ASTROIDE: A UNIFIED ASTRONOMICAL BIG DATA PROCESSING ENGINE OVER SPARK

A PREPRINT

Mariem Brahem
DAVID lab.
Univ. Versailles St Quentin
Paris Saclay University, Versailles France
mariem.brahem@uvsq.fr

Laurent Yeh
DAVID lab.
Univ. Versailles St Quentin
Paris Saclay University, Versailles France
laurent.yeh@uvsq.fr

Karine Zeitouni
DAVID lab.
Univ. Versailles St Quentin
Paris Saclay University, Versailles France
karine.zeitouni@uvsq.fr

October 25, 2018

ABSTRACT

SThe next decade promises to be an exciting time for astronomers. Large volumes of astronomical data are continuously collected from highly productive space missions. This data has to be efficiently stored and analyzed in such a way that astronomers maximize their scientific return from these missions. Recognizing the need to better handle astronomical datasets, we designed ASTROIDE, a distributed data server for astronomical data. We analyze the peculiarities of the data and the queries in cosmological applications and design a new framework where astronomers can explore and manage vast amounts of data. ASTROIDE introduces effective methods for efficient astronomical query execution on Spark through data partitioning with HEALPix and customized optimizer. ASTROIDE offers a simple, expressive and unified interface through ADQL, a standard language for querying databases in astronomy. Experiments have shown that ASTROIDE is effective in processing astronomical data, scalable and outperforms the state-of-the-art.

Keywords Astronomical Survey Data Management · Big Data · Query Processing · Spark Framework

1 Introduction

There has been an accelerating increase of astronomical data produced by advanced telescopes. For instance, the GAIA mission [1] and the future LSST [2] survey are expected to produce Petabytes of data. 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. But, until now it is still a big problem due to the following challenges: The growing scale of astronomical data, the increased accuracy of observation tools brings a change of paradigm for data processing. Also, most astronomical operations are very expensive to process because of their compute-intensive nature [3] especially for complex and costly operations. Among these 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 requires a scalable solution that provides full-fledged spatial data exploration. To resolve 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. However, the specific context of astronomical data handling brings up many questions: (i) how to partition data ? (ii) which data indexing technique should be utilized ? (iii) how to reduce the complexity of astronomical operations ? (iv) and how to provide the astronomers with an expressive and efficient support of complex queries, i.e. combining several operations, without worrying about query optimization ?

Recently, the shared-nothing type of parallel architecture, which uses commodity hardware, is becoming a de facto standard in scientific massive data handling. In this context, the distributed in-memory computing framework Apache Spark [4] 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 because Spark is agnostic of spatial search or cross-matching semantics and optimization rules. 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.

There exists a number of systems [5] [6] [7] [8] that 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 coordinates system, which leads to erroneous query results.
- The query performances remain limited due to unsuitable data partitioning scheme.

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

In this work, we present ASTROIDE, a distributed data server tailored for the management of large volume of astronomical data. ASTROIDE is designed as an extension of Apache Spark, and takes into account the peculiarities of the data and the queries related to astronomical surveys and catalogs. Queries are expressed in Astronomical Data Query Language (ADQL) [9], an SQL extension with astronomical functions. Various logical and physical optimization techniques for the ADQL execution are proposed, and integrated to Spark SQL 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.

In fact, ASTROIDE implements a data partitioner that achieves both spatial locality and load balancing. It also adopts a well-known sky pixelization technique, by using the spherical space filling curve scheme, namely HEALPix [10], to achieve high performance query execution. Furthermore, it transparently translates the user's query into an efficient execution plan, by using these physical optimization techniques within the implementation of the operators.

The main contributions of this paper are as follows:

- We study the requirements of big astronomical data management and show the limitations of the existing solutions.
- We propose ASTROIDE[1] as an extension of Apache Spark, a distributed in-memory computing engine to process and analyze astronomical data.
- We provide a high level access and manipulation by supporting the unified query language ADQL, widely used in the domain.
- In order to speed up query processing, we combine data partitioning with HEALPix pixelization of spherical data.
- We implement a query optimizer within ASTROIDE and provide a set of customized strategies for the optimization of astronomical query plans.
- Experiments on real datasets of up to 1.2 billion objects have demonstrated that ASTROIDE achieves high scalability for querying big astronomical datasets.

[1] `https://github.com/MBrahem/ASRTOIDE`

The rest of this paper is organized as follows. Section 2 introduces the necessary background. Section 3 discusses the related works. Section 4 provides the system architecture overview of ASTROIDE. In Section 5, we explain the execution details of the main astronomical queries. Experimental results over big astronomical datasets are presented in Section 6, while Section 7 summarizes the paper and provides some directions of future work.

2 Background

2.1 HEALPix

The Hierarchical Equal Area isoLatitude Pixelization (HEALPix) [10] [11] [12] 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 a 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 NSIDE parameter, which controls the desired resolution. (see Figure 1)

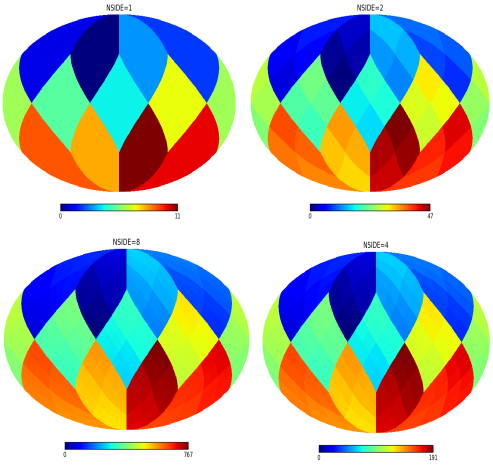


Figure 1: HEALPix partition of the sphere (NSIDE = 1, 2, 4, 8) [10].

We have adopted the HEALPix software for the following reasons:

- 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.
- The software for HEALPix is easily accessible and contains many functionalities that are useful in our context such as filtering neighbor pixels or those in a cone.

Prior work has proposed other spatial indices that are suitable for celestial objects, including Hierarchical Triangular Mesh (HTM) [13]. Comparison between the two indexing techniques is detailed in [14]. 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. The difference between the two schemes is the subdivision method. HEALPix partitions the sphere into exactly equal area quadrilaterals of varying shape whereas HTM subdivides the sphere into spherical triangles of similar, but not identical, shapes and sizes. In this work, we choose HEALPix as our indexing scheme, but another scheme like HTM can be applied on the same principle.

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 1: Used notations.

2.2 Astronomical Queries

For a sake of consistency, we introduce the notations listed in Table 1.

In the International Celestial Reference System (ICRS), the sky is projected on a celestial sphere. The position of a star is recorded as a 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. `dec` is the latitude-like coordinate, it measures the angular distance of a celestial object north or south of the celestial equator.

