memory computing framework Apache Spark [9] has emerged as a fast and general purpose engine for large-scale data processing in memory.

While Spark fits well the large scale nature of astronomical data, it does not provide native support of astronomical queries. Yet, users can rely on UDFs to process astronomical data. For instance, implementing a UDF to execute a spatial filtering or a cross-matching query is possible. However, this leads to an expensive query execution time because Spark considers UDFs as black boxes.

It is not able to optimize spatial search or cross-matching queries. Thus, Spark system invariably applies a full scan for the former, and a Cartesian product when matching collections, because they both involve UDFs. Therefore, advanced physical and logical query optimization techniques are needed in order to query astronomical data seamlessly and efficiently.

In the literature, many existing systems [10; 11; 12; 13] support spatial data and queries over distributed frameworks. These systems allow complex spatial queries and implement spatial indices such as R-tree and R^+-tree. However, they suffer from several limitations with respect to our needs.

- The main issue is the lack of an expressive query language adapted to the astronomical context.
- The proposed built-in functions are not adapted to the spherical coordinate system, which lead to erroneous query results. This is due to the difference between the spherical distance and the euclidean distance i.e., the difference between the length of the great circle arc and of the straight line between two points.
- The query performances remain limited due to unsuitable data partitioning scheme. This issue will be discussed in Chapter 6.

Therefore, there is a need to redesign astronomical operations while taking advantage of the steady progress in big data technologies and tools. The target system should provide an effective, fast and linearly scalable data management for sky surveys.

Indeed, there is a gap between existing distributed systems and astronomical data handling using astronomical query language. Our goal is to fill this gap and introduce a new framework for the management of large volume of astronomical data. Efficient optimization techniques to reduce search space and lighten the cost of distance computation are necessary.

1.4 Objectives and Contributions

This work has been motivated and supported by the DAVID lab of the University of Versailles SQY [1] and the Centre National d'Etudes Spatiales (CNES) [2].

[1] http://www.uvsq.fr/
[2] https://cnes.fr/

The main objective of this dissertation is to explore the research challenges for providing a high performance engine for querying astronomical data, to propose novel methods for high level query support and optimization, and to implement the proposal in a framework that provides support for these queries. Our contribution lies in the implementation of the ADQL language and its optimization within ASTROIDE, an open source framework that we developed based on modern big data technology. Our design takes full advantage of the extensibility features of Spark [9], without changing its source code, which allows more flexibility and portability on new versions of the underlying system. Precisely, the support of astronomical data involves extending several levels of Spark. At the language level, the query language is extended to enable astronomical functions. We also extend the Dataframe API with the same functions at the programming level. In ASTROIDE, queries are expressed using ADQL [7], an SQL-Like language with astronomical functions. At the storage level, more appropriate techniques are needed for organizing and accessing astronomical data. ASTROIDE implements a data partitioner that achieves both spatial locality and load balancing. It also adopts a well-known sky pixelization technique, combined with a spherical space filling curve scheme, namely HEALPix [14], to achieve high performance query execution. At the intermediate level, the query language is parsed and optimized seamlessly. Various logical and physical optimization techniques for the ADQL execution are proposed, and integrated into Spark SQL [15] thanks to the extensibility of its optimizer. This includes the control of the data partitioning mechanism and spatial indexing as well as the customization of the query plan. Our query optimizer is responsible for generating an efficient execution plan for the given ADQL query; it injects spatial-aware optimization, which avoids reading unnecessary data as much as possible; it evaluates and selects the best plan. Therefore, the optimizer has been extended with various rules tailored for astronomical queries.

In a nutshell, this dissertation includes the following contributions:

- **Support of astronomical query processing over Spark**. This work is the first study that explores the extension of Spark to process astronomical queries. It provides a detailed system architecture and its evaluation on real data in order to show its efficiency and its scalability.
- **A comprehensive study of big data management frameworks**. This dissertation discusses the ability of existing big data management frameworks

to support astronomical data and presents new systems that have been implemented to support geo-spatial data.

- **Index and partitioning support for astronomical data.** Our system combines data partitioning with an indexing technique adapted to astronomical data (HEALPix pixelization), in order to speed-up query processing time.

- **Support of the most representative astronomical operators.** We design efficient and scalable cone search, kNN search, cross-match, and kNN join algorithms tailored to our physical data organization.

- **High-level data access.** Our framework supports the data access and query using the astronomical query language ADQL that is recommended by the IVOA.

- **Astronomical query optimization.** We extend the Spark Catalyst [16] optimizer and exploit partition pruning for astronomical operators. We introduce new physical and logical optimizations to optimize query execution plans using transformation rules and strategies.

1.5 Dissertation Outline

This dissertation is composed of seven chapters. Figure 1.2 outlines the organization of this dissertation. The current chapter introduces the general context, describes the motivations and summarizes our contributions. Chapter 2 studies the state-of-the-art. We present existing technologies for performing distributed processing on big data and discuss their ability to deal with huge volume of astronomical data. We also describe the state-of-the-art approaches for distributed spatial query processing. Inspired by the challenges presented in this Chapter and the limitations of existing systems, we present in Chapter 3 the overall architecture of our proposal called ASTROIDE from the physical level to the high level language. We also tackle some important background and definitions that are used in this dissertation. Chapter 4 and Chapter 5 are dedicated to explain more details about ASTROIDE. In Chapter 4, we study partitioning and indexing methods for astronomical query processing. Then, we present our partitioning approach based on HEALPix, a sky indexing scheme tailored for astronomical

data. In Chapter 5, we present our query processing module that exploits partition pruning for astronomical operators. We introduce transformations rules to optimize astronomical query execution. Chapter 6 provides an experimental study of ASTROIDE performances in details. We compare the effectiveness and the efficiency of ASTROIDE with the closest prototype in the state-of-the-art. Finally, we conclude this thesis in Chapter 7 by summarizing our main contributions and indicating some interesting perspectives.

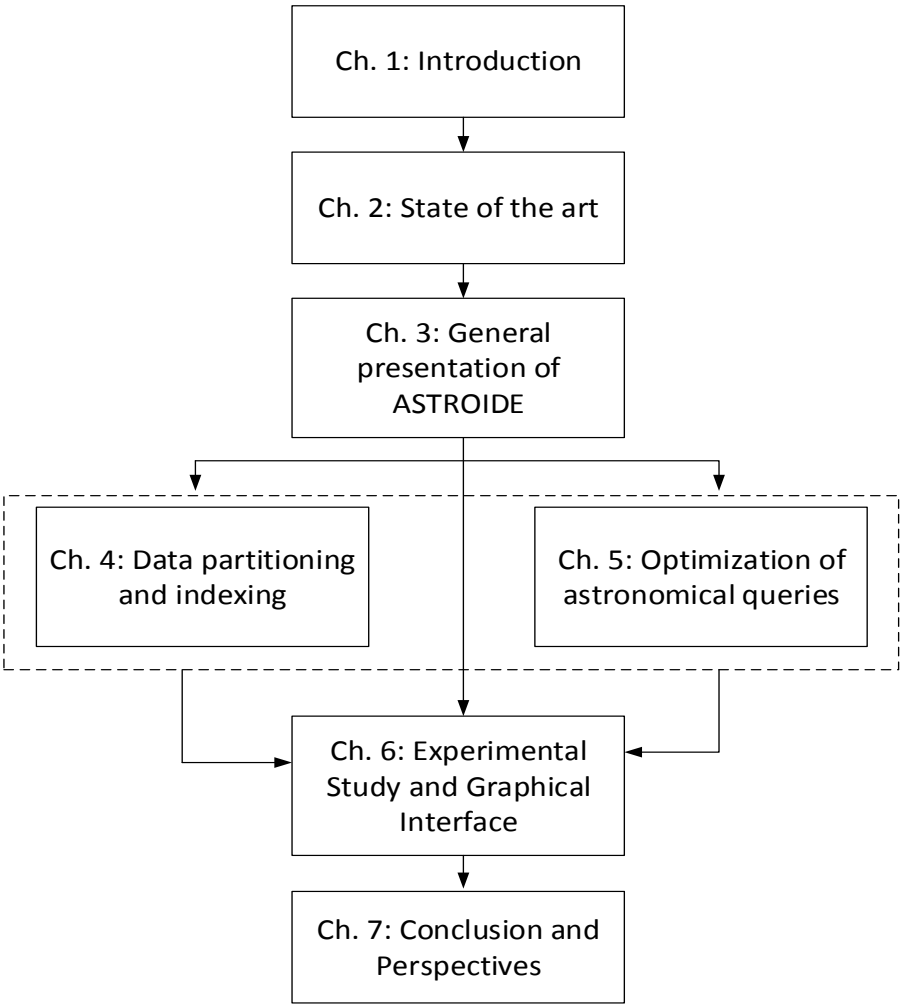


Figure 1.2 Organization of the Dissertation.

CHAPTER 2

State of the Art

Contents

The world is one big data problem.

Andrew McAfee

2.1 Introduction

Over the past few years, data deluge has become a reality. The amount of data generated is astonished, resulting in what is called big data. Big data refers to large amounts of data for which traditional database management tools have become inefficient in terms of storage, processing and analysis. The International Data Corporation (IDC), a pioneer in studying big data defines big data in 2011 report [17] as:

Big data describes a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis.

This definition announces that big data is characterized by 4V's: Volume, Velocity, Variety and Value. Volume refers to the vast amounts of data that are being generated whereas Velocity is the rate of growth and how fast the data is gathered. Variety provides information about the types of data such as structured, unstructured and semi-structured. However, Value refers to the outcome and added-value that the collected data can bring. Given its popularity, many definitions [18; 19; 20] are found for big data and reaching a consensus is difficult. The above definition of big data allows us to compare big data management with traditional data management. First, traditional data management tools are not designed to manage such large volume of data. Second, big data comes with different types whereas traditional data is typically structured and can be easily stored and processed.

Thus, the management of big data has revealed new techniques and technologies to handle the characteristics of big data. In this chapter, we study the existing big-data systems which include a set of tools and mechanisms to load, extract and perform parallel query processing. We also evaluate the ability of these systems to support huge volume of astronomical data and describe the state-of-the-art approaches for spatial query processing.

2.2 Big Data Management

Big data management is a broad practice that encompasses the policies, procedures and technology used for the collection, storage, organization, administration and delivery of large repositories of data. In particular, The National Institute

of Standards and Technology (NIST) categorized big data management into big data science and big data frameworks [21]. Big data science is *the study of techniques covering the acquisition, conditioning, and evaluation of big data*, whereas big data frameworks are *software libraries along with their associated algorithms that enable distributed processing and analysis of big data problems across clusters of computer units*. It concerns the convenient organization of data for the efficient process of large volume. In the following subsections, we describe relational DBMSs and discuss their limitations to cope with big data challenges and present the most popular big data frameworks.

2.2.1 Relational DBMSs

Since they were invented by Edgar Codd in 1970, Relational Database Management Systems (RDBMSs) have been the dominant model for managing, organizing, and retrieving data. They are based on a relational model that stores data in form of relational tables and queries structured data. The main focus of Transactional RDBMS is on ACID properties [22]:

- Atomicity - Transaction is either completely done or it is completely rolled back.
- Consistency - Every transaction is subject to a consistent set of rules.
- Isolation - No transaction should interfere with another transaction.
- Durability - Committed changes are never lost.

The main issue of RDBMSs is that they cannot address the variety and scalability required by big data. The volumes of data generated cannot be managed by a RDBMS. The Relational Data Model defines only tabular data structure and it does not allow representing new needs such as graph. In addition, RDBMSs are increasingly utilizing more and more expensive hardware [23]. Hence, several solutions to manage big data came into existence to satisfy these challenges. Distributed systems provide the basis of those technologies [24].

2.2.2 Hadoop MapReduce

MapReduce was invented by Google to index the large volume of data on the web. MapReduce [25] is the dominant model for batch processing. The key

idea is that data is divided into partitions and these partitions are processed in parallel, assuming that data is stored in a distributed file system. MapReduce refers to two separate steps: *map* and *reduce*. The *map* phase takes an input pair and produces a set of intermediate key/value pairs:

$$map : (K_1, V_1) \rightarrow list(K_2, V_2)$$

The intermediate values associated with the same key K_2 are grouped together and passed to the *reduce* function. The *reduce* is called for each key K_2 and a list of values for that key to merge these values and generate a possibly smaller list of values:

$$reduce : (K_2, list(V_2)) \rightarrow list(K_3, V_3)$$

The process of transferring data from the map task to the reduce task is known as the shuffle. Figure 2.1 describes a workflow of a common MapReduce job.

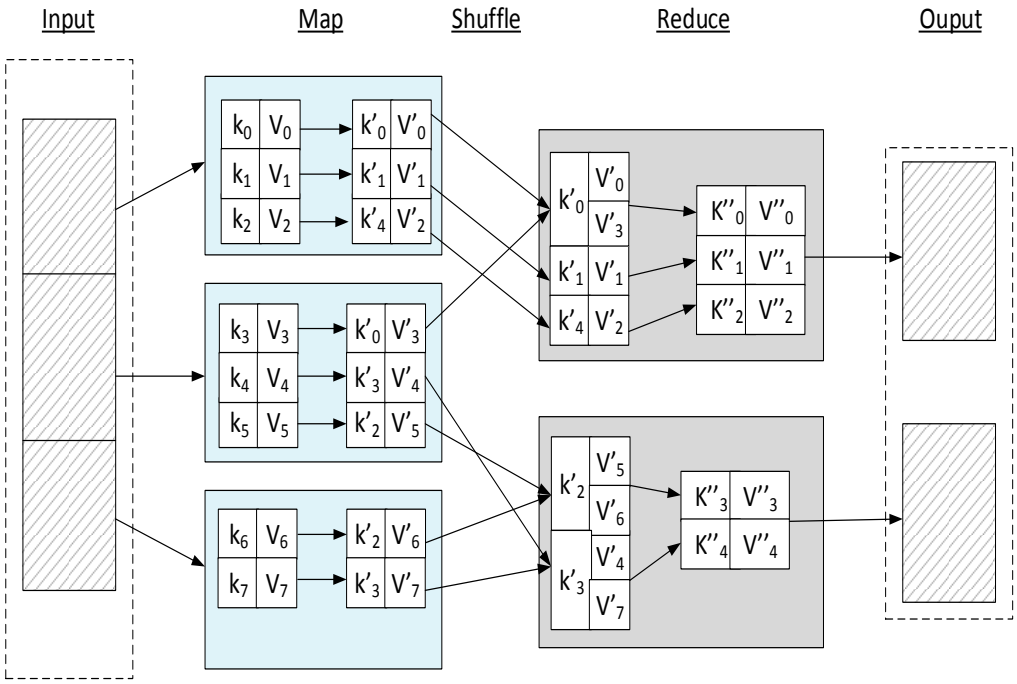


Figure 2.1 Workflow of a MapReduce Job.

One of the most used implementation of the MapReduce model is Apache Hadoop [26]. Hadoop has long been the mainstay of the big data movement.

Apache Hadoop is an open-source software framework that supports large data storage and processing. Hadoop is suitable for big data management because of the following advantages:

- Scalability. This is the major force that drives Hadoop to popularity. It is possible to scale out a Hadoop cluster by adding more nodes. Indeed, adding commodity servers allows Hadoop to scale computation and storage capacities. Hadoop is also able to ensure data locality by pushing operations to nodes on which data resides.
- Cost efficiency. Hadoop uses commodity hardware as a node, this can decrease costs and makes the process of massive data affordable. Hadoop is also able to adapt the processing load according to the power of each node.
- Fault tolerance. If any node goes down, data retrieval can be done through other nodes due to data replication.

Moreover, the Apache Hadoop software is composed of additional modules: Hadoop Distributed File System (HDFS) [27] and Hadoop YARN as the resource's manager. HDFS is the primary data storage system of Hadoop applications. However, other data storage systems can be integrated into the Hadoop framework such as Amazon S3 [28].

HDFS is designed to reliably store very large files across machines in a large cluster. HDFS adopts a master-slave architecture as shown in Figure 2.2 in which the NameNode manages the file system metadata and many DataNodes store the data. The NameNode is a central part as it is responsible for the coordination in the cluster. It allocates resource storage and splits each file into blocks. These blocks are stored in a set of DataNodes. HDFS creates multiple replicas of data blocks in different DataNodes. This increases availability because even if some nodes fail, data can still be accessed by retrieving one of the copies stored in other nodes. HDFS achieves efficiency by distributing data which allows parallel processing on the nodes where the data is located. DataNodes creates, deletes or replicates the actual data blocks based on the instructions received from the NameNode. By default, a HDFS block size is 64 MB and HDFS stores three copies of each data block. Thus, Hadoop provides a reliable shared storage and

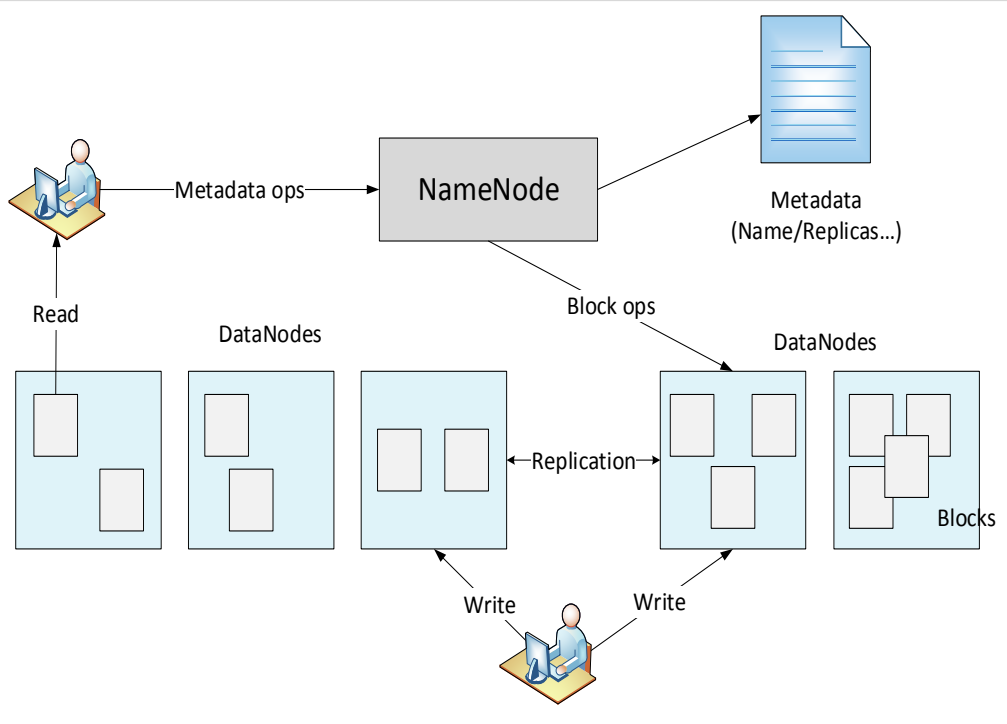


Figure 2.2 HDFS Architecture.

analysis system. The storage is provided by HDFS, and analysis by MapReduce [29].

There are multiple open source projects built on top of Hadoop such as HBase [30], Hive [31] and Pig [32] that we present in the following subsections. These systems are an overlay of Hadoop Map-Reduce, they inherit its main characteristic of scalability to handle massive datasets.

2.2.3 NoSQL-on-Hadoop systems

NoSQL is short for Not Only SQL, it addresses several issues that the relational model is not designed to address. NoSQL systems are designed for large-scale data storage and massively-parallel data processing. They are able to horizontally scale, to replicate and distribute data over many servers and to offer a flexible data schema. Besides, they give up some of the ACID constraints to improve performances. When talking about NoSQL, three basic requirements are involved: Consistency, Availability, Partition Tolerance, known as CAP properties.

- Consistency - Different from the definition of ACID properties. It means that

all nodes see the same data at the same time.

- Availability - Guarantees an answer for each query.
- Partition Tolerance - Operations will complete, even if individual components are unavailable.

However, it is impossible to fulfill all three requirements. Thus, Brewer's CAP theorem [33] states that a NoSQL system has to follow two of these three requirements.

NoSQL systems were classified by Leavitt into four categories [34]:

- Key-value stores. A system that stores values indexed for retrieval by keys (e.g., SimpleDB [35]).
- Column-oriented databases. A system that stores data in a structured table of columns and rows with uniform sized fields for each record (e.g., HBase [30]).
- Document-based stores. A system that stores data as collections of documents (e.g., MongoDB [36]).
- Graph databases. A system that stores data as a graph. A graph data structure consists of a finite set of ordered pairs, called edges, and entities called nodes. (e.g., Neo4j [37])

Here, we give details about HBase as we are interested in the combination of NoSQL databases and Hadoop in this subsection. HBase is considered as the Hadoop database, it is one of the most popular NoSQL systems that combines the scalability of Hadoop with real-time data access. But, there are many other popular NoSQL databases such as Cassandra [38].

HBase

HBase [30] is a non-relational, distributed database that runs on top of Hadoop and HDFS. HBase is designed to support high table-update rates and to scale out horizontally in distributed computing environment. HBase supports massively parallelized processing via MapReduce for using HBase as both source and sink. It is known for providing strict consistency on reads and writes, which

distinguishes it from other NoSQL databases. In addition, it provides pushdown predicates which improves query performances.

Each HBase table (see Figure 2.3) is stored as a sparse multidimensional sorted map, with rows and columns, different from the structure of tables in conventional relational DBMSs. Each row has a unique row key and an arbitrary number of columns denoted as table cells.

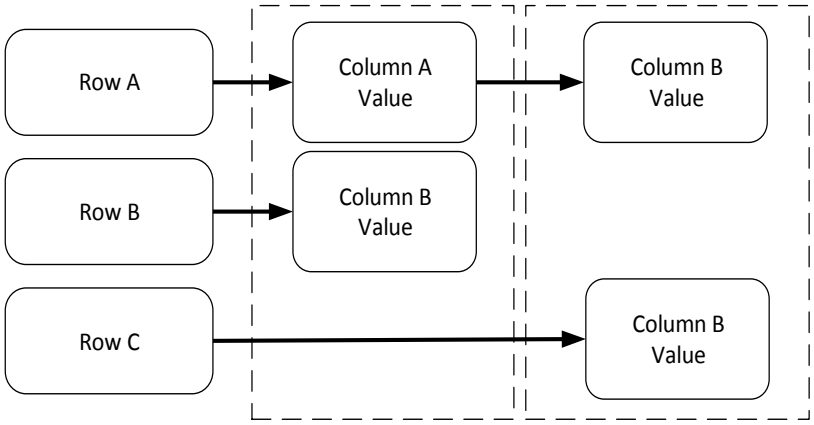


Figure 2.3 HBase Table.

2.2.4 SQL-on-Hadoop Systems

Hive

Hive [31] is a data warehouse system that enables reading, writing, and managing large datasets in distributed storage. It is an SQL-to-MapReduce translator with an SQL dialect. Hive is designed to exploit the scalability of Hadoop while presenting a familiar SQL abstraction. It allows maximizing scalability, performance, extensibility and fault-tolerance. Hive's SQL can be extended via user defined functions (UDFs), user defined aggregates (UDAFs), and user defined table functions (UDTFs). Hive provides the following features:

- Easy access to data via SQL, thus enabling data warehousing tasks such as reporting, and data analysis.
- Access to files stored either directly in HDFS or in other data storage systems such as HBase.

- Query execution via Apache Tez, Apache Spark, or MapReduce procedural language.

Pig

Apache Pig [32] is an open-source platform for analyzing large datasets and representing them as data flows. It consists of a high-level language for expressing data analysis programs, coupled with an infrastructure for evaluating these programs. It enables complex data transformations, aggregations, and analysis using a command language called Pig Latin that are compiled into MapReduce jobs and executed on Hadoop. Pig combines a command based language that is close to a relational algebraic language with operators that are oriented to data analysis (e.g., *coGroup*).

The main features of Pig are:

- Rich set of operators. Pig has a rich set of built-in operators like join, filter ...
- Optimization. Pig optimizes queries before submitting them to Hadoop MapReduce for execution. This allows users to concentrate on semantics rather than creating mapper and reducer functions.
- Extensibility. Pig allows users to write UDFs.
- Lazy evaluation. Pig runs commands only when the output is requested by the user.

2.2.5 Apache Spark

Spark Core

Apache Spark is a new generation tool for big data processing. It was developed by researchers at the University of California at Berkeley in 2009 and open sourced in 2010. Spark [9] is rising in popularity as an alternative to disk oriented processing, due to its ability for executing computations in memory and its expressive API for data processing. It was designed to overcome the disk I/O limitations and to improve the performance of earlier systems. Rather than specifying *map* and *reduce* steps, Spark applies an entire series of data flow transformations

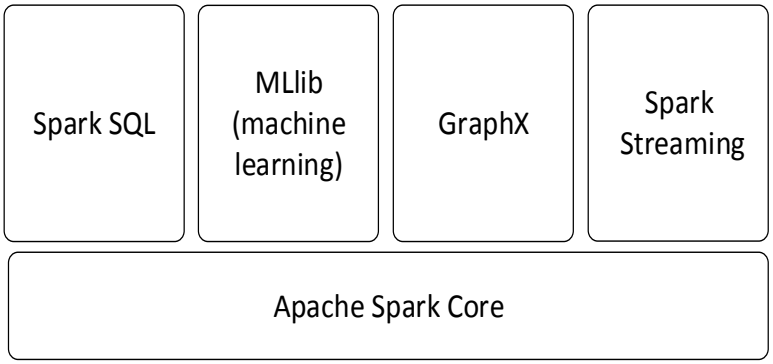


Figure 2.4 Spark Ecosystem.

on the input data. Apache Spark is a fast and general purpose engine for large scale data processing. It combines MapReduce-like capabilities for batch programming, real-time data-processing functions, SQL-like handling for structured data, graph algorithms, and machine learning in a single framework. Spark can run on an existing Hadoop cluster and access any Hadoop data source, including HDFS or HBase. Apache Spark system introduces new upper-level libraries including Spark's MLlib for machine learning, GraphX for graph analysis, Spark Streaming for stream processing and Spark SQL for structured data processing (Figure 2.4).

The entry point to all applications in Spark is the SparkContext (see Figure 2.5). The driver is the host of the SparkContext, it is the workload coordinator in a Spark application, it communicates with each node in the cluster through a cluster manager. The cluster manager analyzes the workload defined by the user, allocate resources and subsequently distributes tasks between the nodes of cluster. Several types of cluster managers: Spark Standalone mode, Mesos or YARN. Once a cluster manager is connected, Spark can begin acquiring executors on the nodes in the cluster. Executors are the processes that do the actual work and run the computations.

Spark mainly offers an immutable distributed collection of objects for in-memory cluster computing called Resilient Distributed Dataset (RDD) [39], which provides an efficient data sharing between computations. An RDD is a read-only, partitioned collection of records. RDDs provide fault-tolerant, parallel data structures that let users store data explicitly on disk or in memory, control its partitioning and manipulate it using a rich set of operators. RDDs are split into multiple

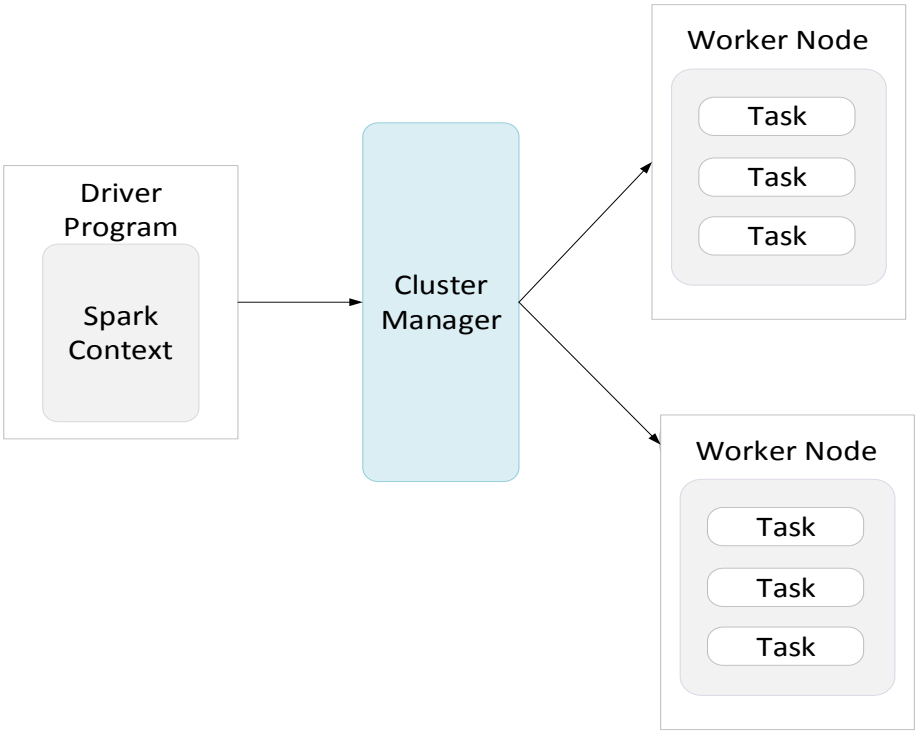


Figure 2.5 Spark Architecture.

partitions and operated on in parallel across processing nodes.

In addition to RDDs, Spark supports two types of operations: transformations and actions. Transformations (e.g., *filter* or *map*) are operations that define (virtually produce) a new RDD by performing some useful data manipulation on another RDD. Actions (e.g., *count* or *foreach*) launch the computation on an RDD by applying the specified set of transformations. Transformations are evaluated lazily, meaning that computation does not take place until an action is invoked. This design enhances the power of Apache Spark since it combines similar transformations into a single operation during query execution.

Once an action is called, Spark builds a Directed Acyclic Graph (DAG) of stages that need to be executed in order to compute the action. A DAG consists of vertices (nodes) and edges (lines). Vertices represent the RDDs and the edges represent the dependencies. Next, it splits the DAG into stages that contain pipelined transformations. Further, it divides each stage into tasks to run in parallel on separate machines. Spark executes all tasks within a stage before moving on to the next stage. With transformations and actions, computations can be organized into multiple stages of a processing pipeline. Every time a transfor-

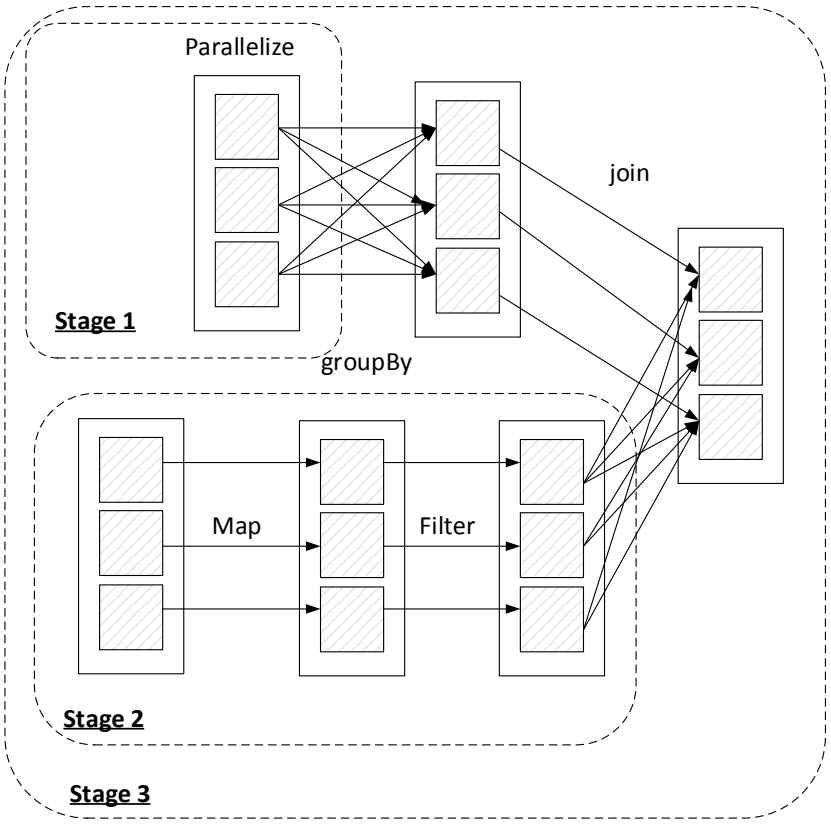


Figure 2.6 Types of Transformation in Spark.

mation is performed on an RDD, a new vertex (a new RDD) and a new edge (a dependency) are created. The DAG can be optimized by re-arranging and combining some operations in a stage when it is possible. This model allows Spark to form consecutive computation stages. Thus, query execution is optimized by minimizing data shuffling. There are two kinds of transformations (as shown in Figure 2.6) that can be applied in Spark: narrow transformations (e.g., *map*, *filter*) and wide transformations (e.g., *groupBy*). They determine whether a shuffle will be performed. If no data transfer between partitions is required, a narrow dependency is created. A wide dependency is created in the opposite case, which means a shuffle is performed [40] which requires the data to be redistributed across the partitions.

