UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MEJORAMIENTO DE UN SERVICIO DE CROSSMATCHING ASTRONÓMICO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

BENJAMÍN EDUARDO CORREA KARSTULOVIC

PROFESOR GUÍA:
AIDAN HOGAN

PROFESOR CO-GUÍA:
FRANCISCO FÖRSTER BURÓN

MIEMBROS DE LA COMISIÓN:
DIONISIO GONZÁLEZ GONZÁLEZ
BENJAMÍN BUSTOS CÁRDENAS

SANTIAGO DE CHILE
2023

Resumen

ALeRCE (Automatic Learning for the Rapid Classification of Events) es un agente de alertas astronómicas chileno encargado de procesar y clasificar alertas provenientes del *Zwicky Transient Facility* (ZTF) y próximamente del *Legacy Survey of Space and Time* (LSST). Estas alertas corresponden a cambios en la naturaleza astrofísica de objetos detectados en el cielo. El objetivo principal de ALeRCE es estudiar tres categorías de objetos: transitorios, estrellas variables y núcleos galácticos activos.

El proceso de *crossmatching* se utiliza para filtrar e identificar los datos que provienen de las alertas astronómicas; este proceso consiste principalmente en buscar objetos en uno o más catálogos astronómicos cruzando los datos de los catálogos con los del objeto en cuestión. Actualmente, ALeRCE utiliza un servicio externo perteneciente al *Centre de Données astronomiques de Strasbourg* (CDS) para realizar el *crossmatching*, el servicio se llama CDS X-Match, pero al no ser un servicio propio de ALeRCE, el uso de CDS X-Match limita el control sobre los catálogos a los cuales se puede consultar y además sobre la disponibilidad general del sistema. Esta falta de control puede llegar a afectar la capacidad de ALeRCE de procesar alertas de forma efectiva, y motiva la búsqueda de alternativas. En este trabajo, se busca una solución que permita garantizar un mejor control sobre los datos y la disponibilidad.

Siguiendo la idea del párrafo anterior, en este trabajo se propone la implementación de un servicio de *crossmatch*. El servicio está compuesto por una API que conecta una aplicación en Spark con los catálogos pre-indexados. El catálogo usado en este trabajo se llama CatWISE2020. La API permite realizar consultas HTTP, para así poder realizar *crossmatch* entre un set de coordenadas con un radio y el catálogo en cuestión. El trabajo busca satisfacer las necesidades de ALeRCE, mejorando así el control y eficiencia sobre su *pipeline* de datos al utilizar Spark con Sedona como *backend* e índices como los Geohashes y Google S2.

Según los experimentos realizados para comprobar la correctitud, el sistema desarrollado tiene un gran margen de error respecto a CDS, en declinaciones cercanas el Ecuador se tiene alrededor de un 1 % de error, y cerca de los polos aumenta a más de un 96 % de error. Se observó que para conjuntos pequeños de coordenadas CDS X-Match era mucho más eficiente que el sistema propio, pero para consultas más grandes la brecha de tiempo tendía a empequeñerse. A pesar de desarrollar una solución completa en Sedona, los resultados obtenidos no cumplen con los requisitos para la integración del servicio de *crossmatch* en la *pipeline* de ALeRCE, por lo que es necesario realizar ajustes para compensar las limitaciones de Sedona, tanto para mejorar los tiempos como para mejorar la precisión.

"The universe is a pretty big place. If it's just us, seems like an awful waste of space."
—Carl Sagan

Agradecimientos

A mi mamá Gerka por siempre insistir en estudiar y no andar flojeando (procrastinando). A mis abuelos Pepe y Meme por recibirme en su hogar mientras estudiaba en Santiago, pero sobre todo quiero destacar al Pepe, ya que con él siempre hacíamos algún trabajo que se inventaba los fines de semana (casi siempre era cambiar un foco o apretar un tornillo en alguna puerta) hasta un punto en que yo creía que el mismo echaba las cosas a perder para que tuviéramos algo que hacer el fin de semana.

Gracias a Jeannette por todas esas conversaciones y copuchas que teníamos, y también los dulces que ella hacía para subirme el ánimo y engordarme.

Quiero agradecer al Richi, que me convenció de hacer un cambio de carrera del DII al maravilloso DCC y con quien jugábamos hasta altas horas de la mañana (o bajas(?)). Él me ayudaba a estudiar y a comprender las cosas que a veces no entendía en ciertos ramos del DCC.

