The Astronomical Journal, 158:37 (14pp), 2019 July https://doi.org/10.3847/1538-3881/ab2384

AXS: A Framework for Fast Astronomical Data Processing Based on Apache Spark

Petar Zečević[1,2], Colin T. Slater[3], Mario Jurić[3], Andrew J. Connolly[3], Sven Lončarić[1], Eric C. Bellm[3], V. Zach Golkhou[3,4], and Krzysztof Suberlak[3]
[1] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia; petar.zecevic@fer.hr
[2] Visiting Fellow, DIRAC Institute, University of Washington, Seattle, USA
[3] DIRAC Institute and the Department of Astronomy, University of Washington, Seattle, USA
Received 2019 March 26; revised 2019 May 17; accepted 2019 May 20; published 2019 July 1

Abstract

We introduce AXS (Astronomy eXtensions for Spark), a scalable open-source astronomical data analysis framework built on Apache Spark, a widely used industry-standard engine for big-data processing. Building on capabilities present in Spark, AXS aims to enable querying and analyzing almost arbitrarily large astronomical catalogs using familiar Python/AstroPy concepts, DataFrame APIs, and SQL statements. We achieve this by (i) adding support to Spark for efficient on-line positional cross-matching and (ii) supplying a Python library supporting commonly used operations for astronomical data analysis. To support scalable cross-matching, we develop a variant of the ZONES algorithm capable of operating in distributed, shared-nothing architecture. We couple this to a data partitioning scheme that enables fast catalog cross-matching and handles the data skew often present in deep all-sky data sets. The cross-match and other often-used functionalities are exposed to the end users through an easy-to-use Python API. We demonstrate AXS's technical and scientific performance on Sloan Digital Sky Survey, Zwicky Transient Facility, *Gaia* DR2, and AllWise catalogs. Using AXS we were able to perform on-the-fly cross-match of *Gaia* DR2 (1.8 billion rows) and AllWise (900 million rows) data sets in $\sim$30 s. We discuss how cloud-ready distributed systems like AXS provide a natural way to enable comprehensive end-user analyses of large data sets such as the Large Synoptic Survey Telescope.

Key words: astronomical databases: miscellaneous – catalogs – methods: data analysis

1. Introduction

The amount of astronomical data available for analysis is growing at an ever increasing rate. For example, the 2MASS survey (in operation 1997–2003) delivered a catalog containing 470 million sources (about 40 GB). More recently, the latest release (DR14) of the Sloan Digital Sky Survey (SDSS; originally started in 2003) contains 1.2 billion objects (about 150 GB of data). *Gaia*, started in 2016, delivered nearly 1.8 billion objects (about 400 GB). Looking toward the future, the Large Synoptic Survey Telescope (LSST) is projected to acquire about 1000 observations of close to 20 billion objects in 10 yr of its operation (with catalog data set sizes projected to be north of 10 PB; LSST Science Collaboration et al. 2009).

Present-day survey data sets are typically delivered through a number of astronomical data archives (for example Strasbourg astronomical Data Centre, SDC;[5] Mikulski Archive for Space Telescopes, or MAST;[6] and NASA/IPAC Infrared Science Archive, or IRSA[7]), usually stored and managed in relational databases (RDBMS). The RDBMS storage allows for efficient extraction of database *subsets*, both horizontally (retrieving the catalog entries satisfying a certain condition) and occasionally vertically (retrieving a subset of columns).

Given their architecture and history, these systems are typically optimized for greatest performance with simple and highly selective queries. End-user analyses of archival data typically begin with subsetting from a larger catalog (i.e., SQL queries). The resulting smaller subset is downloaded to the researcher's local machine (or a small cluster). There it is analyzed, often using custom (usually Python) scripts or, more recently, code written in Jupyter notebooks (Pérez & Granger 2007). This *subset–download–analyze* workflow has been quite effective over the past decade, elevating surveys and archived data sets to some of the most impactful programs in astronomy.

The upcoming increase in data volumes and significant changes in the nature of scientific investigations pose challenges to the continuation of this model. The next decade is expected to be weighted toward exploratory studies examining whole data sets: performing large-scale classification (including using machine learning, ML, techniques), clustering analyses, searching for exceptional outliers, or measuring faint statistical signals. The change is driven by the needs of the big scientific questions of the day. For some, such as the nature of dark energy, utilization of all available data and accurate statistical treatment of measurements are the only way to make progress. For others, such as time series data, detailed insights into the variable and transient universe are now within reach from exploration of properties of *entire populations* of variable sources, rather than a small subset of individual objects. Second, the added value gained by *fusing information from multiple data sets*—typically via positional cross-matching—is now clearly understood. To give a recent example, combining information from the SDSS, Pan-STARRS, and 2MASS has recently enabled the construction of accurate maps of the distribution of stars and interstellar matter in the Milky Way (Green et al. 2015). Finally, a number of existing and upcoming surveys come with a *time domain component*: repeat observations. These result in databases of *time series* of object properties (*light curves*, or *motions*), and sometimes in *real-time alerts* to changes in object properties.

All these elements present significant challenges to classical RDBMS-centric solutions, which have difficulties to scale to PB-level catalog data sets and workflows that require frequent

[4] Moore-Sloan, WRF Innovation in Data Science, and DIRAC Fellow.
[5] http://cds.u-strasbg.fr/
[6] https://archive.stsci.edu/
[7] https://irsa.ipac.caltech.edu

streaming through the entire data set. An alternative may be to consider migrating away from the traditional RDBMS backends, and toward the management of large data sets using scalable, industry-standard, data processing frameworks built on columnar file formats that map well to the types of science analyses expected for the 2020s. Yet these systems have seen limited adoption by the broader astronomy community in large part because they lack domain-specific functionality needed for astronomical data analysis, are difficult to deploy, and/or lack ease of use.

In this work, we tackle the challenge of adapting the one of the industry-standard big-data analytics frameworks—Apache Spark—to the needs of an astronomical end-user. We name the resultant set of changes to Spark, and the accompanying client library, *Astronomy eXtensions for Spark*, or AXS.[8] AXS adds astronomy-specific functionality to Spark that makes it possible to query arbitrarily large astronomical catalogs with domain-specific functionality such as positional cross-matching of multiple catalogs, region queries, and execution of complex custom data analyses over whole data sets. With Spark's strong support for SQL, our hope is to provide a smooth migration path away from traditional RDBMS systems, and a basis for a scalable data management and analysis framework capable of supporting PB+ data set analyses in the LSST era.

We begin by reviewing our prior work in this area and the design drivers for AXS in Section 2. In Section 3, we introduce Apache Spark and discuss its architecture as the foundation that AXS is built on. In Section 4, we describe the algorithm powering AXS distributed cross-match functionality. Section 5 describes the AXS Python API implementation and design choices made therein. We execute and discuss the benchmarks in Section 6, and demonstrate astronomical applications on the Zwicky Transient Facility (ZTF; Bellm et al. 2019; Graham et al. 2019) data set in Section 7. We summarize the conclusion and plans for future work in Section 8.2.

2. Prior Work

In developing the concepts for AXS, we draw heavily on experiences and lessons derived from the eight years of development, use, and support of the Large Survey Database (LSD; Jurić 2010). We begin with a description of the LSD design and lessons learned from its use. We follow by discussing Spark as the basis for a next-generation system, and add an overview of similar work in this area.

2.1. The LSD

The LSD (Jurić 2011, 2012) is a domain-specific Python 2.7 computing framework and database management system for distributed querying, cross-matching, and analysis of large survey catalogs ($>10^9$ rows, >1 TB). It is optimized for fast queries and parallel scans of positionally and temporally indexed data sets. It has been shown to scale to more than $\simeq 10^2$ nodes, and was designed to be generalizable to "shared-nothing" architectures (i.e., those where nodes do not share memory or disk resources and use them independently in order to maximize concurrency). The primary driver behind its development was the immediate need for management and analysis of Pan-STARRS1 data at the Harvard-Smithsonian Center for Astrophysics, and the desire for rapid, early science from the new data set. The LSD code is available at http://github.com/mjuric/lsd.

2.1.1. A Spatially and Temporally Partitioned Database

The LSD is optimized for fast queries and efficient parallel scans of positionally (longitude, latitude) and temporally (time) indexed sets of rows. Its design and some of the terminology have been inspired by Google's BigTable (Chang et al. 2006) distributed database and the MapReduce (Dean & Ghemawat 2004) programming model.

LSD tables are vertically partitioned into *column groups* (`cgroups`): groups of columns with related data (e.g., astrometry, photometry, survey metadata, etc.) and horizontally into equal-area space and time cells (Hierarchical Equal Area and isoLatitutde Pixelization of the sphere, HEALPix; Górski et al. 1999) pixels on the sky, and equal time intervals. On disk, the partitioning maps to a directory structure with compressed, checksummed, HDF5 files (*tablets*) at the leaves. LSD achieves high performance by keeping related data physically stored together, and performing loads in large chunks (optimally $\geqslant$128 MB).

2.1.2. Fast, On-the-fly, Positional Cross-matching

Positional cross-match is a join of two catalogs based on the proximity of their objects' coordinates: in the simplest case, a row in table A will be joined to (potentially k) nearest neighbors in table B. More usefully, the cross-match is made probabilistically, taking measurement errors and other uncertainties into account (Budavári & Basu 2016). Cross-matching is the fundamental operation in survey data analysis. It makes it possible to associate and gather data about the same physical phenomenon (object) observed in different catalogs (wavelength regimes and timescales) or at different times within the same catalog. It allows one to query the *neighborhood* of the objects. Such joined and integrated information can then be fed to higher-level functions to infer the nature of the object, predict its expected behavior, and potentially assess its importance as a follow-up target (including in real time).