ASTROIDE focuses on three main basic astronomical queries, that require more calculations than ordinary search or join in legacy relational databases. They are formally defined hereafter.
We define the spherical distance between r and s as $SD(r, s)$, it reflects the shortest distance between r and s over the celestial sphere. It is calculated with the Harvesine formula [15] as the length of the great circle arc between two points on a sphere:

$$SD(r, s) = 2 \arcsin \sqrt{\sin^2(d)/2 + \cos(d_1) \cos(d_2) \sin^2(a/2)}$$

Where:
d_1 = Declination of r
d_2 = Declination of s
d = d_1 - d_2 = Difference in declination between r and s
a = Difference in right ascension between r and s

Cone Search is one of the most frequent queries in the astronomical domain. It returns a set of stars whose positions lie within a circular region of the sky. 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. Formally [2]:

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

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.

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

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

kNN search: Given a query point p, a dataset S and an integer $k > 0$, the k nearest neighbors from S denoted $k\text{NN}(p, S)$ is a set of k objects such that:

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

Moreover, we can extend these queries to encompass other attributes. For instance, a query example can be: *Selection of sources within a certain angular distance from the specified center position (cone search) fitting as well limitations on a certain magnitude range (or a particular spectral type) ordered by magnitude.*

[2] According to [16] from 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.

Astronomical queries are commonly expressed using the Astronomical Data Query Language (ADQL) [9]. ADQL is a well-known language adapted to query astronomical data, and is promoted by the International Virtual Observatory Alliance (IVOA). 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`, `ENTROID`, `CIRCLE`, `CONTAINS`, etc. For example, `CIRCLE` expresses a circular region on the sky that corresponds to a cone in space. The ADQL expression of the above query example is formulated as follows:

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

A full specification of ADQL is available in [9]. Here, we present and implement the astronomical queries cone search, cross-match, and kNN search, that are expressed using ADQL. These queries use ADQL geometrical functions including `DISTANCE`, `CIRCLE`, `POINT`, `CONTAINS`. For the time being, our prototype does not provide a complete compliance with all the features of ADQL such as geometrical data types. However, the full coverage of ADQL does not raise any research issue.

2.3 Spark

Apache Spark [4] is rising in popularity as an alternative to disk oriented processing, due to its ability for executing most computations in memory and its expressive API for complex data processing. Apache Spark is a fast and general purpose engine for large scale data processing. Spark mainly offers an immutable distributed collection of objects for in-memory cluster computing called Resilient Distributed Dataset (RDDs). RDDs are splitted into multiple partitions and operated on in parallel.

Spark SQL [17], a component on top of Spark core, introduces a new data abstractions called Dataset (a distributed collection of data) and DataFrame. A DataFrame represents a dataset with a defined schema unlike RDDs. Internally, each DataFrame is represented as logical plan in Spark. This logical plan is transformed to a physical plan by applying a set of strategies.

Spark SQL provides also an extensible query optimizer called Catalyst [17], which makes adding new optimization techniques easy. It proposes two ways: by defining custom rules and strategies. Catalyst is a general library for representing logical plans as trees and sequentially applying a number of optimization rules to manipulate them. Recently, a cost based optimization module (CBO) 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. However, Spark SQL does not provide natively efficient astronomical query processing capabilities. For that, we need to design and implement a new framework suitable for astronomical data that leverages the power of this framework and profits from the extensibility of its optimizer.

3 Related Work

Early astronomical servers. The VizieR service [18] developed by the Centre de Données de Strasbourg (CDS) is an on-line database for accessing astronomical data listed in published catalogs and executing cross-matching queries. Linking database and astronomy has been also introduced by Jim Gray in the SkyServer project [19].

An early cross-match algorithm [20] uses a zoning algorithm to implement of points-near-point, spatial cross-match, and self-match queries. The basic idea is to map the sphere into zones, each zone is a declination stripe of the sphere. The implementation of the zoneMatch algorithm is based on Microsoft SQL server where each object is associated to a zoneID and the cross-match is done through SQL statements using predicates on zoneID.

Besides, some recent works [21] [22] [23] [24] [20] proposed customized solutions to execute the main astronomical queries. In [21] and [24], authors introduce a cross-matching function including a partitioning approach with HEALPix. Q3C [22], standing for Quad Tree Cube, provides a sky partitioning schema for PostreSQL and offers an SQL interface for the main astronomical queries. OPEN SKYQUERY [23] allows querying and cross-matching distributed astronomical datasets. The authors propose zoning and partitioning algorithms for parallel query execution using RDBMS technologies.

In [25] [26], authors extend MonetDB, an open source database system based on a column store approach to manage astronomical data. The goal of the authors in [26] is to optimize a column-oriented database to enable the support of large-scale surveys. They implemented a new cross-matching algorithm using hash indexes on a single sorted column of declination values.

AscotDB [27] builds on the combination of several pieces of technology: the SciDB, a scalable database that stores data in distributed and multidimensional arrays, AStronomical COllaborative Toolkit (Ascot) for graphical data exploration, and Python for easy programmatic access. It also indexes the spherical coordinates by using HEALPix. This have led the authors to propose a specific mapping schema between the spherical grid induced by HEALPix and the multidimensional array data structure on which SciDB is based.

However, none of these efforts provide a scalable astronomical server with a unified programming interface like ADQL, as we target. Their performance are still limited because they are mainly based on centralized or rigid server architecture style.

Approaches based on distributed frameworks. In [28], authors report the ability of existing MapReduce management systems (Hive, HadoopDB) to support large scale declarative queries in the area of cosmology. But the query cases used in this work differs from our context because they don not cover the spatial queries, which are typical in astronomical applications.

Recent works have addressed the support of spatial data and queries using a distributed data server. Their architecture have followed the development of the Hadoop ecosystem. We devise these works in four representative proposals.

SpatialHadoop [5] 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 an expressive high level SQL-like language for spatial data types and spatial operations. In the storage layer, SpatialHadoop adapts traditional spatial index structures, Grid, R-tree [29] and R^+-tree [30], to form a two-level index structure, called global and local indexing. SpatialHadoop enriches the MapReduce layer by adding two new components, SpatialFileSplitter and SpatialRecordReader. In the operations layer, SpatialHadoop focuses on three basic operations: range query, spatial join, and k nearest neighbor (kNN).

MD-HBase [7] 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, builds standard index structures like kd-trees and Quad-trees to support range and kNN queries.

GeoSpark [8] 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) concept to support spatial data (SRDDs).

GeoSpark provides native support for spatial data indexing (R-Tree and Quad-Tree) and query processing algorithms (range queries, kNN queries, and spatial joins over SRDDs) to analyze spatial data.

