EFFICIENT ASTRONOMICAL QUERY PROCESSING USING 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

Sky surveys represent a fundamental data source in astronomy. Today, these surveys are moving into a petascale regime produced by modern telescopes. Due to the exponential growth of astronomical data, there is a pressing need to provide efficient astronomical query processing. Our goal is to bridge the gap between existing distributed systems and high-level languages for astronomers. In this paper, we present efficient techniques for query processing of astronomical data using ASTROIDE. Our framework helps astronomers to take advantage of the richness of the astronomical data. The proposed model supports complex astronomical operators expressed using ADQL (Astronomical Data Query Language), an extension of SQL commonly used by astronomers. ASTROIDE proposes spatial indexing and partitioning techniques to better filter the data access. It also implements a query optimizer that injects spatial-aware optimization rules and strategies. Experimental evaluation based on real datasets demonstrates that the present framework is scalable and efficient.

***K*eywords** Astronomical Survey Data Management · Big Data · Query Processing · Spark Framework

1 Introduction

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. The Sloan Digital Sky Survey (SDSS), Large Synoptic Survey Telescope (LSST) and ESA's GAIA mission [1] have been the largest and most accurate three-dimensional maps of the Galaxy ever obtained in the history of astronomy. Traditional relational database systems are no longer adequate, not only because of data explosion, but also because of computational complexity of astronomical queries. Recently, the shared-nothing 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 [2] 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.

Recently, various Spark based systems have been addressed by other disciplines such as geo-spatial data [3] [4]. But, challenges related to astronomical data are unique. Indeed, these systems suffer from several limitations with respect to astronomical data needs. The main issue is the lack of an expressive query language adapted to the astronomical context. In addition, the proposed built-in functions are not adapted to the spherical coordinates system, which leads

to erroneous query results, as shown by our experiments in Section 6. We have also demonstrated that their query performances remain limited due to their data organization scheme which is less suitable for astronomical data. Hence, new techniques and tools are necessary for processing astronomical data. However, 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.

In this work, 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 surveys and catalogs. Astronomical data use spherical coordinates (namely, Right Ascension and Declination) to describe positions of stars in the sky. In particular, common queries use spherical distance as a spatial filtering step. For example, cross-matching queries involve a spatial join according to a spherical distance, which can be a very expensive and complex operation to process. Spatial queries are typically optimized by using spatial indices in a cartesian space. But these indices do not cope well with the spherical coordinates.

Our design takes full advantage of the extensibility features of Spark, 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 language level, the query language should be extended to enable astronomical functions. We also extend the Dataframe API with the same functions at the programing level. At the optimizer level, the parsing, the logical and the physical also need to be customized or extended.

In ASTROIDE, queries are expressed in Astronomical Data Query Language (ADQL) [5], an SQL extension 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 [6], to achieve high performance query execution. At the intermediate level, the query language should be parsed and optimized seamlessly. 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. 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 data.

In a nutshell, our system includes the following extensions over Spark:

- **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 queries processing time.
- **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 most common astronomical query language ADQL.
- **Astronomical query optimization.** We extend the Catalyst optimizer and exploit partition tuning for astronomical operators. We introduce new physical and logical optimizations to optimize query execution plans using transformation rules and strategies.

Besides these contributions, we fully implemented ASTROIDE[1]. Our experiments on real data show its efficiency and its scalability.

The rest of this paper is organized as follows. We first discuss related work in Section 2 and explain necessary background in Section 3. An architectural overview of ASTROIDE is presented in Section 4. Our query processing module is detailed in Section 5 which is followed by performance study in Section 6 and conclusion.

2 Related Work

Early astronomical servers. Traditional DBMS technologies have been widely used for the management of astronomical data. For example, the VizieR service [7] developed by the Centre de Données de Strasbourg (CDS) is available to provide a tool for accessing astronomical data listed in published catalogs and executing cross-matching

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

queries. Q3C [8], standing for Quad Tree Cube, provides a sky partitioning schema for PostreSQL and offers an SQL interface for the main astronomical queries. However, the performance of these systems are limited because they are mainly based on centralized architecture style.

AscotDB [9] builds on the combination of several pieces of technology: the SciDB [10], 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.