Though conceptually simple, spatial cross-matching is non-trivial to implement at scale over partitioned catalogs. To maximize throughput, one wishes to independently and in parallel cross-match on a per-partition (per *cell*, in LSD terminology) basis. If done naively, however, this could lead to incorrect cross-matches near cell edges, where the nearest neighbor resides in the adjacent partition. LSD solves the problem by duplicating in both cells rows that are within a small margin (typically, $30''$) of shared edges (the *neighbor cache*). This copy allows for efficient neighbor lookup (or, for example, for the application of spatial matched filters) without the need to access tablets in neighboring cells. This simple idea allows accurate cross-matches (and matched filter computations) with no inter-cell communication.

2.1.3. Programming Model: SQL and Pipelined MapReduce

The programming workflow model implemented in LSD is a pipelined extension of MapReduce, with an SQL-like query language used to access data. The language implemented is a subset of SQL DML (ISO 2011)—i.e., the `SELECT` statement has no subquery support—with syntax extensions to make it easier to write astronomy-specific queries and freely mix Python and SQL. For more complex tasks, the user can write

[8] The code is available at https://github.com/dirac-institute/AXS and the documentation at https://dirac-institute.github.io/AXS/.

computing *kernels* that operate on subsets of query results on a per-cell basis, returning transformed or partially aggregated results for further processing. These are organized in a lazily computed directed acyclic graph (DAG), similar to frameworks such as Dask (Rocklin 2015) or Spark.[9] Importantly, the framework takes care of their distribution, scheduling, and execution, including spill-over to disk to conserve RAM.

This combination leverages the users' familiarity with SQL, while offering a fully distributed computing environment and the ability to use Python for more complex operations. It reduced the barrier to entry to astronomers already used to accessing online archives using SQL, giving them an acceptable learning curve toward more complex distributed analyses.

2.1.4. Science Applications

Initial science applications of LSD focused on managing and analyzing the 2×10^9 row Pan-STARRS data set. LSD was subsequently applied to catalogs from other surveys, including the time-domain-heavy Palomar Transient Factory (PTF; Law et al. 2009) survey (a precursor to the ZTF), and for aspects of R&D within the LSST Project.

Notable results enabled by LSD included the next-generation dust extinction map (Schlafly et al. 2014), the reconstruction of the three-dimensional distribution of interstellar matter in the Galaxy from joint Pan-STARRS and SDSS observations (Green et al. 2015; Schlafly et al. 2015, 2017), the construction of a *Gaia*–PS1–SDSS proper motion catalog covering 3/4 of the sky (Tian et al. 2017), the DECam Plane Survey delivering optical photometry of two billion objects in the southern Galactic plane (Schlafly et al. 2018), a comprehensive map of halo substructures from the Pan-STARRS1 3π Survey (Bernard et al. 2016), mapping of the asymmetric Milky Way halo and disk with Pan-STARRS and SDSS (Bonaca et al. 2012), the photometric ubercalibration of the Pan-STARRS, the hypercalibration of SDSS (Finkbeiner et al. 2016), the analysis of the first simulated star catalog for LSST (LSST Science Collaboration et al. 2009, Section 4.4 in v2 onwards), and the discoveries and mapping of Galactic halo streams (Sesar et al. 2012, 2013).

2.1.5. Lessons Learned with LSD

LSD was adopted by a number of research teams as it provided them with a way to *query*, *share*, and *analyze* large data sets in a way responsive to their customs and needs. It reduced the barrier to entry for analysis from having the skill to write a complex SQL query (with potentially multiple subqueries and table-valued function tricks) to having the know-how to write a Python function operating in parallel on chunks of the result sets. Importantly, it was *performant* and *transparent*. LSD data set scans typically finished in a fraction of time required for analogous queries in the central RDBMS solution (running low-selectivity queries incurred virtually no overhead over raw `fread`-type I/O). By providing a clear and simple programming model, it avoided the downside of complex query optimizers that sometimes make innocuous changes to queries and lead to orders of magnitude different performance, frustrating users.

Second, LSD served as a test-bed for new concepts and technologies, validating design choices such as column-store data structures, Python-driven distributed workflows organized in DAGs, aggressive in-memory operation with in- and out-of-core caching, mixing of declarative (SQL) and imperative syntax and others. When initially implemented in LSD, these were experimental and controversial choices; today they are broadly used and robustly implemented by frameworks like Spark, Dask, and Pandas and formats like Parquet (Parquet Project 2018).

Just as importantly, the years of "in-the-field" experience gives us an opportunity to understand the major areas in need of improvement.

1. *Fixed partitioning*. LSD implements a fixed, non-hierarchical, partitioning scheme.[10] This leads to significant partitioning skew (factors of 100 between the rarest and densest areas of the sky). While partitioning can be changed on a per-table basis, cross-matching tables with different partitionings is not possible.
2. *Problematic temporal partitioning*. LSD tables are partitioned on time, facilitating performant appends and "time slicing." However, the vast majority of real-world use-cases have users request *all* the data on a particular object (typically to perform classification or some other inference task). Having the time series scattered over a large number of files induces a significant performance hit. A design where time-series is stored as an array column within the object table would perform significantly better.
3. *Lack of robust distributed functionality*. LSD is not resilient to the failures of workers and/or processing nodes. Setting it up and running in multi-node configurations was always experimental and a major end-user challenge. This functionality is crucial for it to continue to scale, however.
4. *Much custom code and no Python 3 support*. LSD contains custom code and solutions for problems where mature, adopted, solutions exist today (Pandas, AstroPy, scikit-learn, and Spark itself). This reduces stability, developer community buy-in, and increases maintenance cost. Also, LSD is written in Python 2.7, for which support will end in 2020.

2.2. Desiderata for a Next-generation System

The issues identified in the previous section, and especially the point about custom code, made us re-examine the development path for LSD. Rather than continuing to maintain an old (and in many ways experimental) code base, a more sustainable way forward would be to build a new system that retains the successful architectural concepts and the user-friendly spirit of LSD while addressing the recognized issues. In particular, we define the following set of key desiderata.

1. *Astronomy-specific operations support*: for an analysis system to be broadly useful to astronomers it must support common astronomy-specific operations, the most important of which are cross-matching and spatial querying;

[9] The similarity is serendipitous; when LSD was written, Spark was still in its infancy and Dask did not exist.

[10] A conscious decision to keep the design simple, and accelerate early development.

2. *Time-series awareness*: the ability to intuitively query or manipulate entire time series of observations of a single object or an entire population,
3. *Ease of use*: to enable the domain scientist to construct and execute arbitrary analyses, *in a distributed fashion* by mixing declarative SQL syntax and Python code, as appropriate;
4. *Efficiency*: fast execution of key operations (positional cross-matching, selective filtering, and scanning through the entire data set);
5. *Scalability*: ability to handle O(10TB) and scale to O(1PB+) tabular data sets, with significant data skews;
6. *Use of industry-standard frameworks and libraries*: building on present-day, proven technologies makes the code more maintainable in the long-run; it also allows us to leverage the R&D, code, and services from other areas of big data analyses.

This last element—the maximal re-use of industry standard frameworks—has been a particularly strong driver in this work. Relative to other similar approaches (e.g., Brahem et al. 2018; see also Section 8.1), we aim to build on top of Apache Spark, with minimal changes to the underlying storage scheme, engine, or query optimizer. We therefore design our changes and algorithms to make as many of them *generic*, and admissible for merging into the Apache Spark mainline. While this restricts our choices somewhat, it enables (the much larger) communities outside astronomy to benefit from our improvements, as well as contribute to them. It also increases long-term maintainability by reducing the astronomy-specific codebase needed to be maintained by the (much smaller) astronomy community.

3. Astronomy eXtensions for Spark

3.1. Apache Spark as a Basis for Astronomical Data Management and Analysis Systems

Apache Spark is a fast and general-purpose engine for big-data processing, with built-in modules for streaming, SQL queries, ML, and graph processing (Figure 1 shows logical components of Spark's architecture; Armbrust et al. 2015; Zaharia et al. 2016). Originally developed at UC Berkeley in 2009, it has become the dominant big-data processing engine due to its speed (10–100× faster than Hadoop), attention to ease of use, and strong cross-language support. Spark supports performant column-store storage formats such as Parquet (Parquet Project 2018). It is scalable, resilient to individual worker failures and provides strong Python interfaces familiar to astrophysicists and other data scientists. Similar to LSD, Spark already "... offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural code" (Armbrust et al. 2015). Relative to comparable projects such as Dask (Rocklin 2015), Spark is more mature and broadly adopted. Furthermore, Spark natively supports *streaming* (and with a unified DataFrames API), allowing applications to real-time use cases.

3.2. Spark Architecture

Fundamentally, Spark is a system that facilitates fault-tolerant, distributed, *transformations* of arbitrarily large data sets.[11] The core Spark abstraction is one of a *resilient distributed data set*, or RDD, a fault-tolerant collection of elements (e.g., table rows, images, or other data) that can be operated on in parallel. Parallelism is achieved through *partitioning*: each RDD is composed of a number of (potentially large) *partitions*, with transformations operating primarily on individual partitions. Spark provides a number of fundamental transformations, such as *map* (where each element in the input data set is operated on by an arbitrary function to be mapped into the output data set), *filter* (where only elements for which the filtering function returns true are placed into the output data set), *reduceByKey* (where a reduction operation is applied to all elements with the same key), and others.[12] Any non-fatal failures (e.g., temporary network outages, or nodes down) are detected automatically, and computation is re-scheduled and re-executed with no need for user involvement.