DAGScheduler (Figure 2.7) is the scheduling layer of Apache Spark that implements stage-oriented scheduling [41]. The fundamental concepts of DAGScheduler are jobs and stages. A job is a top-level work item submitted to DAGScheduler to compute the result of an action. Each Spark job corresponds to one action, and each action is called by the driver program of a Spark application. Every job

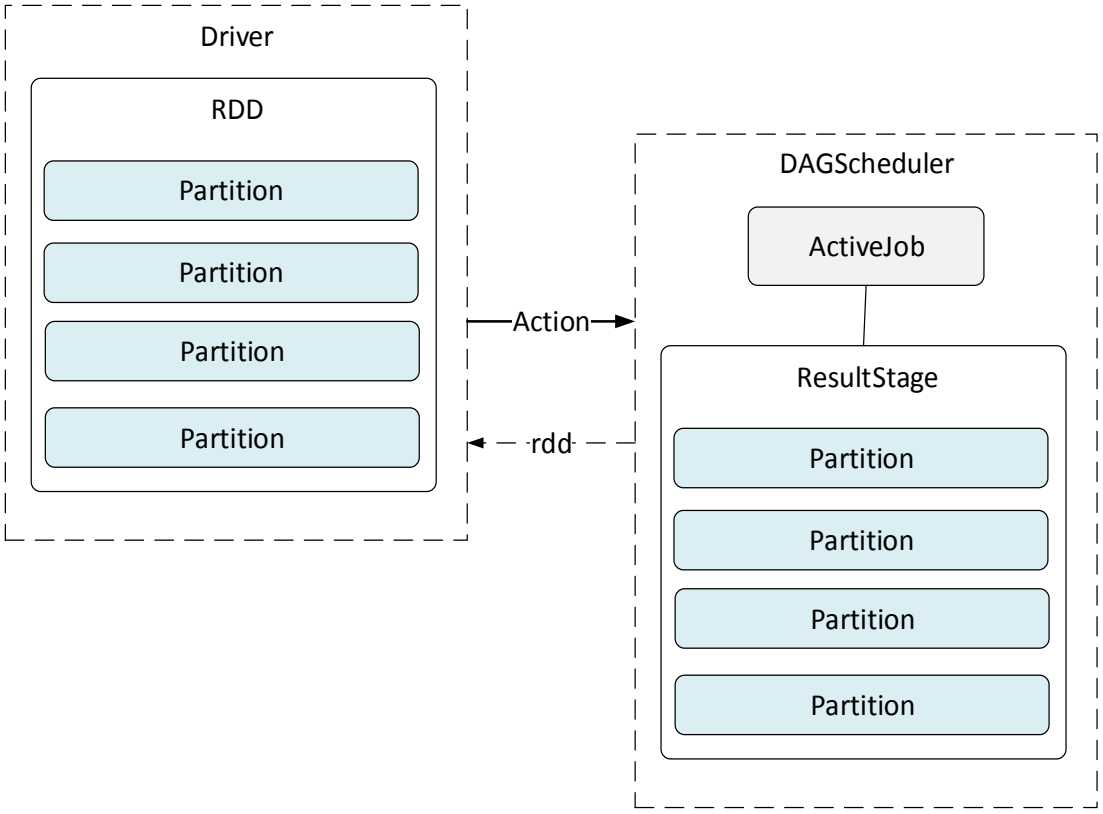


Figure 2.7 DAGScheduler

requires computation of multiple stages to produce the final result. A stage is a physical unit of execution. Each stage is a set of parallel tasks, that correspond to a parallelizable unit of computation done in each stage. There is one task for each partition in the resulting RDD of that stage.

Spark SQL

Spark SQL [15], a component on top of Spark Core, enables the support of relational processing within Spark programs (on native RDDs). Shark [42] was an older SQL-on-Spark project that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark SQL introduces new data abstractions called Dataset (a distributed collection of data) and DataFrame. A Dataset provides the benefits of RDDs and Spark SQL's optimized execution engine. A DataFrame represents a Dataset organized into named columns, unlike RDDs. This provides Spark with more in-

formation about the structure of both the data and the computation. Spark SQL is part of the Apache Spark framework, it allows to run SQL queries on Spark data. The Spark SQL module supports multiple input data formats including Parquet and JSON to allow data to be stored in formats that better represent data. This offers more options to integrate with external systems [43].

Spark SQL provides also an extensible query optimizer called Catalyst [16], which makes adding new optimization techniques easy. Catalyst can be extended in two ways: by defining custom rules, and by adding optimization strategies. Most of the rules are heuristics-based. For example, predicate pushdown is a rule to reduce the number of the qualified records before performing a join operation and project pruning is a rule to reduce the number of the participating columns before further processing.

Spark SQL is agnostic of astronomical query dialect and optimization techniques. Nevertheless, it offers many advantages:

- Queries are integrated with Spark programs. Spark SQL allows to query structured data inside Spark programs, using SQL or a DataFrame API.
- A superior performance compared to other distributed systems
- Its optimizer is powerful and easily extensible.
- It comes with higher-level libraries for advanced analytics.

Catalyst

Catalyst is the optimizer used by Spark SQL. It optimizes all the queries written in SQL and DataFrames using Spark. Catalyst [16] mainly represents the logical plans as trees and sequentially applies a number of optimization rules to manipulate them. Recently, a cost based optimization module (CBO) [44] has been introduced in Catalyst. This module analyzes all execution plans, assigns a cost to each plan and chooses the lowest cost for the physical execution.

The Catalyst process consists of several steps: Analysis, Logical Optimization, Physical Planning and Code Generation.

- Analysis: Catalyst starts from a relation to be computed, either from an abstract syntax tree (AST) returned by the SQL parser, or from a DataFrame object constructed using the API. Analysis allows to:

  - Search relations by names.
  - Map named attributes to the input.
  - Assign a uniqueID to attributes matching the same value.

- Logical Optimization: The logical optimization phase applies standard rule-based optimizations to the logical plan including predicate pushdown, projection pruning, Boolean expression simplification, and other rules. For example, Catalyst can push operations from the logical plan into data sources so that subsequent operations work on smallest datasets.

- Physical Planning: In this step, Catalyst takes a logical plan as input and generates one or more physical plans. After that, it chooses a plan using a cost model. Being a new technology, cost-based optimization is very limited in Catalyst. It is only used to select join algorithms. For example, Catalyst uses broadcast join instead of hash join to optimize join queries when the size of one side data is small. Richer cost based optimization will be implemented in future Spark SQL versions.

- Code Generation: The code generation represents the final point of query optimization where tasks are executed on RDDs. This phase involves generating Java bytecode to run on each machine using a feature of the Scala language, quasiquotes [45]. These quasiquotes allow the programmatic construction of Abstract Syntax Rrees (ASTs) in the Scala language.

2.2.6 Discussion

In this section, we presented the MapReduce framework that offers a powerful programming model and enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. We also introduced tools that allow high-level abstractions of MapReduce with Hive and Pig. Hive provides a declarative SQL based language to query data. Pig provides a procedural interface for writing queries. However, Spark SQL allows the combination of procedural programming and declarative query language. The DataFrame API offers rich relational/procedural integration within Spark programs. Spark SQL introduces also an extensible query optimizer (Catalyst) that allows optimization rules integration.

Spark has drawn a great potential due to its in-memory computing capabilities. However, it is not optimized for astronomical queries that involve geometric computations. It lacks effective partitioning techniques adapted to the spherical space to avoid data skew and balance tasks across nodes. Furthermore, Spark allows to express queries using an SQL interface which is not convenient to express astronomical queries.

In [46], the authors designed a benchmark to report the ability of existing MapReduce based systems to support large scale declarative queries in the area of cosmology. They defined a set of SQL queries and evaluated the performances of these systems in the management (e.g., storage, loading) of astronomical data and their capabilities (i.e., indexing, compression, buffering, and partitioning) to optimize queries. They used datasets simulated in the context of the LSST project and investigated two systems as use cases: Hive and HadoopDB, a Hadoop/ PostgreSQL based system. The aforementioned benchmark highlighted the need for new techniques for astronomical query optimization and partitioning in existing systems. However, the query cases differ from our context, they do not cover queries involving geometrical predicates, which are typical in astronomical applications. In addition, the benchmark does not incorporate any memory oriented distributed computing solution, like Spark.

2.3 Astronomical Servers

Over time, the methods used to collect astronomical data have substantially changed, from hand-drawn illustrations to massive data collected from advanced telescopes. Thus, astronomy was transformed from an observational science into a digital and computational science [47] due breakthroughs in instruments, detectors and computer technologies.

The urgency for new tools to enable data-intensive research has been raised by Jim Gray in 2007. He pointed out that *experimental, theoretical, and computational science were all being affected by the data deluge, and a fourth "data-intensive" science paradigm was emerging*. He believed that these large volumes of data represent a fourth paradigm [48] within various scientific discipline including astronomy. Even though this problem was early announced by Gray, there have been little researches that address these issues due a general lack of understanding of these topics by the scientific community as discussed in [49]. In

the following subsections, we describe some of the existing researches that studied the management of astronomical data, and we highlight their limitations to cope with this data deluge.

2.3.1 SkyServer Project

Linking database and astronomy has been introduced in the SkyServer project [50; 51]. SkyServer is the primary public interface to interact with data from the Sloan Digital Sky Survey (SDSS) using SQL Server as the backend database storage system. It consists of an extensive web application developed in C# and ASP.NET. SkyServer allows interactive navigation through the sky and sophisticated techniques to query SDSS data.

The SkyServer defines 20 typical queries in astronomy that can be expressed using SQL. These queries correspond to typical tasks astronomers would execute to extract and analyze data from astronomical archives. For example, an astronomer is interested in finding all objects that correspond to asteroids. Such query necessitates a table scan, it needs to select objects that have high (but reasonable) velocity. Even if the idea of using databases was not usual to most astronomers, they grew quickly accustomed to SQL and appreciate the language expressivity for astronomical queries. Since 2001, the project has attracted a wide range of users [52; 53] with more than 70% of visits coming from non-astronomers. It has made it possible for anyone to navigate through the sky. SkyServer has revolutionized the interaction between telescopes, its data and user communities. However, complex SQL involving spatial joins and spatial queries were not part of the SkyServer project. This was the main barrier to the wider use of the SkyServer by the astronomy community.

2.3.2 VizieR Service

The VizieR service [54] developed by the Centre de Données de Strasbourg (CDS) is an on-line database for accessing astronomical data listed in a large number of published catalogs. This database was created in 1997 and improved in 2012 to provide better ergonomy and more interactions with users and external applications. The VizieR database allows searching data by content and provides a tool for cross-matching catalog tables from their coordinates. VizieR is based

on the usage of a relational DBMS. Data is stored in relational tables and a set of Meta tables are available to describe tables stored in VizieR such as the number of rows, how to access the actual data ...

To improve query execution, the VizieR service integrates a compression method [55] that stores data into binary files. It groups the objects of the tables by their positions, thus, contiguous records have quite similar values for `ra` and `dec`. It defines a reference position as the header of a group of records. This mechanism allows to index compressed binary files by positions and to reduce I/O costs. It also allows faster response time. To handle queries based on celestial positions, VizieR needs only to scan necessary data files. Several interfaces are currently available to query data stored in VizieR: directly from a Web browser, via a construction of the query using a standardized way of specifying queries in terms of HTTP requests defined by the Astronomical Standardized URL (ASU) conventions, or the developing of XML documents [1].

2.3.3 Q3C in PostgreSQL

Q3C [56], standing for Quad Tree Cube, provides a new sky indexing scheme that overcomes the limitations of HTM sky pixelization scheme in terms of complexity and open source project. The idea is similar to other sky-indexing schemes that we introduce in Section 4.4. However, the individual pixels of Q3C do not have equal areas (as HEALPix). The base of Q3C is the cube inscribed in the sphere, on each face of the cube a quad-tree is constructed. The quad-tree structure allows to create a mapping of the 2D coordinates in the square to an integer number. Since there are six faces, three bits indicating the face number are appended to establish a mapping between the cubes and integer numbers.

Q3C provides the mapping of each point of the sphere to an integer number called IPIX ensuring that nearby points on the sphere have nearby values. The IPIX values allow to create indices on astronomical tables that enable fast searches on the sphere. Every astronomical query is segmented on different pixels, each pixel represents a continuous range of IPIX. This allows to retrieve easily and quickly some regions of the sky with Q3C indices and reduce I/O costs.

Q3C with PostreSQL is an open source database for complex astronomical queries on the sphere for large data. It offers an SQL-Like interface for the main

[1] http://vizier.u-strasbg.fr/cgi-bin/asu-xml

astronomical queries: cone search, spatial searches on the sphere and cross-matches.

Q3C defines a set of specific functions such as *q3c_radial_query, q3c_join, q3c_dist ...* for the main astronomical queries. For example, the following query returns all objects within radius of 0.1° around the position (ra,dec) = (11,12) in the `gaia` table:

Query 2.1 Cone Search (Q3C)

```
SELECT *
FROM gaia
WHERE q3c_radial_query(ra, dec, 11, 12, 0.1);
```

or, to execute the cross-match of `gaia` with a second table `igsl` with a radius of 2 arc-seconds:

Query 2.2 Cross-Match (Q3C)

```
SELECT *
FROM gaia, igsl
WHERE q3c_join(gaia.ra, gaia.dec, igsl.ra, igsl.dec, 2/3600);
```

2.3.4 Open SkyQuery

Open SkyQuery [57; 58] is a web portal that allows querying and cross-matching distributed astronomical datasets using RDBMS technologies. Using Open SkyQuery, users can express their queries in ADQL. The authors propose zoning and partitioning algorithms for parallel query execution using RDBMS technologies. The basic idea is to map the sphere into stripes of a certain height called zones. Then, objects within a zone are stored on disk ordered by zoneID and right ascension using a traditional B-tree index to minimize the number of I/O operations. In order to process spatial searches, Open SkyQuery computes bounding boxes (B-tree ranges), followed by a distance test to discard false positives.

Open SkyQuery aims to provide individual access to astronomical data but the primary goal is to offer cross-matching service. It also introduces a simple optimization process that consists of ordering the cross-match workflow from the service hosting the smallest catalog to the service with the biggest catalog.

2.3.5 MonetDB/SkyServer

Ivanova et al. [59] extend MonetDB [60] to manage astronomical data. MonetDB is an open source database system based on a column store approach where relational tables are broken vertically. MonetDB reduces disk space and offers data compression which improves performance by spending less time in I/O. MonetDB/SkyServer retains the vertical fragmentation of MonetDB and optimizes astronomical queries by fetching only relevant columns from disk. It also offers a horizontal partitioning of data.

The goal of this project is to optimize a column-oriented database to enable the support of large-scale surveys. MonetDB/SkyServer has proven the great potential of MonetDB for the management of scientific databases. The SkyServer functions related to query capabilities were integrated into MonetDB. As a spatial access method, it uses the Zones algorithm because of its simple implementation in SQL. MonetDB/SkyServer enables also the use of the MonetDB/SQL's optimizer. It allows the front-end compiler to activate specific optimization techniques.

2.3.6 AscotDB

AscotDB [61] is build on the combination of three pieces of technology

- SciDB [62], a shared-nothing DBMS that stores data in distributed and multidimensional arrays. SciDB is designed for efficient computing over array data. It was adapted to support the spherical coordinates gathered by astronomical surveys.
- AStronomical COllaborative Toolkit (ASCOT) [63], developed in the astronomy community for graphical data exploration. ASCOT is a web based framework that facilitates collaboration among astronomers. It defines a dashboard of small tools, called gadgets, to share, explore and interact with large astronomical datasets.
- Python, for easy programmatic access.

AscotDB reuses the important features of ASCOT such as data visualization with gadgets and data sharing. However, it introduces new data analysis capabilities that enable users to manipulate raw pixel data. AscotDB stores astronomical

data inside SciDB and translates the queries defined in the graphical interface into operations over SciDB's arrays. SciDB transforms a query into a parse tree defined as a collection of operators in a tree structure, where data is divided into chunks and distributed across nodes of a cluster. To allow the management of spherical coordinates on SciDB, AscotDB adds a middleware layer that maps the spherical coordinates to HEALPix indices. AscotDB proposes also a specific mapping schema between the spherical grid introduced by HEALPix and the multidimensional array data structure on which SciDB is based.

2.3.7 Qserv

Qserv [64] is a distributed shared-nothing SQL database designed to manage the future LSST's data. It was developed during the R&D phase of LSST. Qserv relies on two open-source technologies: MySQL as an SQL execution engine and Xrootd [65] as a distributed file system. It allows to partition data into materialized chunks that are distributed across nodes. Each chunk is further divided into sub-chunks with overlaps. This overlapping data represent the partition's borders defined by a preset spatial distance. This creates a two-level partitioning structure and allows to compute spatial joins correctly, at the condition that the border margin is sufficient.

Qserv offers a query processing module that generates a distributed execution plan. It rewrites user queries for execution on chunk and sub-chunk tables on worker nodes. It splits the query representation into a plan composed of multiple phases of execution, each phase is executed per-data-chunk. The query processing module combines the distributed results to answer the user query. For queries involving spatial restriction (e.g., Query 2.3 asks how many objects within a square degree box in the sky), Qserv does not need to dispatch the execution on all chunks. This enables such queries to be executed on relevant chunks and avoids full-sky queries.

Query 2.3 Spatially-restricted filter (Qserv)

```
SELECT COUNT(*)
FROM Object
WHERE ra BETWEEN 1 AND 2 AND dec BETWEEN 3 AND 4;
```

Qserv defines a set of astronomical queries including queries for object retrieval, time-series measurements on a desired object, full sky filter, spatially-restricted filter, near neighbor and sources not near objects.

2.3.8 Tools for Cross-matching

Investigation of the structure and evolution of the Galaxy through cross-matching queries is one of the main queries in astronomy. Directly after the release of electronic versions of large sky surveys, astronomers started to cross identify catalogs. An early cross-match algorithm [66] uses a zoning algorithm to implement of points-near-point, spatial cross-match, and self-match queries. The basic idea is to map the celestial sphere into zones, each zone is a declination stripe of the sphere. The implementation of the zoneMatch algorithm is integrated with Microsoft SQL server where each object is associated to a zoneID and the cross-match is done through SQL statements using predicates on zoneID. Using the zoning algorithm, the zoneMatch algorithm allows to parallelize cross-match computations by distributing the data and workload among a cluster of database servers.

Pineau et al. [67] employ HEALPix to split the cross-match task into pieces to be processed in parallel. They also use a multithreaded two-dimensional kd-trees (Euclidean space-partitioning data structures) adapted to equatorial coordinates in order to enable efficient neighbors search. However, the large-scale problem is still far from being completely resolved using such algorithms.

Besides, some recent works [68; 56; 57] propose customized solutions to execute cross-matching queries. In [68], the authors introduce a cross-matching function using two partitioning approaches. In the baseline approach, the authors use an algorithm called the Simple Gridding Function that divides the `ra` axes and `dec` axes into equal intervals. However, the limitation of this approach is data skewness. The requirement of creating blocks of approximately the same size could not be ensured. For this reason, they propose a new alternative based on HEALPix by mapping the 2-dimensional space to a line space and use a B-tree index function. Indeed, HEALPix allows to overcome the problem of the spherical-polar distortion. They also propose a solution for the objects on the edge of the computation blocks by expanding the scope of each block.

Furthermore, there are other tools that have to operate with local data. For

example, the Topcat data analysis application [2] that has built-in features for cross-matching data stored locally and Aladin [3] that allows to match a list of objects with a catalog.

2.3.9 Discussion

	DBMS	Language	Indexing	Queries
SkyServer	SQL Server	SQL	HTM, Zones	20 SQL queries defined in a technical report [50]
VizierR	N.A	ADQL	HEALPix	Cone, kNN, Cross-match
Q3C	PostgreSQL	Q3C	Q3C	Cone, kNN Cross-match
Open SkyQuery	N.A	ADQL	Zones	Cross-match
MonetDB/ SkyServer	MonetDB	SQL	Zones	SQL queries
AscotDB	SciDB	ADQL	HEALPix	N.A.
Qserv	MySQL	SQL	Rectangular fragmentation in right ascension and declination	SQL queries (detailed in Section 2.3.7)

Table 2.1 Features of Astronomical Servers.

In the above sub-sections, we described the different aspects of the most used astronomical servers for the management of astronomical data. Table 2.1 resumes the main features of these servers. Bridging the gap between DBMS

[2] http://www.star.bris.ac.uk/ mbt/topcat/
[3] https://aladin.u-strasbg.fr/

technologies and astronomy has been finalized by these servers. A number of RDBMSs (SQL Server, PostreSQL, MonetDB) have been extended to process astronomical data. However, their performances are still limited because they are mainly based on centralized or rigid server architecture style. These systems could not face the challenges required for handling large astronomical data and could not provide the scalability, the availability and the performance required by big data applications.

Furthermore, SciDB, an array-based parallel oriented toward science applications has been also extended by AscotDB to manage data from large sky surveys. However, AscotDB does not propose any solution to handle data skew. Another work, Qserv offers a model that distributes and parallelizes computation among worker nodes. However, the rectangular fragmentation in right ascension and declination causes a distortion problem near the poles. The partitioning approach in Qserv does not use any sky indexing scheme adapted to the spherical space (such as HEALPix or HTM). Moreover, the choice for a shared nothing architecture using MySQL nodes was motivated by the fact that astronomical catalogs are well-defined in a relational model and the lack of indexing support in other existing systems (e.g., Hadoop or NoSQL). However, such implementation incurs significant overhead in dispatching queries and collecting results. Such specific architecture does not allow to take advantages of technological advances in big data management. Moreover, the LSST project has launched other projects [69] based on big data tools like Spark.

Spark provides an alternative solution that offers computational scalability and great flexibility in the management of scientific applications through data partitioning and in-memory computation. Thus, it is useful to determine methods to exploit such big data solutions for the management of data generated by modern telescopes to achieve high level query performances. This idea has been already concluded in the field of geo-spatial data, therefore, we have been interested in studying existing systems in the geo-spatial context.

2.4 Spatial Systems

Recent works have addressed the support of spatial data and queries using a distributed data server. Their architectures have followed the development of the Hadoop ecosystem. We divide these works into seven representative proposals.

2.4.1 Hadoop-GIS

Hadoop-GIS [70; 71] is a scalable and high performance spatial query system over MapReduce. It was the first big data spatial analytics system based on Hadoop that has been proposed in the literature. Hadoop-GIS employs an efficient partitioning approach (SATO) [72] represented by the following steps:

- Sample. It samples a small fraction (1~3%) of input data for analysis. The objective is to evaluate the density distribution of datasets.
- Analyze. It analyzes using the Minimum Bounding Rectangle (MBR) of spatial objects in the sampled dataset to find a global partition strategy.
- Tear. Each global partition is divided into local partitions using the partitioning algorithm that was designated in the analyze step. This allows to further refine the partition
- Optimize. It collects succinct partition statistics (such as the number of objects and the number of objects in the boundaries) to construct a multi-level partition index. The space is partitioned into independent regions and each region is further partitioned into smaller regions so that each small region would fit into a single HDFS file chunk.

Hadoop-GIS extends HiveQL to provide an expressive spatial query language. Users interact with the system using SQL queries with spatial extensions (such as *ST_INTERSECTS*, *ST_DISTANCE* ...). Then, the spatial query translator parses and translates these into operator trees. This translator is an extension of the HiveQL translator to support spatial query operators, spatial functions, and spatial data types. The query optimizer applies rules based optimizations (predicate pushdown or index-only query processing) to generate an optimized query plan. The spatial query engine generates the corresponding MapReduce jobs that are submitted to the Hive engine for execution using spatial data partitioning.

Example of queries

Query 2.4 Example of a Spatial Join query

```
SELECT ST_DISTANCE(ST_CENTROID(tb.polygon),
       ST_CENTROID(ta.polygon)) AS distance,
```

```
FROM polygons ta, polygons tb
WHERE ST_INTERSECTS(ta.polygon, tb.polygon) = 1;
```

Query processing in Hadoop-GIS follows a three-step process: query translation, logical plan generation, and physical plan generation. Query 2.4 is a spatial join query that is translated into a query tree as represented in Figure 2.8. It generates the table scan operators in the first step. Then, it applies filter predicates on required tiles (partitions). The query translator creates a tile based join processing workflow (each tile represents a simple join task). Each task is executed on Hadoop. Finally, Hive execution engine continues the processing task.

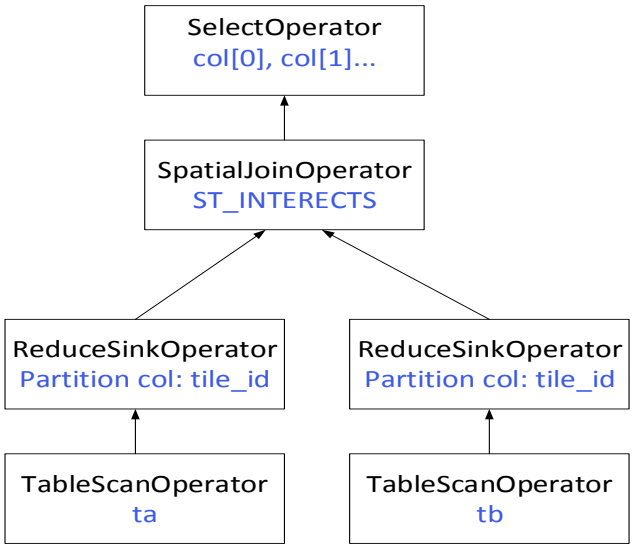


Figure 2.8 Spatial Join Query Plan in HadoopGIS.

2.4.2 SpatialHadoop

SpatialHadoop [10] is an extension of Hadoop that supports spatial data types and operations. It improves each Hadoop layer by adding spatial primitives. SpatialHadoop adopts a layered design composed of four layers: language, storage, MapReduce, and operations layers. For the language layer, it adds a spatial high level query language for spatial data types and spatial operations. In the storage layer, SpatialHadoop adopts traditional spatial index structures (Grid, R-tree [73] and R^+-tree [74]), to form a two-level index structure, called global and local indexing. SpatialHadoop rewrites functions of the Hadoop MapReduce framework by integrating two new components, *SpatialFileSplitter* and *SpatialRecor-*

dReader. *SpatialFileSplitter* exploits the global index to select blocks that contribute to the query result. For example, for range queries, the blocks that are completely outside the query area are not considered in the output result and blocks that are partially or completely overlapping are sent for further processing. *SpatialRecordReader* exploits the local index in the partitions received from the *SpatialFileSplitter* to return exact output records

In the operations layer, SpatialHadoop focuses on three operations: range query, spatial join, and k nearest neighbor (kNN).

Example of queries

A spatial join query takes as input two sets of spatial records R and S and a spatial join predicate (e.g., *overlaps*), and returns the set of all pairs that belong to R and S while satisfying the spatial predicate. In SpatialHadoop, the spatial join algorithm is composed of two steps:

- Global join: The *SpatialFileSplitter* exploits the two spatially indexed files and uses the overlapping filter function to identify the overlapping pairs of blocks that could contribute to the final result. Then, a spatial join algorithm is applied over the two global indexes to produce the overlapping pairs of partitions and the *SpatialFileSplitter* creates a combined split for each pair of overlapping blocks
- Local join: In this step, the *SpatialRecordReader* reads the combined split, extracts the records and local indexes from its two blocks, and sends all of them to the map function for processing. The map function uses the two local indexes to make the process of joining the two datasets faster.

2.4.3 Pigeon

Pigeon [75] is an extension of Pig to support spatial data processing in Hadoop. It is implemented through user defined functions (UDFs) and is compatible with all Pig versions. Pigeon supports spatial data types (*Point, Linestring, Multi-Linestring, Polygon, MultiPolygon, and GeometryCollection*) and spatial functions grouped into four categories:

- Basic Spatial Functions: retrieve basic information about a single spatial object (e.g., the perimeter length or area). These functions take as input a Well-Known Text (WKT) or a Well-Known Binary (WKB) object and convert it into a spatial object.

- Spatial Predicates : return a Boolean value based on the relationship of the input object(s) (e.g., *IsClosed* or *Touches*), these functions are implemented as simple UDFs, the input objects are also converted from WKB or WKT.

- Spatial Analysis : performs spatial transformation on spatial objects. It supports unary functions such as *Centroid* and binary functions such as *Intersection*.

- Aggregate Functions : return a single value that summarizes the input objects (e.g., the function *ConvexHull* returns one polygon representing the minimal convex hull of all input spatial objects). These functions are implemented as algebraic aggregate functions. A given function starts by computing partial results in each machine in the cluster, then the partial results are merged to produce the final answer. For example, multiple local convex hulls are first computed in each machine, then the global convex hull is computed by combining all local hulls.

Example of queries

Spatial join: This query finds the overlapping records in two datasets, it starts by computing the cross product of two relations and then applying a spatial predicate (e.g., overlap) as a post processing filter. Notice the implementation of such query is very basic and does not include any optimization.

2.4.4 MD-HBase

MD-HBase [12] is a scalable multi-dimensional data store for Location Based Services (LBSs), built as an extension of HBase. MD-HBase supports a multi-dimensional index structure over a range partitioned Key-value store. MD-HBase uses a linearization techniques (Z-ordering) [76] to transform a multi-dimensional location into a one dimensional space and uses HBase as a storage back-end.

MD-HBase combines Z-ordering with trie-based quad-trees and k-d trees. It partitions the space into subspaces using trie-based k-d trees and quad-trees. Then, it names each subspace by the longest common prefix of the z-values of points contained in the subspace. This naming scheme was called the *longest common prefix naming*. MD-HBase builds spatial index structures to support range and kNN queries. It introduces an efficient query processing technique that accesses only the index and storage level entries that intersect with the query region.

Example of queries

A spatial range query in MD-HBase is decomposed into several linearized sub-queries. MD-HBase splits the multi-dimensional space recursively into subspaces and organize these subspaces as a search tree. It calculates the z-value range for the query and defines the potential candidate subspaces. Points in a subspace are scanned only if the range of the subspace intersects with the query range. This prunes out all the subspaces that are not relevant.

2.4.5 GeoSpark

GeoSpark [13] extends the core of Apache Spark to support spatial data types, indexes and operations. In other words, the system extends the Resilient Distributed Datasets (RDDs) to support spatial RDDs (SRDDs).

GeoSpark provides the support of spatial data indexing (R-Tree and quad-tree) and query processing algorithms (range queries, kNN queries, and spatial joins over SRDDs). GeoSpark has three main layers:

- Apache Spark Layer: this layer provides the basic Apache Spark features. It consists of loading, saving data from, to persistent storage (e.g., stored on local disk or HDFS)
- Spatial Resilient Distributed Dataset (SRDD) Layer: this layer extends the regular RDD to support geometrical objects (i.e., points, rectangles, and polygons) as well as geometrical operations on these objects. Three new spatial RDDs are proposed: PointRDD, RectangleRDD and PolygonRDD

- Spatial Query Processing Layer: this layer harnesses and extends the SRDD layer to execute spatial queries (e.g., range query, kNN queries and join queries) on large-scale spatial datasets

Example of queries

Spatial join: GeoSpark starts by partitioning the data from the two input SRDDs based on grid partitioning and creates local spatial indexes for the SRDD. Then, it joins the two datasets based on their grid IDs (Grids that have the same ID cover the same space region). For the spatial objects that have the same grid ID, GeoSpark calculates their spatial relations. If two elements from two SRDDS are overlapped, they are kept in the final results. The algorithm continues to group the results for each rectangle and saves the final result to disk.