Besides, some recent works [11] [12] propose customized solutions to execute some astronomical queries in a parallel architecture. In [11], the authors introduce a cross-matching function including a partitioning approach with HEALPix. OPEN SKYQUERY [12] allows querying and cross-matching distributed astronomical datasets. The authors propose zoning and partitioning algorithms for parallel query execution using RDBMS technologies. However, none of these efforts provide a scalable astronomical server with a full and unified programming interface like ADQL, as we target.

Approaches based on distributed frameworks. In [13], the 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 [14] is an extension to 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 and R^+-tree, 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 [15] 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 [4] 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 [3] have 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.

Eldawy et al. [16] 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.

For kNN join approaches, an algorithm based on grid partitioning has been proposed in [17]. It sorts data according to the G-ordering and performs a nested loop join while reducing distance computation. An approximate kNN join has been introduced in [18], which uses z-values to map the multi-dimensional space into one dimension, and transforms the kNN join into a sequence of kNN search.

The aforementioned systems are designed for the geo-spatial context that differs from the astronomical context in its data types and operations. Spatial data types (polygons, rectangles ...) are different from our context. Also, 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 coordinates system, and do not support specific astronomical operators such as cone search queries, cross-match queries that are supported by our system. They use conventional spatial indexing techniques, while we adopt specialized indexing method for astronomical data by using HEALPix.

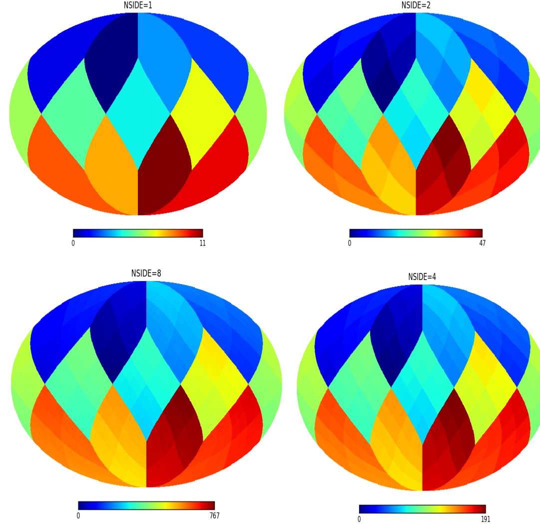


Figure 1: HEALPix partition with NSIDE = 1, 2, 4, 8

3 Background

3.1 HEALPix

The Hierarchical Equal Area isoLatitude Pixelization (HEALPix) [6] 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 3.1)

We have adopted the HEALPix 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 [19] 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) [20]. Comparison between the two indexing techniques is detailed in [21]. 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 that HEALPix partitions the sphere in equal-area cells whereas HTM partitions the sphere in regions with different 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 ϵ as radius
ϵ	Threshold of spherical distance
$SD(r,s)$	Spherical distance between r and s
$\Delta\delta$	Difference in declination between r and s
δ_1 (resp. δ_2)	Declination at position r (resp. s)
$\Delta\alpha$	Difference in right ascension between r and s

Table 1: Notations

3.2 Astronomical Queries

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.

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. For a sake of consistency, we introduce the notations listed in Table 3.2.

- Spherical distance. $SD(r, s)$ defines the spherical distance between r and s, which reflects their shortest distance over the celestial sphere. It is calculated with the Harvesine formula as the length of the great circle arc between two points on a sphere. Formally:

  $SD(r, s) = 2 \arcsin \sqrt{\sin^2 \Delta\delta/2 + \cos\delta_1 \cos\delta_1 \sin^2 \Delta\alpha/2}$

- 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 cone `c` defined by a sky position p and a radius ϵ around that position, a cone search query returns all pairs of points $r \in R$ within c. Formally:

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

- 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.

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

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

- 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.

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

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.*

Astronomical queries are commonly expressed using the Astronomical Data Query Language (ADQL) [5]. 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, CENTROID, CIRCLE, CONTAINS, etc. For example, CIRCLE expresses a circular region in the sky, which corresponds to a cone in space. The ADQL expression of cone search is illustrated in the following example (given that ra an dec are the spherical coordinates in the gaia catalog, it returns all the stars in the cone centered in the point (266, -29) having a radius 0.0833 in the ICRS coordinate system:

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

3.3 Spark

Apache Spark [2] 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 across processing nodes.