Nearly every data analysis algorithm can be expressed as a set of transformations and actions on Spark RDDs. This fact is used to build higher-level abstractions that retain the distributed and fault-tolerant properties of RDDs, but provide a friendlier (or just more familiar) interface to the user. Of particular importance for us are the *Spark SQL* layer and the *DataFrames* abstraction.

Spark SQL implements handling of structured data organized in typed columns—tables—using extended standard SQL language and the matching API. This enables one to write SQL queries over any tabular data set stored in a format that Spark is able to read. The core data structure Spark SQL operates on is the DataFrame (a generalized table). Spark DataFrames are built on RDDs, inheriting their fault tolerance and distribution properties. Otherwise, they are very similar in behavior (and API) to R or Pandas data frames. Beyond a friendlier API, DataFrames enable performance optimizations provided by Spark's Catalyst optimizer: advanced query planning and optimization, translation of SQL queries and transformations into generated Java code compiled and executed on the fly, and a more compact data serialization.

On top of this structure lies a rich set of libraries, used by end-user application programs. For example, Spark's MLlib includes efficient distributed implementations of major ML algorithms. This layer also includes various programming language bindings, including Python. All these characteristics make it an excellent choice as a basis for an end-user framework for querying, analyzing and processing catalogs from large astronomical surveys. They tick off nearly all technical desiderata (efficiency, scalability, and broad support), leaving us to focus on the addition of astronomy-specific operations, time-series awareness, and the ease of use through an enhanced Python API. We describe these in the sections to follow.

4. Extending the Spark SQL Library to Enable Astronomy-specific Operations

To support an efficient and scalable cross-match and region queries of data stored in Spark DataFrames, we minimally extend Spark API with two main contributions: a data partitioning and distribution scheme, and a generic optimization of Spark's sort–merge join algorithm on top of which we build a distributed positional cross-match algorithm.

[11] For example, even an SQL query can be thought of as transforming the original, potentially PB-scale data set, to a new, potentially few-kB data set.

[12] For example, see https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations for a complete list.

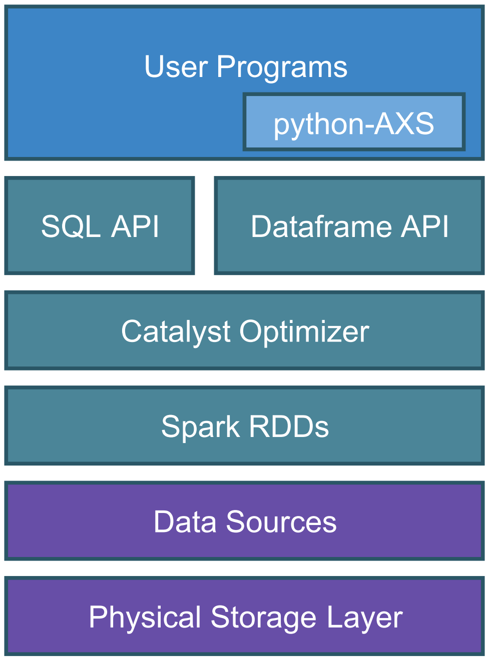

Figure 1. Layered architecture of Spark, with the Python elements of AXS. By the end of the grant period, our entire code will reside in the Python layer. The modifications to Data Sources, RDD, Catalyst, and DataFrame layers will be submitted to upstream projects enabling their broader use.

4.1. *Cross-matching Objects in Astronomical Catalogs*

In astronomy, we are often interested in joining observations from two (or more) survey catalogs that correspond to the same physical objects in the sky. The simplest version of this problem reduces to finding all observations, from two or more catalogs, that are less than some angular distance apart.

More formally, if L is the left relation (catalog) and R is the right one, the cross-matching operation is a set of pairs of tuples l and r such that the distance between them is less than the defined threshold of ϵ:

$$\{(l, r) \mid (l, r) \in L \times R, \operatorname{dist}(l, r) \leqslant \varepsilon\}. \quad (1)$$

This is graphically illustrated in Figure 2.

AXS also supports the nearest-neighbor join where, for each tuple in the left relation, only the tuple in the right relation with the minimum distance is included (which still has to be smaller than the defined ϵ threshold).

4.2. *Distributed Zones Algorithm*

The fundamental problem with cross-matching is how to enable data spatial locality and organize the data for quick searches. Traditionally, two different indexing schemes have been popular: HEALPix (Górski et al. 1999) and HTM (hierarchical triangular mesh; Kunszt et al. 2001). In an earlier paper, Gray et al. (2004) compare HTM, which is used in SDSS's SkyServer (Gray et al. 2002), and the zones approach and find the zones indexing scheme to be more efficient when implemented within relational databases. The zones algorithm is further developed in Gray et al. (2007).

For AXS, we extend the Gray et al. (2007) zones algorithm to adapt it for a distributed, shared-nothing architecture. Zones divides the sky into horizontal stripes called "zones" which serve as indexes into subsets of catalog data so as to reduce the amount of data that needs to be searched for potential matches.

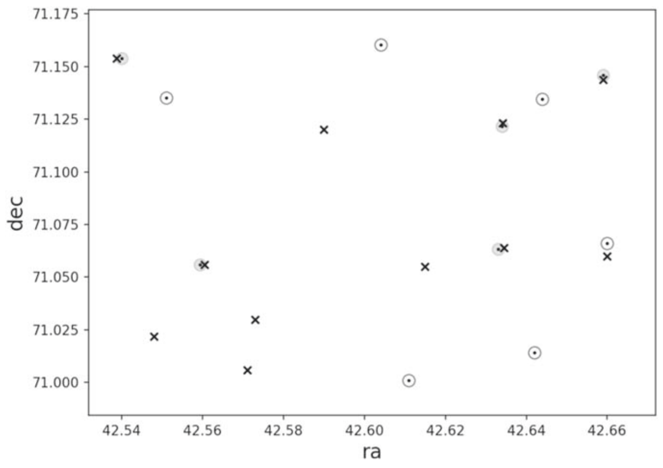

Figure 2. Example of cross-matching two catalogs in a $10'' \times 10''$ region of the sky. Objects of the two catalogs being matched are represented as dots and crosses. The circles show the search region around each object of the first catalog. Circles are filled if their area contains a match.

Given such partitioning, the general idea is to express the cross-join operation as a query of the form shown in Listing 1.

Listing 1. Example of a range query

```
SELECT * FROM GAIA g JOIN SDSS s
  ON g.zone = s.zone
  AND g.ra BETWEEN s.ra - e AND s.ra + e
  AND distance(g.ra, g.dec, s.ra,
    s.dec) <=e
```

(where `e` is a distance that defines the size of the moving window and `distance` is a function which calculates distance between two points), but ensure that (a) the data are partitioned so that this query can be run in parallel over many partitions, and (b) the SparkSQL optimizer is capable of optimizing the query so as to avoid a full Cartesian JOIN within each partition (i.e., that the expensive `distance` function is evaluated only on pairs of objects within a bounding box defined by the `g.ra BETWEEN s.ra - e AND s.ra + e` clause of the query).

4.2.1. *Bucketing Using Parquet Files*

In the distributed zones algorithm we keep the division of sky into N zones but also physically partition data into B buckets, implemented as Parquet bucketed files, so as to enable independent and parallel processing of different zones. Parquet[13] is a distributed and columnar storage format with compression and basic indexing capabilities. Parquet's columnar design is a major advantage for most astronomical catalog usage. Buckets in a Parquet file are actually separate physical Parquet files in a common folder. The Parquet API then handles the whole collection of files as a single Parquet file. Buckets could be implemented in different ways, but we chose Parquet because of its ease of use, level of integration with Spark, and performance.

All the objects from the same zone end up in the same bucket. The zones are placed in buckets sequentially: zone `z` is placed into bucket `b = z % B` (Figure 3 shows an example for

[13] https://parquet.apache.org/

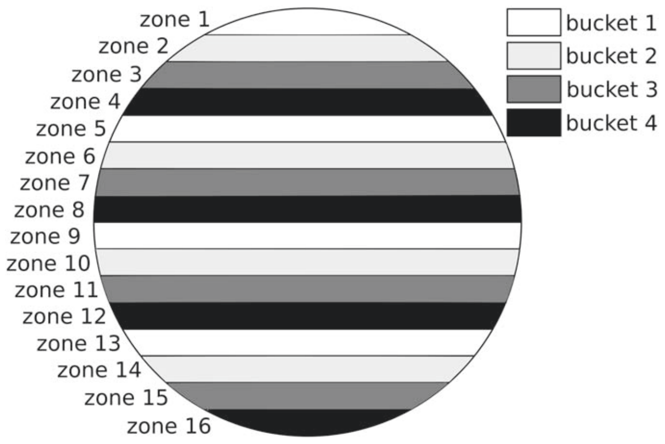

Figure 3. Partitioning the sky into zones and placing zones into buckets. The example shows the sky partitioned into 16 horizontal zones. Objects from each zone get placed into buckets sequentially. In reality, zones are much narrower and are counted in thousands.

16 zones and four buckets, but in reality thousands of zones are used). The reason for placing zones into buckets in this manner is that placing thin neighboring stripes into different buckets automatically reduces data skew, so often present in astronomical catalogs.

If we partition two catalogs in the same way (with the same zone height and the same number of buckets), we can cross-match objects within the same buckets independently of the other buckets. This scheme makes the cross-join operation parallelizable and scalable.