More recently, SIMBA [6] has been proposed as an extension of Spark SQL (not only at the core level of Spark) to support spatial queries and analytics over big spatial data. SIMBA builds spatial indexes over RDDs. It offers a programming interface to execute spatial queries (range queries, circle range queries, kNN, Distance join, kNN join), and uses cost based optimization.

In [31], authors provide a comprehensive tutorial that reviews all existing research efforts in the era of big spatial data and classifies existing works according to their implementation approach, underlying architecture, and system components.

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.

They 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 coordinates system. They use conventional spatial indexing techniques while we adopt specialized indexing method for astronomical data by using HEALPix. We also deal with specific astronomical operations such as cone search queries, cross-match queries, and kNN queries that are not supported by these systems.

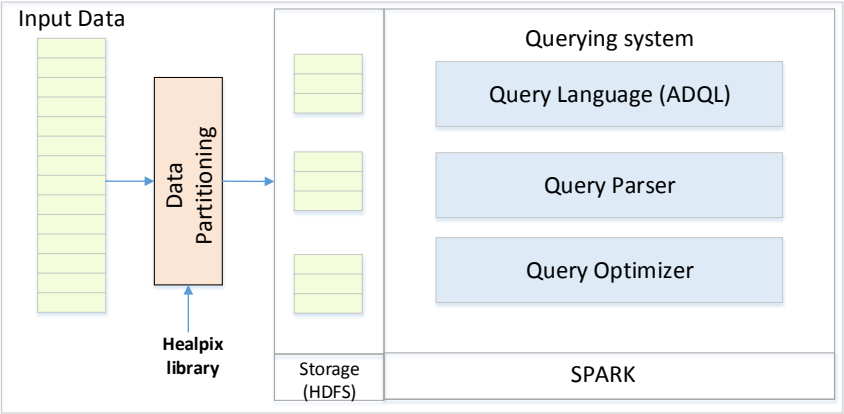


Figure 2: ASTROIDE architecture.

4 ASTROIDE Architecture

ASTROIDE [32] is a distributed data server for big data in astronomy. It is based on the distributed in-memory computing framework Spark, to query huge volume of astronomical data.

Figure 2 shows the overall architecture of ASTROIDE. ASTROIDE adapts data partitioning to efficiently process astronomical queries. Queries are expressed using ADQL, providing astronomical functions. To this end, the query parser is in charge of translating the ADQL query into an internal algebraic representation. ASTROIDE extends the Spark SQL optimizer called Catalyst by integrating specific logical and physical optimization techniques. The query processing of ASTROIDE relies on the code generation engine of Spark to operate on in-memory datasets.

4.1 Data Partitioning

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, this globally improves the query performances.

Spark provides two predefined partitioning mechanisms using DataFrames: hash partitioning and range partitioning. Hash partitioner is the default partitioner. 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. The partition index is determined quasi-randomly; consequently, objects having close key values are likely to end up in different partitions. Whereas the range partitioner divides data into roughly equal ranges. Range partitioner sorts the input records based on their keys and splits the DataFrame into a defined number of partitions. However, these 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 others 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 Spark DataFrames in a way the 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. Close points on the sky have their HEALPix IDs close to each other. So, we sort the data based on their IDs.

To achieve the load balancing, we leverage the Spark range partitioner, which yields data partitions with roughly equal sizes. Then, the partitions are stored on HDFS [33]. We compute the number of partitions using this formula:

$$n = |S|/P * (1 + m)$$

Where $|S|$ is the input size and P is the partition size. m is an adjustment factor to allow for the possible variation in the sizes among the partitions. 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. 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.

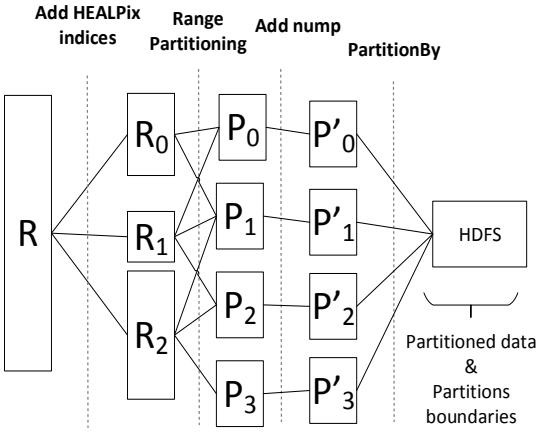


Figure 3: ASTROIDE partitioner.

Figure 3 shows ASTROIDE's partitioner. For partitioning an input DataFrame into a set of partitions R_0, R_1, R_2, ASTROIDE builds indices over raws in each partition R_i using HEALPix. We apply range partitioning on indexed 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 efficiently the data among partitions. Since Spark is an in-memory analytical engine, this efficient distribution is only accessible in-memory. To amortize the construction cost for future queries, we store partitioned data on HDFS. So, our solution is to associate a partition number `nump` to each partition in memory and pack each partition with this number on HDFS.

The DataFrame is partitioned by `nump` using a Spark existing function `partitionBy(nump)`. So, ASTROIDE structures the data on HDFS in such a way that each partition is saved in a separate subdirectory containing records with the same partition number `nump`. ASTROIDE divides also each partition into buckets using the spark function `bucketBy` using HEALPix index as the bucketing column `ipix` . 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/nump=20/ipix=114571)`. 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`.

4.2 Query Parsing

The query parser allows the extension of Spark to deal with astronomical queries and ADQL. We have proposed and evaluated three alternatives of query parsing. The idea is to follow the same Spark interaction ways including SQL and direct DataFrame API:

- Case 1: 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 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.
- Case 2: Query rewriting. If the query is written using ADQL, the query parser verifies first 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 2.2. After that, we convert the initial query into a parse tree by using a tool provided by the CDS [34]. Our query rewriting module receives this processed tree and applies a number of transformations and optimizations to produce a new equivalent, but optimized query compliant with the syntax of Spark SQL. The transformations performed depend on the type of the query and substitute parts of the query matching an astronomical pattern with replacement SQL text. We incorporate filters into the optimized query in order to avoid scanning irrelevant partitions. Query parsing performed in ASTROIDE is completely transparent to the user.

Internally, there is no difference between using the DataFrame API (case 1) or ADQL (case 2) 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.

4.3 Query Optimization

The query optimizer is an important component since it is the responsible for generating a query execution plan that computes the query result efficiently. ASTROIDE extends the Spark Catalyst optimizer by adding custom strategies for astronomical queries. We integrate new strategies for converting Spark non optimized logical plan to an optimized physical plan.