2.4.6 LocationSpark

LocationSpark [77] is a spatial data processing system built on top of Apache Spark. LocationSpark offers a rich set of spatial query operators, e.g., range search, kNN, spatial-join, and kNN-join. It introduces a new layer, termed the query scheduler, to deal with query skew. This query scheduler identifies potential hotspot data partitions by collecting statistical information from each partition (such as number of data points). LocationSpark employs a cost model that evaluates the overhead of repartitioning the hotspot partitions and therefore reallocates these partitions to workers. After data partitioning, the query executor chooses and executes the better execution plan on each slave node. Similar to SpatialHadoop and GeoSpark, LocationSpark employs two layers of spatial indexes (global and local). It offers multiple types of local indices e.g., a grid index, R-tree or quad-tree. It also uses a Spatial Bloom Filter (*sFilter*). *sFilter* is embedded into the global spatial index of LocationSpark to check whether a spatial point is contained inside a spatial range or not.

Example of queries

We choose to describe the kNN join algorithm employed by LocationSpark. It starts by identifying the partitions of each point r_i in R using the global index. Then, the kNN join is executed locally inside each partition to produce the kNN

candidates for each point r_i in R. After that, it calculates the maximum distance from r_i to its kNN candidates. If the calculated radius is overlapping more than one partition, then, the query point r_i is replicated and another set of kNN candidates is identified. To get the final result, LocationSpark merges all the kNN candidates from the identified partitions.

2.4.7 SIMBA

SIMBA [11] has been proposed as an extension of Spark SQL to support spatial queries and analytics over big spatial data. SIMBA builds spatial indexes over RDDs. SIMBA offers an SQL-like interface for spatial queries by adding spatial keywords and grammar (e.g., *POINT, RANGE, KNN, KNN JOIN, DISTANCE JOIN*) in Spark SQL's query parser. In addition to SQL, users can also perform spatial operations over DataFrames. It offers a programming interface to execute spatial queries (range queries, circle range queries, kNN, distance join, kNN join), and uses cost based optimization.
Similar to SpatialHadoop, SIMBA uses the concept of global and local indexing. SIMBA implements several classic index structures including hash maps, tree maps, and R-trees over RDDs. The global index collects statistics from each RDD partition in order to prune out irrelevant partitions. Local indexing is used to accelerate local query processing inside each RDD partition. SIMBA is able to optimize complex spatial queries using indexes and statistics. The logical optimizer applies standard rule-based optimization, such as constant folding, predicate pushdown, to optimize the logical plan. Indeed, predicate and projection pushdown in SIMBA allow to push operations from the logical plan into data sources. In the physical planning phase, SIMBA takes a logical plan as input and generates one or more physical plans based on the spatial operation. Then, it applies cost-based optimizations based on existing indexes and statistics to select the most efficient plan.

Example of queries

Distance join is a θ-join between two tables, it runs in three steps: data partition, global join, and local join.

- Data partition: Simba starts by partitioning the two input tables using R-tree partitioning which ensures load balancing and preserves data locality. It

uses the STR algorithm to get the first level of the tree that represents the partition boundaries.

- Global join: SIMBA generates a list of candidate pairs (R_i, S_j) of partition IDs, and produces a combined partition P = $\{R_i, S_j\}$ for each pair (i, j). After that, the combined partitions are sent to workers for local joins processing.
- Local join: Local join builds a local index over S_j on the fly to find all pairs of points where the euclidean distance is lower than a threshold.

2.4.8 Discussion

In [78], the authors provide a comprehensive tutorial that reviews all existing research efforts in the era of big spatial data and classify existing works according to their implementation approach, underlying architecture, and system components.

Pandey et al. [79] have surveyed the big spatial data analytics systems listed above in a recent paper. They announced that Spark based spatial analytics systems have consistently shown superior performance compared to Hadoop based systems like SpatialHadoop and HadoopGIS. For this reason, they have chosen to evaluate the performance of Spark based systems. They explored in deep these systems and compared them based on features and queries they support, using real-world datasets.

The aforementioned systems are designed for the geo-spatial context that differs from the astronomical context in its data types and operations. Our work considers celestial objects as points in a spherical coordinate system, whereas these systems process other spatial data types (polygons, rectangles ...) in quadrant plane.

These systems do not provide a high level query language adapted to the astronomical context like ADQL. They do not offer astronomical functions tailored for the spherical coordinate system. Common astronomical constructs (notably spherical polar coordinate systems and geometries) can not be expressed in standard SQL. Since ADQL standardizes astronomical queries, the use of ADQL for astronomical query processing is needed.

These systems use conventional spatial indexing techniques while we adopt specialized indexing method for astronomical data using HEALPix. We also demonstrate in Section 4.8 the limitations of these systems in astronomical data

partitioning. They may cause a problem of distortion around the poles. Thus, the partitioning approaches used in these systems are limited to small areas, where the sphere can be projected onto a plane with negligible distortion. Moreover, these systems lead to erroneous results when querying data because of the difference between the spherical distance and the euclidean distance (see Section 1.3). We also deal with specific astronomical operations such as cone search queries, and cross-match queries that are not supported by these systems. As a result, these spatial big data frameworks are not suitable for astronomical applications.

2.5 Summary

The need for astronomical query processing has motivated many projects to extend existing DBMSs. Among these, the SkyServer project, the VizieR service, Q3C, Open SkyQuery ... However, these systems inherit the performance of existing DBMSs and do not allow to take advantages of the technological advances in big data management.

Another category of systems, based on existing big data technologies, adds support for spatial data. These systems include Hadoop-GIS, SpatialHadoop, MD-HBase, GeoSpark, LocationSpark, SIMBA ... However, these systems have focused on the development of techniques that take into consideration the characteristics of geo-spatial data and do not consider the specificities of astronomical queries. Our goal is to provide astronomical query processing, by offering a high performance query processing framework that takes advantages of Spark, a distributed in-memory query processing engine.

CHAPTER 3

General Presentation of ASTROIDE

Contents

> The goal is to turn data into information, and information into insight.
>
> Carly Fiorina

3.1 Introduction

As discussed in the state-of-the-art chapter, Apache Spark has become one of the top big data distributed processing framework in the world. It has the advantage of memory computing compared to traditional MapReduce based technologies. In this chapter, we present ASTROIDE, a distributed data server tailored for the management of large volume of astronomical data and concentrate on query processing and optimization. ASTROIDE stands for ASTROnomical In-memory Distributed Engine. It is designed as an extension of Apache Spark, and takes into account the peculiarities of the data and the queries related to astronomical sky surveys. ASTROIDE introduces effective methods for astronomical query execution on Spark through data partitioning with HEALPix and customized optimizer. Although our current architecture is built on Spark framework, it is necessary to mention that the concepts and algorithms described in this work can be ported to other big data frameworks. This chapter describes the overall architecture of our system from the physical level to the high level language. Before starting to detail our proposal, we will tackle some important background and definitions that are used in this dissertation.

3.2 Background

3.2.1 Query Processing

The objective of query processing is to transform a high-level query (such SQL) into an equivalent lower-level query that implements the execution strategy for the query with efficiency and correctness [80]. The query processor receives a query as input, translates and optimizes this query in several phases into a physical query plan. This physical plan is executed by the query execution engine in order to obtain the results of the query.

Query Processing is divided into three major steps as shown in Figure 3.1. This architecture can be used for any kind of database system including centralized, distributed, or parallel systems. The first phase, the parsing phase, checks the query for syntactic and semantic correctness, performs view resolution, and generates a parse tree that represents the structure of the query in a useful way. During the second phase, the optimization phase, the optimizer chooses the best

plan for the user-submitted query according to statistics (e.g., the histogram of the value distributions in each column), heuristics to reduce disk I/O, CPU ...

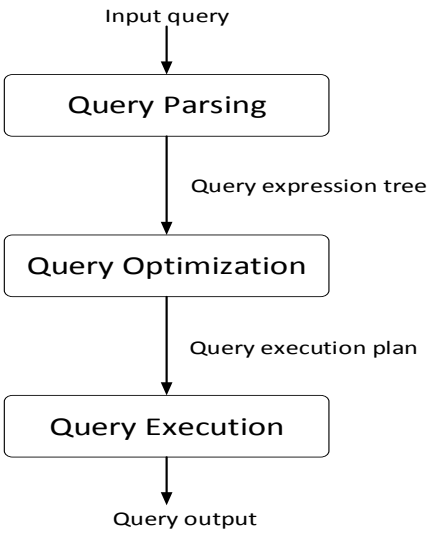


Figure 3.1 Query Processing.

Query optimization is often divided into two phases [81]:

- The logical optimization, which allows to rewrite the parse tree into an initial plan representing an algebraic representation of the query. This plan is transformed into an equivalent plan that is expected to require less time to execute. The goal is to produce the best logical query plan.
- Physical optimization, which transforms a logical query plan into a physical query plan by selecting the best algorithms to implement the operators of the logical plan and an order of execution of these operators. The physical plan includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Based on the information about data representation and its location for distributed systems, the optimizer generates a physical plan. The third phase, called the Execution, executes this physical plan in the query processing engine.

For illustration, let's consider the following SQL query expressing an astronomical query to find the number of stars for each galaxy:

Query 3.1 Example of an Astronomical Query

```
SELECT G.ObjID, COUNT(S.objID)
```

```
FROM galaxy G, star S
WHERE G.parentID = S.parentID
GROUP BY G.ObjID
```

This can be translated into a relational algebra expression using γ to express the grouping and aggregation:

$$\gamma_{G.ObjID,COUNT(S.objID)}(\Pi_{G.ObjID,S.ObjID}(\sigma_{G.parentID=S.parentID}(G \times S)))$$

We can combine the selection and cross-product into an equi-join and generate a projection on G.objID and S.objID which are the only attributes relevant for the γ operation:

$$\gamma_{G.ObjID,COUNT(S.objID)}(\Pi_{G.ObjID,S.ObjID}(S \bowtie_{parentID} G))$$

We can also push the projection below the join as follows:

$$\gamma_{G.ObjID,COUNT(S.objID)}(\Pi_{S.ObjID,G.ObjID}((\Pi_{parentID,ObjID}(S)) \bowtie_{parentID} (\Pi_{parentID,ObjID}(G))))$$

The last expression avoids the expensive Cartesian product and pushes projections. Therefore, it is typically better and should be retained. For a given query, multiple relational algebra expressions representing different orders or combinations of operators are possible as shown in the above example. The objective is to choose the best relational algebra expression that is likely to result ultimately in the cheapest physical plan. The relational algebra expression is represented as a query tree, that can be handled by the query optimizer.

The main components of a query tree are as follows [82]:

- Leaf nodes representing input relations of the query.
- Internal nodes representing intermediate relation that is the output of applying an operations in the relational algebra.
- Root of tree representing the result of the query

- The sequence of operations (data flow) is directed from leaves to the root node.

Below is a brief description of each component of query processing:

Query Parsing.

During this step, the query is parsed, validated and translated into an internal representation from a human readable form to a form easily usable by the query processing engine called parse tree. Query parsing identifies the language tokens such as keywords, attribute names and relation names. It is used for checking syntax and also for relations verification, resolving attributes and checking types.

The same parser can be used in centralized or distributed systems. As discussed in Chapter 1, astronomical queries are difficult to express using SQL. The standard used language for querying astronomical data is ADQL. However, existing big data systems do not provide support for ADQL. Thus, a custom parser is required while taking advantages of built-in query processors of existing systems. For example, the ADQL library [83] developed by the CDS provides parsing and translation of ADQL queries into Postgresql/Q3C or Postgres/pgSphere.

Query Optimization

Selecting the optimal execution strategy for a query is NP-hard in the number of relations [84]. The main focus of query optimization is to minimize the query execution time for a given query and reduce the system resources required to fulfill a query [85]. The role of query optimization is to estimate the cost of alternative query evaluation plans (QEPs) to execute the query, and chooses the cheapest. All plans are equivalent in terms of final output but may be widely different in their costs, i.e. the query execution times.

There are two other major elements which are necessary to completely determine all aspects of query optimization. On the one hand, we need to describe how to search through the set of all possible QEPs. On the other hand, we need to compare different QEPs and decide which one is the best. In general, the decision is based on the costs of various resources, such as CPU, disk I/O ...

The query optimizer is represented by three components as shown in Figure 3.2: search space, cost model and search strategy [80].

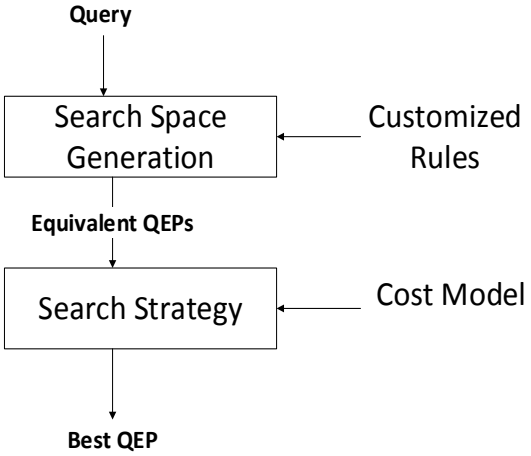


Figure 3.2 Query Optimization.

- Search Space. It is defined as a set of equivalent QEPs for a given query that can be generated using transformation rules [86]. The objective is to reduce the search space by applying heuristics. Typical examples are predicate pushdown, in which predicates are applied as early as possible in the query, commutative (e.g., $R \bowtie S = S \bowtie R$) and associative laws (e.g., $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$), duplicate elimination... Such rules can significantly improve query execution time.

- Search Strategy. It is also called the enumeration algorithm, it determines the algorithms applied to explore the search space. The most popular strategy is dynamic optimization which works in a bottom-up way and builds execution plans starting from base relations and joining one more relation at each step till the complete plans are obtained. Other alternative enumeration algorithms have been proposed, Steinbrunn et al. [87] presents an overview of these algorithms and Kossmann [88] describes the most important approaches in the literature.

- Cost Model. This module requires analytical formula to estimate the size of intermediate data and cost functions to predict the cost of operators. It assigns an estimated cost to each possible QEPs. This cost model relies on [89]:

  – Statistics on the relations which include various metadata (e.g. the number of rows, the number of disk pages of the relations, histogram

of distributions of column values ...) and indexes.

- Formulas to estimate the selectivity of various predicates and the sizes of the output for each operator in the query plan.
- Formulas to estimate the CPU and I/O costs for every operator in query plan.

In distributed systems, cost functions can be expressed with respect to local processing time (CPU time, I/O time) and communication time (time to initiate a transmission, time to transmit the data) [80]. The communication cost is a key factor of distributed systems performances, it is a linear function of the amount of data to be transmitted. Thus, an optimizer has to reduce the amount of data transmitted. In our context, we can take into account partitioning approaches to retrieve only relevant partitions. This allows to reduce the amount of data transmitted. Thus, accessing unnecessary partitions incurs useless communication costs. The role of the cost model is to make the best use of existing indices and statistics in order to select the most efficient QEP. The objective in astronomical systems is to leverage the index and partitioning support.

About big data technologies introduced in the state-of-the-art, Hive introduces a cost model using Calcite's [90]. Calcite is currently the most widely adopted optimizer for big-data analytics in the Hadoop ecosystem. Calcite is adopted by Hive, Drill, Storm, and many other query processing engines [91], providing them with advanced query optimizations. Calcite applies various optimizations such as query rewrite, join reordering and join algorithm selection. Calcite has a plan pruner that can choose the cheapest query plan. The chosen logical plan is then converted by Hive into a physical operator tree, optimized and converted into jobs, and then executed on the Hadoop cluster.

About the Spark optimizer, Catalyst introduces rule and cost based optimization but very little work has been devoted to cost model, it uses a simple cost model. Cost based optimization was added in recent Spark version 2.0 and focused on cardinality estimation, broadcast vs. shuffled join, join reordering. Thus, future Spark versions [1] are continuously evolving to finish with a robust cost model. A recent research [92] proposes a cost model

[1] https://issues.apache.org/jira/browse/SPARK-17129

for Spark SQL that covers the class of queries composed of joins, selection predicates and aggregations. It keeps into account the network and I/O costs as well as CPU costs.

Execution Engine.

The query-execution engine takes the QEP generated by the query optimizer, executes that plan, and returns the answers to the user. Query execution provides generic implementations for every operator (e.g., send, scan, or Nested Loop Join ...) and generates the code for the selected QEP.

3.2.2 Astronomical Queries

ASTROIDE focuses on four main basic astronomical queries (cone Search, kNN Search, cross-match, kNN Join), that could not be, or too costly executed directly on existing systems.

For a sake of consistency, we introduce the notations listed in Table 3.1.

Notation	Description
R, S	Catalogs of stars
r (resp. s)	A star $r \in R$ (resp. $s \in$ S)
c	Circle with $c.p$ as center and $c.sr$ as a radius
ε	Threshold of spherical distance
$SD(r,s)$	Spherical distance between r and s

Table 3.1 Used Notations.

Definition 1: The Spherical distance is calculated with the harvesine formula [93] as the length of the great circle arc between two points on a sphere. For two celestial objects r, s in spherical coordinates (ra_1, dec_1) and (ra_2, dec_2), the spherical distance between these two objects is:

$$SD(r,s) = 2\arcsin\sqrt{\sin^2(d/2) + \cos(dec_1)\cos(dec_2)\sin^2(a/2)}$$

Where:

d = dec_2 - dec_1 = Difference in declination between r and s

a = ra_2 - ra_1 = Difference in right ascension between r and s

Definition 2: Cone Search [94] returns a set of stars whose positions lie within a circular region of the sky (see Figure 3.3).

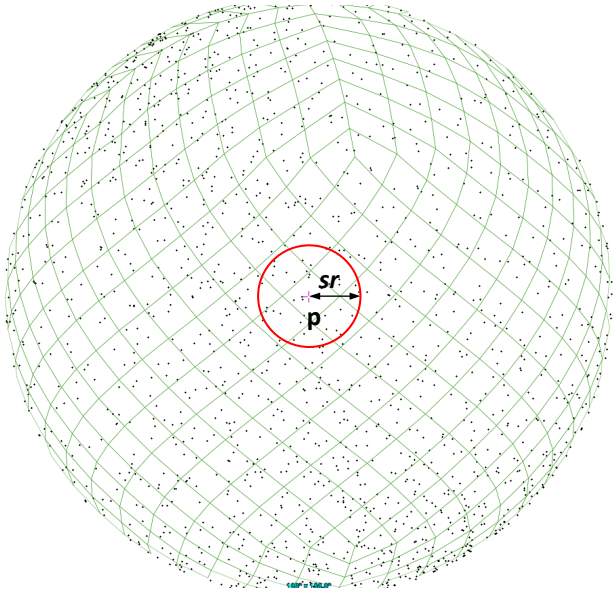


Figure 3.3 Cone Search Definition

Given a dataset R and a circle c defined by a sky position p and a radius sr around that position, a cone search query returns all pairs of points $r \in R$ within c. According to [94] from the IVOA, the Simple Cone Search (SCS) assumes the observer geocentric. We can imagine this in three dimensions as a cone stretching from an observer (such as a telescope) to a circle defined by a point p and a radius sr. Formally :

$$\texttt{Cone-Search}(R,\ c) = \{r \mid r \in R,\ SD(r,c.p) \leq c.sr\}$$

Definition 3: Cross-matching query aims at identifying and comparing astronomical objects belonging to different observations of the same sky region. Cross-matching takes two datasets R, S and a radius ε as inputs and returns all pairs of points (r,s) such as their spherical distance is lower than ε.

$$\texttt{XMATCH}(R,S,\varepsilon) = \{(r,s) \mid (r,s) \in R \times S,\ SD(r,s) \leq \varepsilon\}$$

Cross-Matching is equivalent to a spatial distance join on two datasets R and S ($R \bowtie_{\varepsilon} S$) in database terms.

Definition 4: Given a query point p, a dataset S and an integer $k > 0$, the k nearest neighbors from S denoted kNN(p,S) is a set of k objects such that:

$$\forall o \in kNN(p,S), \forall s \in S - kNN(p,S),\ SD(o,p) \leq SD(s,p)$$

Definition 5: kNN Join takes two datastes R,S and an integer $k > 0$ and returns each object $r \in$ R with each of its kNNs from S.

$$k\texttt{NN-Join}(R,S) = \{(r,s) \mid r \in R, s \in kNN(r,S)\}$$

3.2.3 ADQL

Astronomical queries are commonly expressed using the Astronomical Data Query Language (ADQL) [7]. ADQL is a well-known language adapted to query astronomical data, and is promoted by the International Virtual Observatory Alliance (IVOA) [95]. It is an SQL-Like language improved with geometrical functions which allows users to express astronomical queries with alphanumeric properties. ADQL provides a set of geometrical functions: AREA, BOX, CENTROID, CIRCLE, CONTAINS, etc. For example, CIRCLE expresses a circular region of the sky, which corresponds to a cone in space. The ADQL expression of cone search is illustrated in the following example. Given that `ra` and `dec` are the spherical coordinates in the `gaia` catalog, it returns all stars in the cone centered at the point $(266, -29)$ having a radius 0.0833 in the ICRS coordinate system:

Query 3.2 Cone Search (ADQL)

```
SELECT *
FROM gaia
WHERE 1=CONTAINS(POINT('ICRS', ra, dec),
      CIRCLE('ICRS', 266, -29, 0.0833))
```

3.2.4 DataFrames

Spark SQL employs a programming abstraction called DataFrame. It is conceptually equivalent to a table in a relational database. DataFrame is a distributed collection of data organized into named columns.

DataFrame supports reading data from various data sources, including JSON files, Parquet files, Hive tables. It can read data from local file systems, distributed file systems (HDFS), cloud storage (S3), and external relational database systems via JDBC.

Similar to RDDs, DataFrames are evaluated lazily. This means that computation only occurs when an action (e.g., *count, save*) is required. With DataFrames, users can perform relational operations using a domain-specific language (DSL). Below, we include a basic Scala example to create a DataFrame from a Parquet file and to apply a filter operation over the loaded DataFrame.

Query 3.3 Example with DataFrame

```
val df = spark.read.parquet("examples/gaia.parquet")
df.filter($"magnitude" > 18).show()
```

The main reason for processing data using DataFrames in ASTROIDE is that the Catalyst optimizer allows the optimization of operations that are used to build a DataFrame. Catalyst can apply physical and logical optimizations to speed up computation. In case of Parquet files, entire blocks can be skipped, such optimization allows to improve performance.

3.2.5 Parquet Format

Parquet is a popular columnar storage format, used as a back-end for ASTROIDE. It supports nested data structures and different compression and encoding schemes. We have chosen Parquet format in ASTROIDE because of the following advantages:

- Columns are efficiently compressed which leads to save storage space
- Only required data is scanned, thereby reducing I/O and increasing performance

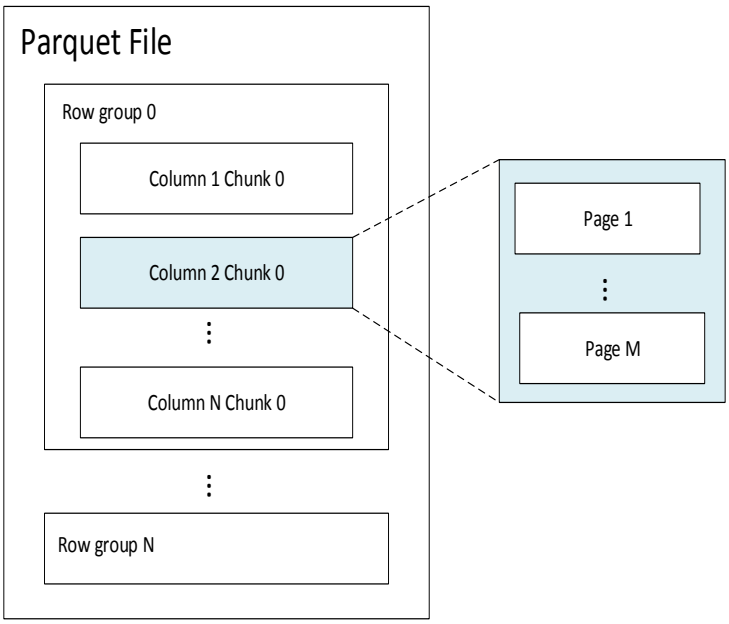


Figure 3.4 Structure of Parquet File

Parquet is represented by a three-level hierarchical structure to organize data. It starts by a horizontal partitioning of the data into rows called row groups which are distributed across a cluster and processed in parallel. These row groups are subsequently divided into column chunks which refers to data in a column within a row group (vertical partitioning). The third level is a page. Each column chunk is in turn split into one or more pages.

Parquet makes queries both fast and light. It stores useful statistics to allow early data filtering. These statistics includes the count and minimum/maximum values. Parquet only pulls data that is filtered for a row group, column chunks, and selected partitions. For example, for a query with a range predicate, we can reference the min and max values and decide whether to read or skip the row group, column chunk or page. This will reduce the amount of data to be load and to be processed during query execution.

3.3 Overview of the Proposed Framework

The primary goal of ASTROIDE [96; 97; 98] is to provide a scalable and efficient query processing system for astronomical data. In view of the foregoing, we have based ASTROIDE on Spark framework. Our design takes full advantages

of Spark features.

Figure 3.5 shows the architectural components of ASTROIDE, available as open-source [2]. Our framework is composed of two modules: data partitioning and query processing. The data partitioning module is responsible for managing the partitions in a way that ensures data locality, load balancing and task parallelization. On the right of the Figure 3.5 is the query processing module. ASTROIDE allows to execute astronomical queries using ADQL or DataFrames. The query parser translates ADQL queries into SQL queries with UDFs which are in turn transformed into Abstract Syntax Trees (ASTs). The query optimizer integrates specific logical and physical optimization techniques to generate an efficient QEP that is executed by the Spark engine.

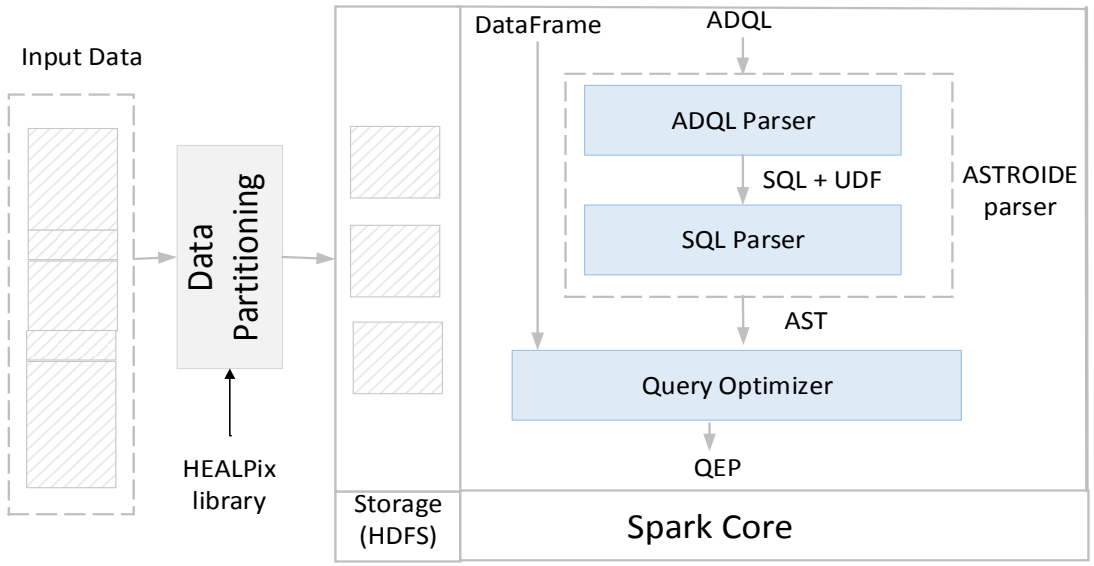


Figure 3.5 ASTROIDE Architecture

3.4 Data Partitioning

Astronomical queries as described in chapter 1 are expensive to process and may lead to computation skew in some nodes of a cluster. Data skew is observed in distributed systems when certain data partitions are overloaded during query processing. In the astronomical context, a deep knowledge about data distribution is necessary to provide efficient algorithms for query processing.

[2] https://github.com/MBrahem/ASTROIDE

Partitioning is a fundamental component for parallel data processing. It reduces computing resources when only a sub-part of relevant data are involved in a query, and distributes tasks evenly when the query concerns a large number of partitions. Hence, partitioning globally improves the query performances.

Spark partitioning methods are only applicable when the partition key is a scalar one dimensional value. In order to make use of partitioning techniques in our case, we need first to adapt it for the 2D coordinates as they are intensively used in typical astronomical queries. With this regard, we establish two main requirements:

- Data locality: points that are located close to each other should likely fall in the same partition, that is a partition has to represent a portion of the sky.

- Load balancing: the partitions should be roughly of the same size to avoid data skewness and efficiently distribute tasks between nodes of the cluster. A poor load balancing leads to imbalance among workers in a cluster, which globally slows down the execution time.

To achieve the first requirement, a spatial grouping of the data is necessary. Nevertheless, a basic spatial partitioning may lead to imbalanced partitions due to the typical skewness of astronomical data. Therefore, the partitioning should be also adaptive to the data distribution. ASTROIDE partitions astronomical data in a way that partitions are balanced while favoring data locality. To this end, we employ HEALPix as an indexing scheme to map the two-dimensional spherical coordinates into a single dimensional ID, this ensures the data locality requirement, i.e., close stars in the sky are likely to have close HEALPix IDs. To achieve load balancing, we leverage the Spark range partitioner, which yields data partitions with roughly equal sizes. In the final step, the partitions are stored using Parquet format (Section 3.2.5) in HDFS [27] to amortize the construction costs for future queries. We structure partitioned files in such a way that each partition is divided further into buckets. The bucketed column is the HEALPix index so that rows in the same cell are always stored in the same bucket. We also determine partition boundaries and store them as metadata.

3.5 Query Processing

In this section, we provide an overview of query processing in ASTROIDE. Query processing in our context is the transformation of the query from an ADQL dialect into an execution plan that performs the required manipulations within Spark. When ASTROIDE receives an ADQL query, it goes through a series of steps as shown in the right part of Figure 3.5. In the first phase, the system parses the query, verifies the ADQL syntax rules and performs appropriate query rewriting by translating geometrical functions into equivalent internal UDFs. Then, it matches the translated SQL query to an abstract syntax tree generated by the Spark SQL parser. In the third phase, the query optimizer takes this tree as input and performs optimization on the query by using customized rules and strategies. The query optimizer returns an execution plan that minimizes the query processing time.

3.5.1 Query Interface Levels

ASTROIDE allows the extension of Spark to deal with astronomical queries using ADQL and DataFrames. We have proposed and evaluated two alternatives as a query interface. The idea is to follow the same Spark interaction ways including SQL and direct DataFrame API:

- **DataFrame API**. If the query is written using the DataFrame API, we provide a query interface by extending this API with the astronomical operations listed in Section 3.2.2. Each DataFrame is internally represented as a logical plan that describes the computation required to produce the data. In this case, no data parsing is required. For example, a cross-matching between two DataFrames `df1` and `df2` is expressed with the following Scala code in ASTROIDE. This produces a new DataFrame called `output` which is the result of cross-matching.

Query 3.4 Cross-Match using DataFrames

```
val output = df1.XMatch(SparkSession, df2, 0.002)
```

- **ADQL Queries**. If the query is written using ADQL, ASTROIDE parser transforms the ADQL query into an abstract syntax tree as explained in Section 3.5.2.

Internally, there is no difference between using the DataFrame API or ADQL as the same execution engine will be used for both. However, the ADQL option remains more expressive with an easier syntax. ADQL queries can be also executed without any modifications in the program.

3.5.2 ASTROIDE Parser

ASTROIDE parser is divided into two steps: ADQL parser and SQL parser. In the first step, the parser verifies that the query is syntactically correct. Then, it extracts tables names, columns names and some keywords such as `CONTAINS`, `JOIN`, `POINT`, `CIRCLE` from the input query. This extraction helps ASTROIDE to fetch the query types described in Section 3.2.2. The query parser translates the ADQL query into a valid SQL query understandable by the spark SQL parser. In the second step, the Spark SQL parser transforms the translated SQL query into the corresponding AST.

3.5.3 Query Optimizer

This module is an extension of the Spark Catalyst optimizer. We integrate new rules and strategies for converting Spark non optimized logical plans to optimized physical plans. The decision to exploit Catalyst [15] as a backbone for our optimizer was driven by the fact that astronomical databases contain both astronomical and relational queries, as opposed to developing a new optimizer. Building a new optimizer is a complex engineering task. Instead, Catalyst allows executing relational queries and enables adding new optimization rules as explained in Section 2.2.5. Our optimizer still leverages all the benefits of Spark's optimizer like predicate pushdown, projection pruning and join reordering. We distinguish two types of operators: traditional algebraic operators, and astronomical operators. Here, we integrate optimizations for astronomical operators and let Catalyst performs traditional operators.