Spark SQL, 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 [2], which makes adding new optimization techniques easy. Catalyst can be extended in two ways: by defining custom rules, and by adding optimization strategies.

Catalyst 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) has been introduced in this optimizer. This module analyzes all execution plans, assigns a cost to each plan and chooses the lowest cost for the physical execution.

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

- A superior performance compared to other distributed systems
- Its optimizer is powerful and easily extensible.

This has encouraged its use as a base framework in many projects [4][3].

4 Overview of ASTROIDE

The primary goal of ASTROIDE [22] 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 advantage of Spark features.

Figure 2 shows the architectural components of ASTROIDE. Our framework is composed of two modules: data partitioning and querying system. The data partitioning module is responsible for managing the partitions in a way to ensure data locality, load balancing and task parallelization. Our partitioning algorithm runs as follows. First, we apply a linearization technique to map the two dimensional spherical coordinates representing positions of stars on the sky into a single dimensional ID using HEALPix scheme. As explained in Section 3.1, HEALPix indexing is adapted to astronomical data and fulfills the data locality requirement, i.e., close stars in the sky are likely to have close HEALPix IDs. Next, we leverage a range partitioner, which yields data partitions with roughly equal sizes. This stores items sorted by HEALPix index and organizes data by cells intervals. Then, we store partitioned files on HDFS to amortize the construction cost 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. In the final step, we determine partition boundaries and stores them as metadata.

On the right of the Figure 2 is the querying system. It includes the query interface for processing astronomical data. The query parser module translates ADQL queries into an internal algebraic representation described by an operator

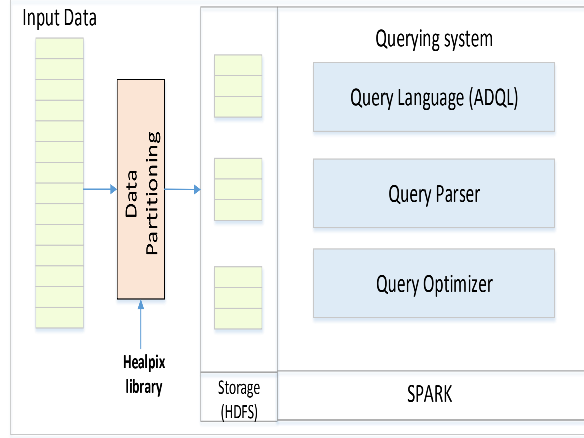


Figure 2: ASTROIDE architecture

tree. After that, the query optimizer optimizes this tree while integrating specific logical and physical optimization techniques. Then Spark engine will execute the optimized plan. The next section explains in more details this module.

5 Query processing techniques

In this section, we provide an abstraction of query processing in ASTROIDE. Query processing in our context is the transformation of the query from ADQL dialect into an efficient execution plan that performs the required manipulations within Spark. When ASTROIDE receives a new query, it goes through a series of steps as shown in the right part of the architecture 2. In the first phase, the system parses the query, verifies the ADQL syntax rules and performs appropriate query rewriting by translating geometrical functions to equivalent internal UDFs. Then, it matches the translated SQL query to a relational algebraic expression. In the second phase, the query optimizer takes this algebraic representation 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.

5.1 Query Parser

The query parser allows the extension of Spark to deal with astronomical queries and ADQL. It takes an ADQL query from the user, parses it into ADQL objects and makes sure that the ADQL grammar rules are correct. At this step, the query parser uses the ADQL library to check the grammar of ADQL. If an error is found, the submitted query is rejected. Otherwise, it extracts tables names, columns name and some keywords such as CONTAINS, JOIN, POINT, CIRCLE from the input query. This extraction helps ASTROIDE to fetch the query type. After that, we apply some transformations to the original query by replacing all parts of the query matching an astronomical pattern with replacement SQL text.