The output physical plan is the actual plan which ASTROIDE executes for the final data processing. The query optimizer performs the implemented strategies on the input query to transform the query plan tree into equivalent, but with optimized form by rewriting costly operators or by adding filter operator on our data structure. For instance, Figure 4 shows the transformation of a cross-matching query planning.

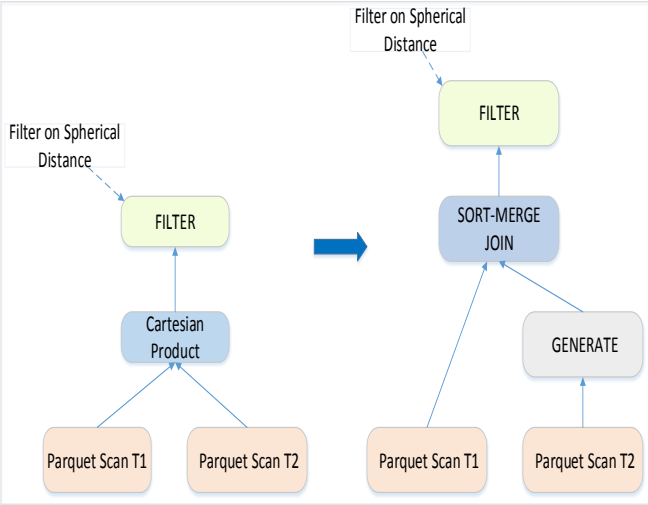


Figure 4: Cross-matching execution plan.

The naive approach of cross-matching uses a cartesian product to execute a join on the two inputs datasets, and then filters the output according to the join condition. The cartesian product is inefficient for big datasets. Instead, ASTROIDE's optimizer adds new optimizations strategies that transform all logical plans that implements a cartesian product with filter on a spherical distance formula to an optimized physical plan that we will explain in Section 5.2.

5 Astronomical queries in ASTROIDE

5.1 Cone Search Query

A cone search query takes a dataset R, a point p and a radius sr as inputs and returns the records in R that overlap the cone defined by p and sr. Spark SQL has to scan the entire input table to execute a cone search query Without any

accelerating data structure (e.g. index). For instance, the following SQL query finds all rows with positions within `0.0833` degrees of `ra=266` and `dec=-29` in a table called `gaia`:

```
SELECT *
FROM gaia
WHERE (2 *ASIN(SQRT(SIN((dec-(-29)/2)*
      SIN((dec-(-29))/2) + COS(-29)*
      COS(-29)* SIN((ra-266)/2)*
      SIN((ra - 266)/2))) <= 0.0833)
```

In contrast, ASTROIDE introduces an efficient plan that handles the query using partitioning. ASTROIDE starts by finding all pixels within the query cone using the HEALPix library, group these cells by range and uses the partition boundaries to obtain IDs of partitions which overlap these ranges and thus avoid reading irrelevant partitions. Finally, ASTROIDE filters each selected partition with the spherical distance formula condition.

Using ADQL or DataFrame API to execute the cone search gives the same execution time, because both generates the same physical plan (as shown in Figure 5) . The ADQL code of the cone search query is:

Query X

```
SELECT *
FROM gaia

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

If we apply the above method, the query X is rewritten in Y. Notice that we integrate in the transformed query the filtering of relevant partitions.

Query Y

```
SELECT *
FROM gaia
WHERE (nump IN (100)
      AND ipix IN (114571,114572,114580)
      AND SphericalDistanceICRS(ra,dec,266,-29)
      <0.0833))
```

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

Figure 5 explains the cone search execution plan. ASTROIDE applies Parquet *pushed filters* and *partitions filters* by evaluating the filter expressions in the query above. ASTROIDE drives Spark to prune the data and reduces the amount of data that Spark must read during query runtime. Parquet *filters pushdown* relies on buckets pruning on the HEALPix indices (which corresponds to a column called `ipix`). The *pushdown* is a logical optimization rule that consists on applying filtering operation as early as possible in order to prevent loading unneeded rows. Whereas, *partition filters* is based on partitions pruning, ASTROIDE skips scanning partitions that don't match the values specified in the query. Finally, a filtering step is required to filter objects based on spherical distance.

5.2 Cross-matching Query

To avoid a cartesian Product, our idea is to first limit the distance computation to pairs belonging either to the same cell or to neighboring cells by using the HEALPix indices, thus generating matching candidates. Then a refinement step computes their exact distance and filters the actual matching pairs

In the first step, the objective is to match the data in the same cell or in neighboring cells based on their HEALPix indices. We basically leverage a traditional equi-join of the datasets R and S on HEALPix indices to generate the

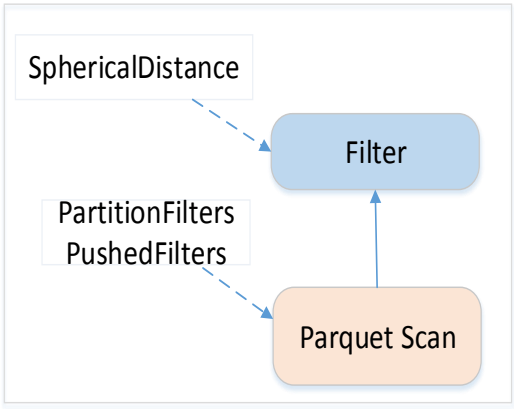


Figure 5: Cone search execution plan.

Algorithm 1 : HX-MATCH(R,S)

Input: Input datasets R and S, search radius ϵ
Output: Matching pairs of stars that satisfy the predicate $sphericalDistance(r, s) < \epsilon$

1: $S^+ \leftarrow \emptyset$
2: **for** each t $\in$ S **do**
3: **for** each n $\in$ Neighbors(t.ipix) **do**
▷ Neighbors is a user defined function which returns the current cell and all the neighboring cells.
4: $S^+ = S^+ \cup \{ t' \mid t'$.ipix = n $\wedge$ t'.a = t.a, a $\neq$ ipix$\}$
▷ Clone each object in S and assign it the HEALPix number of its neighbor cells
5: **end for**
6: **end for**
7: candidates = Join(R, S^+, $R.ipix$= $S^+.ipix$)
8: matched = Filter(candidates, $sphericalDistance < \epsilon$)
9: return matched
▷ sphericalDistance is a user defined function which computes the harvesine distance between two data points

candidates sharing the same cell. As for the candidates belonging to neighbor cells of an object, a naive approach would be to do a θ-join of the two datasets where the criteria is the membership of the HEALPix indice in the one to the set of HEALPix indices of the neighbors in the other dataset. But θ-join does not benefit from the optimization techniques, which exist for equi-join. To cope with this problem, we augment one of the datasets by replicating all objects on the fly as the cross matching query is issued. 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 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.

The pseudo-code of cross-matching (HX-MATCH) is explained in Algorithm 1. It takes as input two partitioned files R and S and a search radius ϵ and returns all matching points that satisfy a distance criteria. The algorithm runs as follows:

1. 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, where we substitute the HEALPix value (here ipix) by the current cell and the one of each neighbor cell (Line 4).
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 starts by generating a logical plan that is resolved to a customized physical plan represented in Figure 4. This physical plan defines computation operators on DataFrames. ASTROIDE starts by scanning the two input parquet files. Then, it generates neighbors of each object, runs a sort merge join algorithm, and finally, filters the output result according to the spherical distance predicate.

Besides the previous form based on Spark API, users can also express a cross-matching query between R and S as follows (here with an ϵ distance of 2 arc-second):

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

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