The input of this module is an abstract syntax tree generated after query parsing from an ADQL expression. The query optimizer transforms this tree into an equivalent query tree, but with optimized form by rewriting costly operators or by adding filter operator on our data structure. This query tree is represented by a

logical plan, we distinguish three types of plans: analyzed logical plan, optimized logical plan, physical plan. The parsed plan goes through a series of analyzer rules to produce the analyzed logical plan. The analyzed plan is converted in turn into an optimized logical plan using optimization rules. The optimized plan is transformed into a physical plan using strategies. The output physical plan is the actual plan which ASTROIDE executes for the final data processing. To optimize astronomical queries, we inject a set of optimization rules, analyzer rules or strategies to produce the final QEP. We demonstrate these rules and strategies by giving examples in Chapter 5.

Rule based optimization in our context exploits spatial partitioning to access the smallest possible number of partitions, avoiding cartesian product or performing projection on spatial indices as early as possible ... Indeed, our query optimizer injects transformation rules to avoid scanning all records. It uses indices to prune out partitions that do not contribute to the query result and scans only relevant partitions.

3.6 Summary

In this chapter, we presented ASTROIDE, a solution that combines the scalability of distributed data processing engines using Spark with the expressiveness of astronomical data servers using ADQL. ASTROIDE achieves this objective through efficient partitioning, customized rules and strategies in the Catalyst optimizer and an expressive query interface. We proposed an approach that builds indices using HEALPix combined with data partitioning to support astronomical queries. We presented a query processing module that improves the execution of astronomical queries through dynamic query rewriting and efficient optimizer.

CHAPTER 4

Data Partitioning and Indexing

Contents

> Numbers have an important story to tell. They rely on you to give them a voice.
>
> Stephen Few

4.1 Introduction

Data partitioning is a key feature for efficient query processing in distributed systems. It is even more crucial in astronomical big data management. Data partitioning is based on dividing data into smaller subsets and processing partitions in parallel on multiple nodes. Partition pruning can also speed up query performance drastically by eliminating unnecessary and enabling index scans as well as reducing memory needs, disk I/O ... Partitioning is a powerful mechanism to improve the overall manageability of big data systems [99]. It aims to reduce query execution time and facilitates the parallel execution of queries. Therefore, partitioning plays an essential role for achieving optimal system performance. It can efficiently simplify the complexity of managing massive data.

In this chapter, we discuss the importance of partitioning in distributed systems and present the challenges related to the partitioning of astronomical data. We study existing approaches for sky indexing and partitioning. We also detail the partitioning approach in ASTROIDE.

4.2 Importance of Partitioning

Performance, scalability, and manageability are critical in any computing environment. Partitioning is one of the most efficient features to address these requirements. It enhances the manageability of data by dividing large tasks into small tasks. It improves scalability and performance by pruning unneeded data. This enables to access only the partitions that contains the necessary data and not the whole table, only one or few partitions are scanned. It eliminates, or ignores partitions that are irrelevant to the criteria that we have set in the input query. Hence, it can significantly reduce the amount of data scanned to return results, provides faster data loading and limits the communication costs. It offers a fine-grained control over the physical design for performance tuning. It also enables parallel query execution, and allows to control parallelism which leads to better cluster utilization (fewer costs and better execution time).

4.3 Challenges in Astronomical Data Partitioning

Partitioning of astronomical data poses several challenges related to their characteristics as exposed in Chapter 1.

4.3.1 Data Skew

Data skew in distributed systems occurs when a node or a set of nodes have to deal with larger partitions or more complex computations. Unbalanced partitions can have a very negative impact on query performance, resource usage and scalability. It could lead to serious overload across tasks, which would eventually result in some tasks taking longer time to complete than others and increase the overall job latency. In general, distributed systems use a hash function to partition data. This function is expected to generate equal structure of partitions. However, in some cases (e.g., regions with high density of stars), the hash function fails to achieve equal partitioning, resulting in unbalanced partition sizes.

There are many reasons why skew may arise in astronomical parallel query processing. When some sky regions contain significantly more data objects than others, this could lead to the overload of some nodes while others can remain idle. Thus, sophisticated partitioning approaches have become necessary to achieve load balancing and avoid data skew. Load balancing is fundamental in order to affect a similar amount of data to nodes of the cluster.

4.3.2 Objects on the Boundaries

In distributed systems, partitioning of astronomical data can generate multiple objects around the borders of partitions (e.g., the object surrounded in Figure 4.1). However, those objects of different HEALPix cells could match with objects from adjacent partitions. Thus, to get a correct cross-matching result, we need to deal with the matched objects along the borders, i.e., those having an ε-distance but belonging to different (neighbor) partitions. So, join should be extended to neighbors by replicating objects on the borders to multiple neighbor partitions. This process adds extra query processing overhead which increases with the volume of boundary objects and data volume.

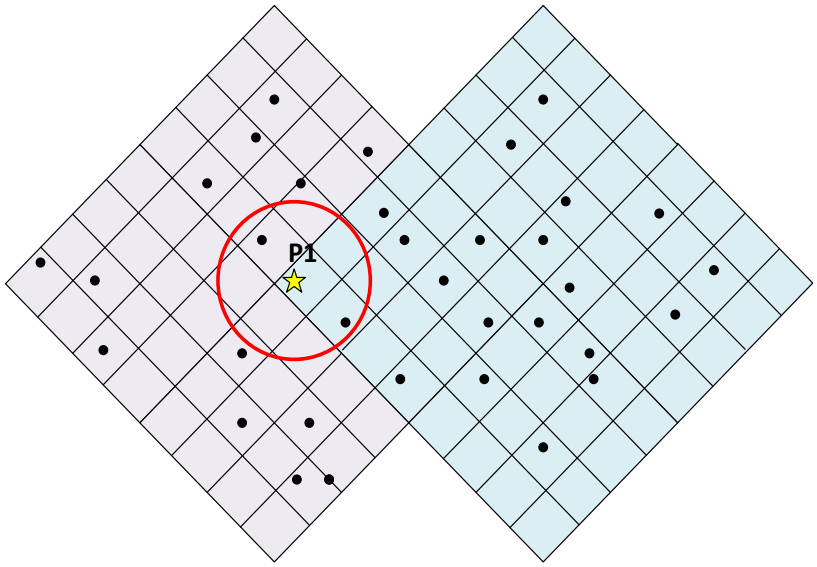


Figure 4.1 Objects on the Boundaries

4.3.3 Partitioning Cost

Partitioning algorithms of multi-dimensional data are expensive to process. This is not only driven by the massive volume of data, but also with the high computational complexity. Most of the existing partitioning approaches are designed for a centralized paradigm where query processing is performed on a single server. Among these the SkyServer [51], the VizieR service [54], Q3C/PostreSQL [56] where communication cost is not taken into consideration. Such servers simply employ horizontal or vertical partitioning approaches using sky indexing scheme which are significantly differentiated from partitioning techniques in a distributed context because of the skewed distribution and the large volume of astronomical data.

To the best of our knowledge, there is no astronomical system that provides a partitioning approach based on big data technologies that is able to handle data skew and distortion problems around the poles.

4.4 Sky Indexing

Data describing an astronomical object are represented by a celestial location right ascension and declination as well as other information related to the obser-

vation such as magnitude or spectral type. Sky indexing transforms data associated to these locations into cells (also known as pixels). It provides a single identifier for each cell on the sky. It is a simple way to represent the sky coverage of a catalog. Different data structures were designed to store spherically mapped data, they differ in their way to divide the sphere's surface. HTM and HEALPix are the most mature data structures that are widely used as indexing schemes for astronomical data. They have been put forward to meet different needs of astronomical applications for the following reasons:

- Uniform coverage of the sky with no singularities around the poles.
- Fast algorithm to compute the corresponding indices.

4.4.1 HTM

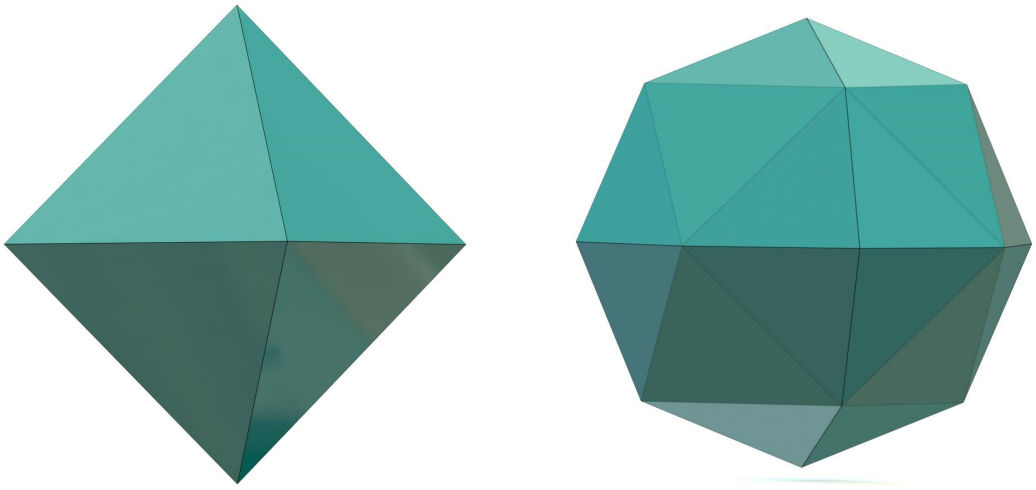


Figure 4.2 HTM Partition

Szalay et al. developed the Hierarchical Triangular Mesh (HTM) [100] and applied it to index the Sloan Digital Sky Survey (SDSS) [101]. HTM is a multi-level, recursive decomposition of the sphere. HTM starts with a spherical octahedron, which identifies 8 spherical triangles of equal sizes. Each octahedron has six vertices, given by the intersection points of the x, y, z axes, enumerated from v_0 to v_5:

$$v_0 : (0,0,1) \quad v_3 : (-1,0,0)$$
$$v_1 : (1,0,0) \quad v_4 : (0,-1,0)$$
$$v_2 : (0,1,0) \quad v_5 : (0,0,-1)$$

The first 8 triangular surfaces at level 0 are defined as:

$$S_0 : (v_1, v_5, v_2) \quad N_0 : (v_1, v_0, v_4)$$
$$S_1 : (v_2, v_5, v_3) \quad N_1 : (v4, v_0, v_3)$$
$$S_2 : (v_3, v_5, v_4) \quad N_2 : (v3, v_0, v_2)$$
$$S_3 : (v_4, v_5, v_1) \quad N_3 : (v2, v_0, v_1)$$

These spherical triangles N0 - N3 and S0 - S3 represent respectively the 4 northern and southern spherical triangles (called level 0 trixels). Further, each trixel is recursively subdivided into four children by bisecting the parent's edges (4 x 8 triangular surfaces are generated at level 1). The process is repetitive as depicted in Figure 4.2. The name of a trixel (e.g., $N20351$) uniquely defines its depth (number of digits in the name) and its location on the sky. A spherical triangle is given by three points on the unit sphere connected by great circle segments.

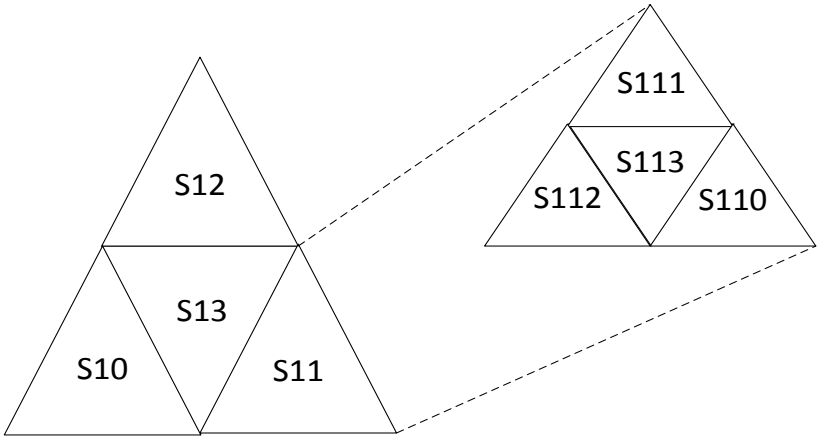


Figure 4.3 HTM Procedure

This indexing scheme leads to efficient mapping of triangular regions of the sphere to unique identifiers while keeping data locality. The number of triangles N at a given depth $d > 0$ is equal to:

$$N = 8 \times 4^{d-1} \tag{4.1}$$

4.4.2 HEALPix

The Hierarchical Equal Area isoLatitude Pixelization (HEALPix) [14; 102; 103] is one of the most popular indexing methods for astronomical data, and is commonly used to index the catalogs. In our context, we used HEALPix as a linearization technique to transform the two-dimensional data points (represented by spherical coordinates) into a single dimension value represented by a pixel identifier (HEALPix ID). HEALPix allows spatial splitting of the sky into 12 base pixels (cells). HEALPix nested numbering scheme has a hierarchical design, where the base pixels are recursively subdivided over the spherical coordinate system into four equal size pixels. These subspaces are organized as a tree and the amount of subdivisions (i.e., the height of the tree) is given by the N_{side} parameter, which controls the desired resolution (see Figure 4.5). The total number of cells per level of resolution is:

$$N_{nested} = 12 \times (N_{side})^2 \tag{4.2}$$

Where $N_{side} \in \{1, 2, 4, 8, ..., 2^k\}$

The order or the level of resolution in HEALPix is given by k (the limit is 29). Each pixel area is equal to:

$$area_{pixel} = \frac{\pi}{3N_{side}^2} \tag{4.3}$$

The HEALPix angular resolution is defined by:

$$\theta_{pixel} = \sqrt{area_{pixel}} \tag{4.4}$$

The HEALPix nested scheme follows a “Z” pattern to spatially label sequence in a specific sequence (a similar methodology of Z-ordering [104]). The details to increase the resolution are represented in Figure 4.4. Every pixel is split into 4 daughter pixels, these daughters inherit the pixel index of their parent and acquire two new bits. This nested scheme allows to represent data using a hierarchical structure and to give unique identifiers to each pixel in the map.

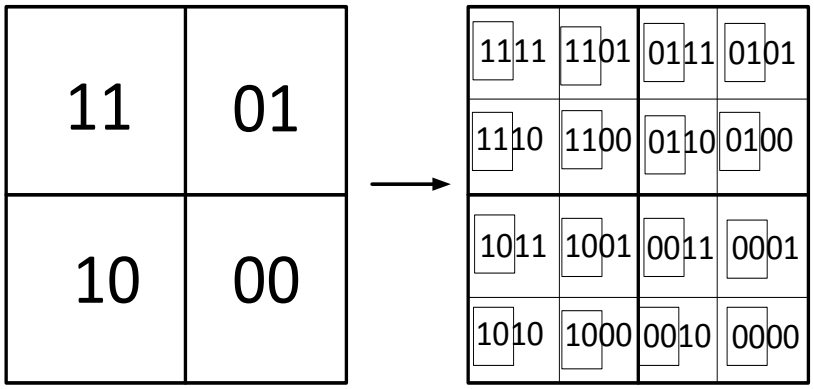


Figure 4.4 HEALPix Procedure

HEALPix offers also a second ordering scheme with rings. Ring scheme has not a hierarchical structure but it allows to align pixels on rings. It provides techniques to count the pixels moving down from the north to the south poles along each iso-latitude ring. The number of iso-latitude rings is:

$$N_{ring} = 4(N_{side} - 1)$$

Both the ring and the nested indexing schemes map the two-dimensional distribution of discrete cells to a one dimensional array of size Ncell.

As presented above, prior work has proposed other spatial indices that are suitable for celestial objects, including Hierarchical Triangular Mesh (HTM) [100]. Comparison between the two indexing techniques is detailed in [105]. Both HTM and HEALPix hierarchically partition the sky using a fixed number of cells, and each object is associated to the index of the cell that contains the object.

In this work, we choose HEALPix as our indexing scheme, but another scheme like HTM can be applied on the same principle. Compared to the astronomical servers presented in the state-of-art, AscotDB [61] and the VizieR service [54] use HEALPix as an indexing scheme, whereas the SkyServer project [51] uses HTM.

To summarize, HEALPix has the following characteristics:

- HEALPix is a mapping technique adapted to the spherical space, with equal areas per cell all over the sky, a unique identifier is associated to each cell of the sky. This id allows us to index and retrieve the data according to the spatial position efficiently.
- Data linearization with HEALPix ensures preserving data locality, neighboring points in the two-dimensional space are likely to be close in the corresponding one dimensional space. Data locality helps us to organize spatially close points in the same partition or in consecutive partitions, and thus optimizes query execution by grouping access to close objects and avoiding access to irrelevant partitions.
- There exist a ready for use library for HEALPIX [106], provided as open source by NASA. It contains many features that are useful in our context such as filtering neighbor pixels or those in a cone.

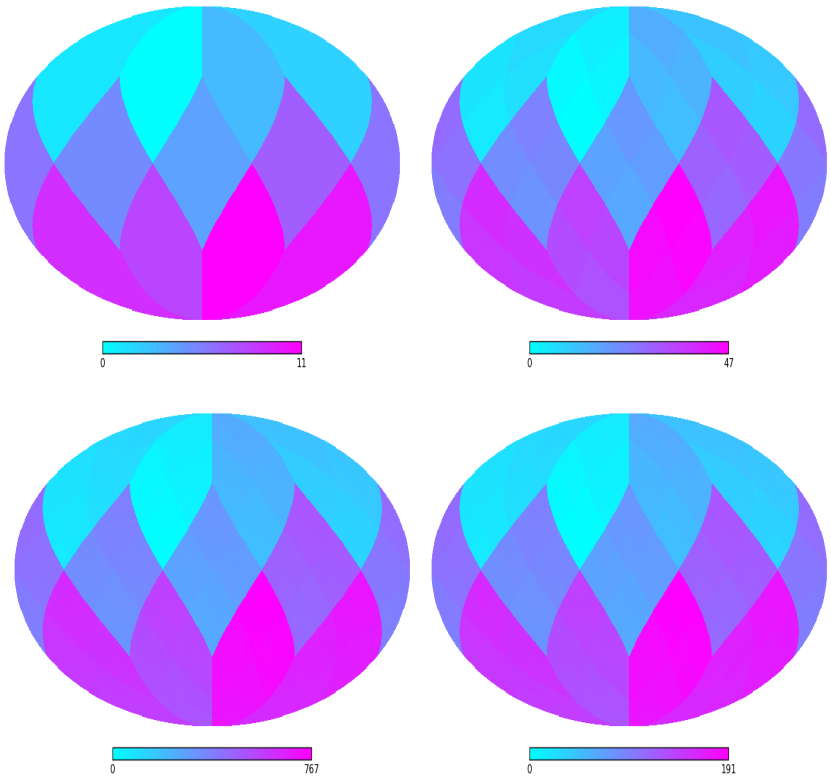


Figure 4.5 HEALPix Partition with $N_{side} = 1, 2, 4, 8$

4.5 Related Approaches

4.5.1 Astronomical Partitioning

A logical approach for speeding up astronomical queries within large datasets is to apply partitioning algorithms. This had been issued by Jim Gray in 2006 [107] by the zoning algorithm. Other partitioning algorithms were proposed in the literature:

Zones Algorithm

Jim Gray proposes a zones algorithm [107; 66] that maps the celestial sphere into stripes called zones. Each object at position (ra, dec) is assigned into a zone using this formula:

$$ZoneID = floor(\frac{dec+90}{h}) \tag{4.5}$$

Where h is the zone height (see Figure 4.6), and $floor(x)$ gives the largest integer less than or equal to x.

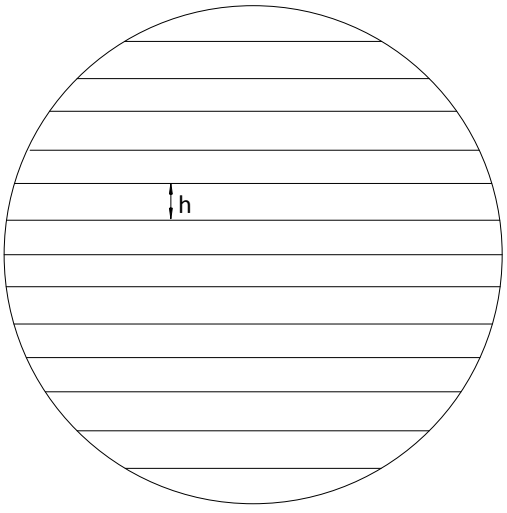


Figure 4.6 Zones Partitioning

Nieto-Santisteban et al. [107] used the zones algorithm for cross-matching queries with multiple SQL servers. It allows query execution in parallel by dis-

tributing data among a cluster of database servers. The zone algorithm helps to speed up neighborhood searches by discarding objects beyond some radius.

To look for objects within a certain radius (θ) of point (ra, dec), we need to consider zones such as:

$$\frac{dec - \theta}{h} \leq zoneID \leq \frac{dec + \theta}{h}$$

However, when the radius is much larger than the zone height, many zones are included in the computation. Thus, the zones algorithm become less efficient.

kd-tree Algorithm

A k-dimensional tree (kd-tree) (Figure 4.7) is a space-partitioning data structure for organizing points in a k-dimensional space [108], in such a way that once built, whenever a query arrives requesting a list of all points in a neighborhood, the query can be answered quickly without needing to scan every single point. It splits the multi-dimensional space recursively into subspaces in a systematic manner and organizes these subspaces as a search tree.

This structure is suitable for spatial data partitioning. But, it was also used to partition astronomical data in the literature. Gao et al. [109] propose an algorithm based on kd-trees using HTM [100] as a spatial index. The idea is to divide the sky into small equal triangles by HTM. They map each object `(ra,dec)` of the dataset to a HTM index, and for each triangle, they build a kd-tree. The use of HTM with kd-trees was also proposed by Kunszt et al. [101] to execute queries on data from the SDSS archive.

Pineau et al. [67] use also kd-trees to partition data. They divide the sky into HEALPix pixels that can be processed independently, one by one on a single machine and simultaneously on a cluster of machines. They propose an independent pixel cross-match. For example, to cross-match the sources contained in a cell of a catalog A with a catalog B, they load only the sources of the catalog B which are overlapping this cell. To overcome the border issue, they also load an extra border around this cell. Then, they build a 2d-tree containing the sources retrieved from the catalog B. For each source in the catalog A, they perform a cone search query in the 2d-tree. The process is multi-threaded and stops when all sources are processed.

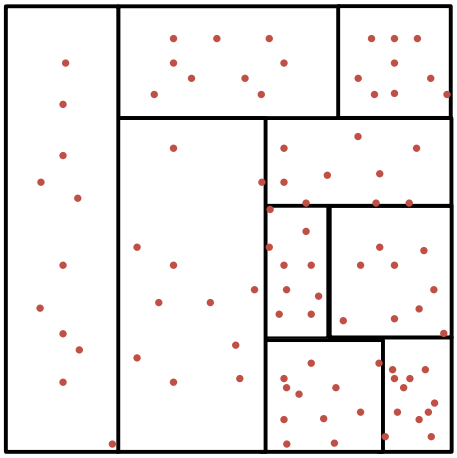


Figure 4.7 kd-tree Partitioning

4.5.2 Spatial Partitioning and Indexing

Many spatial indexing algorithms were proposed in the literature such as grid, R-tree, and R+-tree, Z-curve, Hilbert curve, quad-tree, and k-d tree. They were presented in details in [110; 111; 112]. It is not simple to apply such algorithms in a distributed context because of the granularity of data management. A unique index is not suitable. Thus, the general idea (as discussed in Chapter 2), is to define a two level indexing approach of global and local index.

- The global index partitions data at node level, contributes in the organization of partitions, and helps to identify the partitions that are relevant for the query range.

- The local index organizes data inside each node and limits the access to blocks within each partition to accelerate range filtering.

HadoopGIS [71], GeoSpark [13], SpatialHadoop [10] and SIMBA [11] employ R-trees global and local indexing. The partitioning approach in ASTROIDE is inspired from this idea. ASTROIDE adopts a similar approach by using range partitioning to organize partitions at the node level and HEALPix indexing to organize data inside each partition.

Grid Indexing

Grid indexing [113] is a simple space-oriented, in which the spatial universe is partitioned into n equal sized grids. It is a fast and easy to implement technique. However, it cannot handle the spatial data skewness. If the data distribution is not uniform, it fails at balancing the workload among the nodes of the cluster. Grid indexing is reserved for applications that have uniform data distribution or if the choice of a cell size gives an approximate equal density that absorbs the skew.

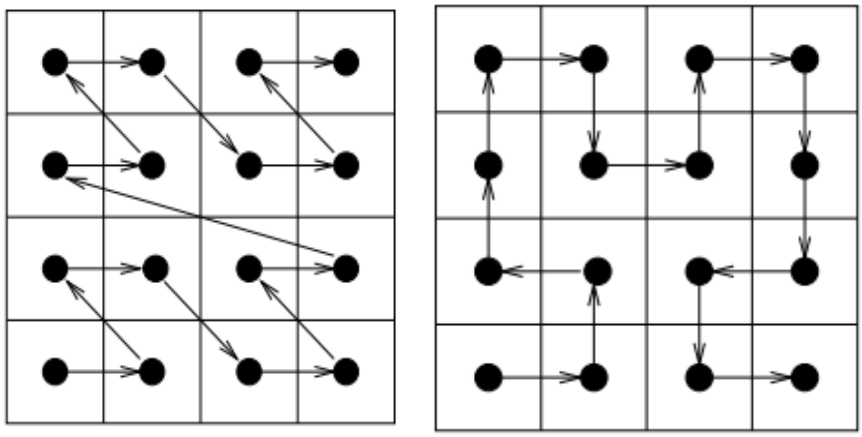
Space Filling Curve

A Space Filling Curve (SFC) is a way of mapping the multi-dimensional space to the one-dimension space. It passes through every cell element in the D-dimensional space so that every cell is visited exactly once [114]. Different SFC techniques were proposed in the literature. The difference between such curves is in their ways of mapping to the one dimensional space (Figure 4.8).

One of the most desired properties of such mapping techniques is data locality, it means that the locality between objects in the multidimensional space is being preserved in the linear space. The transformed data can be stored in a traditional one-dimensional database, allowing the efficient process of spatial queries, such as range query [115] or kNN query [116]. Sophisticated mapping functions have been proposed in the literature such as:

- Z-curve: Based on interleaving bits from coordinates values [104]
- Hilbert Curve: A continuous fractal space-filling curve, introduced by David Hilbert in 1891 [117]. It is represented by a sequence of curves defined iteratively. [118].

Yao et al. [116] propose efficient algorithms using Z-curve space filling curve techniques to implement kNN queries with SQL operators in relational databases. Lawder et al. [115] employ the Hilbert Curve to index multi-dimensional data and execute range queries. In MD-HBase [12], a Z-curve is used for data partitioning. The data storage layer stores the items sorted by their keys that correspond to the Z-values of the dimensions being indexed and range-partitions the key space. MD-HBASE proposes also efficient range queries algorithms using Z-curve detailed in Section 2.4.4.



(a) Z-curve (b) Hilbert Curve

Figure 4.8 Space Filling Curve

R-trees

R-trees are hierarchical data structures derived from the B-trees, originally introduced by Guttman [119]. R-tree is the preferred method of spatial data indexing in which each geometric object is represented by its minimum bounding rectangle (MBR). Leaf nodes contain entries of the form (id, R) where id is the identifier of the spatial object being indexed, and R is the MBR of the object. Non-leaf nodes contain entries of the form (ptr, R) where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. R-tree follows a height-balanced tree, all leaves of the R-tree are at the same level.

Figure 4.9 depicts some objects on the left and the corresponding R-tree on the right. Data rectangles D,E,F,G are stored in leaf nodes. Whereas MBRs A, B and C are the root nodes. A, for instance, covers child nodes D, E, F and G.

A Sort-Tile-Recursive (STR) algorithm is used to build an R-tree. The general method is similar to building a B-tree where we start by creating the leaf level nodes and then creating each successively higher level until the root node is created. Considering N the number of rectangles in the two-dimensional space, the idea is to tile the space using vertical slices, so that each slice contains enough rectangles to create roughly $\sqrt{(N/M)}$ nodes, where M is the R-tree node capacity. Then, we sort the rectangles by x-coordinate and S slices are created ($P = \lceil N/M \rceil$ and $S = \lceil P \rceil$). A slice consists of $S.M$ rectangles. In each slice, objects are sorted by y-coordinate and packed into nodes. This procedure continued recursively.

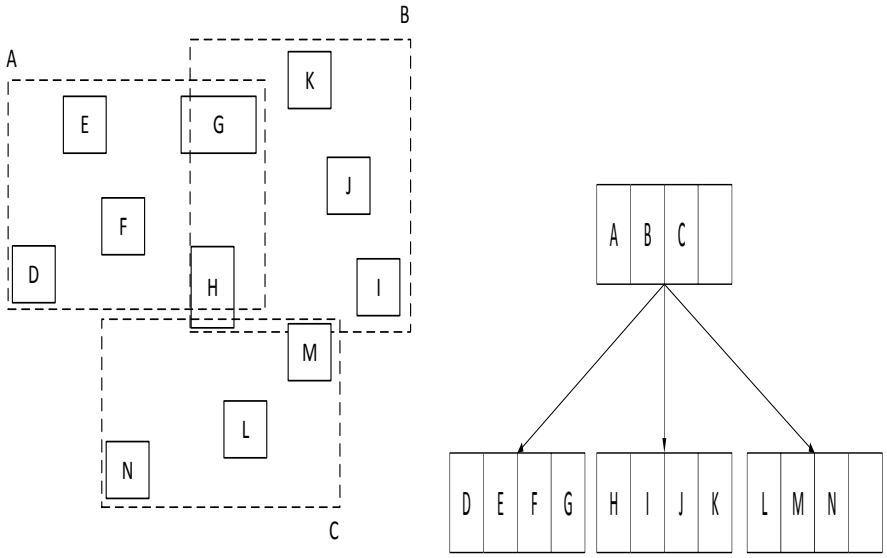


Figure 4.9 Example of R-tree

It should be noted that the MBRs of different nodes may be overlapping. This means that a spatial search may visit many nodes. R^+-trees [120] were proposed as a structure to overcome this limitation. They improve retrieval performance by avoiding visiting multiple paths. This is achieved by using the clipping technique. The MBRs of the nodes at the same level are disjoints. So, inserted objects have to be divided into two or more MBRs, causing duplication of some object's entries.

SpatialHadoop [10] and SIMBA [11] use R-Tree and R^+-Tree for data partitioning and indexing. These methods have been proven to be efficient for spatial data processing compared to other spatial partitioning algorithms.

The main problem of applying these algorithms in our context is that the positions of astronomical objects are given in spherical coordinates. Their use is restricted to tasks inside small areas, where the sphere can be projected onto a plane with negligible distortion. Thus, most of the systems operating with astronomical data use sky indexing techniques by subdividing the sky into numbered small fragments.

4.6 Spark Partitioning Approaches

Because we have chosen Spark as our back-end framework, we started by studying the partitioning approaches offered by such framework. Our objective is to

extend the built-in partitioning methods in Spark to support astronomical data partitioning.

To support data partitioning, Spark divides the input RDD into a collection of partitions. Each machine allows the process of many partitions. The number of launched Spark tasks is equal to the number of the Spark partitions.

Spark provides two data partitioning schemes, namely hash partitioner and range partitioner that can be executed over RDDs or DataFrames.

- **Hash partitioner**. The default partitioner in Spark. It calculates a partition index based on an element's Java hash code, and puts the keys that have the same hash index in the same partition. Hash partitioning does not necessarily distribute data uniformly. Consequently, it can end up with few partitions containing most of the records.
- **Range partitioner**. It divides data into roughly equal partitions. Range partitioner sorts the input records based on their keys and splits the RDD into a defined number of partitions, each of which contains data with keys within a specific range. Range partitioner provides better workload balance. However, it does not support the two-dimensional nature of astronomical data. It also lacks of mechanisms to determine the best number of partitions for a specific job.

4.7 ASTROIDE Partitioning

Our partitioning approach follows a similar approach of the two level partitioning model used in the geo-spatial context (global and local indexing). The general idea is to leverage the spark range partitioning algorithm to efficiently distribute data across nodes and avoid accessing unnecessary partitions. We have also adopted HEALPix sky indexing scheme as a linearization technique to organize data inside each partition into buckets.