4.2.2. Joining Data within Buckets

Data in each bucket are sorted first by `zone`, then `ra` column, which serve as indexing columns for cross-matching operations. Every table handled by AXS needs to have `zone` and `ra` columns if it is to be used for subsequent cross-matching with other tables (also, zone height and number of buckets in the two tables being joined need to be the same).

In order to efficiently join data within buckets, we chose to use an "epsilon join" (Silva et al. 2010) implementation where two tables are joined quickly in one pass by maintaining a moving window over the data in the right table, based on an equi-join condition on the primary column (`zone` in our case) and a range condition on the secondary column (`ra` in our case). Importantly, we extended the Spark's sort–merge join implementation[14] to recognize an optimization opportunity and avoid calculating the distance function for all object pairs from the two zones, but only for those which match the prior `BETWEEN` condition.

As a consequence of the data bucketing scheme and the epsilon join optimization, Spark executes a query shown in Listing 1 as multiple parallel and fast join operations on bucket pairs from the two tables. The join is performed without data exchanges (shuffles) between nodes, and the data need to be read only once. Figure 4 shows the process for a bucket pair graphically.

4.3. Correctness at Zone Boundaries

An important point for cross-match completeness and performance is joining data from neighboring zones. Objects

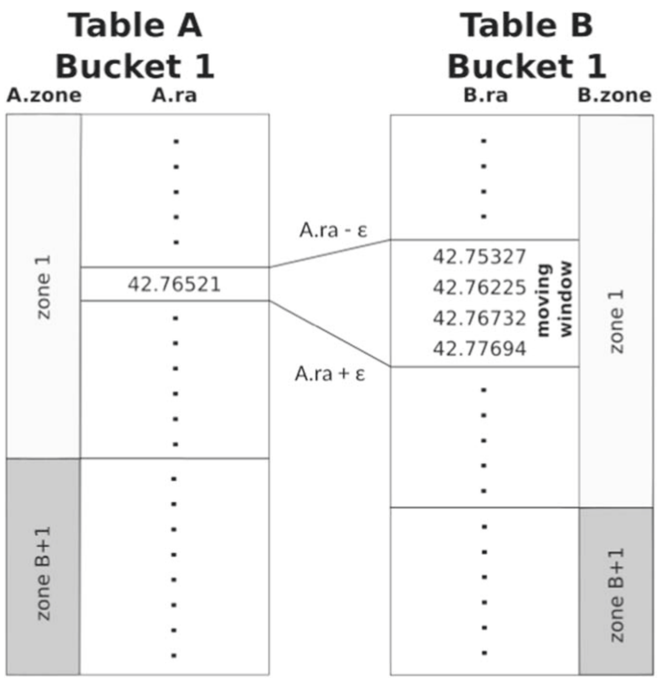

Figure 4. Using the "epsilon join" to reduce the number of rows for which distance is calculated. For the match candidate row in the figure, only four distance calculations are performed (B stands for the number of buckets).

residing near zone borders might have matching objects in the border area of the neighboring zone. Joining only objects from the matching zones would miss these matches in the neighboring zones. In order to maintain the possibility to join objects within a single zone independently of other zones, we chose to duplicate objects from the lower border stripe of each zone to the zone below it. This marginally increases the amount of data stored on disk, but allows us to run cross-joins without data movement between processes (and possibly nodes) during query processing.

The duplication does have consequences for ordinary table queries. First, queries for objects in the catalog now need to be modified so as not to include the duplicated data. Second, cross-matching results must be pruned for objects from duplicated border stripes that have been cross-matched twice (once inside their original zone and once inside the neighboring zone). In our implementation, this is transparently handled in the Python AXS API layer.

Finally, we note that the "height" of stripe overlap needs to be selected such that it covers the maximum expected cross-join search radius. We find that $10''$ is a reasonable default, capable of handling data sets ranging from SDSS to ZTF and *Gaia* scales.

4.4. Zone Height and Number of Buckets

Choosing the number of buckets is a trade-off between the bucket size (and the amount of data that need to be read as part of a single Spark task) and the maximum parallelism achievable. This is because a single task can process all buckets serially, but several tasks cannot process a single bucket. Data are read from buckets in a streaming fashion, so larger buckets do not place a much larger burden on memory requirements.

Larger zones reduce the number of zones which means that fewer data will need to be duplicated. However, smaller zones

[14] https://github.com/apache/spark/pull/21109

reduce data skew. Data skew can significantly affect processing, so it is advisable not to use zones that are too large. Furthermore, larger zones will have more rows in their "moving windows" during the cross-match operation and will hence require more memory for cross-matching.

4.5. Data Skew Considerations

Large (especially all-sky) astronomical data sets often include highly skewed data because of the highly uneven distribution of astronomical sources on the sky. Under such skew and with naive spatial partitioning, processing of queries on very populous partitions would use large amounts of memory and last much longer than what would typically be the case. This would considerably degrade the overall query performance.

As was already mentioned in Section 4.2.1, a convenient property of data placement according to the distributed zones algorithm is that, when zones are sufficiently narrow, the data get naturally distributed more or less equally among the buckets. In practice, even for highly skewed catalogs such as the SDSS and *Gaia*, we observed the maximum ratio of sizes of different bucket files of a factor of two.

5. Ease of Use: AXS Python API

As the main end-user interface to AXS we provide a Python API. This API is a thin layer on top of `pyspark`, Sparks's own Python API. We designed it to expose the added astronomy-specific functionality, while abstracting away the implementation details such as partitioning or the underlying cross-match algorithm.

In this section we highlight some of the key elements of the Python API. The full documentation is available at https://dirac-institute.github.io/AXS/.

5.1. A Simple Example

The two main classes the user interacts with in AXS are `AxsCatalog` and `AxsFrame`. These serve as extensions of Spark's `Catalog` and `DataFrame` interfaces, respectively.

By analogy with Spark's `Catalog`, an instance of `AxsCatalog` is constructed from an instance of `SparkSession`:

```
from axs import AxsCatalog
axs_catalog = AxsCatalog(spark)
```

The `SparkSession` object is similar to a connection object from standard Python database API.[15] It represents the connection to the Spark metastore database, and enables manipulation of Spark tables. The `AxsCatalog` instance adds awareness of tables that have been partitioned using the distributed zones algorithm (Section 4.2), and the ability to retrieve their instances as shown in the following snippet:

```
axs_catalog.list_tables() # output omitted
sdss = axs_catalog.load(''sdss'')
gaia = axs_catalog.load(''gaia'')
```

The returned objects above are `AxsFrames`. These extend Spark `DataFrame` objects with astronomy-specific methods. One of these is `crossmatch` which performs cross-matching between two `AxsFrame` tables.

```
from axs import Constants
gaia_sd_cross = gaia.crossmatch(sdss,
  r = 3 * Constants.ONE_ASEC, return_min = False)
gaia_sd_cross.select(
  ''ra,'' 'dec,'' 'g,'' 'phot_g_mean_mag'').
  save_axs_table(''gaiaSdssMagnitudes'')
```

The snippet above sets up a pipeline for positional cross-matching of the *Gaia* and SDSS catalogs. The resulting catalog is then queried for a subset of columns, which are finally saved into a new, zones-partitioned, table named `gaiaSdssMagnitudes`. We note that the graph above executes only when `save_axs_table` is called; in other words, `crossmatch` and `select` are *transformations* while `save_axs_table` is an *action*.[16]

Had the `return_min` flag in the above snippet been set to `True`, `crossmatch` would return only the nearest neighbor for each row in the *Gaia* catalog.

5.2. Support for Spatial Selection

Region queries in AXS are queries of objects in regions of the sky based on boundaries expressed as minimum and maximum R.A. and decl. angles. Having sky partitioned in zones and zones stored in buckets, region queries get translated into several searches through the matching bucket files. The underlying Spark engine is able to execute these in parallel. And since the bucket files are sorted by `zone` and `ra` columns, the zone being derived from decl. values, these searches can be performed quickly. We take advantage of the fact that Spark can push column filters down to Parquet file reading processes so that whole bucketed files can be skipped if they are not needed. Similarly, reading processes can skip parts of files, based on sort columns.

These optimizations are hidden from the AXS user and region queries in AXS API are as simple as this:

```
region_df = gaia.region(ra1 = 40, dec1 = 15,
      ra2 = 41, dec2 = 16)
```

Cone search is implemented using the `cone` method. It requires a center point and a radius and returns all objects with coordinates within the circle thus defined.

More complex selections (e.g., support for polygons) are possible to implement using these primitives coupled with additional Python code. These may be added at a later date, if there is sufficient user demand.

5.3. Support for Time Series

Support for time series data that is both performant and user friendly is increasingly important given the advent of large-scale time-domain surveys. A prototypical time series is a light curve: a series of flux (magnitude) measurements of a single object. Measurements typically include the measurement itself, the time of the measurement, the error, and possibly other metadata (e.g., flags, or the filter band in which the measurement was taken).

With AXS, we recommend storing time series data as *a set of vector (array) columns in an object catalog*. This is a departure

[15] https://www.python.org/dev/peps/pep-0249/

[16] See https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations for details.

from the more classical RDBMS layout employing an “object” and “observation” table, where the light curve would be constructed by JOIN-ing the two. The classical approach has two disadvantages: converting the returned JOIN into a more natural “object + light curve” representation is left to the user, and may not be trivial (especially to an inexperienced user). Second, the need to perform a JOIN, and the fact that the observation data are physically separate (on disk) from object data, reduces the overall query performance. The downside of our approach is that updates to time-series columns are expensive; this, however, is not an issue when AXS is used to manipulate large *data release* data sets, which are static by definition. Finally, nothing precludes users from structuring their data set across two (object, observation) tables if it is better for their particular use case. Various time-series support functions assume the vector-column format at this time, however.