```
SELECT *
FROM R JOIN
    (SELECT *,explode(Neighbors(S.ipix))
    AS ipix_nei
    FROM S) AS SA
  ON (R.ipix=SA.ipix_nei)
WHERE SphericalDistanceICRS(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 the HEALPix cell with its neighbors and outputs the elements of the array as separate rows. Neighbors 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.

5.3 k Nearest Neighbor Query

A kNN query takes a dataset R, a point p and an integer k as inputs and returns the k closest objects to p in R. Using Spark SQL, the baseline approach is to scan all objects in R, to calculate their distances to p and selects the top-k objects. An example of a relational query expressing a `6` nearest neighbor query for a point p=`(44.97,0.09)` from a table `gaia` is:

```
SELECT *
FROM gaia
ORDER BY (2*ASIN(SQRT(SIN((dec-0.09)/2)
         *SIN((dec-0.09)/2) + COS(dec)
         *COS(0.09)*SIN((ra-44.97)/2)*
         SIN((ra-44.97)/2))))
LIMIT 6
```

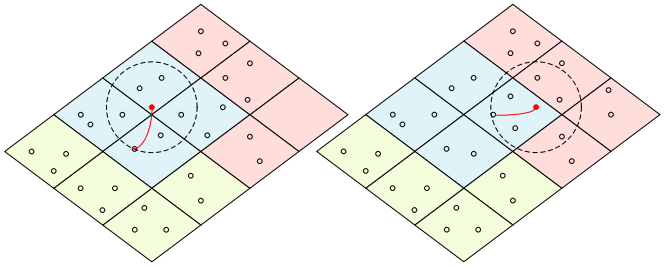
Such a query requires at least a sort with a complexity in $O(n * log_n)$. In ASTROIDE, we avoid this and design a more efficient algorithm for kNN queries. First, we identify the cell that includes the point p using the HEALPix library, we use the function `getPix` which returns the HEALPix index of an object represented by spherical coordinates at a defined resolution. Then, we locate the partition that intersects this cell using partitions metadata that we created using our partitioner.

The initial result is a traditional kNN algorithm inside the target partition. For objects close to the borders, the obtained answer can not be considered final since potential neighbors from other partitions might be closer than the k^{th} one in the current partition. So, a verification step is necessary.

Therefore, in our approach, we consider the distance to the k^{th} neighbor of p from the initial answer as a search radius sr (it is determined by the farthest point from p). Then, we draw a cone centered at p with the calculated radius sr.

If all the cells in the cone overlap the target partition, then the algorithm returns the initial answer as final (Figure 6b). If not, we execute a cone search query using the calculated radius and we take the top-k records.

Figure 6 shows an example of a kNN query with $k = 6$. The first case represents the initial result where the search circle overlaps only one partition. In Figure 6b, the search circle intersects more than one partition, so a cone search query is run, and a refinement step is performed to select the k closest points to p.



(a) Case 1 (b) Case 2

Figure 6: kNN query (k=6).

The query execution plan is presented in Figure 7. It shows how this algorithm starts loading into memory only the target partition using the Parquet *partition filters* and uses the *takeOrdered* API to select the top-k records based on their distance to p.

6 Experiments

This section provides an evaluation study of ASTROIDE performance. We compare the effectiveness and the efficiency with the closest prototype in the state-of-the-art, i.e. SIMBA [6] and Spark SQL. Experiments were performed over a distributed system composed of 6 nodes having 80 cores with one Gigabit network. The main memory reserved for Spark is 156 GB in total.

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 [1] [35] 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 [36], 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.

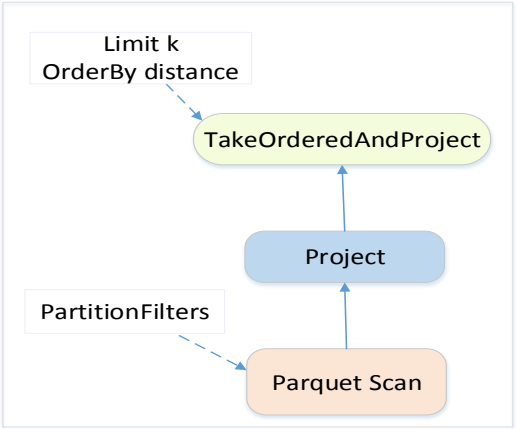
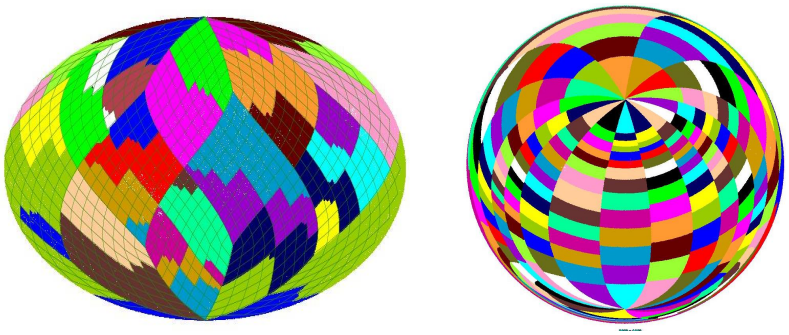


Figure 7: kNN execution plan.



(a) Partitioning in ASTROIDE.

(b) Partitioning in SIMBA.

Figure 8: Partitions visualization with Aladin.

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 2: Main characteristics of the datasets.

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.

6.1 Partitioning

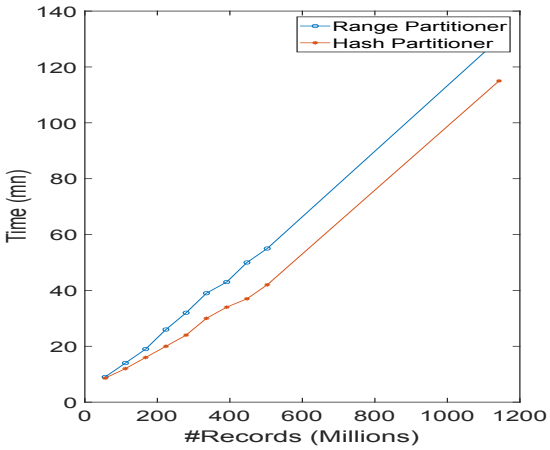


Figure 9: Effect of data size on partitioning (GAIA DR1).