In ASTROIDE, it is important to determine how many partitions are needed. The partitioning algorithm has to ensure that all the created partitions would fit into memory. In Spark, the number of partitions determines the level of parallelism. Currently, Spark uses a defined number of partitions. Large partitions leads to long task execution. Therefore, a heuristic calculation must be used in

ASTROIDE. The number of partitions is automatically determined based on the dataset size, which adjusts also the number of Spark tasks. The number of partitions can be obtained by dividing the total dataset size by the target partition size. However, a variation in the sizes among the partitions can occur. This may happen when the density of the first or the last cell, i.e., corresponding to the first or the HEALPix ID in the partition is dense. Thus, we allow a margin with m factor in order to absorb such unpredictable cases. In the experiments, it has been empirically fixed to 0.3. We compute the number of partitions using this formula:

$$n = |IS|/PS * (1+m)$$

Where $|IS|$ is the input size.
PS is the partition size.
m is an adjustment factor.

Algorithm 1 Data partitioning and indexing

Input: $InFile$: Input File, PS: Partition size
Output: $OutFile$: Partitioned output file, $boundaryList$: Partition boundaries

1: $dataFrame$ = *Load*($InFile$)
   /* Index creation using the HEALPix Library */
2: $IndexedDF \leftarrow \emptyset$
3: **for** each row in $dataFrame$ **do**
4: $ipix$=*toHealpix*(ra,dec)
5: $IndexedDF = IndexedDF \cup \{row + (ipix)\}$
6: **end for**

7: $n \leftarrow \frac{Size(InputF)}{PS} * 1.3$
8: $partitionedDF$ = *RangePartitioning*($IndexedDF$, $ipix$, n)

9: $OutputDF \leftarrow \emptyset$
10: **for** each partition P in $partitionedDF$ **do**
11: $nump$ = *getPartitionId(P)*
12: $OutputDF = OutputDF \cup \{P + (nump)\}$
13: **end for**

14: $OutFile$ = *Save*($OutputDF$)

15: /* Partition boundaries creation */
16: $boundaryList$ = *getBoundaries*($OutputDF$)

The data partitioning and indexing processes are shown in Algorithm 1. Input files are stored using parquet format, a column-oriented binary file format. Parquet with compression reduces data storage, it allows reading only records of interest through selected columns. Input files are loaded as DataFrames (Line 1), equivalent to relational tables (see Section 3.2.4 for more details). The two-dimensional coordinates are mapped to a single dimensional ID represented by the HEALPix value (Line 2-6). This library helps us to linearize data since the partitioning key in Spark should be in one dimension. So, a new Healpix column is added to the input DataFrame. Then, we apply a range partitioner that divides the input DataFrame into n partitions (Line 8). The records are partitioned by ranges into roughly equal partitions. Since Spark is an in-memory analytical engine, this efficient organization is only accessible in-memory. So, our solution is to associate a partition number $nump$ to each partition in memory (Line 9-13) and pack each partition with this number in HDFS. The function *getPartitionId* tracks the partition number in memory. This allows to ensure a correspondence between the partitions created in memory and the partitions that are physically created in HDFS. The output DataFrame is saved using Parquet format in HDFS. The function *save* (Line 14) divides each partition of $partitionedDF$ into buckets and uses $ipix$ as the bucketing column (the values of HEALPIx cells inside each partition are hashed by a user-defined number into buckets). This function creates a file in HDFS with $nump$ as the top-level partition and $ipix$ as the second-level partition. Finally, a sorted list of the range boundaries for each partition is created (Line 15).

Figure 4.10 shows ASTROIDE's partitioner. For partitioning an input data into a set of partitions R_0, R_1, R_2, ASTROIDE builds HEALPix indices over raws inside each partition R_i. We apply range partitioning on partitions R_i to create sorted partitions $P_0, P_1, ..., P_n$ given the number of partitions n. This causes a shuffling and distributes evenly the data among partitions. We store partitioned data in HDFS to amortize the construction cost for future queries. The DataFrame is partitioned by $nump$ using a Spark existing function *partitionBy*. So, ASTROIDE stores the data in such a way that each partition is saved in a separate subdirectory containing records with the same partition number $nump$. Each partition is further divided into buckets using the spark function *bucketBy* using HEALPix index as the bucketing column. This structure provides a two-level partitioning structure as shown in Figure 4.11. Astronomical datasets are partitioned into independent partitions.

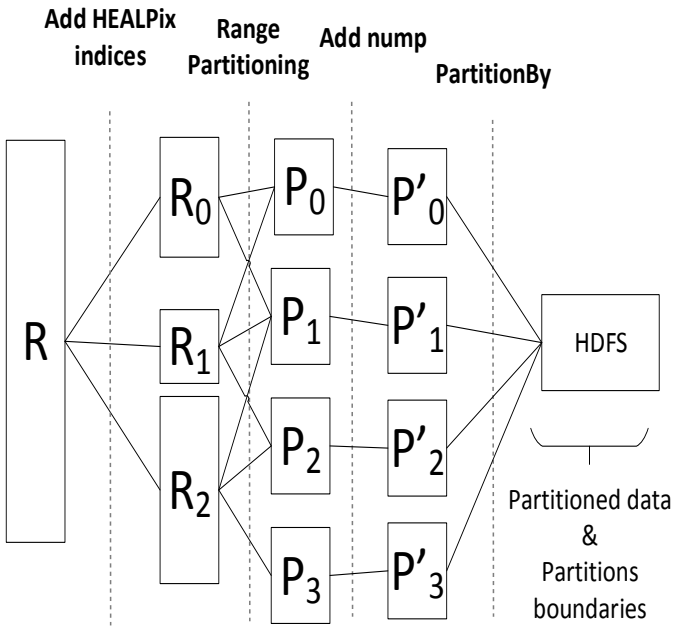


Figure 4.10 ASTROIDE Partitioner.

For simplicity, the space is partitioned into three regions in Figure 4.11. The sizes of these regions are decided such that each region would fit into a partition size PS. Each region can be further divided into even smaller regions called buckets. In Figure 4.11, a single partition is split into 3 tiles (green, yellow, red). Such two-level partitioning could save large I/O cost and improve query performance.

This technique optimizes query execution in a way that makes it efficient to retrieve the contents of a bucket and obviate scanning irrelevant partitions. Records with the same HEALPix indices will be stored in the same bucket. This structure is very useful, for example, if a query limits for a catalog GAIA only records located in cell `ipix=114571`, ASTROIDE will scan the contents of one bucket in one subdirectory (e.g., `gaia.parquet/nump=20/ipix=114570-114575`).
In the final step, ASTROIDE determines partition boundaries and stores them as metadata. Note that in our case, all we need to store are the three values `(np, l, u)` where `np` is the partition number, `l` is the first HEALPix cell of the partition number `np` and `u` is the last HEALPix cell of the partition number `np`.

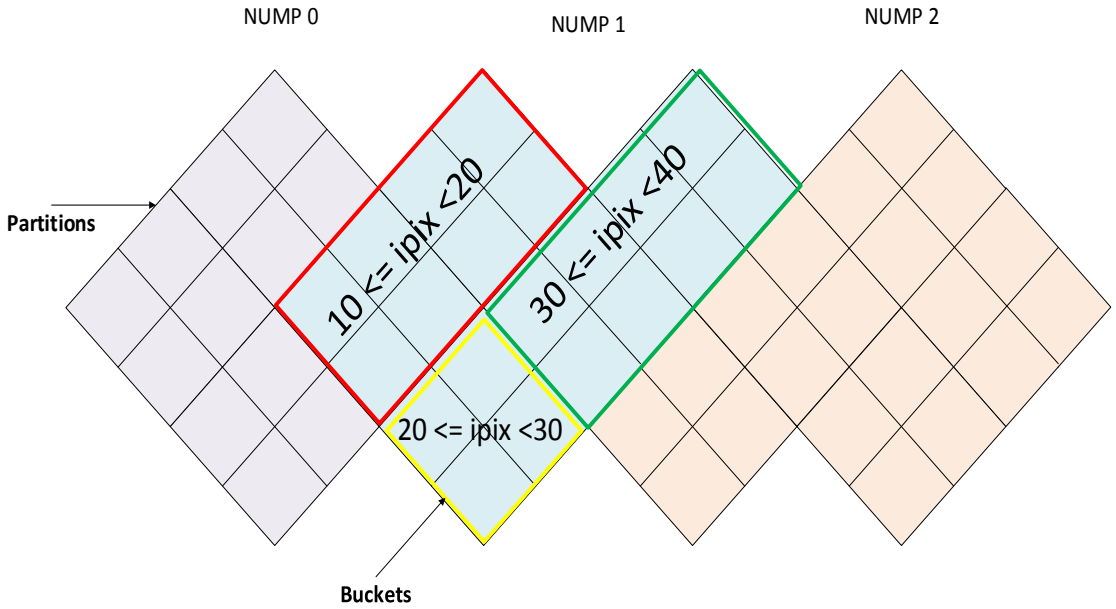
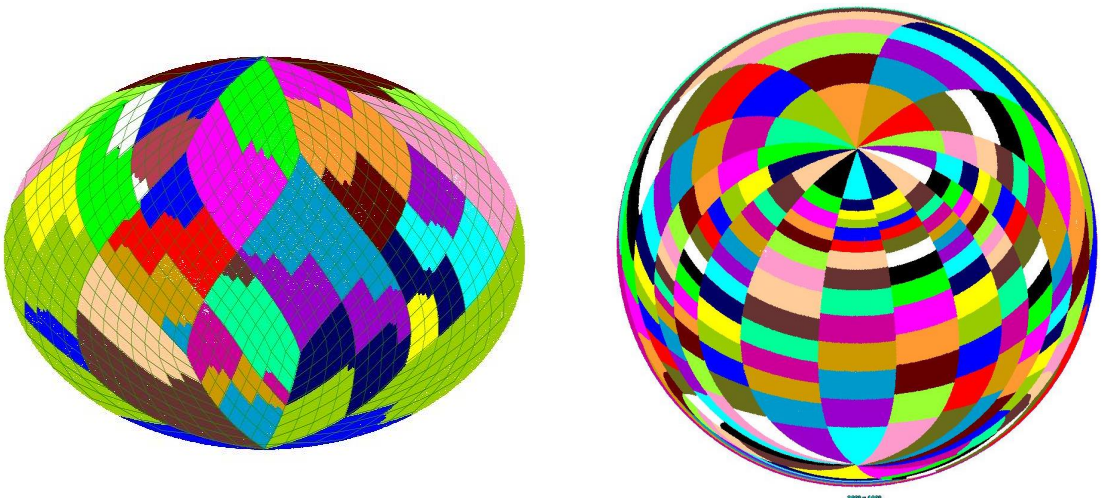


Figure 4.11 Two-level Partitioning.

4.8 Partitions Visualization

We visualized the created partitions in ASTROIDE using Aladin [121], a tool for viewing astronomical data and acquiring sky maps. As shown in Figure 4.12a, each partition is represented by a color, a partition corresponds to a region of the sky and all created partitions cover all the sky. An approximate balance between different partitions is considered. Our partitioning algorithm preserves spatial locality which means that nearby objects are assigned to the same partition. For each object of a given partition, the corresponding HEALPix value is assigned to a specific range, and when a data entry fits that range, it is assigned to that partition; otherwise it is placed in another partition where it fits. Thus, partitions are contiguous but not overlapping, each partition has an exclusive upper bound HEALPix value.

Figure 4.12b illustrates partitioning in SIMBA. It uses an R-tree based on STR partitioning [122]. This data structure groups nearby objects and represents them with their minimum bounding rectangles. We can observe that the bounding boxes become very elongated around the poles. This may hinder the query performances, due to the increase of the objects along the border, which entails multiple partitions access.



(a) Partitioning in ASTROIDE.

(b) Partitioning in SIMBA.

Figure 4.12 Partitions Visualization with Aladin.

4.9 Summary

In this chapter, we presented a study of different partitioning and indexing algorithms for astronomical data. Our study reveals the important challenges that should be considered for effective partitioning algorithms for astronomical data. We also discussed the most important partitioning algorithms in the context of geo-spatial big data. We described our new partitioning algorithm based on HEALPix indexing scheme to achieve efficient astronomical query processing.

CHAPTER 5

Optimization of Astronomical Queries

Contents

> Information is the oil of the 21st century, and analytics is the combustion engine.
>
> Peter Sondergaard

5.1 Introduction

In Spark SQL, a user expresses a query that the final response must satisfy without describing the computation. The optimizer responsibility is to translate this query into a query execution plan that returns the requested result efficiently. However, the application of standard query optimization techniques in the context of astronomical query processing is not sufficient due to astronomical data challenges (see Section 1.2). Furthermore, very little attention was devoted to astronomical query processing using big data systems. Existing astronomical servers discussed in Section 2.3 are not able to handle the colossal increase of large scale data and neglect the optimization issues. This chapter presents our query optimization module which allows to produce an optimized query execution plan for the most frequent and challenging astronomical queries.

5.2 Query Processing

Query processing in our context focuses on the design of efficient algorithms for astronomical operators. The performance enhancement provided by these algorithms includes access methods using sky indexing techniques and partitioning algorithms.

Query processing in ASTROIDE follows a two-step process comprised of filtering and refinement steps [123]:

- Filtering step. It does not return the exact result for the original query but a set of candidates objects that is a superset of the result objects. This step takes advantages of our partitioning and indexing model to obtain candidates objects. It reduces not only I/O time but also CPU time and communication costs.
- Refinement step. The output of the filter step is fed into the refinement step. The refinement step finds exact answer to the original query where candidate objects are processed using spherical distance filtering.

The objective is to discard as early as possible unnecessary data to optimize query execution. We do not need to call the refinement step on non candidates objects as we are certain that they definitely do not belong to the query result.

5.2.1 Query Parsing

ASTROIDE begins query processing either from an ADQL query or from a DataFrame object. The ADQL query is analyzed in order to identify geometrical and traditional algebraic predicates. The query parser checks input queries for syntactic correctness and translates them into SQL queries with UDFs. The transformations performed depend on the type of the query and substitute parts of the query matching ADQL's geometrical terms with replacement SQL terms using UDFs that are already defined in ASTROIDE. This substitution is based on an existing library [83] developed by the CDS. We extend this library to integrate ASTROIDE UDFs.

For example, the ADQL query 3.2 used to express a cone search query is transformed into a Spark SQL-compliant query by replacing the `CONTAINS` function with a UDF expressing a spherical distance as follows:

Query 5.1 Query 3.2 after Parsing

```
SELECT *
FROM gaia
WHERE SphericalDistance(ra, dec, 266, −29) < 0.0833;
```

5.2.2 Query Optimization

Our query optimizer uses different methods to optimize astronomical queries: with query rewriting or with transformation rules.

Optimization with Query Rewriting

In this subsection, we consider the case of query rewriting with the integration of transformations that optimize the query. The first step is to check whether query rewriting is needed. After that, the optimizer must determine how it will rewrite the query. It makes this type of determination by defining the type of the query. This is accomplished by individually checking the content of various clauses (`SELECT`, `FROM`, `WHERE`, `ORDER BY`, or `LIMIT`). Depending on the type of the query, the query optimizer adds to the query expression new optimization clauses.

This approach consists in rewriting astronomical queries into optimized SQL queries with UDFs. The basic idea is to apply a number of transformations on

the query expression to produce a new equivalent, but optimized query compliant with the syntax of Spark SQL. We incorporate filters into the optimized query in order to avoid scanning irrelevant partitions and buckets (see Query 5.3 & Query 5.5). We also transform the cross-match query into an SQL query that uses an equi-join on HEALPix indices (see Query 5.7).

Query rewriting performed in ASTROIDE is completely transparent to the user. It does not require any change inside the Catalyst optimizer and executes exactly the same execution plans of those generated by the second approach (using transformation rules) that we present in the next section.

Optimization with Transformation Rules

In ASTROIDE, the second approach for query optimization makes use of the Catalyst optimizer as a back-end and introduces new features that extend its base components. Once the AST is generated from the ASTROIDE parser, the query optimizer performs several operations on the query tree by applying transformation rules and strategies. Using Catalyst, we take advantages of existing techniques for non spatial query optimization and provide an efficient implementation for astronomical queries using indexing and partitioning techniques.

5.3 Query Optimization Worflow

The workflow of query optimization in ASTROIDE is represented in Figure 5.1. Our query optimizer consists of four major steps: Extended Analysis, Extended logical-physical optimizations, Physical planning and Code Generation.

5.3.1 Extended Analysis

Spark SQL begins with a relation to be computed, the relation may contain unresolved attribute references. An attribute is called unresolved if we do not know its type or have not matched it to an input table. In addition to analyzing the compatibility with metadata, Spark SQL allows to overcome some current limitations by injecting resolution rules. For example, the kNN join query (see Query 5.11) uses correlated sub-queries to select data from the table R referenced in the outer query. However, the current version of Spark SQL does not support such

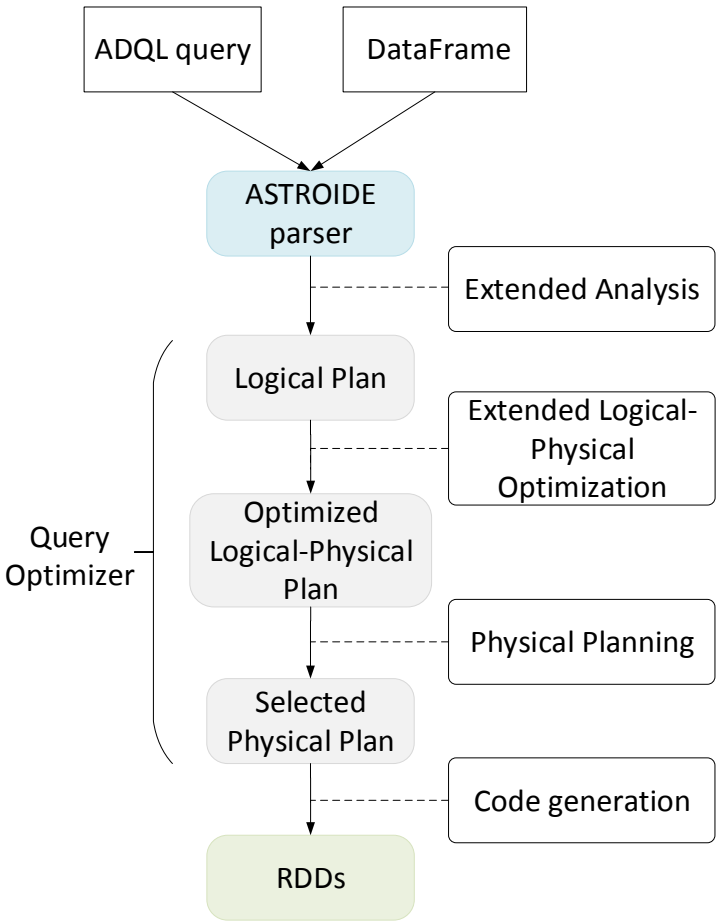


Figure 5.1 Query Optimization Worflow

sub-queries with UDFs. Thus, attributes $R.ra$ and $R.dec$ can not be resolved. An "unresolved logical plan" tree is built, then, we apply a special type of rules called analyzer resolution rules using the method *injectResolutionRule* to transform an impossible-to-solve plan into an analyzed logical plan.

5.3.2 Extended Logical-Physical Optimizations

The logical optimization applies rules based optimization to the logical plan. In our context, we exploit this feature, not only for a pure logical optimization, but also to transform the logical plan. The logical plan is transformed into an optimized logical-physical plan using indexing and partitioning metadata. In fact, using these rules allow us to solve the filtering phase of most queries by either transforming the spatial predicates and join to scalar counterparts (efficiently processed in Spark SQL), and/or filtering the relevant partitions by using HEALPix

and the metadata. This will be illustrated for each query type in Section 5.5, Section 5.6, Section 5.7 and Section 5.8. We identify sub-trees that match astronomical predicates and let ASTROIDE applying multiple optimization rules as explained in Section 5.4. Each rule focuses on a specific optimization to optimize astronomical queries and allows to map one query plan to another semantically equivalent plan. Sub-trees corresponding to non-spatial predicates are processed by Catalyst. This step is essential to benefit from this back-end as much as possible.

5.3.3 Physical Planning

In the physical planning phase, ASTROIDE takes an output query plan and generates a physical plan. The logical plans that are usually ineffective are pruned by heuristics, the use of a cost model, or both. Physical operators used in the physical plan are selected in ASTROIDE using Catalyst. At the moment, we didn't implement any cost model. We rely on Catalyst to select the best operators for the physical plan. For example, the actual proposed cost model in Catalyst can be used to select equi-join algorithms: for small relations, Spark SQL uses a broadcast join on Spark. The selected physical plan is executed on Spark RDDs and required information is generated to the user.

5.4 Rule-based Optimization in ASTROIDE

Ramakrishnan et al. [124] states query optimization as the process of selecting the most efficient query evaluation plan for a query. The optimizer considers multiple query plans for a given input query. These plans are equivalent in the sense that they generate the same result but differ in the execution order of the operators and therefore on performances. The query optimizer identifies which of those plans will be the most efficient.

Our query optimizer uses rule-based optimization that utilizes transformation rules to transform an initial query tree into an optimized query tree. Beyond inheriting Catalyst rules, it injects new customized rules. It introduces relevant methods that triggers the new rules to optimize astronomical queries. Indeed, we propose

a partition-aware optimizer that uses laws for improving query plans as listed below. We show how to apply rules to improve the query tree. This section presents the rules that turn one query tree into another query tree that may have a more efficient physical query plan. The result of applying these transformations is represented by the QEP.

5.4.1 Partitions Pruning

This optimization consists in pruning unnecessary partitions from consideration. This limits the number of partitions that ASTROIDE has to load into memory. ASTROIDE only reads the records in the partitions that satisfy the ADQL query and performs operations only on those partitions. Partitions pruning can reduce the amount of data retrieved from HDFS, shortens the query execution time, improves query performance and optimizes resource utilization. We inject new optimization rules which determine partitions that need to be scanned, and hence partitions that can be pruned, to satisfy the astronomical query.

5.4.2 HEALPix Pushdown

Predicate pushdown is a logical optimization that consists in pushing down filtering operations into the data source. In ASTROIDE, we add new rules that use indices defined on HEALPix values to eliminate loading cells that do not contribute to the query result. We inject new filters in the operator tree that aim to remove extraneous objects and reducing the amount of loaded data at the data source level. This is a great optimization that increases the performance of astronomical queries since the new filters on HEALPix cells are performed at the low level rather than dealing with the entire dataset which can help to avoid memory issues. Pruning data reduces I/O, communication and CPU costs to optimize ASTROIDE's performance. This optimization is possible thanks to Catalyst. The injection of such rules in the logical plan allows the use of parquet metadata in the physical plan. This provides a way to skip row groups based on the min/max values. Parquet can store statistics, in particular the min/max values in the row group metadata. This can be used to filter out the rows that do not match the condition.

5.4.3 Merge non-spatial and geometrical Filters

This optimization occurs when a query has multiple predicates including a geometrical predicate. ASTROIDE can efficiently handle ADQL clauses with multiple filters that are separated by an operator such AND, OR. The idea is to combine the refinement step on spherical distance and the non-spatial predicates in the same filter operation (see second operation in Figure 5.2b). In other terms, non-spatial operators can be merged with the refinement step into one physical operator.

5.4.4 Avoid Cartesian Product

Complex astronomical queries may involve spatial joins with spherical distance functions. These functions are expressed as UDFs that are considered as black boxes. Thus, complex queries can be conceptually formulated as Cartesian products. Avoiding a Cartesian product is a common heuristic to reduce the search space. Our optimizer utilizes this optimization and replaces the Cartesian product with an equi-join on HEALPix indices. For instance, for cross-matching queries, our optimizer do not consider the query execution plan represented in Figure 5.5a as part of the search space.

We demonstrate the rules presented above by giving examples of the main astronomical queries in the following sections.

5.5 Cone Search

The cone search query is written using the following ADQL expression. Query 5.2 allows to select sources within a certain angular distance from a specified center position. It also uses a predicate to filter objects fitting a certain magnitude range [10,18].

Query 5.2 Cone Search with Filter (ADQL)

```
SELECT *
FROM gaia
WHERE magnitude>=10 AND magnitude<=18
    AND 1=CONTAINS(POINT('ICRS',ra,dec),
    CIRCLE('ICRS',266,-29, 0.0833));
```

5.5.1 Baseline Approach

A plan without optimization requires traversing all the data in R and collecting the objects that lie inside the specified circle with the specified magnitude range. It has to scan the entire input table to execute a cone search query without any accelerating data structure (e.g., index). A baseline plan is represented by the QEP presented in Figure 5.2a.

5.5.2 Optimization with Query Rewriting

By leveraging the proposed data organization, such query can be processed using the filter-refine paradigm. In the filtering step, partitions that are disjoint from the query cone can be filtered out. In the refinement step, exact candidates objects can be returned with accurate geometry test.

If we apply query rewriting, Query 5.2 is transformed into Query 5.3. Notice that we integrate in the transformed query the filtering of relevant partitions and buckets which are programmatically computed using our query optimizer.

Query 5.3 Query 5.2 after Rewriting

```
SELECT *
FROM gaia
WHERE (magnitude>=10 AND magnitude<=18
      AND nump IN (100)
      AND ipix IN (114571,114572,114580)
      AND SphericalDistance(ra,dec,266,-29)
       <0.0833));
```

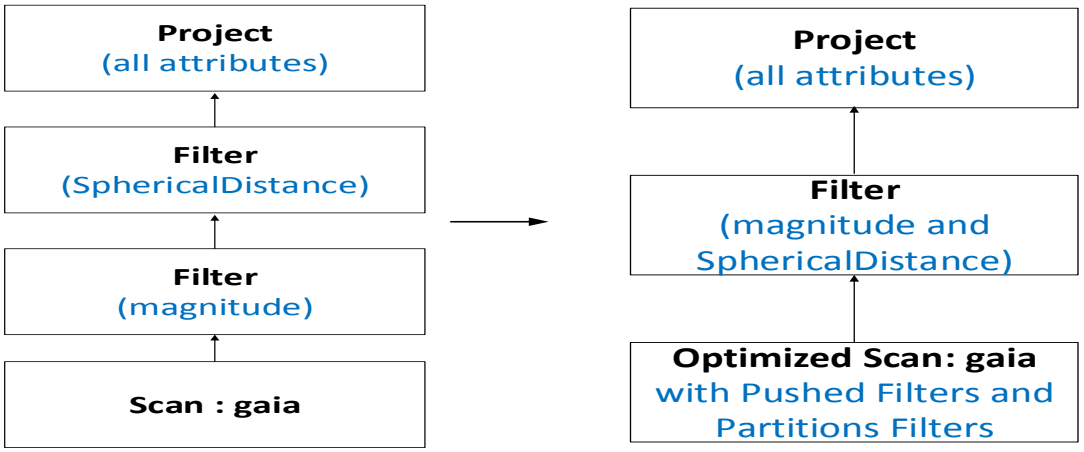
The filtering condition on `nump` ensures that only partition number `100` is loaded into memory, the filtering condition on `ipix` uses our partitioning model based on buckets to push the predicate down into the data source rather than dealing with the entire partition. The two filtering conditions ensure that only the sky region intersecting the cone is loaded into memory. Finally, another filtering step is required to select exact objects' harvesine distance using the registered UDF called SphericalDistance.

5.5.3 Optimization with Transformation Rules

Given a cone search query, the second alternative is to use transformation rules by Catalyst. Our optimizer first identifies the pixels without accessing the data, then uses the indexing and partitioning scheme to retrieve the actual data having these HEALPix IDs, and finally applies the harvesine distance with the candidates to get the result. We leverage the facility offered by Spark to only access the data belonging to some given partitions. Thus, we inject optimizer rules to:

- Find all pixels (i.e. cells) within the query cone using the HEALPix library,
- Group these cells by range and use the partition metadata to obtain IDs of partitions which overlap these ranges,
- Filter only required partition(s)
- Use our local indexing over HEALPix IDs, based on buckets, to push the predicate down into the data source rather than dealing with the entire partition(s).
- Select exact objects inside the cone according to the harvesine distance.

These optimizations consist in using both pushed filters (Section 5.4.2) and partition filters (Section 5.4.1) to prune the search space, as shown in Figure 5.2b.



(a) Without Optimization (b) After Optimization

Figure 5.2 Plan Transformation of Query 5.2

5.6 kNN Search

We considered an example of a kNN query that aims to find the 10 closest sources to a star represented by a position p=(44.97, 0.09) in the `gaia` catalog .

Query 5.4 kNN (ADQL)

```
SELECT TOP 10 ra,dec, DISTANCE(POINT('ICRS', ra, dec),
        POINT('ICRS', 44.97, 0.09)) AS dist
FROM gaia
ORDER BY dist;
```

5.6.1 Baseline Approach

In general, the baseline approach to execute a kNN query requires to scan all objects in R, to calculate their distances to p, to order them by distance, and finally to take the top-k objects. Such query requires at least a sort with a complexity in $O(n * log_n)$.

5.6.2 Optimization with Query Rewriting

Similar to spatial selection queries, the filtering phase for relevant partitions can be also applied to avoid scanning the entire input dataset. But, unlike the cone search, the identification of the required HEALPix cells and the concerned partition is far from being obvious (see Section 5.6.3).

The kNN search query is rewrote into an equivalent SQL query as:

Query 5.5 Query 5.4 after rewriting

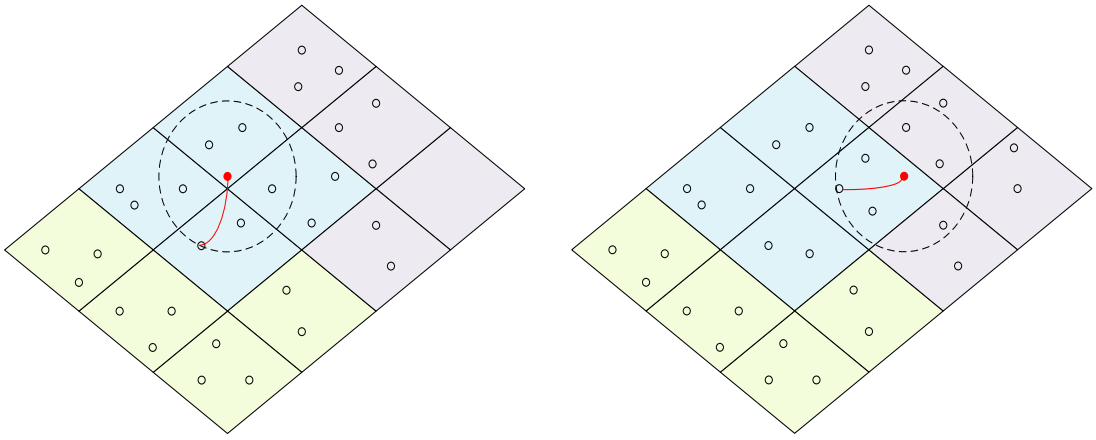
```
SELECT ra,dec, SphericalDistance(ra,dec,44.97,0.09) AS dist
FROM gaia
WHERE nump = 120
ORDER BY dist
LIMIT 10
```

For this query, we replace the TOP keyword of ADQL with LIMIT keyword to make the query comprehensible by the Spark SQL parser. After that, we add a filtering condition to the query that is programmatically calculated. This filter

discards unnecessary partitions before calculating the distance function, leading to an overall improved run-time.

5.6.3 Optimization with Transformation Rules

We propose an algorithm, which, first, identifies the cell that includes the point p using the HEALPix library. Then, it locates the partition that intersects this cell using the partition metadata that we created using our partitioner. The initial result is set as kNN objects restricted to the partition of p. For objects close to the borders of the partition, the obtained answer can not be considered final since potential neighbors from other partitions might be closer than the kth one in the current partition. Therefore, in our approach, we consider the distance to the kth neighbor of p from the initial answer as a search radius (it is determined by the farthest point from p). Then, we draw a cone centered at p with the calculated radius. If all the cells in the cone belong to the target partition (Case 1 in Figure 5.3a), then the algorithm considers the initial answer as final. If not (Case 2 in Figure 5.3b), we execute a cone search query using the calculated radius, and we take the top-k records. Figures 5.3a and 5.3b illustrate the two cases of a kNN query with k = 6.