We illustrate a few time-series support functions on an example of handling a light curve. To make it simpler to support multi-wavelength use cases (where the light-curve across all bands is passed to a user-defined function for, e.g., classification), we recommend storing observations in a tuple of vector columns such as `(time, mag, magErr, band)`. This does reduce performance if the user wishes to perform analysis on a single filter at a time, but we increasingly see a demand for simultaneous cross-band analysis. To support maximum performance, we provide two helper array functions usable from within queries: `ARRAY_ALLPOSITIONS`, which returns an array of indexes of all occurrences of an element, and `ARRAY_SELECT`, which returns all elements indexed by the provided index array. These two functions, combined with other Spark SQL functions, can be used for querying and manipulating light-curve data in AXS tables.

For example, to get the number of *r*-band observations for all objects in a catalog `ztf`, assuming that column `band` contains the filter name used for each measurement, one can use a snippet similar to the one in Listing 2.

Listing 2. Example of using `array_allpositions`

```
from pyspark.sql.functions import size,
  array_allpositions
ztf_rno = ztf.select(size(
  array_allpositions(ztf(``band''), `r'')))
```

`array_allpositions` returns an array of indices into the *band* column, each corresponding to the *r*-band value. Spark’s built-in function `size` returns the length of the indices array.

5.4. Fast Histograms

A common summarization technique with large survey data sets is to build *histograms* of a statistic as a function of some parameters of interest. Examples include sky maps (e.g., counts or metallicity as a function of on-sky coordinates) and Hess diagrams (counts as a function of color and apparent magnitude). AXS Python API offers a support for straightforward creation of histograms. These are implemented as relatively straightforward wrappers around Spark API, making their execution fully distributed and fault-tolerant.

The functions of interest are `histogram` and `histogram2d`. When calling the `histogram` method, users pass a column and a number of bins into which the data are to be summarized. Similarly, `histogram2d(cond1, cond2, numbins1, numbins2)` bins the data in two dimensions, using the two provided condition expressions.

An example of histogramming is given in code Listing 3. The code results in a 2D histogram graph showing the density of differences in observations in *g band* between the SDSS and *Gaia* catalogs, versus the same differences between the *WISE* and *Gaia* catalogs. Both differences are binned into 100 bins.

Listing 3. An example of using `histogram2d`

```
from pyspark.sql.functions import coalesce
import matplotlib.pyplot as plt
cm = gaia.crossmatch(sdss).crossmatch(wise)
(x, y, z) = cm.histogram2d(
  cm.g - cm.phot_g_mean_mag,
  cm.w1mag - cm.phot_g_mean_mag,
  100, 100)
plt.pcolormesh(x, y, z)
```

5.5. Saving Intermediates

By default, Spark eschewes saving intermediate results of computations, unless an explicit request has been made to do so.

We have found there are common use cases where saving intermediate calculation results and reusing them later is useful. One example is a result of a cross-match of large tables, which will then be further joined with or cross-matched to other catalogs. To be subsequently cross-matched, these intermediate tables need to be partitioned in the same distributed zones format and stored in the `AxsCatalog` registry. To support this common operation, we provide an `AxsFrame.save_axs_table` method. It saves the underlying `AxsFrame`’s data as a new table and partitions the data as was described in Section 4.2.

5.6. Support for Python User-defined Functions (UDFs)

`AxsFrame` class’s methods `add_column` and `add_primitive_column` are thin wrappers around Spark’s `pandas_udf` and `udf` functions. They are only intended to make it a little easier for astronomers to run custom data-processing functions on a row-by-row basis (i.e., to avoid using `@pandas_udf` and `@udf` annotations) and make their code more readable. They are applicable only when handling data row by row, which corresponds to Spark’s `udf` function and Spark’s `pandas_udf` function of type `PandasUDFType.SCALAR` and cannot be used for `PandasUDFType.GROUPED_MAP` nor `PandasUDFType.GROUPED_AGG` UDFs.

Both functions accept a name and a type of the column to be added, the function to be used for calculating the column’s contents, and names of columns whose contents are to be supplied as input to the provided function. The difference between the two methods is that `add_primitive_column` supports only outputting columns of primitive types, but is significantly faster because it uses Spark’s `pandas_udf` support under the hood. `add_column` method uses the scalar `udf` functions, making it slower, but supports columns of complex types. `pandas_udf` is faster because it is able to

Table 1
Catalog Used for Performance Tests, with Number of Rows, Number of Non-duplicated Rows, and Compressed Data Size

Catalog	Row Count	R. cnt. no dup.	Size
SDSS	0.83 Bn	0.71 Bn	71 GB
Gaia DR2	1.98 Bn	1.69 Bn	464 GB
AllWISe	0.87 Bn	0.75 Bn	384 GB
ZTF	3.13 Bn	2.93 Bn	1.17 TB

handle blocks of rows at once by utilizing the Python Pandas framework (and its vectorized processing).

For an example and discussion of using UDFs see Section 7.

5.7. Adding New Data to AXS Catalogs

While our primary use-case at present is to support analysis of large, static (i.e., data release) data sets, we have found it useful to be able to incrementally add data to AXS tables (e.g., to facilitate incremental ingestion).

For this purpose, AXS provides the method `add_increment`:

```
add_increment(self, table_name, increment_df,
  rename_to = None, temp_tbl_name = None)
```

This will add the contents of the `increment_df` Spark DataFrame to the AXS table with the name `table_name`, taking care to calculate zones, and to bucket and sort the data appropriately. Users can customize the table name for the copy of the old data (`rename_to`) and the temporary table name used in the process (`temp_tbl_name`).

Adding an increment to an existing table means that the new and old data need to be merged, repartitioned, and saved as a separate table. On top of physical movement of data during this operation, time has to be spent on data sorting and partitioning, which makes this operation the most expensive of all those presented so far. Data partitioning and sorting is a necessary part of the cross-match operation, but is performed in advance and only once, so that the latter part (table joins) can be performed online as needed.

We measured the time AXS needs to partition the different catalogs. The results can be found in Section 6.1.

6. Cross-matching Performance

To test AXS's cross-matching performance, we used the catalogs listed in Table 1. We tested both the scenario where all matches within the defined radius were returned, and the scenario where only the first nearest neighbor was returned for each row. The number of resulting rows for each catalog cross-match combination is given in Table 2, for both scenarios. Furthermore, we compared cross-matching performance for both scenarios in cases when the data were cached in the OS buffers (warm cache) and when thy were not cached (the cache was empty, or "cold cache").[17] This caching mechanism works on OS level and is separate from Spark's caching mechanism (which we did not use) and from its memory handling.

For each scenario we also varied the number of Spark executors (level of parallelism) in the cluster (going from 1 to 28). Each executor was given 12 GB of Java memory heap and 12 GB of off-heap memory.[18] The results are shown in Figure 5, and the raw results are in Tables 3 and 4 for the first scenario, and Tables 5 and 6 for the second scenario. Each data point in the figure and in the tables is an average of three tests.

The results demonstrate the performance that can be expected when running AXS on a single machine. The scalability is constrained by the single machine's shared resources and, consequently, the performance does not improve much beyond 28 executors. It should also be noted that for these tests we used a local file system, not Hadoop File System

[17] For clearing the OS buffers and creating a "cold cache" situation we wrote "3" to the */proc/sys/vm/drop_caches* file.

Table 2
Catalog Combinations Used for Cross-match Performance Tests, with Numbers of Resulting Rows When Returning All Matches or Only the First Nearest Neighbor

Left cat.	Right cat.	Results—All	Results—NN
Gaia DR2	AllWISE	320 M	320 M
Gaia DR2	SDSS	227 M	126 M
ZTF	AllWISE	109 M	109 M
ZTF	SDSS	273 M	168 M
Gaia DR2	ZTF	92 M	49 M
AllWISE	SDSS	235 M	119 M

Table 3
Averaged Raw Cross-match Performance Results (in Seconds), When Returning All Matches, for the First Three Catalog Combinations, Depending on the Number of Executors and Whether Cold or Warm OS Cache Was Used

Execs.	*Gaia*-AllWISE		*Gaia*-SDSS		ZTF-AllWISE	
	Warm	Cold	Warm	Cold	Warm	Cold
1	2438	2276	1441	1412	3487	3283
2	313	1276	300	744	1772	1732
4	168	685	155	401	202	1007
8	99	425	90	235	110	650
12	58	338	61	209	76	491
16	48	284	51	187	57	412
20	40	254	42	166	47	346
24	35	226	37	153	40	329
28	31	226	37	148	38	296

Table 4
Averaged Raw Cross-match Performance Results (in Seconds), When Returning All Matches, for the Last Three Catalog Combinations, Depending on the Number of Executors and Whether Cold or Warm OS Cache Was Used

Execs.	ZTF-SDSS		*Gaia*-ZTF		AllWISE-SDSS	
	Warm	Cold	Warm	Cold	Warm	Cold
1	2885	2690	3084	3115	1469	1461
2	499	1372	402	1567	239	745
4	239	743	209	904	129	436
8	125	489	123	617	65	268
12	88	394	77	512	49	198
16	66	341	60	431	39	162
20	57	282	51	373	35	140
24	52	271	45	361	30	129
28	47	239	39	315	29	122

[18] The ability to directly allocate memory off Java heap was introduced into Spark as part of Tungsten project's changes and can considerably improve performance by avoiding Java garbage collection.