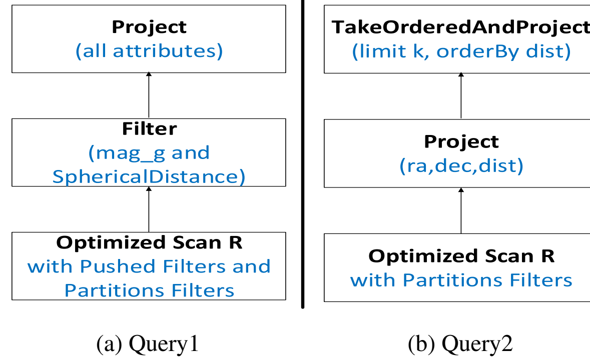
For example, an ADQL query used to express a cone search query as follows:

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

is transformed into a Spark SQL-compliant query by replacing the CONTAINS function with a UDF expressing a spherical distance as follows:

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

At the end of the query parsing, the SQL query is mapped to a query tree that corresponds to a relational algebra expression. The transformation performed by our query parser depends only on the type of the query and do not take into account the cost of query plans.



(a) Query1 (b) Query2

Figure 3: Plan after optimizations

5.2 Query optimizer

The query optimizer is an important component since it is the responsible for generating a query execution plan that computes the query result efficiently. This module is an extension of the Spark Catalyst optimizer. The decision to exploit Catalyst [2] a backbone for our optimizer was driven by the fact that astronomical databases contain both astronomical and non-astronomical queries, as opposed to developing a new optimizer.

Our optimizer still leverages all the benefits of Spark's optimizer like predicate pushdown, projection pruning, join reordering, ... We distinguish two types of operators: relational operators and astronomical operators. Here, we integrate optimizations for astronomical operators and let Catalyst performs relational operators. In the remainder of the paper, we present the techniques of extending Catalyst to support astronomical queries.

The input query is supposed to be an operator tree after query parsing and rewriting from an ADQL expression.

This query tree is represented by a logical plan, we distinguish three types of logical plans: parsed logical plan, analyzed logical plan, optimized logical plan. The parsed logical plan goes through a series of analyzer rules to produce the analyzed logical plan. This analyzed plan is converted in turn to an optimized logical plan using optimization rules. This optimized plan is transformed to a physical plan using strategies.

To optimize astronomical queries, we plug in logical plans with a set of optimization rules, analyzer rules or strategies to produce the execution plans.
We demonstrate these rules and strategies by giving examples. Query examples can be divided in three categories: pure spatial queries; queries on non-spatial properties; hybrid queries which combine spatial and non-spatial properties. In the following sub-sections, some examples of the first two categories are explained.

5.3 Source Selection Queries

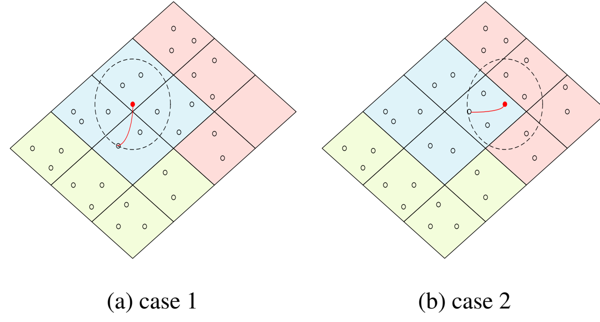
5.3.1 Query 1

Selection of sources within a certain angular distance from the specified center position **(Cone Search)**, fitting as well limitations on a certain magnitude range [10,18].

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

A plan without custom optimizations requires traversing all the data in R and collect the objects that lie inside the specified circle with the specified magnitude range. It can be represented by the following operations: `Project(Filter(Filter(Scan (R), magnitude ), cone_condition), all attributes)`

However, 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 the query cone can be filtered out. In the refinement step, exact candidates objects can be returned with accurate geometry test. Precisely, given a query cone, our optimizer first identifies the pixels without accessing the data, then uses the indexing and partitioning scheme to retrieve the actual data



(a) case 1 (b) case 2

Figure 4: kNN query

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 to the data source rather than dealing with the entire partition(s).
- Select exact objects inside the cone according to the harvesine distance.

This optimizations consists in using both pushed filters and partition filters to drastically prune the search space, as shown in Figure 3a.

5.3.2 Query2

Find the 6 closest sources to a star represented by a position p=(44.97, 0.09) in gaia catalog **(kNN)**.

```
SELECT TOP 6 ra,dec, DISTANCE(Point('ICRS', ra, dec),
        Point('ICRS', 44.97, 0.09)) AS dist
FROM gaia ORDER BY dist;
```

The baseline approach to execute this query is to scan all objects in R, to calculate their distances to p, to order them by distance, and finally take the top-6 objects.