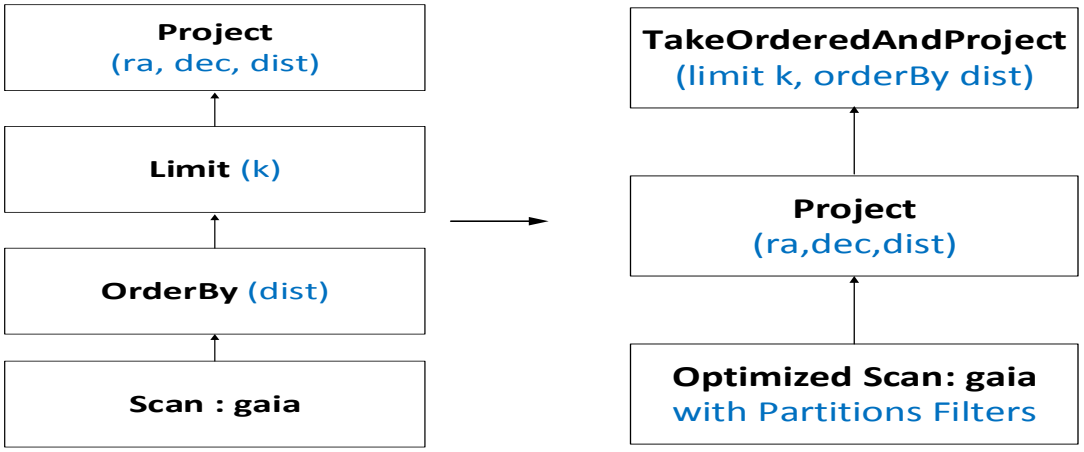


(a) Case 1 (b) Case 2

Figure 5.3 kNN Cases

The physical plan of a kNN query is represented in Figure 5.4b. It starts by applying the partition filter early. By the time the candidates objects are fetched by the optimizer, it projects data on columns `(ra, dec, dist)`. It reads only the columns that the query needs to process and skips the rest of the data. The

column projection is implemented using the Parquet format. Thus, ASTROIDE and Parquet can optimize I/O costs and reduce the amount of data read from storage. Finally, ASTROIDE uses the *takeOrderedAndProject* API to select the top-k records based on their distance to p. This API is equivalent to having a limit operator after a sort operator.



(a) Without Optimization (b) After Optimization

Figure 5.4 Plan Transformation of Query 5.4

To recapitulate, this section shows two QEPs of kNN queries with ASTROIDE. The first example (Figure 5.4a) is the baseline workload, performing a full scan of the entire table, the second example (Figure 5.4b) shows partitions pruning when a filter on the partitioning key is injected.

5.7 Cross Match

In this subsection, we considered an example of a cross-matching that takes two catalogs `gaia` and `igsl`, and returns the set of all pairs (r, s) where $r \in$ `gaia`, $s \in$ `igsl` and the spherical distance between r and s is lower than a radius of 2 arc-seconds.

Query 5.6 CrossMatch (ADQL)

```
SELECT * FROM gaia R JOIN igsl S
ON 1=CONTAINS(POINT('ICRS', R.ra, R.dec),
   CIRCLE('ICRS', S.ra, S.dec, 2/3600));
```

5.7.1 Baseline Approach

Cross-match is one of the most imperative operations in processing astronomical data. In practice, it involves a cartesian product, which leads to a highly expensive query execution. For large catalogs, this execution becomes intractable.

5.7.2 Optimization with Query Rewriting

This query can be programmatically rewritten by our optimizer into a Spark SQL expression as follows:

Query 5.7 Query 5.6 after rewriting

```
SELECT *
FROM gaia R JOIN
    (SELECT *, explode(lpixNeighbors(S.ipix))
    AS ipix_nei
    FROM igsl S) AS SA
  ON (R.ipix=SA.ipix_nei)
WHERE SphericalDistance(R.ra,R.dec,
      SA.ra,SA.dec) < 2/3600
```

Notice that this query uses *explode*, a built-in Spark function that flattens the array containing each HEALPix cell with its neighbors and outputs the elements of the array as separate rows. *lpixNeighbors* refers to a user-defined function that takes as input a HEALPix value and creates an array composed of this value and its neighbors. The original table has been replaced by an extended table with neighbors and the ADQL θ-join predicate has been substituted by an equi-join on HEALPix IDs and a filter predicate on spherical distance. More details about this query transformation are discussed in the next subsection.

5.7.3 Optimization with Transformation Rules

Our solution consists, first, to limit the distance computation to pairs belonging either to the same cell or to neighboring cells, based on HEALPix indices, thus generating matching candidates. Then, a refinement step computes their exact distance and filters the actual matching pairs.

In the first step, to force the matching between the data in the same cell or in neighboring cells, we use a trick: we augment one of the datasets by replicating all objects on the fly. In order to facilitate their matching with objects from different cells of the second dataset, we substitute the HEALPix value of the replicates by those of the neighboring cells. Thus, a simple equi-join query on HEALPix value (by far more efficient than the original query) suffices to generate all the candidate pairs. Furthermore, the fact that the datasets are partitioned according to HEALPix order contributes to improve the performances of this operation. The second step computes the distance and removes the false positives to get the final result.

Algorithm 2 : HX-MATCH(R,S)

Require: Input datasets R and S, search radius ε
Ensure: Result of HX-MATCH Matching pairs of stars that satisfy the predicate *sphericalDistance*$(r,s) < \varepsilon$

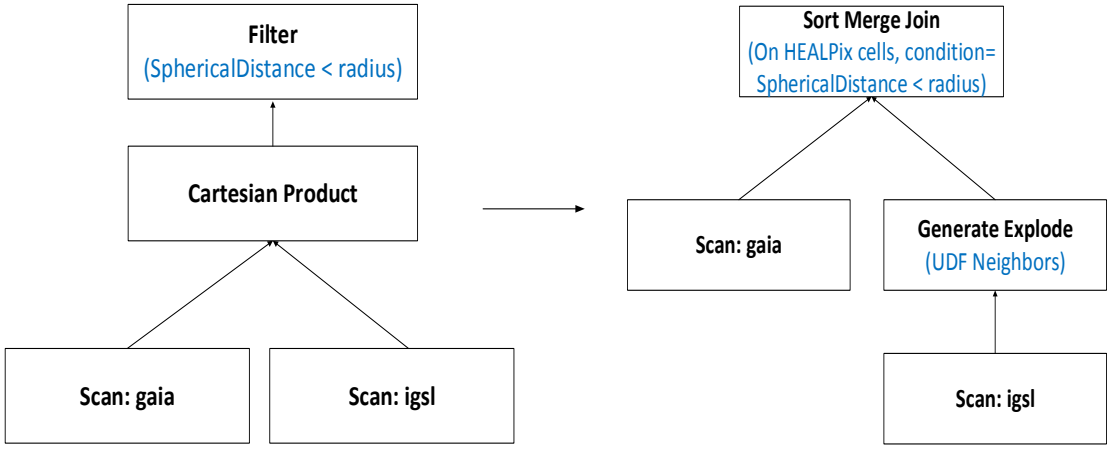
1: $S^+ \leftarrow \emptyset$
2: **for** t $\in S$ **do**
3: **for** N $\in$ *IpixNeighbors*(t.ipix) **do**
▷ IpixNeighbors is a user defined function which returns the current cell and all the neighboring cells.
4: $S^+ = S^+ \cup \{\, t \oplus ipix = n \mid \forall n \in N\}$
▷ Clone each object in S and assign it the HEALPix number of its neighbor cells
5: **end for**
6: **end for**
7: c = Join(R, S^+, $R.ipix = S^+.ipix$)
8: **return** Filter(c, *sphericalDistance*$(c.R, c.S^+) \leq \varepsilon$) ▷ sphericalDistance is a user defined function which computes the harvesine distance between two data points

The pseudo-code of cross-matching (HX-MATCH) [125] is explained in Algorithm 2. It takes as inputs two partitioned files and a search radius ε and returns all matching points that satisfy a distance criteria. The algorithm runs as follows:

1. Since the reference dataset need to be duplicated, we chose the smallest dataset as a reference, let say S. We augment S (into S^+), by creating for each object as many replicates as the number of neighboring cells (Line 4), where we substitute the HEALPix value (here ipix) by the current cell and the one of each neighbor cell (Line 4). Here, *IpixNeighbors* is a user defined function which returns the current cell and all the neighboring cells.

2. We apply an equi-join query between R and this augmented dataset S^+ to get the candidate pairs (Line 7). At last, the refinement step checks the exact objects' harvesine distance and returns the cross-match results (Line 8).

To execute the cross-matching query, ASTROIDE integrates strategies to limit pairwise computations. It starts by scanning the two input files. Then, it applies the *Generate* logical operator which is internally defined in catalyst to calculate the list N of neighboring cells for each row in S and generate a new row for each element in N. This operator duplicates all objects of the reference catalog in the neighboring cells, runs a sort merge join algorithm on HEALPix indices, and finally, filters the output result according to the spherical distance predicate (see physical plan in figure 5.5b).



(a) Without Optimization (b) After Optimization

Figure 5.5 Plan Transformation of Query 5.6

5.8 kNN Join

The objective of this query is to associate each star in a dataset R with its k nearest neighbors stars from a dataset S:

Query 5.8 kNN Join (ADQL)

```
SELECT R.id, S.id FROM gaia R, igsl S
WHERE S.id IN (SELECT Top 10 Sl.id FROM igsl Sl
```

```
ORDER by DISTANCE(POINT('ICRS', R.ra, R.dec),
POINT('ICRS', SI.ra, SI.dec)) );
```

Efficient process of kNN join queries is challenging since it involves both the join and the kNN query. It's worth noticing that the query expression generated by the ADQL parser is not supported by Spark SQL (Spark V2.2), because it involves a correlated subquery that uses attributes from the outer query (R.ra, R.dec) in a UDF, which is not allowed in the current version of Spark SQL. However, ASTROIDE provides a way to cope with this limitation by injecting resolution rules.

In this work, we propose a novel kNN join algorithm tailored for astronomical queries. The most trivial approach would be to execute a kNN query for each element $r_i \in R$. However, this is not efficient because it ignores the shared neighbors in S among close objects of R and entails multiple scans of the partitions. A possible solution could be to minimize the search space by defining a pruning area where all pairs (r,s) can be considered as candidates. However, due to data skewness, predicting a distance threshold is difficult in real astronomical applications, and a uniform distance is inappropriate for effective pruning. Hence, we need a specific approach to support such complex query.

Algorithm 3 : kNN-Join(R,S,k)

Input: R, S datasets, and k
Output: L: result of kNN-Join

1: $R^+ \leftarrow \emptyset$; L={}
2: **for** each t $\in R$ **do**
3: $Res = \{\}; nb = 0$
4: $PrevNeighbors = \{t.ipix\}$
5: **while** $nb < k$ **do**
6: $V = IpixNeighbor(PrevNeighbors) - Res$
7: $nb = nb + count(S, ipix)$, $\forall ipix \in V$
8: $PrevNeighbors = V$
9: $Res = Res \cup V$
10: **end while**
11: $R^+ = R^+ \cup \{ t \oplus ipix = n \mid \forall n \in Res\}$
12: **end for**
13: c = Join(R^+, S, $R^+.ipix = S.ipix$)
14: XM=OrderBy(c, c.R.id,c.sphericalDistance (c.R, c.S))
15: **return** = $L \cup \{g1, .., g_k | \forall g_1, .., g_n \in Group(XM, XM.R.id)\}$

Algorithm 3 outlines our kNN join approach. The first phase is a preprocessing step to calculate the neighboring cells of each ipix (i.e., HEALPix value) belonging to R allowing to reach k objects in S. For each cell $ipix_R$ in R, we examine whether it contains k objects in S. If not, this means that we need to look in the neighboring cells. Lines 3-10 allow to retrieve the list of neighboring cells for each cell in R in order to filter the kNN candidates. To this end, we maintain a histogram of S containing the number of objects for each cell. For illustration, Figure 5.6 represents a 3NN join where R is defined by 3 objects ($P1$,$P2$,$P3$). In the preprocessing step, we look for these 3NN candidates in the 8 direct neighboring cells for each object in R. For $P2$ and $P3$, the first level of neighborhood (colored in yellow) is sufficient to get the 3 closest neighboring points. However, for $P1$, we need to check the second level of neighborhood (colored in red). A similar procedure is repeated until we find the k points.

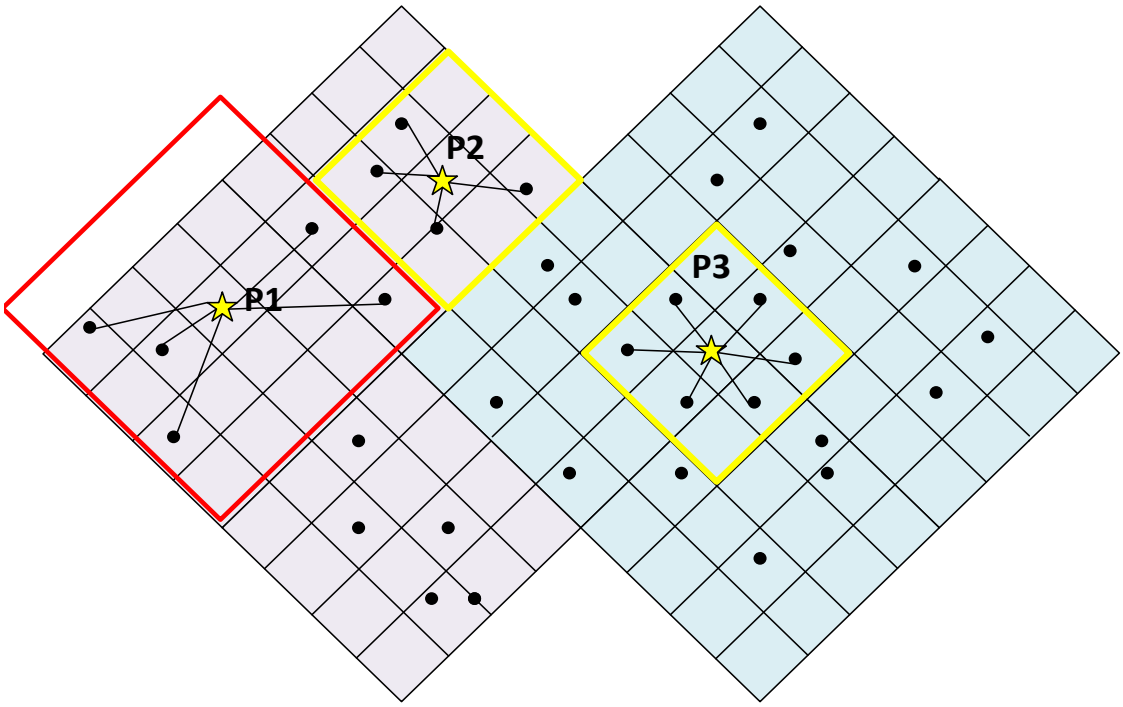


Figure 5.6 3NN Join Query.

In the second step, we associate each object in R with the calculated neighboring cells and clone objects of R in these cells (Line 11) (similarly to our process in cross-matching algorithm). A traditional equi-join is then performed between the extended objects of R and S (Line 13). The result is table c containing pairs of objects as possible candidates. To take the top-k records in S, we partition c

into independent groups of source identifier. Within each partition, the rows are ordered by the spherical distance between objects of R and S (Line 14). Finally, a *rank* function is computed with respect to this order to take no more than k objects.

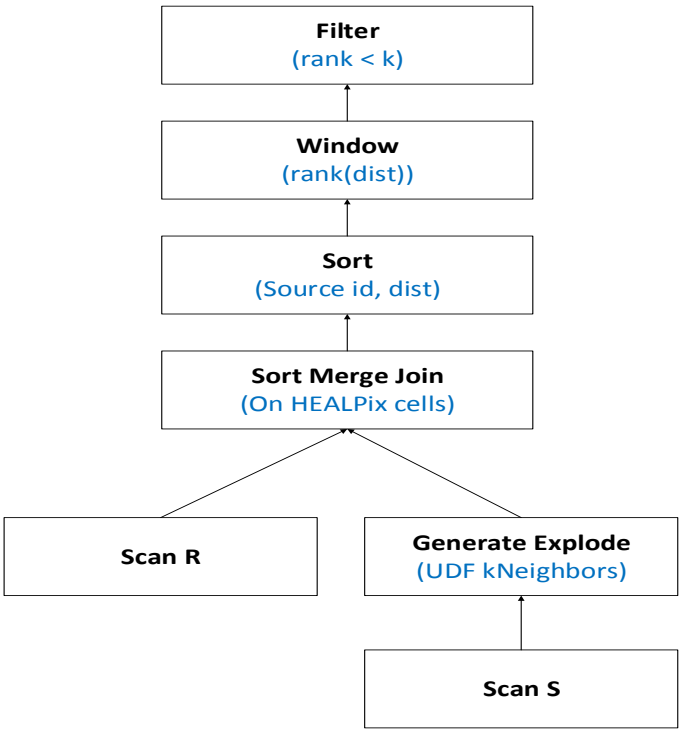


Figure 5.7 Plan Transformation of Query 5.11

The idea is to generate a completely new physical plan (Figure 5.7) for the kNN join query with the following operations:

- Scan both datasets R and S.
- Extend each cell of R with a list of neighboring cells. This is done by maintaining a histogram of S that is broadcast to every node of the cluster. The UDF *kNeighbors* uses this histogram to calculate the cells allowing to reach k objects in S. Then, the operator *Generate Explode* replicates each object of R in these cells.
- Execute a sort merge join on HEALPix indices.
- Add a *window* operator which is based on two base concepts: partitioning and ordering. The first concept logically partitions tuples into groups of

$sourceid$ and ordering defines how the tuples of the table (after the join) are ordered by distance during *window* function evaluation.

- Add a filter operator to take no more than k objects in S (using a *rank* function).

Even though Catalyst can be simply extended, astronomical rules implementation is complex. This complexity arises from the fact that we need to find and replace subtrees with specific logical operators. We have chosen this solution to allow optimization for both relational and astronomical operations indistinctly. Catalyst identifies which part of the tree can be optimized using ASTROIDE's rules and automatically skips over subtrees that do not match.

5.9 Combination with other Attributes

In this section, the objective is to show that ASTROIDE benefits from Spark SQL while allowing the combination of astronomical queries with other attributes, and their integration in the query optimizer. We describe the behavior of our framework in the management of several scenarios. We consider three scenarios:

- *Scenario 1.* kNN with a filter on a certain magnitude range.

Query 5.9 kNN with Filter (ADQL)

```
SELECT TOP 10 ra,dec, DISTANCE(POINT('ICRS', ra, dec),
        POINT('ICRS', 44.97, 0.09)) AS dist
FROM gaia
WHERE magnitude >= 18
ORDER BY dist;
```

- *Scenario 2.* Cross-matching with a filter on a certain magnitude range.

Query 5.10 CrossMatch with Filter (ADQL)

```
SELECT * FROM gaia R JOIN igsl S
ON 1=CONTAINS(POINT('ICRS', R.ra, R.dec),
   CIRCLE('ICRS', S.ra, S.dec, 2/3600));
WHERE R.magnitude >= 18;
```

- *Scenario 3.* kNN Join with a filter on a certain magnitude range.

Query 5.11 kNN Join with Filter (ADQL)

```
SELECT R.id, S.id FROM gaia R, igsl S
WHERE S.id IN (SELECT Top 10 SI.id FROM igsl SI
              ORDER by DISTANCE(POINT('ICRS', R.ra, R.dec),
              POINT('ICRS', SI.ra, SI.dec)) )
AND R.magnitude >= 18;
```

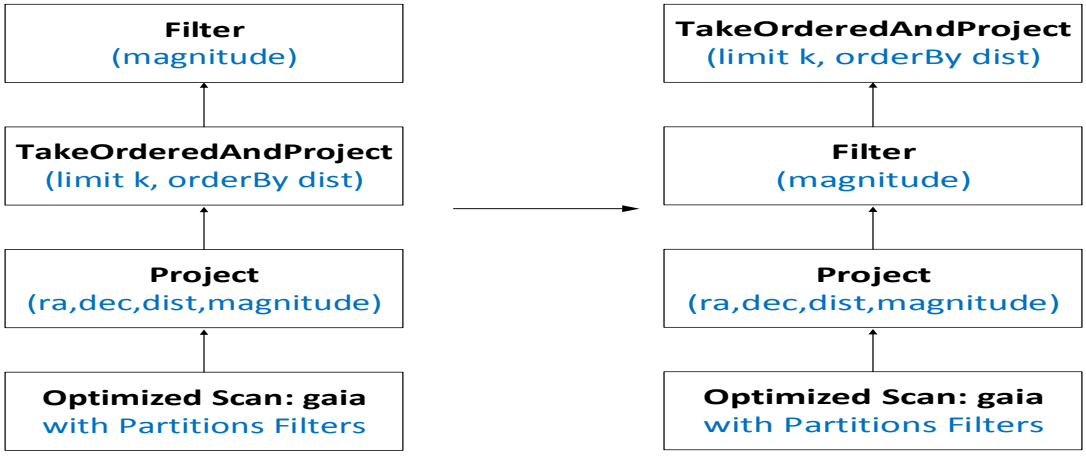
5.9.1 Scenario 1

ASTROIDE takes advantage of Catalyst optimizations to execute these scenarios. It uses a general set of guidelines to choose the best method for accessing data in each table.

As showed in Figure 5.8a, kNN queries involve the use of a relatively expensive operation in the plan, particularly CPU-consuming with sorting (*orderBy*). Combining sorting and filtering predicates in ASTROIDE causes a reorganization in the plan to reorder operations in order to reduce the amount of data required for sort. In other terms, the filtering operator is applied early in the plan to reduce the size of the input size before executing the sorting operation (Figure 5.8b). The predicate that filters out rows on magnitude is applied as soon as possible. The goal is to choose the best plan from all the plans examined and to minimize CPU, I/O resources and communication costs as much as possible.

5.9.2 Scenario 2 & 3

In the case of the combined cross-matching and filtering query, the expected execution (left side in Figure 5.9) calculates the matching result between the two tables and then computes a filter on magnitude. In the alternative scheme of Figure 5.9, the conditional statement is evaluated before the join occurs. We first eliminate sources that do not satisfy the predicate on magnitude range before joining the input tables. The predicate filter is pushed below the join as it reduces the input size of the join. It is important to reduce record numbers from large tables as much as possible before performing the *SortMergeJoin* operation.



(a) Before Optimization (b) After Optimization

Figure 5.8 Plan Transformation of Query 5.9

Because data is filtered out in early stages of query processing, subsequent joins will be faster. This transformation improves significantly the query execution time because joins are applied on smaller tables. The same idea is applied in Scenario 3, that is the kNN-join is applied after filtering the input.

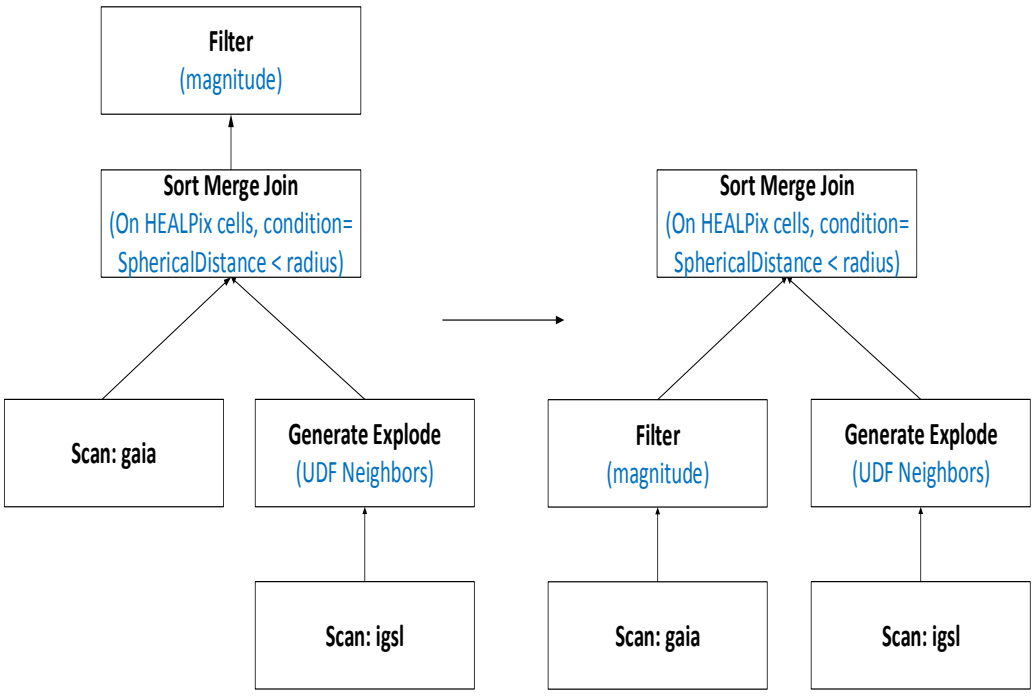


Figure 5.9 Plan Transformation of Query 5.10

5.10 Summary

As discussed in Chapter 2, existing big data technologies such as Spark are unsuitable for astronomical applications. In this chapter, we presented our query processing module and highlighted its differences with the query processing module in conventional distributed systems. Our main contribution is the implementation of an astronomical query optimizer as an extension of an existing query optimizer (Catalyst). We have shown how standard query processing and optimization rules can be adapted to astronomical query processing. We introduced an ADQL parser that accommodates astronomical operators. Our query optimizer uses HEALPix indexing structure and partitioning to facilitate the query retrieval. Future research will include building a cost model for analyzing the suggested query execution plans.

CHAPTER 6

Experimental Study and Graphical Interface

Contents

Data is the new science. Big Data holds the answers.

Pat Gelsinger

6.1 Introduction

In the state-of-the-art chapter, we studied existing DBMSs technologies that support astronomical data management. We compared their main features and highlighted their limits in big data processing. The overwhelming flow of astronomical data has made DBMSs technologies no longer adequate for extremely large scale astronomical data. As a result, researchers worldwide have started to take advantages of big data technologies to cope with the challenges related to data processing problems on massive datasets.

To the best of our knowledge, there is no system based on big data technologies that can process large astronomical data. For this reason, we were interested in a similar context that is the geo-spatial domain in order to find a reference for performance evaluation. SIMBA [11] is the most complete and recent work in the geo-spatial context. It offers better performances compared to all existing spatial systems. It also presents a query optimizer adapted to the geo-spatial context.

Thus, in this chapter, we compare the effectiveness and the efficiency of ASTROIDE with the closest prototypes in the state-of-the-art, i.e. SIMBA. We also compare ASTROIDE with Spark SQL as a baseline in order to illustrate the gain in performance provided by the proposed optimizations. This chapter provides an experimental study of ASTROIDE. The tests are based on three real world datasets, with a set of astronomical queries presented in Chapter 5. We also present and analyze the results of our comparisons in two different infrastructures: real system cluster and cloud cluster to evaluate the ability of ASTROIDE to perform complex and large scale data processing using a cloud infrastructure. We describe also our GUI that enable easily visualization and manipulation of astronomical data.

6.2 Experimental Setup

6.2.1 Local Cluster Description

Experiments were performed over a distributed system composed of 6 nodes having the following characteristics:

- 4 workers, each worker with:
  - CPU. 8 cores
  - RAM. 30 GB
- Two other nodes, each node with:
  - CPU. 24 cores
  - RAM. 46 GB

In total, 80 cores are used with one Gigabit network for node communication. The main memory reserved for Spark is 213.7 GB in total. We used Spark V2.2.1 with a spark driver memory of 10 GB and kept other configurations parameters as their default values. Input datasets are uploaded in HDFS with a replication factor of 3 on each DataNode.

6.2.2 Cloud Cluster Description

In Section 6.4, we compare the execution of ASTROIDE on a conventional (local) cluster and an OpenStack platform. For this, we created two clusters with the same configurations, one as an OpenStack cluster using the Galactica platform [126] and other as a conventional cluster. Each cluster is composed of 6 nodes, Figure 6.1 shows the common master/slave architecture of the deployed clusters.

We tried to have similar configurations in both clusters in terms of number of nodes, RAM and number of cores:

Table 6.1 Configuration Details

Property	Conventional cluster & Cloud Cluster
Number of nodes	6
Node characteristics	28 GB RAM | 6 cores
Spark version	2.2.1
HDFS replication factor	3
Driver memory	10 GB

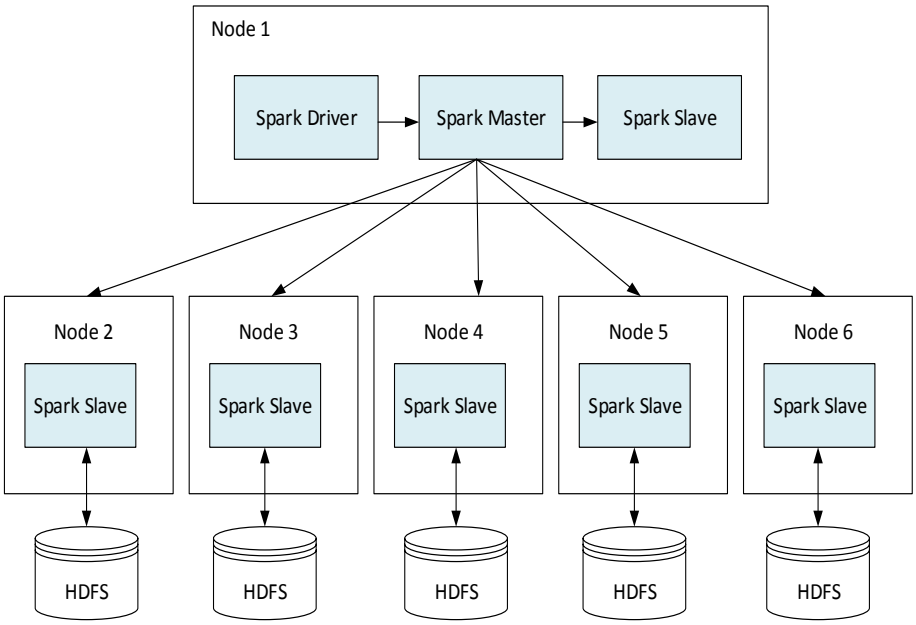


Figure 6.1 Cluster Architectures

6.2.3 Datasets Description

We used three datasets in our experiments: Each record contains a `sourceId`, a two-dimensional coordinate `(ra` and `dec)` representing the star position on the sky and other attributes including `magnitude`, `metalicity`.

- GAIA. The public GAIA DR1 dataset [3] [127] describes the positions of more one billion sources and represents the most detailed all-sky map in the optical to date.
- IGSL. Short of the Initial GAIA Source List [128], it is a compilation catalog produced for the GAIA mission.
- Tycho-2. An astrometric reference catalog related to prior surveys containing positions and proper motions of more than two million brightest stars in the sky.

The characteristics of these datasets are resumed in the following table:

In all experiments, since we focus on the query processing cost, and not the result materialization cost, we use *COUNT(*)* to return the total number of rows in each output DataFrame. The objective of our experiments is to provide an

Dataset	# of Objects	File Size	# attributes
GAIA DR1	1,142,461,316	498.5 G	57
IGSL	1,222,598,530	323.2 G	43
Tycho-2	2,539,893	501.4 M	32

Table 6.2 Main Characteristics of the Datasets.

experimental study of ASTROIDE using the query execution time for partitioning, cone search, kNN, cross-matching and kNN Join queries. By default, the partition size is 256 MB, the HEALPix order is 12. A cone search query is characterized by the radius of its cone, which is set to 2 arc-second. The default cross-match distance threshold is 2 arc-second. For kNN queries, k is set to 10. For kNN join queries, k is equal to 5. The default data size is 225 million records.

6.3 Result Analysis using a Local Cluster

6.3.1 Partitioning

Figure 6.2 measures the partitioning time in ASTROIDE using both hash partitioner and range partitioner on GAIA DR1 dataset. The difference between the two partitioners is explained in Section 3.4. The time for reading data from HDFS and writing data to HDFS is included in the performance measurement. We also included the time for deriving the partitions boundaries. The two partitioners show a linear growth when the data size increases, the hash partitioner is slightly faster as it partitions data quasi-randomly without keeping data locality. However, the range partitioner is more efficient in querying data. It divides the dataset into approximately equal-sized partitions, each of which contains records with HEALPix indices within a specified range. For these reasons, we choose to use the range partitioner in our experiments.