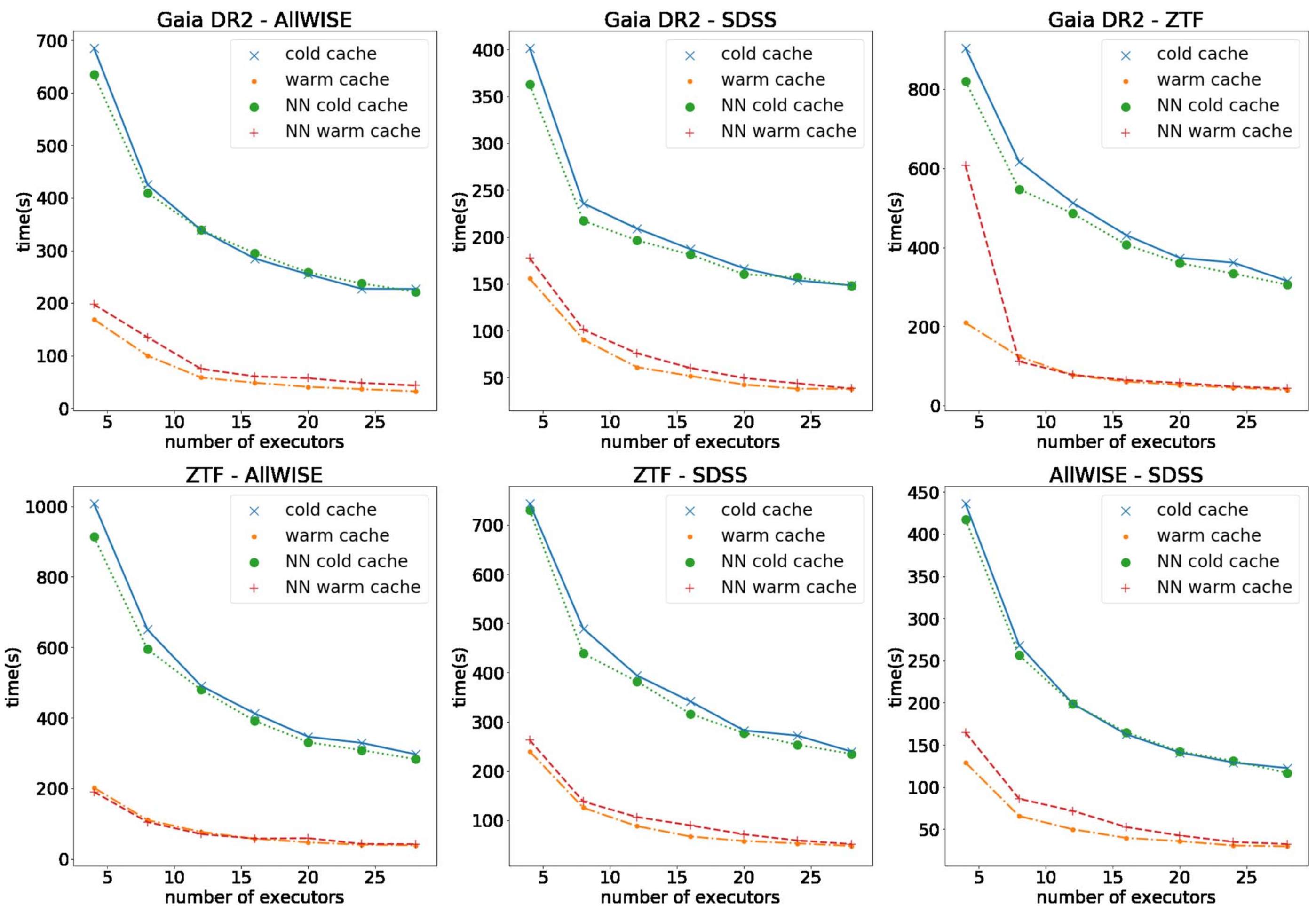

Figure 5. Performance tests of cross-matching various catalogs in scenarios with file system buffers empty or full (we used Linux OS-level caching, not Spark caching), when returning all matches or just the first nearest neighbor ("NN" results).

Table 5
Averaged Raw Cross-match Performance Results (in Seconds), When Returning Only the First Nearest Neighbor, for the First Three Catalog Combinations, Depending on the Number of Executors and Whether Cold or Warm OS Cache Was Used

Execs.	*Gaia*-AllWISE		*Gaia*-SDSS		ZTF-AllWISE	
	Warm	Cold	Warm	Cold	Warm	Cold
1	723	2152	642	1371	3130	2991
2	427	1184	362	702	363	1569
4	197	634	177	362	190	913
8	134	409	101	217	104	594
12	74	338	75	196	70	479
16	60	295	60	181	57	391
20	57	258	49	160	59	330
24	47	237	43	156	42	308
28	42	221	38	147	42	283

Table 6
Averaged Raw Cross-match Performance Results (in Seconds), When Returning Only the First Nearest Neighbor, for the Last Three Catalog Combinations, Depending on the Number of Executors and Whether Cold or Warm OS Cache Was Used

Execs.	ZTF-SDSS		*Gaia*-ZTF		AllWISE-SDSS	
	Warm	Cold	Warm	Cold	Warm	Cold
1	2769	2617	2994	2639	637	1341
2	508	1462	363	1433	291	730
4	262	729	607	820	164	417
8	137	438	111	547	86	256
12	106	381	77	485	71	198
16	89	316	64	406	52	164
20	71	277	56	360	42	142
24	58	253	47	333	34	130
28	51	234	42	305	32	116

(HDFS; tests in a distributed environment using data from S3 storage and HDFS are planned for the future and will be published in a subsequent paper).

The warm cache results show the "raw" performance of the cross-matching algorithm, not including the time required for reading the data from disk. It should be noted, however, that the cache did not have the capacity for complete data sets and that data were partly read from disk in all cases. The cold cache results show the performance users can expect if they do not have much memory available.

These cross-matching results outperform other systems, although direct comparisons are difficult because of different architectures, data sets, and algorithms used. Other teams report best cross-matching times on the scale of tens of minutes, while our best results are in tens of seconds. The comparisons are discussed in Section 8.1.

Table 7
Data Partitioning: Size of the Partitioned Catalogs (in GB) and Time Needed to Partition the Data (in Minutes) Depending on the Number of Zones Used

Catalog	5400 z.		10800 z.		21600 z.	
	Size	Time	Size	Time	Size	Time
SDSS	66	12	71	12	82	12
Gaia	430	89	464	86	532	150
Allwise	352	120	384	119	444	133
ZTF	1124	547	1169	545	1334	523

Note. All tests shown here used 500 buckets for partitioning data.

Table 8
Data Partitioning: Size of the Partitioned Catalogs (in GB) and Time Needed to Partition the Data (in Minutes) Depending on the Number of Buckets Used

Catalog	250 b.		500 b.		750 b.	
	Size	Time	Size	Time	Size	Time
SDSS	71	12	71	12	72	10
Gaia	464	88	464	86	464	86
Allwise	384	125	384	119	384	116
ZTF	1169	557	1169	545	1169	514

Note. All tests shown here used 10,800 zones for partitioning data.

6.1. Data Partitioning Performance

We investigated the effects different zone heights and number of buckets have on data preparation. By data preparation we mean sorting and bucketing of the data as was described previously in Section 4.2.1. This is an operation that needs to be done only once for each catalog (or each new version of a catalog) so that cross-matches can be done online, without additional data movement.

In Table 7 we list the time needed for partitioning each catalog (in minutes) and the size of the partitioned (compressed) Parquet files on disk (in GB), depending on the number of zones used, while using the fixed number of buckets (500, which is the default). We used 28 Spark executors for the tests. The middle columns show the data for the number of zones AXS uses by default (10,800 zones, which corresponds to a zone height of one arcminute). The other two columns show the results for twice as many and half as many zones. As can be seen from the results, the partitioning times depend roughly on the total size of the data but have large fluctuations, so more tests would be needed to make more accurate measurements. However, these numbers are intended to be only informational as users are not expected to need to do this on their own, or at least not often.

Compressed size of partitioned catalogs increases with the number of zones because of increased data duplication, as was explained in Section 4.4.

Table 8 shows the same tests, but this time depending on the number of buckets used, while using the fixed number of zones (the default of 10800). The middle columns are the same as in Table 7 (they correspond to the same number of buckets and zones). The size of the catalogs is the same regardless of number of buckets used (Parquet compression and size of indexes do not depend greatly on the number of files). However, partitioning time obviously decreases with more buckets used. We believe this is because data shuffling is more efficient with smaller files.

Table 9
Cross-matching Duration (in Seconds) Depending on the Number of Zones and Whether Cold or Warm OS Cache Was Used for Each Catalog Combination

Catalogs	5400 z.		10800 z.		21600 z.	
	Warm	Cold	Warm	Cold	Warm	Cold
G—A	32	207	31	226	36	240
G—S	33	128	37	148	36	151
Z—A	47	260	38	296	39	283
Z—S	48	209	47	239	49	227
G—Z	37	271	39	315	44	326
A—S	27	114	29	122	29	130

Note. All tests shown here used 500 buckets for partitioning data.

Table 10
Cross-matching Duration (in Seconds) Depending on the Number of Buckets and Whether Cold or Warm OS Cache Was Used for Each Catalog Combination

Catalogs	250 b.		500 b.		750 b.	
	Warm	Cold	Warm	Cold	Warm	Cold
G—A	32	211	31	226	32	2314
G—S	33	146	37	148	37	159
Z—A	37	292	38	296	36	289
Z—S	47	237	47	239	47	234
G—Z	39	323	39	315	40	313
A—S	28	119	29	122	28	132

Note. All tests shown here used 10,800 zones for partitioning data.