Similar to spatial selection queries, the filtering phase for relevant partitions can be also applied to avoid scanning the entire dataset. But, unlike the cone search, the identification of the right HEALPIX indices and the concerned partition are far from being obvious.

Therefore, 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), then the algorithm returns the initial answer as final. If not (case 2), we execute a cone search query using the calculated radius and we take the top-k records. Figures 4a, 4b illustrate the two cases of kNN query with k = 6.

The physical plan of a kNN query is represented in Figure 3b. It starts by applying the partition filter early. By the time the candidates objects are fetched by the optimizer, it uses the *takeOrderedAndProject* API to select the top-k records based on their distance to p.

5.4 Join Queries

5.4.1 Query3

Cross-matching two catalogs `gaia` and `igsl`, with a radius of 2 arc-seconds **(cross-matching query)**.

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

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.

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.
Notice that for the candidates belonging to neighbor cells, a naive approach would be to perform a θ-join where the criteria is the membership to the neighbors of each object cell. But θ-join does not benefit from the optimization techniques, unlike equi-join.

The second step computes the distance and removes the false positives to get the final result.

Algorithm 1 : HX-MATCH(R,S)

Require: Input datasets R and S, search radius ϵ
Ensure: Result of HX-MATCH
1: $S^+ \leftarrow \emptyset$
2: **for** t $\in S$ **do**
3:     **for** N $\in$ IpixNeighbour(t.ipix) **do**
4:         $S^+ = S^+ \cup \{ t \oplus ipix = n \mid \forall n \in N\}$
5:     **end for**
6: **end for**
7: c = Join(R, S^+, $R.ipix$= $S^+.ipix$)
8: **return** Filter(c, $sphericalDistance(c.R, c.S^+) \leq \epsilon$)

The pseudo-code of cross-matching (HX-MATCH) [23] is explained in Algorithm 1. It takes as input two partitioned files 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 (Line 4), where we substitute the HEALPix value (here ipix) by the current cell and the one of each neighbor cell (Line 4). Here, `IpixNeighbour` 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)

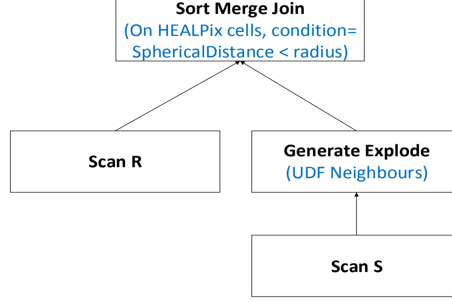


Figure 5: Query3 plan after optimizations

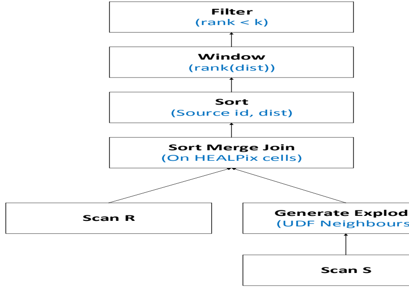


Figure 6: Query4 physical plan after optimizations

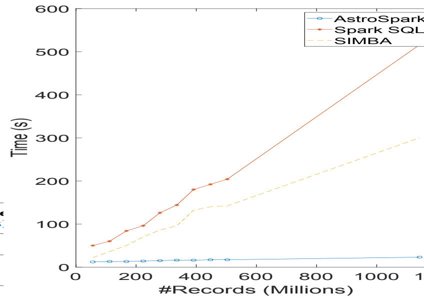


Figure 7: Effect of data size on cone search

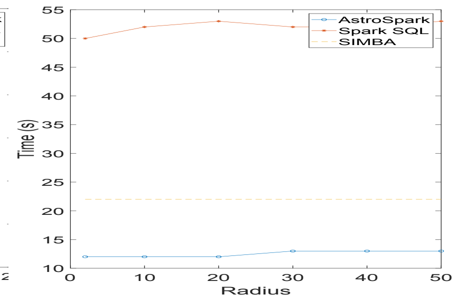


Figure 8: Effect of radius on cone search

5.4.2 Query4

Associate each star in `gaia` catalog with its k nearest neighbors stars in `igsl` catalog (**kNN join query**):

```
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)) );
```

Efficient process of a kNN join query 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, because it involves correlated subquery that uses attributes from the outer query (R.ra, R.dec), which is not allowed in the current version of Spark SQL. However, catalyst provides a way to cope with this limitation.