Figure 9 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 4.1. 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 NSIDE = 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 millions of records.

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. 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.

We visualized the created partitions using Aladin[3], a tool for viewing astronomical data and acquiring sky maps. As shown in Figure 8a, each partition is represented by a color, a partition corresponds to a region of the sky and all created partitions cover all the sky. For partitioning, SIMBA uses an STR-Tree based partitioning [37] as illustrated in Figure 8b. 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 partition access.

6.2 Cone Search Query

Figure 10 presents how the data size affects the performance of cone search queries on GAIA DR1 datasets. 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 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. Figure 11 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.

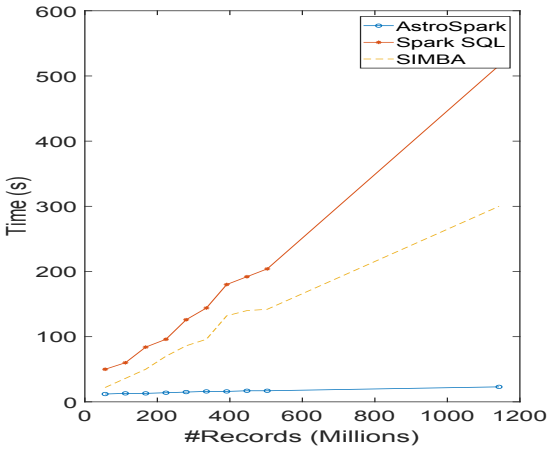


Figure 10: Effect of data size on cone search (GAIA DR1).

6.3 Cross-Matching Query

The default search radius is set to 2 arc-seconds, a value recommended by an astronomer.

We started by comparing the performance of the cross-matching using partitioned files with different HEALPix resolution (Figure 12). 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 12 shows how the cross-matching time is influenced by the HEALPix resolution. As the HEALPix order increases, it is interesting that the performance of cross-matching becomes almost constant. Besides, for value greater than 16, some records are lost in the output result, the reason is that the HEALPix cell area becomes bigger than the area of the search circle defined by ϵ. In our previous work, we explained the cross-matching algorithm [38] using the HEALPix order value 8. In this paper, 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 in our case, it ensures the fastest query as well as the correctness of the result.

Next, we evaluated the performance of ASTROIDE, SIMBA and Spark SQL for executing cross-matching queries. As shown in Figure 13, ASTROIDE is scalable and efficient. The performance shows a linear trend. Our approach shows

[3] ALADIN. http://aladin.u-strasbg.fr/

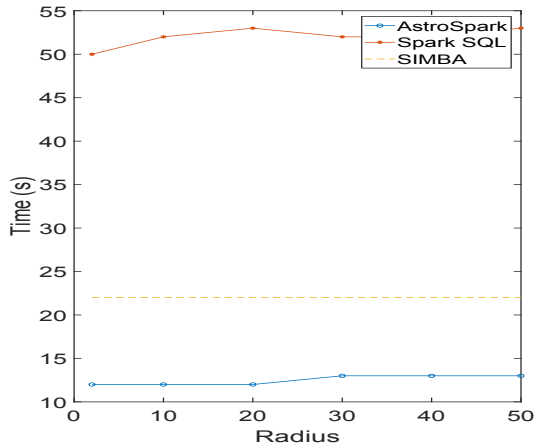


Figure 11: Effect of radius on cone search (GAIA DR1).

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 50 million is about 3000 objects.

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. 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.

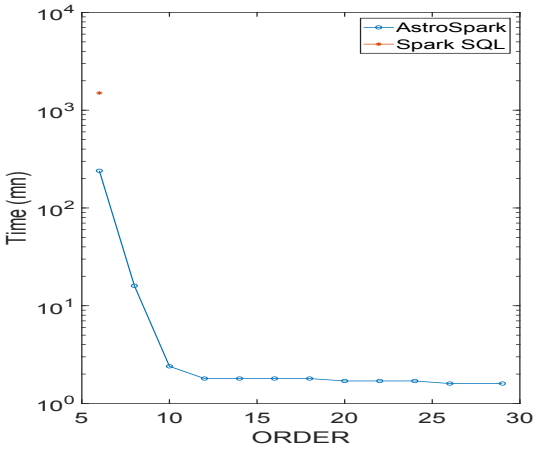


Figure 12: Effect of HEALPix resolution on HX-Match
(GAIA DR1/Tycho-2).

We have also studied the performance of the cross-matching algorithm as the search radius increases. Figure 14 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. We compare 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.

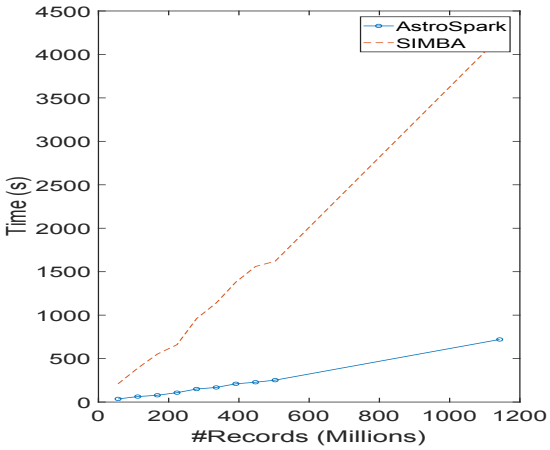


Figure 13: Effect of data size on Cross-matching
(GAIA DR1/Tycho-2).

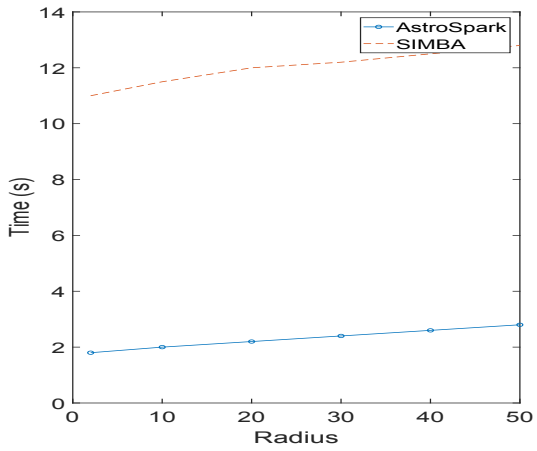


Figure 14: Effect of radius on Cross-matching
(GAIA DR1/Tycho-2).