Al Javier con quien siempre jugábamos a cualquier hora y cualquier día a los juegos más extraños y escondidos que existían, pero sobre todo a esas tardes y madrugadas que jugábamos tanto el ROR1 como el ROR2, y en donde siempre terminaba yo salvaba el juego pues él moría todo el tiempo.

Gracias a Aidan y Francisco, quienes me brindaron su mentoría a lo largo de este proyecto, por responder mis dudas y preocupaciones, por apoyarme cuando pasé por tiempos difíciles durante la escritura de este trabajo, y por facilitarme el servidor para poder realizar el trabajo. Quiero agradecer a Kay, con quien siempre conversábamos y quien me ayudó a crear los conjuntos de datos para hacer los experimentos.

Al grupo de "La Pera" (actualmente se llama FRIENDS), quienes fueron mis primeros amigos que hice en la universidad y con quienes pasé grandes y memorables momentos.

Quiero agradecer a la firma en donde trabajo McKinsey & Company, que es en donde encontré un lugar seguro y me han apoyado cuando pasé por momentos difíciles. También fueron quienes me ayudaron a ser un mejor profesional y crecer como persona. A Matheus quien es mi mentor dentro de la firma, y me ha ayudado mucho durante mi corta trayectoria en esta consultora.

Por último quiero agradecer a Belén, quien ha sido mi compañera en esta trayectoría universitaria, es quien me ha apoyado y confiado en mí durante todo este proceso.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Motivación

La astroinformática surgió aproximadamente en la década de 1990 gracias al proyecto de observación del espacio *Sloan Digital Sky Survey* (SDSS) diseñado por el astrofísico James Gunn en conjunto con otros científicos y llevado a cabo en el observatorio Apache Point en Nuevo México, Estados Unidos. Este proyecto tenía la capacidad de sondear un tercio de la esfera celeste, obteniendo así información de miles de millones de objetos, tales como, estrellas, galaxias y agujeros negros supermasivos en lugares distantes del espacio. Con esto se marcó un hito importante en la astronomía, expandiéndose hacia el campo del *big data* debido a que el SDSS recolecta cerca de 200 GB de datos por noche, los cuales se agregan a una base de datos que contiene actualmente cerca de 50 TB [5].

La astroinformática se enfoca principalmente en desarrollar herramientas, métodos y aplicaciones de ciencia de la computación, ciencia de datos, aprendizaje de máquinas y estadística para la investigación y educación en la astronomía orientada a los datos. En esta área se abordan además los 4 desafíos del *big data*, los cuales son:

1. Volumen: Se genera una gran cantidad de información por noche en los observatorios distribuidos alrededor del mundo.
2. Velocidad: En muchas ocasiones se deben tomar decisiones en tiempo real, por lo que existe un desafío adicional, el cual es procesar un alto flujo de alertas de telescopios. Es decir, procesar un alto flujo de eventos astrofísicos, tales como, supernovas, núcleos activos de galaxias, estrellas variables, asteroides, y otros.
3. Variedad: Existen muchos catálogos de objetos astronómicos que se generan y alimentan a partir de distintos proyectos de observatorios; a lo largo del tiempo, pueden existir varios registros de un mismo objeto.
4. Veracidad: Se debe asegurar que los elementos hayan sido clasificados correctamente una vez buscados en un catálogo astronómico. Es decir, tener en cuenta que existe ruido en las observaciones y que puede afectar a las consultas, sumado a esto, los márgenes

de error generados por aproximaciones también se deben tener en cuenta al momento de realizar estudios.

Sumado a lo anterior se encuentra la llegada de una nueva generación de telescopios, los cuales producen un flujo mayor de datos; por ejemplo, Atacama Large Millimeter Array (ALMA) produce alrededor de 1 terabyte de datos diariamente[1]. Además es conocido que Chile es una potencia mundial en cuanto a estudio de la astronomía, ya que el desierto de Atacama provee un ambiente idóneo para la observación. Otro desafío es poder captar cambios en el cielo asociados a una gran variedad de fenómenos astrofísicos y poder analizarlos en tiempo real antes que estos dejen de ser visibles[2].