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 $r \in R_i$. However, this is not efficient because it ignores the shared neighbors in S among close objects of R, which 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 the need for a specific approach to support this query.

Algorithm 2 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. Second, we associate each object in R with the calculated neighboring cells and clone each object 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.

Algorithm 2 : 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: $PrevNeighbours = \{t.ipix\}$
5: **while** $nb < k$ **do**
6: $V = IpixNeighbour(PrevNeighbours) - Res$
7: $nb = nb + count(S, ipix), \forall ipix \in V$
8: $PrevNeighbours = V$
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** = $L \cup \{g1, .., g_k | \forall g_1, .., g_n \in Group(XM, XM.R.id)\}$

To make Catalyst supporting such query, we inject 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. The idea is to generate a completely new physical plan (Figure 6 for the query with the following operations:

- Extend each cell of R with its neighboring cells containing possible candidates and replicate each object 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. Partitioning logically partitions tuples into groups of source id, ordering defines how the tuples of c table are logically ordered by distance during window function evaluation.
- Add a filter operator to select only required k records in S.

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 applied by our customized rules and automatically skips over subtrees that do not match.

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 [3] 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 many other non spatial attributes including magnitude, metalicity, and so on.

- *GAIA*. The public GAIA DR1 dataset [1], which describes more than one billion sources and represents the most detailed all-sky map in the optical to date.
- *IGSL*. Short of the Initial Gaia Source List [24]. It is a compilation catalog produced for the purpose of comparison with the observations collected in GAIA mission.
- *TYCHO-2*. An astrometric reference catalog related to prior surveys containing positions and proper motions of more than two millions brightest stars in the sky.