We have also investigated the impact of the HEALPix resolution parameter on the partitioning, given that $N_{side} = 2^{order}$ and the maximum value of $order$ specified by the HEALPix library is 29. We have found that the costs are equivalents with the increase in the HEALPix order, the reason is that the computation costs associated to HEALPix indices is kept constant while varying the resolution. The partitioning time is about 20mn for a file of 225 million of records. Thus, the performance of our partitioning algorithm does not depend on the HEALPix granularity.

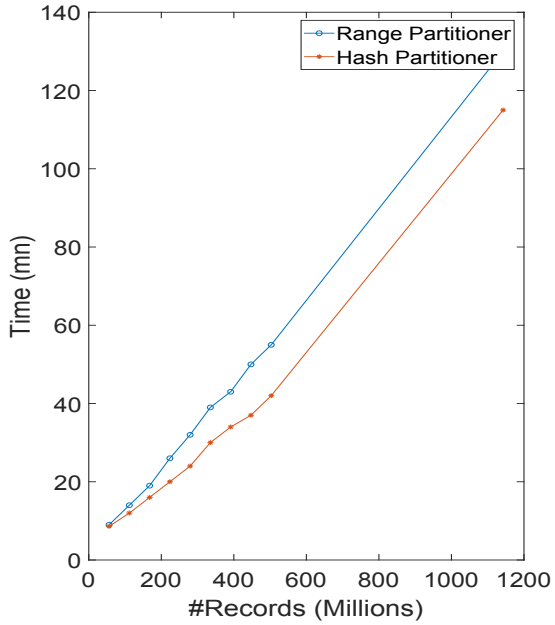


Figure 6.2 Effect of Data Size on Partitioning (GAIA DR1).

However, a finer resolution entails more CPU cycles. For following experiments, we fixed the HEALPix resolution to 12, experiments have demonstrated that this value makes query execution more efficient without losing cells in output result (see Section 6.3.3 for more details).

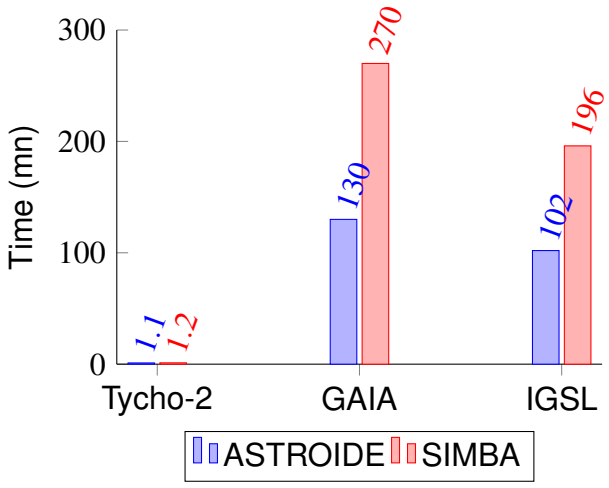


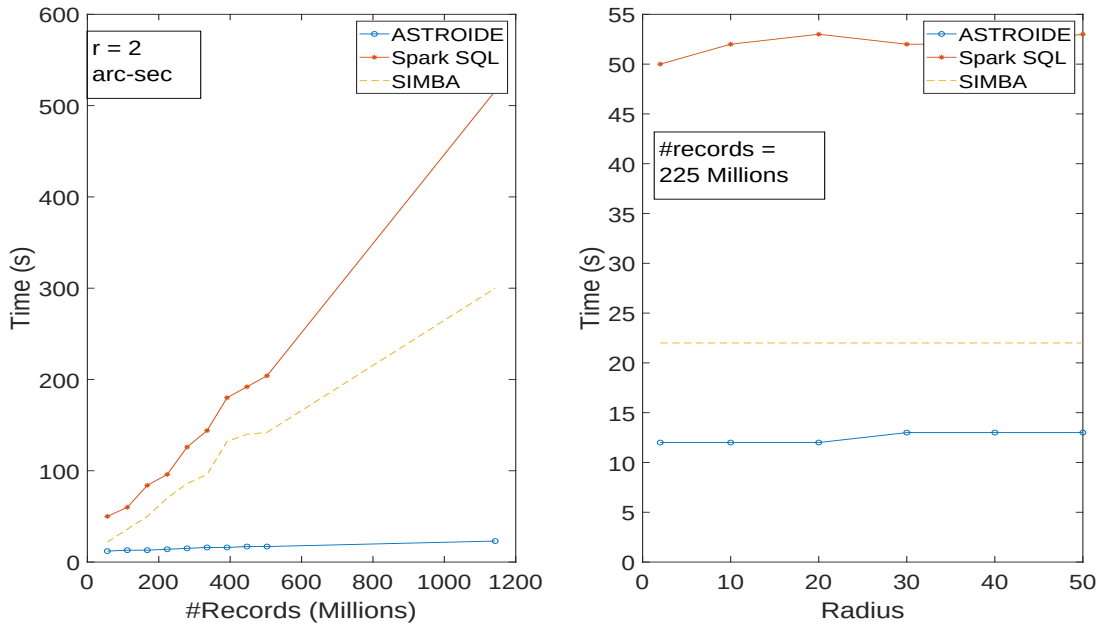
Figure 6.3 Partitioning with Different Datasets

We have also compared the performance of partitioning across three datasets (Tycho-2, GAIA, IGSL) using ASTROIDE and SIMBA. Figure 6.3 shows the efficiency of our partitioning algorithm. Partitioning of massive astronomical data is

a time-consuming process that can take hours as discussed in Section 4.3.3. We developed a HEALPix based partitioning algorithm to improve the performance of astronomical queries. Our approach is based on data indexing to create a spatial value that keeps data locality and then sorting data on these values. We have shown that such approach is efficient for astronomical data partitioning. We can see that the execution time is reduced roughly by half using ASTROIDE compared to SIMBA. Later, we will demonstrate the importance of our partitioning algorithm for scalable query processing.

Note that the partitions construction is a one shot process, since we chose to store the partitioned files in HDFS and use them for future queries.

6.3.2 Cone Search Query



(a) Effect of Data Size on Cone Search. (b) Effect of Radius on Cone Search.

Figure 6.4 Cone Search Performance

Figure 6.4a presents how the data size affects the performance of cone search queries. The horizontal axis represents the number of objects in the GAIA dataset (DR1), and the vertical axis represents the query execution time.

The query execution time increases when the data size increases, this is due to the processing of more partitions when increasing the data size. ASTROIDE

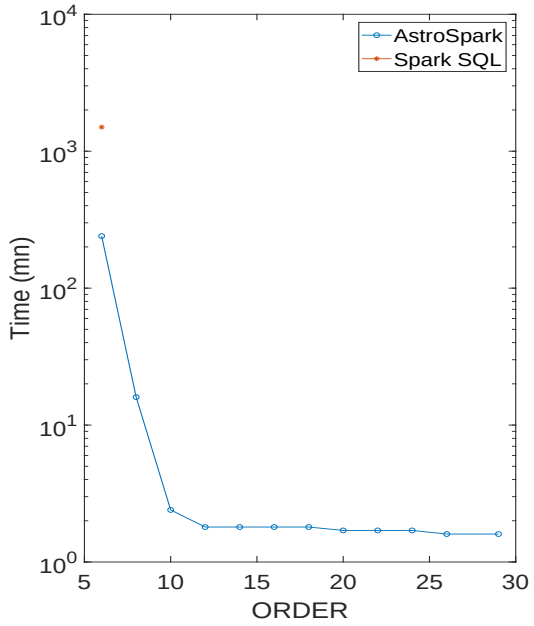
and SIMBA exhibit good scalability because they need to scan only few partitions to execute the query. Indeed, ASTROIDE requires less access to partitions than SIMBA but Spark SQL has to scan all objects in the dataset. It is noticeable that the query runtime in ASTROIDE increases linearly as the data size increases.

Figure 6.4b studies the impact of query radius on execution time, we increased the radius from 2 arc-seconds to 50 arc-seconds. The performance of ASTROIDE, SIMBA and Spark SQL remains constant. ASTROIDE is 2x faster than SIMBA and 5x faster than Spark SQL. This is due to partition pruning in ASTROIDE. If a reasonably large radius is intersecting only one partition, we do not need to scan more partitions. However, enlarging much more the query radius can deprive ASTROIDE from optimization opportunities, reduce the power of partition pruning and lead to performance deterioration.

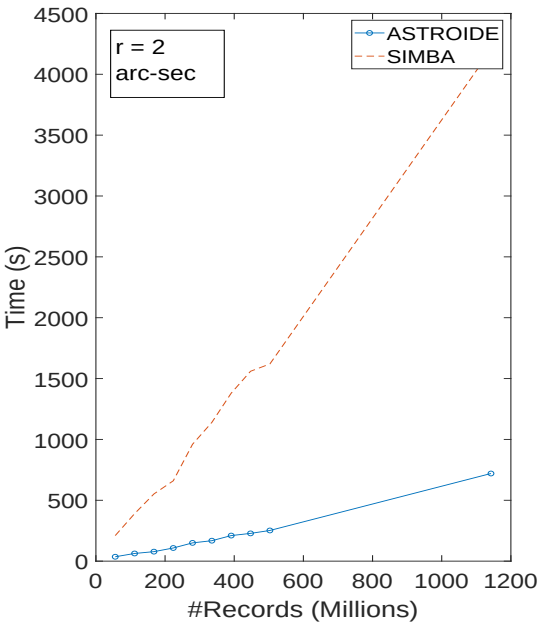
6.3.3 Cross-Matching Query

The default search radius is set to 2 arc-seconds, a value that is significant for an astronomer.

We started by comparing the performance of the cross-matching using partitioned files with different HEALPix resolution. For this test, input files are indexed using HEALPix and organized using range partitioning. Then, we run the cross-match query on these files. Figure 6.5a shows how the cross-matching time is influenced by the HEALPix resolution. As the HEALPix order increases, it is interesting to observe that the performance of cross-matching becomes almost constant. Besides, for values greater than 16, some records are lost in the output result, the reason is that the search circle defined by ε becomes bigger than the HEALPix cell area. Indeed, for $order$ less or equal to 15, the HEALPix angular resolution (defined in Formula 4.4) is lower than 6,44 arc-second and for $order$ equal to 16, the HEALPix angular resolution is 3,22 arc-second. However, the square root of the area of the search radius is 3,544 arc-second. In a previous work, we explained the cross-matching algorithm [125] using the HEALPix order value 8. In this dissertation, we improved the execution time of our algorithm using a different HEALPix resolution based on new experiments. The HEALPix order value 12 is the best choice according to several measurements, it ensures the fastest query as well as the correctness of the result.



(a) Effect of HEALPix Resolution on HX-Match.



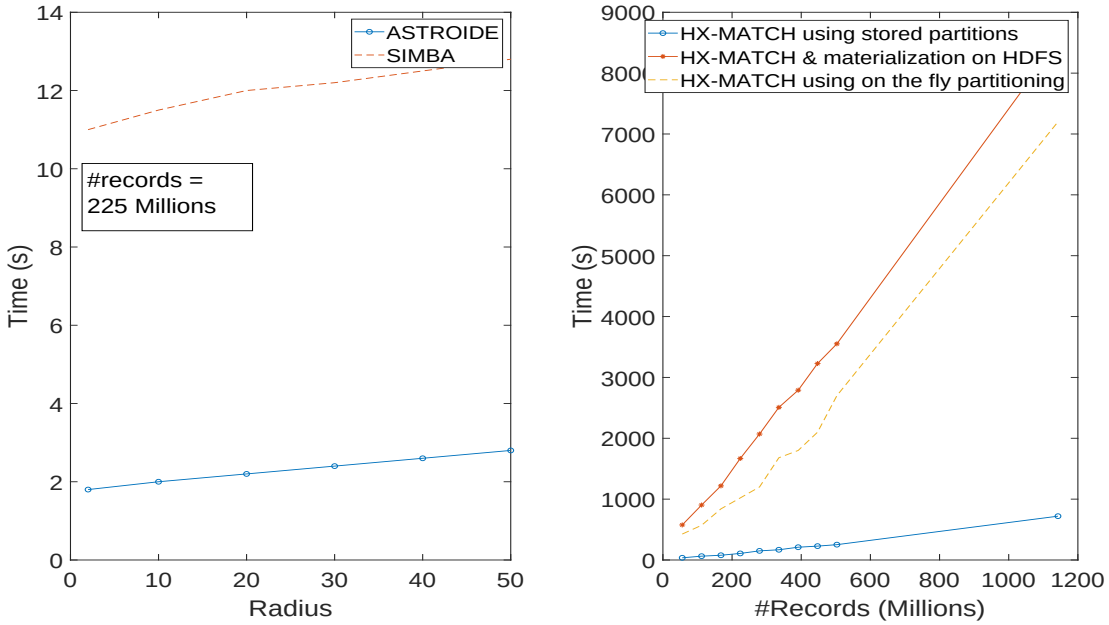
(b) Effect of Data Size on Cross-Match.

Figure 6.5 Cross-Match Performance

Next, we compared the performance of ASTROIDE, SIMBA and Spark SQL for executing cross-matching queries. As shown in Figure 6.5b, ASTROIDE is scalable and efficient. The performance shows a linear trend. Our approach shows the best performance compared to SIMBA because it requires less access to partitions and fewer objects along the borders. Since ASTROIDE is using the HEALPix library for its indexing module, a sky partitioning technique adapted to astronomical data, experiments show that ASTROIDE outperforms SIMBA for astronomical queries. Furthermore, SIMBA only implements the Euclidean distance, which leads to erroneous result when cross-matching. For instance, the difference in terms of number of outputs for a file of 55 million is 3367 objects. Spark SQL is worse, because it performs a cartesian product. As an example, the execution time of a cross-match between 200,000 records of GAIA and Tycho-2 takes 13,6 hours.

We have also studied the performance of the cross-matching algorithm as the search radius increases. Figure 6.6a shows that ASTROIDE is 6x faster than SIMBA and the performance gap remains constant with bigger radius.

We have also validated our choice of materializing partitioned files on HDFS.



(a) Effect of Radius on Cross-match. (b) Effect of Partitioning on HX-MATCH.

Figure 6.6 More Cross-Match Performance

We show three different costs.

- Option 1: Cost of cross-matching and materializing partitioned files on HDFS.
- Option 2: Cost of partitioning and cross-matching using on the fly partitioning (without materializing on HDFS).
- Option 3: Cost of cross-matching when reading already partitioned files from HDFS.

Figure 6.6b represents the performance of the cross-matching algorithm with different approaches. Option 2 is slightly faster than the option 1 because of the write overhead of the partition. However, the former solution obliges to repartition the data at each query execution. In contrast, the option 3 that reuses an existing partition (in blue in Figure 6.6b) is extremely fast. This shows how the cost of partitioning is immediately amortized by the subsequent queries.

Next, we continue to investigate the cost of our HX-MATCH algorithm by pre-computing the cross-matching with a relatively large radius lr. We thus build an intermediary DataFrame M which contains the IDs of the matched records from R

and S according to lr along with their distance and store it in HDFS. The dataset schema is represented by the following attributes $(sourceID_1, sourceID_2, dist)$ where $sourceID_1$ and $sourceID_2$ correspond to source identifier in $R$and S respectively and $dist$ is the spherical distance between the actual objects identified by $sourceID_1$ and $sourceID_2$. Thus, the subsequent cross-matching queries can be replaced by a simple selection on M and equi-join queries, which is much more efficient than the initial computation of a distance matching. Precisely, all needed is to filter M on $dist \leq \varepsilon$ where ε is the requested radius in the cross-matching query, and to retrieve other attributes from R and/or S to make a join with R and/or S on respective $sourceIDs$. This can be expressed as follows:

$$R \bowtie_{R.sourceId_1=M.sourceId_1} (\sigma_{\text{dist}<\varepsilon}(M)) \bowtie_{M.sourceId_2=S.sourceId_2} (S)$$

To study the impact of this pre-computing technique on our algorithm, we matched the two biggest catalogs GAIA DR1 and IGSL with a radius of 7 arcseconds. We first used HX-MATCH to build the intermediary dataset M. This tooks 3 hours, including the materialization of M on HDFS. Although this seems high compared to a one shot cross-matching with a smaller radius, it will be amortized when it comes to repeated queries with various distance criteria, as long as this distance is lower than the one used to build M.

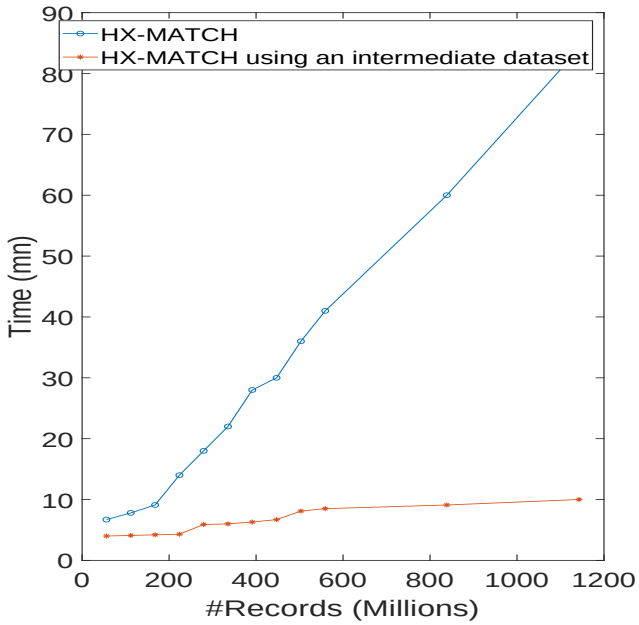


Figure 6.7 Effect of Use of Intermediate Dataset (GAIA DR1/IGSL).

Figure 6.7 shows the overall speedups of this optimization. The comparison between the two options shows that using the intermediary dataset, the cross-matching computation achieves 4x speedups. However, this optimization incurs a storage overhead, and is limited with the chosen maximum radius.

The average number of matching pairs produced after cross-Matching is reported in Table 6.3:

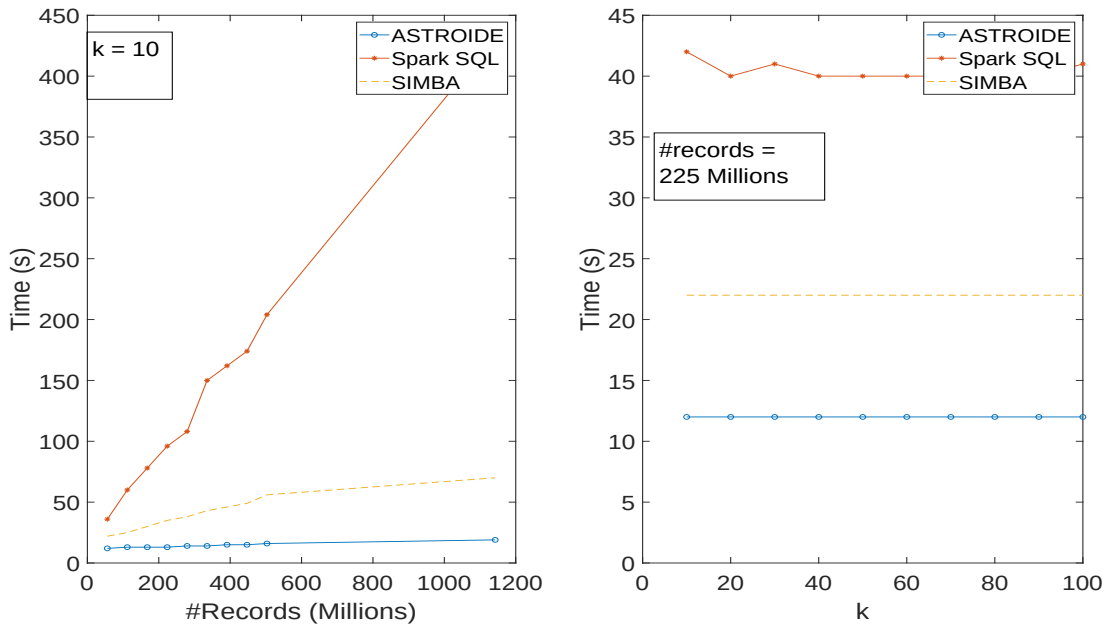
Dataset1	Dataset2	# of matches	radius
GAIA DR1	Tycho-2	2,421,938	2"
GAIA DR1	IGSL	2,661,125	2"

Table 6.3 Average Number of Matching Pairs.

6.3.4 kNN Search Query

Figure 6.8a shows the performance of the kNN query on GAIA datasets in ASTROIDE, Spark SQL and SIMBA. We select a random point from the input dataset, fix k to 10 and measure the execution time of the query. In Figure 6.8a, we study the effect of increasing the data size from 50 millions to 1.2 billions. ASTROIDE outperforms Spark SQL since Spark SQL requires scanning the whole dataset to execute a kNN query. ASTROIDE achieves better performance than SIMBA, because it scans fewer partitions. ASTROIDE brings more optimization opportunities by pruning more partitions. In general, one or two partitions are sufficient to cover the kNN result. In addition, we study the effect of increasing k, we varied k from 10 to 100 and fixed the data size to about 50 million of objects. Figure 6.8b shows that the performance of ASTROIDE, SIMBA and Spark SQL are not affected by k. Spark SQL scans all the objects regardless of k values. For ASTROIDE and SIMBA, the change of k does not affect the number of partitions read, they maintain a constant speed. Besides, ASTROIDE is 2x faster than SIMBA.

ASTROIDE is able to effectively reduce the number of partitions scanned by partition pruning. This is very efficient for relatively small value of k (when the number of records involved in the query is small). However, for larger values of k, ASTROIDE has to scan more partitions, so that there are less optimizaton opportunities for our optimizer. Thus, the performance gain from using partition pruning is not as significant as represented in Figure 6.8b.



(a) Effect of Data Size on kNN queries.

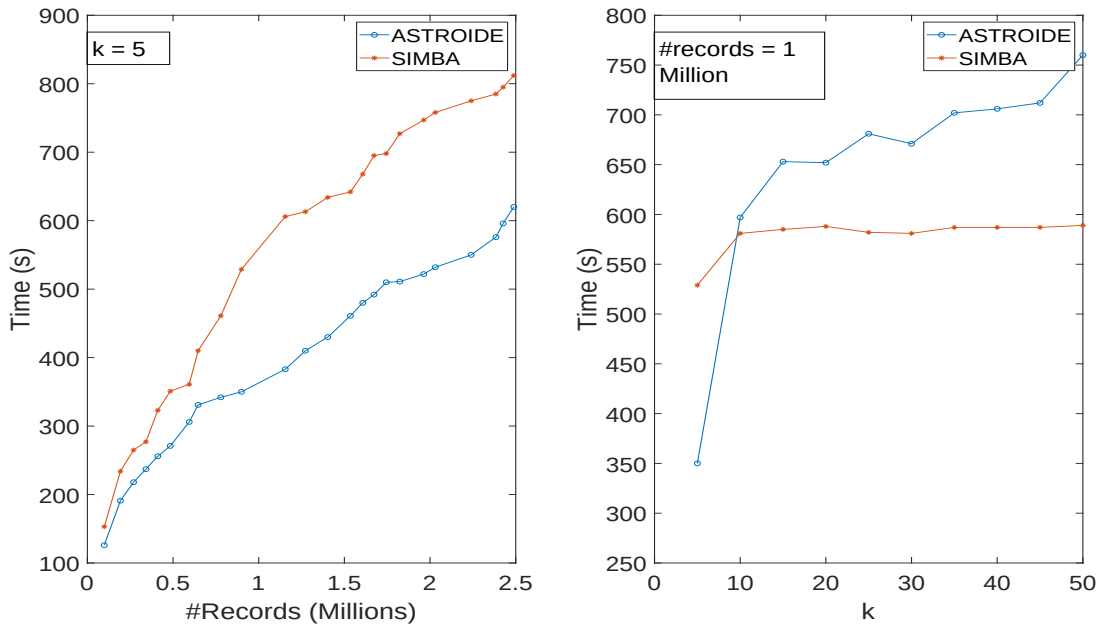
(b) Effect of k on kNN queries.

Figure 6.8 kNN Performance

6.3.5 kNN Join Query

The kNN join query is impossible to execute with Spark SQL. That's why we choose to compare the performance of kNN join in ASTROIDE with SIMBA. We tested the performance of this operator when increasing the number of objects of IGSL catalog while having the number of object fixed to 1 million for GAIA dataset. The results are presented in figure 6.9a. Both ASTROIDE and SIMBA show a linear scale up, but ASTROIDE outperforms SIMBA for k = 5.

The result of varying k on kNN join are illustrated in figure 6.9b. As we can see, when k increases, the performance of SIMBA remains nearly the same. Thus, SIMBA exhibits better scalability when scaled out with larger value of k. This is because SIMBA partitions the dataset and applies a sampling technique to compute a distance bound for each partition. For ASTROIDE, the execution time increases linearly, which indicates that data replication is sensitive to k. With increase of k, there is a continuous increase in the query execution time. This is mainly due to data replication that involves much more records in query processing. It is unavoidable that retrieving neighboring cells of certain cells with larger k values could have significantly larger number of duplication level. However, it



(a) Effect of Data Size on kNN Join.

(b) Effect of k on kNN Join.

Figure 6.9 kNN Join Performance

should be noted that in most astronomical applications, the common value used to execute kNN join is k = 5, for which ASTROIDE provides better performance.

6.4 Cloud Based Implementation and Tests

Addressing big data challenges discussed in Section 1.2 requires a large computational infrastructure to perform complex and compute intensive queries on large sky surveys. Cloud computing is an emerging trend for performing large-scale computing. It eliminates the need to maintain expensive computing hardware, dedicated space, and software [129]. This section focuses on the integration of ASTROIDE with cloud computing.

The objective of this section is to study the effectiveness of ASTROIDE in a cloud computing platform, to compare the trends of performances when moving to a cloud infrastructure and to evaluate the cost of this move. To this end, we deployed ASTROIDE in two clusters: a virtual system using a cloud infrastructure and a real system cluster. We evaluated the ability of ASTROIDE to perform complex and large scale astronomical data processing using OpenStack platform [130]. OpenStack is one of the most widely used open source platform for build-

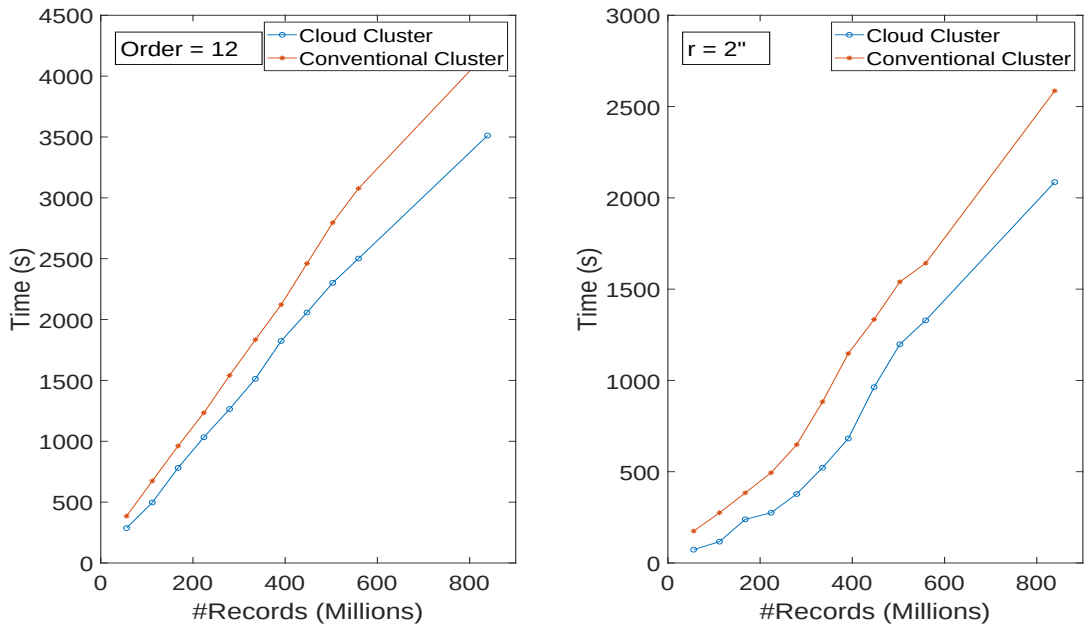
ing and managing cloud computing platforms for public and private clouds. OpenStack is revolutionizing the cloud computing landscape in order to provide a cloud operating system that controls large pools of compute, storage, and networking resources.

In order to evaluate the performance of ASTROIDE in both infrastructures, we have focused on the most complex and used operation in astronomy which is cross-matching. We have also studied the performance of our partitioning algorithm. After the successful creation of the clusters, we run two types of jobs on the clusters:

- The first job partitions samples of the GAIA dataset using our partitioning algorithm. It consists of indexing data using HEALPix, and applying a range partitioning to ensure load balancing while keeping data locality. The execution time includes the materialization of partitioned files, along with the metadata, on HDFS to use them for subsequent queries.
- The second job executes our cross-matching algorithm HX-MATCH between partitioned samples of GAIA DR1 and Tycho2 catalogs. Indeed, HX-MATCH astutely combines HEALPix based indexing and partitioning, to achieve an efficient pruning and to remedy the potential skew of celestial objects. It maps the original predicate into an equi-join on HEALPix indices (which is much faster than a distance join) in order to generate candidate pairs sharing the same cell.

Figure 6.10 shows the outcomes of these experiments using both clusters. When ASTROIDE is deployed on OpenStack, it shows a promising outcome compared to a conventional cluster. The partitioning algorithm maintains a linear trend, similar results can be also observed in Figure 6.10b for cross-matching queries.

Equivalent clusters in terms of number of nodes, RAM and number of cores (see Table 6.1) were configured in two environments to evaluate ASTROIDE performance. Our experiments demonstrate that ASTROIDE on cloud is faster than its implementation on real system cluster. However, this is due to the fact that the cloud infrastructure is more powerful. Table 6.4 shows data that describes the cluster performances in terms of CPU, memory and I/O. Besides, deploying



(a) Data Partitioning on OpenStack.

(b) HX-MATCH on OpenStack.

Figure 6.10 Performance Comparison using ASTROIDE

ASTROIDE on a cloud platform could be a promising alternative that provides different advantages compared to traditional clusters including lower costs, ease of deployment and flexibility.

Table 6.4 Cluster Performances

Property	Conventional cluster	Cloud Cluster
CPU (s)	44,56	60,18
Memory (MB/sec)	2589,62	1928,69
File I/O (Mb/sec)	11,09	27,47

6.5 ASTROIDE GUI

While astronomical big data analysis is a vital task, another important component is the ability after analysis to visually interact and explore data result. In this section, we focus on the Graphical User Interface (GUI) offered by ASTROIDE to enable easily visualization and manipulation of astronomical data. The GUI of ASTROIDE receives two kinds of tasks : data partitioning or astronomical queries. The partitioning task goes through our partitioning module to create partitioned

files in HDFS, while, astronomical queries passes through two modules: querying and visualization.

6.5.1 Querying Module

The querying module supports ADQL queries (see Figure 6.11) and allows their execution on ASTROIDE query processing module. It allows the processing of astronomical queries by exploiting our query optimization module and query algorithms described in Chapter 5. Figure 6.11 is a screen-shot of our graphical interface that enables users to easily express ADQL queries and hides the complexity of the system. The querying module answers user queries using partitioned files in HDFS. It uses ASTROIDE engine to process astronomical queries.