6.2. Cross-matching Performance Depending on the Number of Zones and Buckets

We also investigated the effect different number of zones and different number of buckets have on cross-matching performance. Table 9 shows cross-matching performance results when using different numbers of zones while keeping number of buckets fixed at the default value of 500. Table 10 shows the same when using different numbers of buckets while keeping number of zones fixed at the default value of 10,800. Values in the middle columns in both tables are the same as those in Tables 3 and 4 (because those tests used the default values for both the number of zones and the number of buckets). All tests in this section were done using 28 executors and with the queries returning all matching results.

The results show that increasing zone height improves cross-matching performance and that number of buckets does not greatly influence cross-matching times.

7. Demonstrating a Complete Use Case: Period Estimation for ZTF Light Curves

Integrating AXS and Spark with astronomical applications and tools will enable an ecosystem that can query, cross-match, and analyze large astronomical data sets. We use the example of the analysis of the time series data described in Section 5.3 to illustrate how existing tools can be integrated with AXS. For this case we use Gatspy (General tools for Astronomical Time Series in Python; Vanderplas 2015), a suite of efficient algorithms for estimating periods from astronomical time-series data. Gatspy is written in Python and contains fast implementations of traditional Lomb–Scargle periodograms (Lomb 1976; Scargle 1982) and extensions of the Lomb–Scargle algorithm for the case of multi-period estimation, and multi-band observations. The implementations of these algorithms scale as $O(N)$ for small numbers of points within a light curve or times series (i.e., $<10^4$ points) to $O(N^2)$ for large times series. A 10^3 point light curve requires approximately 10^{-2} s to estimate the best-fitting period.

Python functions are accessed through Spark using UDFs. There are two basic mechanisms for constructing UDFs. In each case, a decorator is used to annotate the Python function (specifying the return type of the output of the function). The earliest implementation of UDFs serialized and distributed the data from a Spark or AXS `DataFrame` to a function row-by-row. More recently the `pandas_udf` has been implemented within Spark 2.4 which enables vectorized operations. Data partitions are converted into an Arrow format as columns, serialized, passed to the Python function, and operated on as a `Pandas.Series`. Returned data are converted to an Arrow format and passed back to the Spark driver over a socket. Because of the vectorized operations and improved serialization, the `pandas_udf` is much more efficient than earlier UDFs (Databricks 2017) and it is this type of UDF that we will consider here. Still, users can expect a degraded performance of UDFs with respect to Spark native functions because UDFs are black boxes for the Spark Catalyst optimizer. Python UDFs can add even more performance degradations because of data serialization and deserialization between different data representations.

For the `pandas_udf` there are two `PandasUDFTypes`, `SCALAR` and `GROUPED_MAP`. For `SCALAR` functions the data are passed to and returned from the Python function as `Pandas.Series` (with the output being a `Pandas.Series` of the same length as the input). A more flexible approach is provided by the `GROUPED_MAP` type. For this case the spark `DataFrame` is subdivided into subsets using a `groupby` function, serialized and passed to the Python function using the Arrow format, and the output returned to the driver as a Spark `DataFrame`. The flexibility of the `GROUPED_MAP` comes from the fact that arbitrarily complex data structures can be returned as a `DataFrame` (e.g., a single period, a vector of periods and scores for the period, or vector arrays for the periodogram and associated frequencies). A limitation on the current `GROUPED_MAP` implementation is that the `apply` function cannot pass arguments to the Python function.

For our example application we adopt the `GROUPED _MAP` data type. The structure of the `pandas_udf` is shown below for a Gatspy function that returns two arrays, a periodogram and the associated frequencies. The format for the returned data is described by the schema.

Listing 4. Applying Pandas functions to partial results

```
schema = StructType(
  [StructField('Frequency',
    ArrayType(DoubleType()),False),
   StructField('Periodogram',
    ArrayType(DoubleType()),False)])

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def LombScargle_periodogram(data):
  model = periodic.LombScargle(
    fit_period=True, Nterms = n_term)
   model.fit(data['mjd'],
    data['psfmag'], data['psfmagerr'])
   periodogram, frequency=
```

(Continued)

```
    model.periodogram_auto()
  return pd.DataFrame(
    dict(Frequency = frequency,
      Periodogram = periodogram))

results = df.groupby('matchid').
  apply(LombScargle_periodogram)
```

8. Discussion

8.1. Similar Systems

The closest to our approach is a system called ASTROIDE, by Brahem et al. (2018). ASTROIDE is also based on Apache Spark and offers an API for cross-matching and processing astronomical data. The main difference compared to AXS is the data partitioning scheme used: AXS partitions the data using a "distributed zones" partitioning scheme, while ASTROIDE uses HEALPix partitioning in a way that closely resembles our prior LSD product (Section 2.1). Comparing cross-matching performance results between ASTROIDE and AXS is not easy because of the differences in benchmarking environments (ASTROIDE authors used 80 CPU cores spread over six nodes, compared to a maximum of 28 cores on a single machine for AXS) and different data sizes (the most relevant ASTROIDE tests were done cross-matching 1.1 billion with 2.5 million objects; the smallest data set in our tests had ~0.7 billion objects). ASTROIDE shows the best results when cross-matching is performed in advance and saved separately (with "materialized partitions"); we prefer online cross-matching to avoid the $O(N^2)$ problem with given an increased number of data sets available. These differences aside, we find that AXS outperforms ASTROIDE in cross-matching performance (even with materialized partitions): for cross-matching 1.2 billion with 2.5 million objects on 80 cores ASTROIDE needs about 800 s. That said, performance was not a main driver in our partitioning approach; AXS has been intentionally designed as a minimal extension to Spark's existing support for SQL and bucketing, rather than a more encompassing rework needed by ASTROIDE. This makes it easier to achieve our goal of upstreaming all Scala-level changes, and eventually having to maintain only the Python API library.[19]

More broadly, the problem of data storage and indexing has been solved within large surveys in various ways. For example, the QServ database (Wang et al. 2011) is being built for the LSST project to enable the scientific community to query the LSST data. A unique aspect of QServ is that it implements *shared scans*, a technique which allows simultaneous execution of whole-database queries by large numbers of users. QServ does not yet implement capabilities for executing more complex data analytics workflows, or image processing functions, something that we have found useful in ZTF work.

The *Gaia* survey data release pipeline implements a cross-match function described in Marrese et al. (2017). Their algorithm is based on a sweep line technique which requires the data to be sorted by decl. The data then only need to be read once. They differentiate between good and bad match candidates and have separate resulting tables for each. They

[19] In case our changes do not get merged into the main Spark distribution, maintaining this patch is still a small task relative to building a completely new data management tool (such as QServ).

also utilize position errors in determining the best match. The algorithm was implemented in MariaDB, with custom performance optimizations. The authors report cross-match time between *Gaia* DR1 data set (1.1 billion objects) and SDSS DR9 (470 million objects) of 56 minutes.

There has been a number of other work focused at providing efficient solutions for cross-matching and handling large astronomical catalogs.

1. *catsHTM* is a recent (2018) tool for "fast accessing and cross-matching large astronomical catalogs" (Soumagnac & Ofek 2018). It stores data in HDF5 files and uses HTM for partitioning and indexing data. They report that cross-matching 2MASS and *WISE* catalogs takes about 53 minutes (without saving the results).
2. Nieto-Santisteban et al. (2007) describe cross-match implementation in *Open SkyQuery*. It is based on the *zones* algorithm and implemented on Microsoft SQL Server. They report SDSS–2MASS cross-match duration of about 20 minutes when using eight machines.
3. In Jia et al. (2015) the authors develop an algorithm for cross-matching catalogs in heterogeneous (CPU–GPU) environments. They index the data using HEALPix indexing method and report the best time of 10 minutes for cross-matching 1.2 billion objects of SDSS data with itself in a multi-node setup with several high-end GPUs.
4. Dobos et al. (2012) implement a probabilistic cross-matching algorithm. They also partition the data based on zones, but each machine contains the full copy of the data. Their cross-match is not only based on distances, but on several criteria which all contribute to the final likelihood calculation. They orchestrate multi-node SQL queries using a complex workflow framework, but do not report any performance numbers.

8.2. Summary and Future Directions

In this paper we presented Astronomy eXtensions for Spark, or AXS, a data management solution capable of operating on distributed systems and supporting complex workflows intermixing SQL statements and Python code. AXS enables scalable and efficient querying, cross-matching, and analysis of astronomical data sets in the $O(10^{10})$ row regime. It is built on top of Apache Spark, with minimal extensions to the core Spark code that have been submitted for merging upstream. We described AXS's data partitioning and indexing scheme and the applied epsilon-join optimization which enable fast catalog cross-matching and reduce data skew. We also described AXS Python API which exposes AXS cross-matching, light-curve querying functions, histograming, and other functionality. The cross-matching performance testing results show that AXS outperforms other systems, with hardware differences taken into account. The tests done so far have all been performed on a single large machine; work is ongoing to deploy and benchmark AXS in a fully distributed setting.

As we discussed in Section 1, the exponential growth of data being generated by large astronomical surveys, and the shifting research patterns toward statistical analyses and examinations of whole data sets, present challenges to management of astronomical data in classic RDBMS systems. We argue that systems like Spark and AXS, or perhaps in the future similar systems built on Dask, may serve as a capable, open-source solution to the impending crisis.