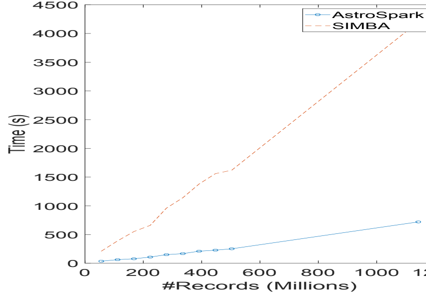


Figure 9: Effect of data size on cross-matching

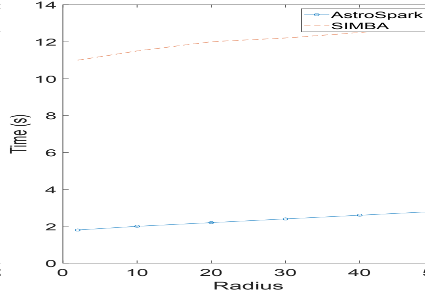


Figure 10: Effect of radius on cross-matching

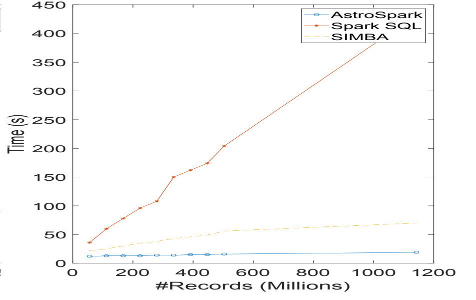


Figure 11: Effect of data size on kNN queries

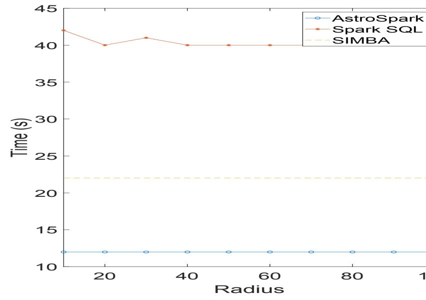


Figure 12: Effect of k on kNN queries

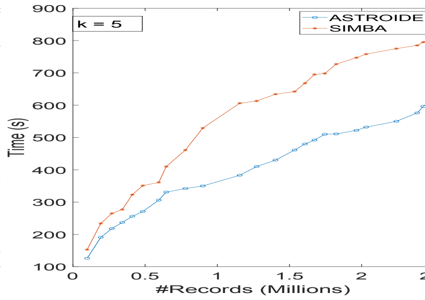


Figure 13: Effect of data size on kNN Join

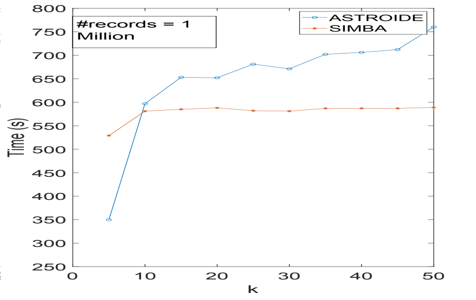


Figure 14: Effect of k on kNN Join

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

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
TYCHO2	2,539,893	501.4 M	32

In all the 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 Cone Search Query

Figure 7 relates how the data size affects the performance of cone search queries on GAIA DR1 datasets. Predictably, the query execution time increases when the data size increases, due to the access cost to the partitions and the communication costs when the query result spread over several partitions. We observe that ASTROIDE is much more efficient than SIMBA, which in turn, outperforms SPARK SQL. Indeed, ASTROIDE requires less access to partitions than SIMBA but Spark SQL has to scan all objects in the dataset. Figure 8 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 two orders of magnitude faster than SIMBA and five orders of magnitude faster than Spark SQL.

6.2 Cross-Matching Query

We evaluated the performance of ASTROIDE, SIMBA and Spark SQL regarding cross-matching queriy performance. As shown in Figure 9, ASTORIDE 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. 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 catalog of 50 million stars 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. We have also studied the performance of the cross-matching algorithm by varying the search radius. Figure 10 shows that ASTROIDE is six orders of magnitude faster than SIMBA and the performance gap remains constant with bigger radius.

6.3 kNN Query

Figure 11 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 11, 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 two orders of magnitude 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 k parameter, by varying k from 10 to 100 (we fixed the data size to about 50 millions objects). Figure 12 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. Besides, ASTROIDE maintains two orders of magnitude better performance than SIMBA.

6.4 kNN Join

The kNN join query is impossible to execute with Spark SQL. That is 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 result are presented in figure 13. We can observe that ASTROIDE outperforms SIMBA for k = 5. ASTROIDE shows a very scale up feature: it achieves a linear speed-up with the increase of the data size. The result of varying k on kNN join are illustrated in figure 14. As we can see, when k increases, the performance of SIMBA remains nearly the same. This is because SIMBA partitions the dataset 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. It 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.

7 Conclusion

In this paper, we presented a query processing module that improves the execution of astronomical queries using ASTROIDE, an astronomical in-memory processing framework. ASTROIDE achieves scalability through astronomical indexing and partitioning, dynamic query rewriting and efficient optimizer based on customized rules and strategies. Experiments have shown that our querying system is comparable to an existing spatial in-memory distributed framework. ASTROIDE is a workable solution intended for use by astronomers or as a backed for other services as virtual observatories or graphical tools.

In future work, we will complement our approach with a deep cost analysis, and we will investigate the integration of a cost model into the optimizer. Another interesting perspective is to couple our system with a test and query profiling tool, which would facilitate the experiments, but also give insight into the optimization process.

8 ACKNOWLEDGMENT

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 for her 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 ESA mission *Gaia*.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] SciDB.

[11] 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.

[12] María A Nieto-Santisteban, Aniruddha R Thakar, and Alexander S Szalay. Cross-matching very large datasets. In *National Science and Technology Council (NSTC) NASA Conference*, 2007.

[13] 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.

[14] 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.

[15] 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.

[16] Ahmed Eldawy and Mohamed F. Mokbel. The era of big spatial data. *Proc. VLDB Endow.*, 10(12):1992–1995, 2017.

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

[18] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

[19] HEALPix Softaware. http://healpix.sourceforge.net/.

[20] Alexander S Szalay, Jim Gray, George Fekete, Peter Z Kunszt, Peter Kukol, and Ani Thakar. Indexing the sphere with the hierarchical triangular mesh. *arXiv preprint cs/0701164*, 2007.

[21] 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.

[22] 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.

[23] 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.

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