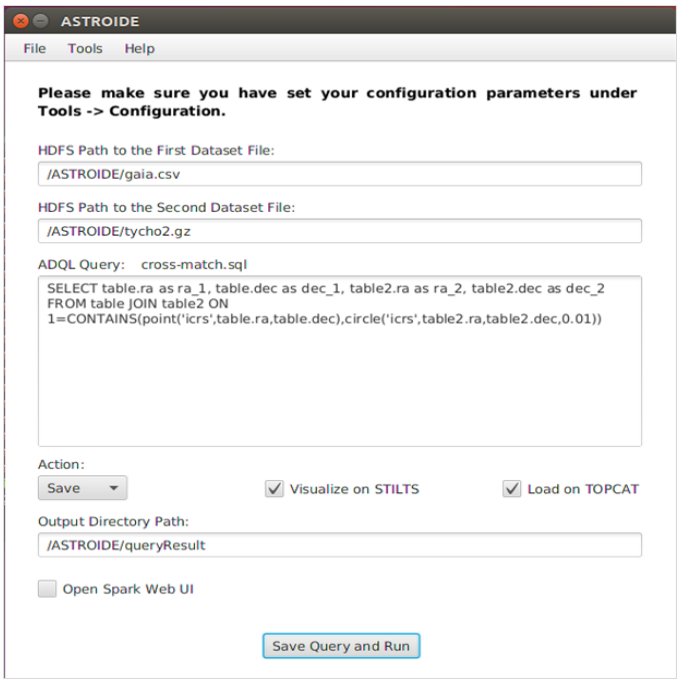


Figure 6.11 ASTROIDE GUI

6.5.2 Visualization Module

Visualization is a desirable feature for data management in general, and for astronomical data in particular. We propose a visualization module that runs on top of our querying module and enables users to visualize data using TOPCAT [131], a Tool for OPerations on Catalogues And Tables [1]. TOPCAT is a GUI application for

[1] http://www.star.bris.ac.uk/ mbt/topcat/

retrieval, analysis, and manipulation of tables. It has been developed to provide a toolkit for astronomers to enable interactive exploration of tables in the context of astronomy. The results returned by our querying system are loaded on TOPCAT in order to overview and explore their contents. In our visualization module, we use STILTS [132] (STIL Tool Set) [2], a scriptable access to most of the features available in TOPCAT. We use this library to represent query results using a visual form that is intuitive and easily comprehensible by the user. Our visualization module supports both sky map visualization and data list visualization.

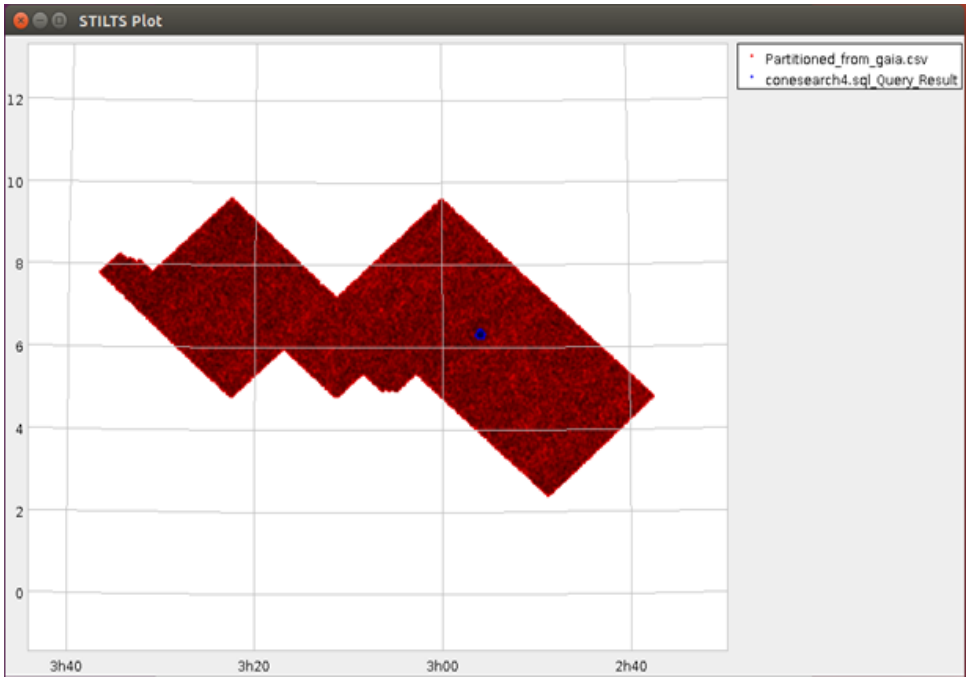


Figure 6.12 Cone Search Visualization

Figure 6.12 allows the visualization of the cone search result, when a user query looks for stars within a circular region of the sky. The map shows in blue the position of stars within the given circular region, while, the red points represent the accompanied partition(s) to which the result belongs. Users can get more detailed information about the location of the stars when the mouse hovers over a sky position.

For cross-matching and kNN join queries, ASTROIDE GUI allows to visualize the matching pairs, in different colors for each dataset, along with a matching line between each pair. For example, a zoom-in visualization of a 2NN join query is represented in Figure 6.13. We notice that, every red point is connected to two

[2]http://www.star.bris.ac.uk/ mbt/stilts/

blue points, its two nearest neighbors from the second dataset, as required by the query. The blue points are not clear due to the enormous number of green lines, but their location is clear, where all the green lines coming from different red points join.

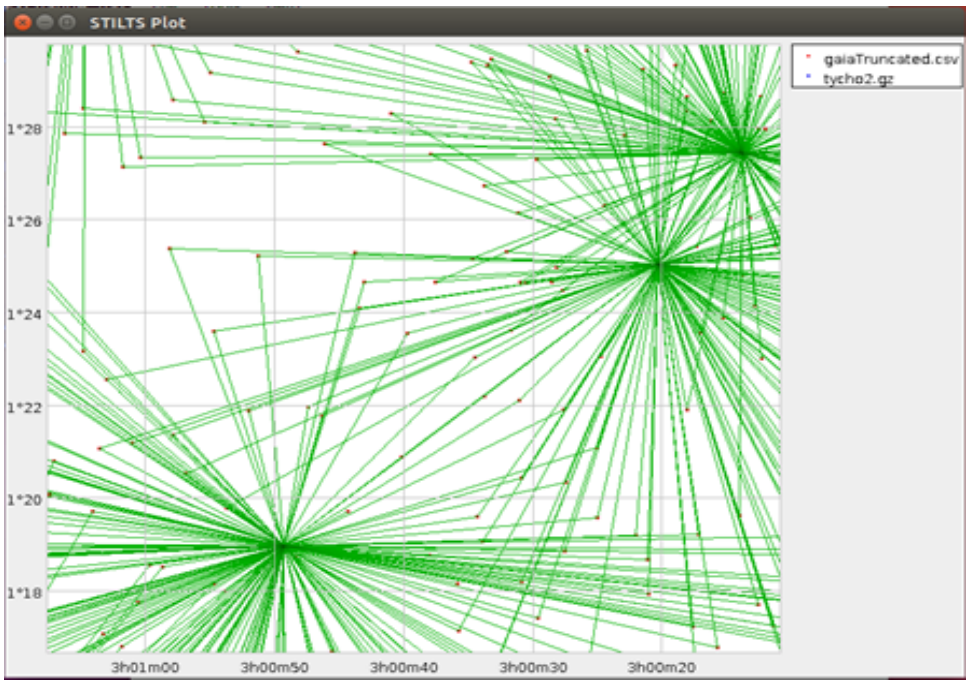


Figure 6.13 2NN Join Visualization

Although the proposed GUI is a simple and efficient tool to express astronomical queries and visualize query results, it suffers from some limitations caused by the limitations of the TOPCAT tool. The huge size of astronomical dataset makes it too difficult to be processed by a desktop application not designed for a distributed context. The actual version represents visually only a sample of output data and does not allow the visualization of the entire output result. The vision is to run ASTROIDE as a web application, where everything happens on the side of the server while using a more powerful tool for data visualization.

6.6 Summary

Experiments on real datasets from the ongoing spatial mission GAIA demonstrate that ASTROIDE is obviously much faster than Spark SQL. It also outperforms SIMBA (which is not specialized in astronomical data) thanks to the use of a HEALPix based partitioning approach and efficient query algorithms.

ASTROIDE extends Spark to support main astronomical queries. It achieves scalability through astronomical indexing and partitioning, dynamic query rewriting and efficient optimizer based on customized rules and strategies.

CHAPTER 7

Conclusions and Perspectives

In this chapter we review the work presented in this thesis with an emphasis on achieved contributions, the lessons learned and the future works. In particular, Section 7.1 summarizes the main research achievements, and Section 7.2 presents the future directions of our research.

7.1 Summary of our Contributions

At present, the continuous progress in telescopes, detectors, and computer technologies has given birth to large sky surveys with information measured in Perabytes and billions of detected sources. This data avalanche brings astronomy into the big data era. However, the traditional astronomical data analysis methodologies using DBMSs technologies are inadequate to cope with the big data characteristics. Thus, this new digital sky necessitates changes to the means and methods used for handling and exploring large amounts of astronomical data. To cope with the challenges related to the data explosion in the amounts of astronomical data and the computational complexity of the astronomical operations, we address four main contributions in this thesis:

- We proposed ASTROIDE, a unified astronomical big data processing engine over Spark. The goal is to provide a scalable, efficient and expressive astronomical query processing system. ASTROIDE achieves scalability and efficiency by combining the benefits of cost-effective data processing with Spark and customized astronomical query engine. ASTROIDE provides

an expressive query interface by unifying the data interaction method with ADQL. We automated ADQL queries parsing, rewriting and execution. ASTRODE supports spatial selection queries (Cone Search and kNN) and join queries (Cross-match and kNN Join). ASTROIDE exposes also a visualization module capable of exploring the query result returned by our query processing system.

- We proposed an approach for astronomical data partitioning combined with building indices using HEALPix. Our partitioning module is allows to improve query performances by enabling query processing in parallel, avoiding data skewness and pruning out irrelevant partitions. Besides, it reduces the amount of data scanned to get the query result and limits the communication costs. To reach the above goals, we use HEALPix as an indexing scheme to map the two-dimensional spherical coordinates into a single dimensional ID. We also leverage a range partitioner to achieve load balancing. We use a hierarchical structure to organize partitions in HDFS in order to reduce I/O, CPU and communication costs.
- We implemented a query optimizer that extends the Catalyst optimizer to handle astronomical data. We presented a variety of transformation rules for answering astronomical queries in the ASTROIDE querying engine. The objective of our optimizer is to bridge the gap between the high level query language required by astronomical applications (ADQL) and the low-level operators required by the Spark execution engine. This has the advantage to benefit from the expressiveness of ADQL, but necessitates the integration of customized rules and strategies inside the Spark's optimizer.
- We conducted an extensive experimental study, and demonstrated the effectiveness and the efficiency of our framework. We compared our solution to the state-of-the-art, and showed its superior performance for the main astronomical queries using real datasets.

7.2 Perspectives

In this thesis, we have addressed the problem of large scale astronomical data processing using Spark, a distributed in-memory computing engine. We have developed a spatial-aware query optimization module that leverages the index

support in ASTROIDE and make the best use of ASTROIDE's partitioning approach.

For future work, we would like to integrate a cost model to estimate the run time of astronomical queries in ASTROIDE. For example:

- We could define a selectivity parameter to decide whether we need to use HEALPix indexing or not. If a cone search query is not selective and returns a large fraction of data, we may decide to skip the filtering step and perform a full table scan. Such optimization [133] was recently included in GeoSpark.

- A second interesting approach could be to better handle data skew. For kNN join queries, some data partitions could be overwhelmed using big k values. To optimize such queries, we would like to integrate a cost model that takes into consideration the skewness of data partitions by supporting two types of partitions: skewed partitions and non skewed partitions. Given the estimated run-time over skewed partitions, the optimizer could split the skewed partitions into sub-partitions. A similar approach was integrated in the query optimizer of LocationSpark [134].

We would also like to test further the scalability by increasing both the data size using the second release of GAIA DR2 and the cluster size to verify that ASTROIDE inherits from the scalability of the underlying framework, Apache Spark. This could be done once we get access to a larger size cluster.

Another potential perspective could be to extend the benchmark designed in [46] to a complete benchmark that includes performance evaluation of other systems including Qserv.

It would be also interesting to replace the Catalyst optimizer in Spark with a more portable optimizer such as the Apache Calcite [1] optimizer [90]. The integration of Calcite in Spark SQL could improve the query execution time. Catalyst is essentially a heuristic optimizer, very little work has been devoted to the cost model. While Calcite combines heuristic and cost-based optimization. A benchmark [135] on TPC-DS data demonstrated that Calcite on Spark SQL could help to achieve performance improvements of two orders of magnitude.

[1] https://calcite.apache.org/

A promising idea could be to integrate Spark-fits [136] in ASTROIDE. Spark-fits was recently developed by the LAL lab of the university of Paris-Sud and offers a native Spark connector to manipulate FITS data in a distributed environment. FITS is a well-established format for exchanging and archiving astronomical data. It stands for 'Flexible Image Transport System' supported by NASA and was brought under the auspices of the International Astronomical Union (IAU).

It would be also interesting to explore new systems as a back-end such as Hops [2]. Hops is a new trend of Apache Hadoop Distribution, which offers a distributed metadata service built on a NewSQL database and ensures consistency for concurrent updates. In Hops, metadata is stored in MySQL Cluster and can scale out to many tens of nodes and tens of TBs of RAM. A promising approach could be to integrate astronomical query processing in such platform.

Finally, there is a great interest in the support of other types of astronomical data such as photometric and spectral data. This will lead to revisit our model and the framework. For instance, transients are mainly represented by multivariate time series and require advanced data manipulation for preprocessing and analytics tasks.

[2] http://www.hops.io/

List of Abbreviations

SDSS Sloan Digital Sky Survey

LSST Large Synoptic Survey Telescope

IVOA International Virtual Observatory Alliance

ADQL Astronomical Data Query Language

ICRS International Celestial Reference System

UDF User Defined Function

SQL Structured Query Language

IDC International Data Corporation

NIST National Institute of Standards and Technology

DBMS Database Management System

HDFS Hadoop Distributed File System

CDS Centre de Données de Strasbourg

RDD Resilient Distributed Dataset

QEP Query Execution Plan

ESA European Space Agency

Q3C Quad Tree Cube

FITS Flexible Image Transport System

IGSL Initial Gaia Source List

Publications

- M. Brahem, K. Zeitouni, and L. Yeh. *ASTROIDE: A unified astronomical big data processing engine over spark*. IEEE Transactions on Big Data, 2018.
- M. Brahem, K. Zeitouni, and L. Yeh. *Efficient astronomical query processing using spark*. In 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL, 2018.
- M. Brahem, K. Zeitouni, and L. Yeh. *HX-MATCH: In-memory cross-matching algorithm for astronomical big data*. In Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD, 2017
- M. Brahem, K. Zeitouni, and L. Yeh. *Large scale data management of astronomical surveys with astrospark*. In 10th Extremely Large Databases Conference, XLDB, 2017.
- K. Zeitouni, M. Brahem, and L. Yeh, *Large Scale Data Management of Astronomical Surveys with AstroSpark*, European Week of Astronomy and Space Science, EWASS, 2017.
- M. Brahem, K. Zeitouni, and L. Yeh. *Astrospark: towards a distributed data server for big data in astronomy*. In Proceedings of the 3rd ACM SIGSPATIAL PhD Symposium, ACM, 2016.
- M. Brahem, K. Zeitouni, and L. Yeh. *Large scale data management of astronomical surveys with astrospark*. In Conference on Big Data from Space, BIDS, 2017.
- M. Brahem. *Adaptative performance optimization for distributed big data server*. In Journées CNES Jeunes chercheurs, JC2, 2017.
- M. Brahem, K. Zeitouni, and L. Yeh. *Astrospark: towards a distributed data server for big data in astronomy*. In National Conference, Doctoral Session BDA, 2016.
- M. Brahem. *Astrospark: towards a distributed data server for big data in astronomy*. In Junior Conference on Data Science and Engineering of University Paris Saclay, 2016.

Bibliography

[1] D. G. York, J. Adelman, J. E. Anderson Jr, S. F. Anderson, J. Annis, N. A. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman, *et al.*, "The sloan digital sky survey: Technical summary," *The Astronomical Journal*, vol. 120, no. 3, p. 1579, 2000.

[2] "LSST." `https://www.lsst.org/`.

[3] "GAIA." `http://sci.esa.int/gaia/`.

[4] N. Astronomy, "New worlds, new horizons in astronomy and astrophysics," 2010.

[5] G. B. Berriman and S. L. Groom, "How will astronomy archives survive the data tsunami?," *Communications of the ACM*, vol. 54, no. 12, pp. 52–56, 2011.

[6] A. S. Szalay, J. Gray, P. Kunszt, A. Thakar, and D. Slutz, "Large Databases in Astronomy," in *Mining the Sky*, pp. 99–116, Springer, 2001.

[7] "ADQL." `http://www.ivoa.net/documents/latest/ADQL.html`.

[8] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[10] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pp. 1352–1363, IEEE, 2015.

[11] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1071–1085, ACM, 2016.

[12] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "*MD*-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services," *Distributed and Parallel Databases*, vol. 31, no. 2, pp. 289–319, 2013.

[13] J. Yu, J. Wu, and M. 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*, p. 70, ACM, 2015.

[14] K. M. Gorski, E. Hivon, A. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann, "HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere," *The Astrophysical Journal*, vol. 622, no. 2, p. 759, 2005.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, 2015.

[16] "Catalyst." `https://databricks.com/session/catalyst-a-query-optimization-framework-for-spark-and-shark`.

[17] J. Gantz and D. Reinsel, "Extracting value from chaos," *IDC iview*, vol. 1142, no. 2011, pp. 1–12, 2011.

[18] P. Zikopoulos, C. Eaton, *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[19] D. Laney, "3d data management: Controlling data volume, velocity and variety," *META group research note*, vol. 6, no. 70, p. 1, 2001.

[20] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[21] "M. Cooper and P. Mell. (2012). Tackling Big Data." `https://bigdatawg.nist.gov/_uploadfiles/M0065_v1_4451775754.pdf`.

[22] E. F. Codd, *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[23] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[24] S. Mazumder, R. S. Bhadoria, and G. C. Deka, *Distributed Computing in Big Data Analytics: Concepts, Technologies and Applications*. Springer, 2017.

[25] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] “Hadoop.” http://hadoop.apache.org/.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10, Ieee, 2010.

[28] “Amazon S3.” https://aws.amazon.com/fr/s3/.

[29] T. White, *Hadoop: The definitive guide*. " O’Reilly Media, Inc.", 2012.

[30] “Hbase.” https://hbase.apache.org/.

[31] “Hive.” https://hive.apache.org/.

[32] “Pig.” https://pig.apache.org/.

[33] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[34] N. Leavitt, “Will nosql databases live up to their promise?,” *Computer*, vol. 43, no. 2, 2010.

[35] “SimpleDB.” https://aws.amazon.com/fr/simpledb/.

[36] “MongoDB.” https://www.mongodb.com/.

[37] “Neo4j.” https://neo4j.com/.

[38] “Cassandra.” http://cassandra.apache.org/.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, pp. 2–2, USENIX Association, 2012.

[40] P. Zecevic and M. Bonaci, *Spark in Action*. Manning Publications Co., 2016.

[41] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. " O’Reilly Media, Inc.", 2017.

[42] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: Sql and rich analytics at scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pp. 13–24, ACM, 2013.

[43] M. Frampton, *Mastering Apache Spark*. Packt Publishing Ltd, 2015.

[44] "Cost based optimizer in Apache Spark." https://spark-summit.org/2017/events/cost-based-optimizer-in-apache-spark-22/, 2017.

[45] D. Shabalin, E. Burmako, and M. Odersky, "Quasiquotes for scala," tech. rep., 2013.

[46] A. Mesmoudi, M.-S. Hacid, and F. Toumani, "Benchmarking SQL on MapReduce systems using large astronomy databases," *Distributed and Parallel Databases*, vol. 34, no. 3, pp. 347–378, 2016.

[47] A. A. Goodman and C. G. Wong, "Bringing the night sky closer: Discoveries in the data deluge," *The Fourth Paradigm: Data-Intensive Scientific Discovery*, pp. 39–44, 2009.

[48] T. Hey, S. Tansley, K. M. Tolle, *et al.*, *The fourth paradigm: data-intensive scientific discovery*, vol. 1. Microsoft research Redmond, WA, 2009.

[49] G. Bell, T. Hey, and A. Szalay, "Beyond the data deluge," *Science*, vol. 323, no. 5919, pp. 1297–1298, 2009.

[50] J. Gray, D. Slutz, A. Szalay, and A. Thakar, "Jan vandenberg, peter kunszt, and chris stoughton. data mining the sdss skyserver database," tech. rep., Technical Report MSR-TR-2002-01, MSR, 2002.

[51] "SkyServer." http://skyserver.sdss.org/dr6/en/proj/.

[52] V. Singh, J. Gray, A. Thakar, A. S. Szalay, J. Raddick, B. Boroski, S. Lebedeva, and B. Yanny, "Skyserver traffic report - the first five years," *CoRR*, vol. abs/cs/0701173, 2006.

[53] "SkyServer Traffic." http://skyserver.sdss.org/log/en/traffic/.

[54] F. Ochsenbein, P. Bauer, and J. Marcout, "The VizieR database of astronomical catalogues," *Astronomy and Astrophysics Supplement Series*, vol. 143, no. 1, pp. 23–32, 2000.

[55] S. Derriere, F. Ochsenbein, and D. Egret, "On-line access to very large catalogues," in *Astronomical Data Analysis Software and Systems IX*, vol. 216, p. 235, 2000.

[56] S. Koposov and O. Bartunov, "Q3C, Quad Tree Cube–the new sky-indexing concept for huge astronomical catalogues and its realization for main astronomical queries (cone search and Xmatch) in open source database PostgreSQL," in *Astronomical Data Analysis Software and Systems XV*, vol. 351, p. 735, 2006.

[57] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay, "Cross-matching very large datasets," in *National Science and Technology Council (NSTC) NASA Conference*, 2007.

[58] T. Budavári, A. Szalay, J. Gray, W. O'Mullane, R. Williams, A. Thakar, T. Malik, N. Yasuda, and R. Mann, "Open skyquery–vo compliant dynamic federation of astronomical archives," in *Astronomical Data Analysis Software and Systems (ADASS) XIII*, vol. 314, p. 177, 2004.

[59] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten, "Monetdb/sql meets skyserver: the challenges of a scientific database," in *Scientific and Statistical Database Management, 2007. SSBDM'07. 19th International Conference on*, pp. 13–13, IEEE, 2007.

[60] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database," 2012.

[61] J. VanderPlas, E. Soroush, K. S. Krughoff, M. Balazinska, and A. Connolly, "Squeezing a Big Orange into Little Boxes: The AscotDB System for Parallel Processing of Data on a Sphere.," *IEEE Data Eng. Bull.*, vol. 36, no. 4, pp. 11–20, 2013.

[62] "SciDB." `https://www.paradigm4.com/try_scidb/`.

[63] D. Marcos, A. Connolly, K. Krughoff, I. Smith, and S. Wallace, "Ascot: a collaborative platform for the virtual observatory," in *Astronomical Data Analysis Software and Systems XXI*, vol. 461, p. 901, 2012.

[64] D. L. Wang, S. M. Monkewitz, K.-T. Lim, and J. Becla, "Qserv: A distributed shared-nothing database for the lsst catalog," in *State of the Practice Reports*, p. 12, ACM, 2011.

[65] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky, "Xrootd-a highly scalable architecture for data access," *WSEAS Transactions on Computers*, vol. 1, no. 4.3, 2005.

[66] J. Gray, M. A. Nieto-Santisteban, and A. S. Szalay, "The zones algorithm for finding points-near-a-point or cross-matching spatial datasets," *CoRR*, vol. abs/cs/0701171, 2006.

[67] F.-X. Pineau, T. Boch, and S. Derriere, "Efficient and scalable cross-matching of (very) large catalogs," in *Astronomical Data Analysis Software and Systems XX*, vol. 442, p. 85, 2011.

[68] Q. Zhao, J. Sun, C. Yu, C. Cui, L. Lv, and J. Xiao, "A paralleled large-scale astronomical cross-matching function," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 604–614, Springer, 2009.

[69] "AstroLab Software." https://astrolabsoftware.github.io/.

[70] A. Aji, X. Sun, H. Vo, Q. Liu, R. Lee, X. Zhang, J. Saltz, and F. 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*, pp. 528–531, ACM, 2013.

[71] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[72] H. Vo, A. Aji, and F. Wang, "Sato: a spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 545–548, ACM, 2014.

[73] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, vol. 19, pp. 322–331, Acm, 1990.

[74] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.," tech. rep., 1987.

[75] A. Eldawy and M. F. Mokbel, "Pigeon: A spatial mapreduce language," in *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pp. 1242–1245, IEEE, 2014.

[76] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[77] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: a distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 2016.

[78] A. Eldawy and M. F. Mokbel, "The Era of Big Spatial Data," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1992–1995, 2017.

[79] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, 2018.

[80] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[81] J. Widom, H. Garcia-Molina, and J. D. Ullman, “Database systems the complete book,” 2009.

[82] S. K. Singh, *Database systems: Concepts, design and applications*. Pearson Education India, 2011.

[83] G. Mantelet, “ADQL library.” `https://github.com/gmantele/taplib`.

[84] T. Ibaraki and T. Kameda, “On the optimal nesting order for computing n-relational joins,” *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 3, pp. 482–502, 1984.

[85] S. T. Shenoy and Z. M. Ozsoyoglu, “Design and implementation of a semantic query optimizer,” *IEEE transactions on Knowledge and data Engineering*, vol. 1, no. 3, pp. 344–361, 1989.

[86] R. Elmasri and S. Navathe, *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[87] M. Steinbrunn, G. Moerkotte, and A. Kemper, “Heuristic and randomized optimization for the join ordering problem,” *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 6, no. 3, pp. 191–208, 1997.

[88] D. Kossmann, “The state of the art in distributed query processing,” *ACM Computing Surveys (CSUR)*, vol. 32, no. 4, pp. 422–469, 2000.

[89] S. Chaudhuri, “An overview of query optimization in relational systems,” in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, ACM, 1998.

[90] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, “Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources,” in *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230, ACM, 2018.

[91] “Calcite.” `http://calcite.apache.org/docs/powered_by`.

[92] M. Golfarelli and L. Baldacci, “A cost model for spark sql,” *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[93] J. J. Mwemezi and Y. Huang, “Optimal facility location on spherical surfaces: algorithm and application,” *New York Science Journal*, vol. 4, no. 7, pp. 21–28, 2011.

[94] R. Plante, R. Williams, R. Hanisch, and A. Szalay, “Simple cone search version 1.03,” *IVOA Recommendation 22 February 2008*, 2008.

[95] "IVOA." http://www.ivoa.net/.

[96] M. Brahem, S. Lopes, L. Yeh, and K. Zeitouni, "AstroSpark: towards a distributed data server for big data in astronomy," in *Proceedings of the 3rd ACM SIGSPATIAL PhD Symposium*, ACM, 2016.

[97] M. BRAHEM, K. Zeitouni, and L. Yeh, "Astroide: A unified astronomical big data processing engine over spark," *IEEE Transactions on Big Data*, 2018.

[98] M. Brahem, , K. Zeitouni, and L. Yeh, "Efficient astronomical query processing using spark," in *In 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ACM, 2018.

[99] H. Herodotou, N. Borisov, and S. Babu, "Query optimization techniques for partitioned tables," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 49–60, ACM, 2011.

[100] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar, "Indexing the sphere with the hierarchical triangular mesh," *CoRR*, vol. abs/cs/0701164, 2005.

[101] P. Kunszt, A. Szalay, I. Csabai, and A. Thakar, "The indexing of the sdss science archive," in *Astronomical Data Analysis Software and Systems IX*, vol. 216, p. 141, 2000.

[102] R. W. Youngren and M. D. Petty, "A multi-resolution HEALPix data structure for spherically mapped point data," *Heliyon*, vol. 3, no. 6, p. e00332, 2017.

[103] P. Fernique, M. Allen, T. Boch, A. Oberto, F. Pineau, D. Durand, C. Bot, L. Cambresy, S. Derriere, F. Genova, *et al.*, "Hierarchical progressive surveys-Multi-resolution HEALPix data structures for astronomical images, catalogues, and 3-dimensional data cubes," *Astronomy & Astrophysics*, vol. 578, p. A114, 2015.

[104] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *ACM Sigmod Record*, vol. 15, pp. 326–336, ACM, 1986.

[105] W. O'Mullane, A. Banday, K. Gorski, P. Kunszt, and A. Szalay, "Splitting the sky-htm and healpix," in *Mining the Sky*, pp. 638–648, Springer, 2000.

[106] "HEALPix Softaware." http://healpix.sourceforge.net/.

[107] M. A. Nieto-Santisteban, A. R. Thakar, A. S. Szalay, and J. Gray, "Large-scale query and xmatch, entering the parallel zone," in *Astronomical Data Analysis Software and Systems XV*, vol. 351, p. 493, 2006.

[108] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[109] D. Gao, Y. Zhang, and Y. Zhao, “The Application of kd-tree in Astronomy,” in *Astronomical Data Analysis Software and Systems XVII*, Astronomical Society of the Pacific Conference Series, 2008.

[110] A. Eldawy, L. Alarabi, and M. F. Mokbel, “Spatial partitioning techniques in spatialhadoop,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1602–1605, 2015.

[111] H. Vo, A. Aji, and F. Wang, “Sato: A spatial data partitioning framework for scalable query processing,” in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL ’14, (New York, NY, USA), ACM, 2014.

[112] H. Singh and S. Bawa, “A survey of traditional and mapreducebased spatial query processing approaches,” *ACM SIGMOD Record*, vol. 46, no. 2, pp. 18–29, 2017.

[113] W. Wang, J. Yang, R. Muntz, *et al.*, “Sting: A statistical information grid approach to spatial data mining,” in *VLDB*, vol. 97, pp. 186–195, 1997.

[114] M. F. Mokbel, W. G. Aref, and I. Kamel, “Performance of multi-dimensional space-filling curves,” in *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pp. 149–154, ACM, 2002.

[115] J. K. Lawder and P. J. H. King, “Querying multi-dimensional data indexed using the hilbert space-filling curve,” *ACM Sigmod Record*, vol. 30, no. 1, pp. 19–24, 2001.

[116] B. Yao, F. Li, and P. Kumar, “K nearest neighbor queries and knn-joins in large relational databases (almost) for free,” in *Data engineering (ICDE), 2010 IEEE 26th international conference on*, pp. 4–15, IEEE, 2010.

[117] D. Hilbert, “Über die stetige abbildung einer linie auf ein flächenstück,” in *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pp. 1–2, Springer, 1935.

[118] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, “Analysis of the clustering properties of the hilbert space-filling curve,” *IEEE Transactions on knowledge and data engineering*, vol. 13, no. 1, pp. 124–141, 2001.

[119] A. Guttman, *R-trees: A dynamic index structure for spatial searching*, vol. 14. ACM, 1984.

[120] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects.," tech. rep., 1987.

[121] "Aladin." http://aladin.u-strasbg.fr/.

[122] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *Data Engineering, 1997. Proceedings. 13th international conference on*, pp. 497–506, IEEE, 1997.

[123] J. A. Orenstein and F. A. Manola, "Probe spatial data modeling and query processing in an image database application," *IEEE transactions on Software Engineering*, vol. 14, no. 5, pp. 611–629, 1988.

[124] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw Hill, 2000.

[125] M. Brahem, K. Zeitouni, and L. Yeh, "HX-MATCH: In-Memory Cross-Matching Algorithm for Astronomical Big Data," in *International Symposium on Spatial and Temporal Databases*, pp. 411–415, Springer, 2017.

[126] "Galactica." https://galactica.isima.fr//.

[127] A. G. Brown, A. Vallenari, T. Prusti, J. De Bruijne, F. Mignard, R. Drimmel, C. Babusiaux, C. Bailer-Jones, U. Bastian, M. Biermann, *et al.*, "Gaia Data Release 1-Summary of the astrometric, photometric, and survey properties," *Astronomy & Astrophysics*, vol. 595, p. A2, 2016.

[128] "IGSL." http://cdsarc.u-strasbg.fr/viz-bin/Cat?I/324.

[129] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, 2015.

[130] "OpenStack." https://www.openstack.org/.

[131] M. Taylor, "Topcat: Desktop exploration of tabular data for astronomy and beyond," in *Informatics*, vol. 4, p. 18, Multidisciplinary Digital Publishing Institute, 2017.

[132] M. B. Taylor, "Stilts-a package for command-line processing of tabular data," in *Astronomical Data Analysis Software and Systems XV*, vol. 351, p. 666, 2006.

[133] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: the geospark perspective and beyond," *GeoInformatica*, pp. 1–42, 2018.

[134] M. Tang, Y. Yu, W. Aref, A. Mahmood, Q. Malluhi, and M. Ouzzani, “In-memory distributed spatial query processing and optimization,” tech. rep., Purdue technical report, 2016.

[135] “Grunion.” https://www.datascience.com/blog/grunion-data-science-tools-query-optimizer-apache-spark/.

[136] J. Peloton, C. Arnault, and S. Plaszczynski, “Fits data source for apache spark,” *Computing and Software for Big Science*, vol. 2, no. 1, p. 7, 2018.