More broadly, the scale of future data sets and the demand for large-scale and complex operations on them poses significant challenges to the typical "subset–download–analyze" analysis paradigm common today. Rather than downloading (now large) subsets, there are strong arguments to "bring the code to the data" instead, and remotely perform next-to-the-data analysis via *science platforms* (e.g., Juric et al. 2016). However, this would place new demands on astronomical archives: the kinds of analyses and the size of the community to be supported would require petascale-level end-user compute resources to be deployed at archive sites. Furthermore, to enable efficient joint analyses/data set fusion, large data sets of interest would eventually need to be replicated across all major archives, adding to storage requirements. Taking this route is possible, but brings with it all operational issues typically encountered in user-facing HPC deployments (in addition to the broader question of utilization and cost-effectiveness). Is turning astronomical archives into large data-center operators the right way to go?

An alternative may be to consider migrating away from the traditional RDBMS backends, and toward the management of large data sets using solutions such as AXS, built on scalable, industry-standard, data-processing frameworks that map well to the types of science analyses expected for the 2020s. As these solutions are designed to operate in distributed, cloud environments, so they would enable utilizing the cloud to satisfy the new computational demands. In the best case, the shift would be dovetailed by a move toward physical co-location of data sets on public cloud resources as well (or a hybrid, private–public, solution, that would allow the analysis to begin at archive centers, but then effortlessly spill over into the cloud). Such a *cloud-native* approach would offer tremendous benefits to researchers: elasticity and cost effectiveness, scalability of processing, shareability of input data sets and results, as well as increased reproducibility.

P. Z., C.T.S., M.J., and A.J.C. acknowledge support from the University of Washington College of Arts and Sciences, Department of Astronomy, and the DIRAC Institute. University of Washington's DIRAC Institute is supported through generous gifts from the Charles and Lisa Simonyi Fund for Arts and Sciences, and the Washington Research Foundation.

P.Z. and S.L. acknowledge support from University of Zagreb Faculty of Electrical Engineering and Computing and the European Regional Development Fund under the grant KK.01.1.1.01.0009 (DATACROSS).

M.J. acknowledges the support of the Washington Research Foundation Data Science Term Chair fund, and the UW Provost's Initiative in Data-Intensive Discovery.

A.J.C. acknowledges support from NSF awards AST-1715122 and OAC-1739419,

Based on observations obtained with the Samuel Oschin Telescope 48 inch and the 60 inch Telescope at the Palomar Observatory as part of the Zwicky Transient Facility project. ZTF is supported by the National Science Foundation under grant No. AST-1440341 and a collaboration including Caltech, IPAC, the Weizmann Institute for Science, the Oskar Klein Center at Stockholm University, the University of Maryland, the University of Washington, Deutsches Elektronen-Synchrotron and Humboldt University, Los Alamos National

Laboratories, the TANGO Consortium of Taiwan, the University of Wisconsin at Milwaukee, and Lawrence Berkeley National Laboratories. Operations are conducted by COO, IPAC, and UW.

ORCID iDs

Petar Zečević https://orcid.org/0000-0002-2651-243X
Colin T. Slater https://orcid.org/0000-0002-0558-0521
Mario Jurić https://orcid.org/0000-0003-1996-9252
Andrew J. Connolly https://orcid.org/0000-0001-5576-8189
Eric C. Bellm https://orcid.org/0000-0001-8018-5348
V. Zach Golkhou https://orcid.org/0000-0001-8205-2506
Krzysztof Suberlak https://orcid.org/0000-0002-9589-1306

References

Armbrust, M., Xin, R. S., Lian, C., et al. 2015, in Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD'15, ed. T. Sellis et al. (New York: ACM), 1383
Bellm, E. C., Kulkarni, S. R., Graham, M. J., et al. 2019, PASP, 131, 018002
Bernard, E. J., Ferguson, A. M. N., Schlafly, E. F., et al. 2016, MNRAS, 463, 1759
Bonaca, A., Jurić, M., Ivezić, Ž., et al. 2012, AJ, 143, 105
Brahem, M., Lopes, S., Yeh, L., & Zeitouni, K. 2018, in Proc. 3rd ACM SIGSPATIAL PhD Symposium, ed. E. Hoel (New York: ACM), 3
Budavári, T., & Basu, A. 2016, AJ, 152, 86
Chang, F., Dean, J., Ghemawat, S., et al. 2006, in Proc. 7th USENIX Symp. Operating Systems Design and Implementation—Vol. 7, OSDI '06 (Berkeley, CA: USENIX Association), 15, http://dl.acm.org/citation.cfm?id=1267308.1267323
Databricks 2017, Introducing Pandas UDF for PySpark, https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html
Dean, J., & Ghemawat, S. 2004, in Proc. 6th Conf. on Symp. on Opearting Systems Design and Implementation—Vol. 6, OSDI'04 (Berkeley, CA: USENIX Association), 10, http://dl.acm.org/citation.cfm?id=1251254.1251264
Dobos, L., Budavari, T., Li, N., Szalay, A. S., & Csabai, I. 2012, in Scientific and Statistical Database Management, ed. A. Ailamaki & S. Bowers (Berlin: Springer), 159
Finkbeiner, D. P., Schlafly, E. F., Schlegel, D. J., et al. 2016, ApJ, 822, 66
Górski, K. M., Hivon, E., & Wandelt, B. D. 1999, in Evolution of Large Scale Structure: From Recombination to Garching, ed. A. J. Banday, R. K. Sheth, & L. N. da Costa (Garching: ESO), 37
Graham, M. J., Kulkarni, S. R., Bellm, E. C., et al. 2019, PASP, 131, 078001
Gray, J., Nieto-Santisteban, M. A., & Szalay, A. S. 2007, The Zones Algorithm for Finding Points-Near-a-Point or Cross-Matching Spatial Datasets MSR TR 2006 52
Gray, J., Szalay, A. S., & Thakar, A. R. 2002, Data Mining the SDSS SkyServer Database MSR TR 02 01
Gray, J., Szalay, A. S., & Thakar, A. R. 2004, There Goes the Neighborhood: Relational Algebra for Spatial Data Search MSR-TR-2004-32
Green, G. M., Schlafly, E. F., Finkbeiner, D. P., et al. 2015, ApJ, 810, 25
ISO 2011, **ISO/IEC 9075-1:2011** Information technology–Database languages–SQL–Part 1: Framework (SQL/Framework), https://www.iso.org/standard/53681.html
Jia, X., Luo, Q., & Fan, D. 2015, 2015 IEEE 21st Int. Conf. on Parallel and Distributed Systems (ICPADS) (Piscataway, NJ: IEEE), 617, https://ieeexplore.ieee.org/abstract/document/7384346
Jurić, M. 2010, Large Survey Database Website, http://lsddb.org
Jurić, M. 2011, BAAS, 43, 433.19
Jurić, M. 2012, LSD: Large Survey Database Framework, Astrophysics Source Code Library, ascl:1209.003
Juric, M., Axelrod, T., & Becker, A. C. 2016, Large Synoptic Survey Telescope (LSST) Systems Engineering Data Products De1nition Document LSE-163
Kunszt, P. Z., Szalay, A. S., & Thakar, A. R. 2001, in Mining the Sky, ed. A. J. Banday, S. Zaroubi, & M. Bartelmann (Berlin: Springer), 631
Law, N. M., Kulkarni, S. R., Dekany, R. G., et al. 2009, PASP, 121, 1395
Lomb, N. R. 1976, Ap&SS, 39, 447
LSST Science Collaboration, Abell, P. A., Allison, J., et al. 2009, arXiv:0912.0201
Marrese, P. M., Marinoni, S., Fabrizio, M., & Giuffrida, G. 2017, A&A, 607, A105
Nieto-Santisteban, M. A., Thakar, A. R., & Szalay, A. S. 2007, National Science and Technology Council (NSTC) NASA Conference, https://esto.nasa.gov/conferences/nstc2007/papers/Nieto-Santisteban_Maria_A10P2_NSTC-07-0074.pdf
Parquet Project 2018, http://parquet.apache.org/documentation/latest/
Pérez, F., & Granger, B. E. 2007, CSE, 9, 21
Rocklin, M. 2015, in Proc. 14th Python in Science Conf. No. 130–136, Citeseer, ed. K. Huff & J. Bergstra (Austin, TX: SciPy), 126
Scargle, J. D. 1982, ApJ, 263, 835
Schlafly, E. F., Finkbeiner, D. P., Jurić, M., et al. 2012, ApJ, 756, 158
Schlafly, E. F., Green, G., Finkbeiner, D. P., et al. 2014, ApJ, 789, 15
Schlafly, E. F., Green, G., Finkbeiner, D. P., et al. 2015, ApJ, 799, 116
Schlafly, E. F., Green, G. M., Lang, D., et al. 2018, ApJS, 234, 39
Schlafly, E. F., Peek, J. E. G., Finkbeiner, D. P., & Green, G. M. 2017, ApJ, 838, 36
Sesar, B., Cohen, J. G., Levitan, D., et al. 2012, ApJ, 755, 134
Sesar, B., Grillmair, C. J., Cohen, J. G., et al. 2013, ApJ, 776, 26
Silva, Y. N., Aref, W. G., & Ali, M. H. 2010, in 2010 IEEE 26th Int. Conf. on Data Engineering (ICDE 2010) (Piscataway, NJ: IEEE), 892
Soumagnac, M. T., & Ofek, E. O. 2018, PASP, 130, 075002
Tian, H.-J., Gupta, P., Sesar, B., et al. 2017, ApJS, 232, 4
Vanderplas, J. 2015, gatspy: General Tools for Astronomical Time Series in Python, Zenodo, doi:10.5281/zenodo.14833
Wang, D. L., Monkewitz, S. M., Lim, K.-T., & Becla, J. 2011, in State of the Practice Reports, SC '11, ed. S. Lathrop et al. (New York: ACM), 12:1
Zaharia, M., Xin, R. S., Wendell, P., et al. 2016, Commun. ACM, 59, 56