Figure 15 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 15) 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 on 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 simple selection in 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 $dist \leq \epsilon$ 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:

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

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 arc-seconds. We first used HX-MATCH to build the intermediary dataset M. This tooks 3

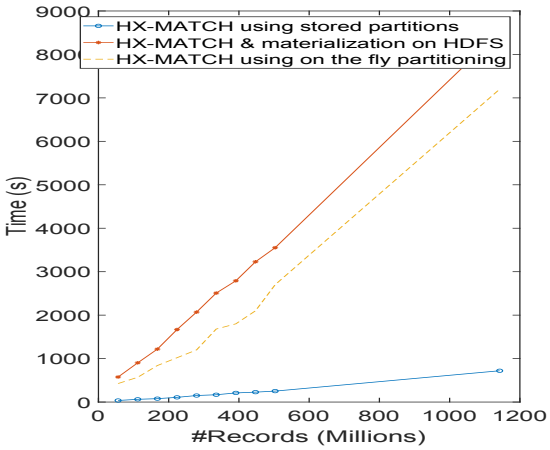


Figure 15: Effect of use of materialized partitions (GAIA DR1/Tycho-2) on HX-MATCH.

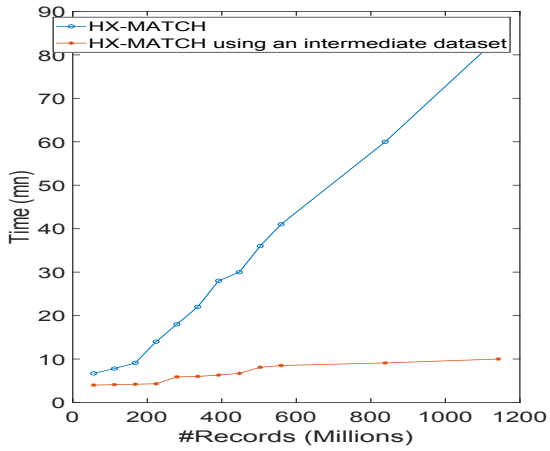


Figure 16: Effect of use of intermediate dataset (GAIA DR1/IGSL).

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 16 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 3:

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

Table 3: Average number of matching pairs.

6.4 kNN Query

Figure 17 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 17, we

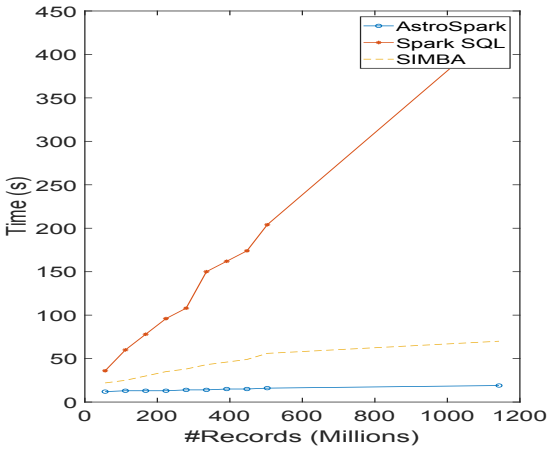


Figure 17: Effect of data size on kNN queries (GAIA DR1).

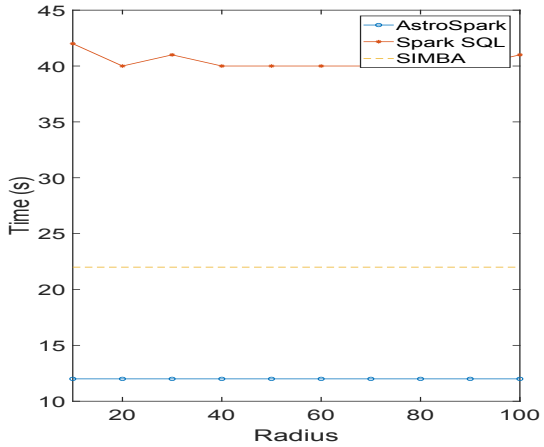


Figure 18: Effect of k on kNN queries (GAIA DR1).

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 less 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 millions of objects. Figure 18 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.

6.5 Queries involving 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 attribute queries. We describe the behavior of our framework in the management of several scenarios. We consider three scenarios:

- *Scenario 1.* Cone search with a filter on a certain magnitude range.
- *Scenario 2.* kNN with a filter on a certain magnitude range.
- *Scenario 3.* Cross-matching with a filter on a certain magnitude range.

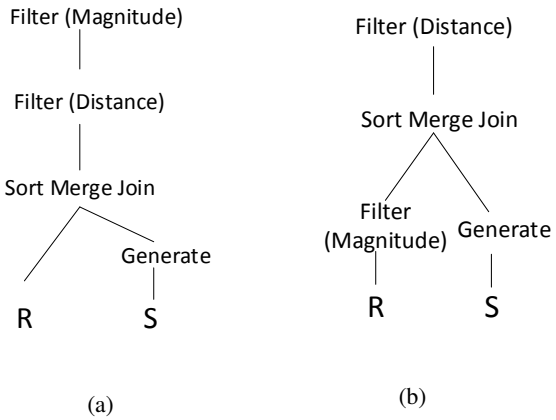


Figure 19: Transformations of cross-matching.

6.5.1 Scenario 1 & Scenario 2

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:

- Determines the selectivity factor for each predicate in the selection clause.
- Determine the cost of the filter operation.
- Determine the cost of the astronomical query (cone search or kNN).
- Compare the costs.
- Start by executing the cheapest one.

In other terms, if executing a kNN query reduces the input size more than the attribute filter predicate, then ASTROIDE will execute the kNN query before the filter.

For instance, an astronomical query as below is used to express a 20 nearest neighbor query for a source defined by position `(164.835, -54.618)` from a table `gaia` with a filter on `magnitude`.

```
SELECT TOP 20 *,
  DISTANCE(POINT('ICRS', ra, dec),
    POINT('ICRS', 164.835,-54.618)) as dist
FROM gaia
WHERE magnitude < 18
ORDER BY dist
```

The number of records returned by the filter on magnitude is about 300 million records on table `gaia`, the number of records returned by the kNN search is 20 records. So, the predicate that filters out the most rows is the astronomical predicate on neighbors, it should be applied as soon as possible. As a general rule, the most-filtering astronomical predicates should be applied before the least-filtering astronomical predicates. The goal is to minimize CPU costs and to choose the best plan from all the plans examined.

6.5.2 Scenario 3

In such a case, the traditional execution (Figure 19a) calculates the matching result between the two tables and then computes a filter on magnitude. In the alternative scheme of Figure 19b, 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. These transformations improves significantly the query execution time because Catalyst takes care of finding the most efficient way to run the query.

7 Conclusion

In this paper, we presented ASTROIDE, a solution that combines the scalability and cost-effectiveness of distributed data processing engines with Spark and the expressiveness of astronomical data servers with ADQL. ASTROIDE achieves this objective through efficient partitioning using HEALPix, customized strategies in the Catalyst optimizer and an expressive query interface. ASTROIDE extends Spark to support main astronomical queries. Experiments on real datasets from the on-going 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 an adapted HEALPix partitioning algorithm.

We plan improve our query optimizer by adding optimization rules and strategies to execute queries. We will also integrate a cost based optimization module in ASTROIDE to select the best query execution plan. This feature has been recently proposed in Spark [39], and an accurate cost model for the spatial operators in the query plan would be useful for complex queries. We also intend to test further the scalability by increasing both the data size and the cluster size. A long term perspective is to extend ASTROIDE to deal with advanced analytics, such as moving objects and pattern detection, or object classification. Most data analytics algorithms leverage the pairwise distance computation and filtering, which can reuse the solutions proposed in this paper.

Acknowledgments

The present paper has been partially supported by CNES (Centre National d'Etudes Spatiales) and by the MASTER project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie-Slodowska Curie grant agreement N. 777695.
We would like to thank Veronique Valette from the CNES and Frederic Arenou at Paris Observatory for their cooperation. We are also thankful to François-Xavier Pineau and colleagues from the CDS (Centre de Données astronomiques de Strasbourg) for their help regarding the ADQL parser.
This work has made use of data from the European Space Agency (ESA) mission GAIA (`https://www.cosmos.esa.int/gaia`), processed by the GAIA Data Processing and Analysis Consortium (DPAC, `https://www.cosmos.esa.int/web/gaia/dpac/consortium`). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the GAIA Multilateral Agreement.

References

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

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

[3] Alexander S Szalay, Jim Gray, Peter Kunszt, Anirudha Thakar, and Don Slutz. Large Databases in Astronomy. In *Mining the Sky*, pages 99–116. Springer, 2001.

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

[5] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1352–1363. IEEE, 2015.

[6] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1071–1085. ACM, 2016.

[7] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. *MD*-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, 31(2):289–319, 2013.

[8] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 70. ACM, 2015.

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

[10] Krzysztof M Gorski, Eric Hivon, AJ Banday, Benjamin D Wandelt, Frode K Hansen, Mstvos Reinecke, and Matthia Bartelmann. HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

[11] Robert W Youngren and Mikel D Petty. A multi-resolution HEALPix data structure for spherically mapped point data. *Heliyon*, 3(6):e00332, 2017.

[12] P Fernique, MG Allen, T Boch, A Oberto, FX 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*, 578:A114, 2015.

[13] Peter Z Kunszt, Alexander S Szalay, and Aniruddha R Thakar. The hierarchical triangular mesh. In *Mining the sky*, pages 631–637. Springer, 2001.

[14] William O'Mullane, AJ Banday, KM Gorski, Peter Kunszt, and AS Szalay. Splitting the sky-htm and healpix. In *Mining the Sky*, pages 638–648. Springer, 2000.

[15] Jovin J Mwemezi and Youfang Huang. Optimal facility location on spherical surfaces: algorithm and application. *New York Science Journal*, 4(7):21–28, 2011.

[16] Raymond Plante, Roy Williams, Robert Hanisch, and Alex Szalay. Simple cone search version 1.03. *IVOA Recommendation 22 February 2008*, 2008.

[17] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[18] François Ochsenbein, Patricia Bauer, and James Marcout. The VizieR database of astronomical catalogues. *Astronomy and Astrophysics Supplement Series*, 143(1):23–32, 2000.

[19] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 570–581. ACM, 2002.

[20] Jim Gray, María A. Nieto-Santisteban, and Alexander S. Szalay. The zones algorithm for finding points-near-a-point or cross-matching spatial datasets. *CoRR*, abs/cs/0701171, 2006.

[21] Qing Zhao, Jizhou Sun, Ce Yu, Chenzhou Cui, Liqiang Lv, and Jian Xiao. A paralleled large-scale astronomical cross-matching function. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 604–614. Springer, 2009.

[22] 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*, volume 351, page 735, 2006.

[23] Tamás Budavári, AS 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*, volume 314, page 177, 2004.

[24] F-X Pineau, Thomas Boch, and Sebastien Derriere. Efficient and scalable cross-matching of (very) large catalogs. In *Astronomical Data Analysis Software and Systems XX*, volume 442, page 85, 2011.

[25] Milena Ivanova, Niels Nes, Romulo 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*, pages 13–13. IEEE, 2007.

[26] Bart Scheers, Steven Bloemen, Hannes Mühleisen, Pim Schellart, Arjen van Elteren, Martin Kersten, and Paul J Groot. Fast in-database cross-matching of high-cadence, high-density source lists with an up-to-date sky model. *Astronomy and computing*, 23:27–39, 2018.

[27] Jacob VanderPlas, Emad Soroush, K Simon Krughoff, Magdalena Balazinska, and Andrew Connolly. Squeezing a Big Orange into Little Boxes: The AscotDB System for Parallel Processing of Data on a Sphere. *IEEE Data Eng. Bull.*, 36(4):11–20, 2013.

[28] Amin Mesmoudi, Mohand-Saïd Hacid, and Farouk Toumani. Benchmarking SQL on MapReduce systems using large astronomy databases. *Distributed and Parallel Databases*, 34(3):347–378, 2016.

[29] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

[30] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, 1987.

[31] Ahmed Eldawy and Mohamed F. Mokbel. The Era of Big Spatial Data. *Proc. VLDB Endow.*, 10(12):1992–1995, 2017.

[32] Mariem Brahem, Stephane Lopes, Laurent Yeh, and Karine Zeitouni. AstroSpark: towards a distributed data server for big data in astronomy. In *Proceedings of the 3rd ACM SIGSPATIAL PhD Symposium*, page 3. ACM, 2016.

[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[34] Grégory Mantelet. ADQL library. `https://github.com/gmantele/taplib`.

[35] Anthony GA Brown, A Vallenari, T Prusti, JHJ De Bruijne, F Mignard, R Drimmel, C Babusiaux, CAL Bailer-Jones, U Bastian, M Biermann, et al. Gaia Data Release 1-Summary of the astrometric, photometric, and survey properties. *Astronomy & Astrophysics*, 595:A2, 2016.

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

[37] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Data Engineering, 1997. Proceedings. 13th international conference on*, pages 497–506. IEEE, 1997.

[38] Mariem Brahem, Karine Zeitouni, and Laurent Yeh. HX-MATCH: In-Memory Cross-Matching Algorithm for Astronomical Big Data. In *International Symposium on Spatial and Temporal Databases*, pages 411–415. Springer, 2017.

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