Debido a las razones anteriores es que ha surgido la necesidad de crear agentes de alertas astronómicas, y así es como surge el *Automatic Learning for the Rapid Classification of Events* (ALeRCE) [7], el cual es un agente de alertas astronómicas chileno diseñado para proveer una clasificación de largos flujos de alertas de telescopios, tales como los proveídos por el *Zwicky Transient Facility* (ZTF), y en el futuro, el *Legacy Survey of Space and Time* (LSST) del observatorio Vera C. Rubin en el norte de Chile.

Dentro de la línea de 11 procesos que realiza ALeRCE se encuentra el *crossmatching* de cuerpos astronómicos, que consiste en encontrar pares de coincidencias entre dos o más catálogos correspondientes a un mismo objeto[3]. Para esto se realiza una búsqueda proyectando un cono con centro en la posición de cada punto y se localizan todos los puntos en el radio cercano; este proceso es llamado *conesearch*.

Actualmente, uno de los problemas que enfrenta ALeRCE es el uso de servicios externos, por ejemplo, el servicio de X-Match del Centre de Données astronomiques de Strasbourg (CDS) [8] como parte de su cadena de procesamiento de los datos, lo cual puede presentar inconvenientes, tales como, la interrupción del servicio externo y la limitación de los datos disponibles.

Es por esto que ALeRCE decidió generar su propio servicio de *crossmatching* [1], servicio que tenga una alta disponibilidad, sea capaz de procesar las alertas de ZTF y LSST, y además pueda realizar *crossmatching* de estas alertas sobre varios catálogos, un proceso que involucra encontrar coincidencias de cuerpos astronómicos entre dos fuentes mediante distintos algoritmos de búsqueda, por ejemplo, mediante *conesearch*. Actualmente el sistema propio es capaz de realizar *crossmatching*, pero uno de los problemas principales de éste es que el rendimiento es inferior al que se requiere y cuenta con un pequeño margen de error, por lo que se necesita mejorar dicho servicio tanto en precisión como en velocidad de procesamiento.

[1] `https://www.almaobservatory.org/en/factsheet/`

[2] `https://www.nlhpc.cl/alerce-un-broker-astronomico-chileno-nuevas-herramientas-para-comprender-el-universo-dinamico/`

[3] `https://ipg.fer.hr/ipg/resources/astronomical_cross-matching`

1.2. Objetivos

El objetivo general del trabajo es poder mejorar de forma general el servicio de *cross-matching* implementado por el broker ALeRCE, para que logre cumplir con sus requisitos, es decir, que sea escalable, certero y competitivo en términos de tiempos de búsqueda respecto a la solución actual que es CDS, pero que además esté bajo el control del equipo de ALeRCE y provea la posibilidad de indexar catálogos nuevos.

Para lograr el objetivo general, se consideran varias alternativas para implementar *cone-search*, las cuales están todas basadas en Apache Spark [22], excepto la solución realizada en un trabajo de memoria anterior [1], la cual se basa en MongoDB. En este contexto, los objetivos específicos son:

1. Elegir un motor de consultas geoespaciales que pueda resolver consultas relacionadas con un punto y radio de forma eficiente y a gran escala.
2. Comprender la estructura de CatWISE2020 y desarrollar funciones de preprocesamiento que faciliten la creación de *hashes* o indexación utilizando diferentes métodos y múltiples niveles de profundidad.
3. Diseñar y crear funciones para un *pipeline* que realice las consultas en cada motor.
4. Analizar los distintos métodos de consulta y hacer pruebas de rendimiento para obtener los mejores métodos y técnicas de indexación sobre cada motor utilizando un conjunto de muestra.
5. Crear una API con los respectivos *endpoints* para consultar cada método.
6. Procesar la base de datos completa para poder evaluar la correctitud de cada método utilizando CDS como referencia.
7. Desarollar una metodología para comparar velocidad y correctitud entre los distintos métodos.
8. Comparar y analizar los resultados obtenidos entre los distintos métodos y verificar que se alcance un mínimo de 350 consultas por segundo para un radio de 1 arcosegundo.

1.3. Estructura del documento

Este documento cuenta con 7 capítulos, en conjunto con una Tabla de Contenidos e índices para las Tablas y Figuras, con el fin de ayudar a comprender el documento y el análisis de los resultados obtenidos.

En el Capítulo 1, el cual corresponde al capítulo actual, se presentó las motivaciones y los objetivos para el estudio del área, lo que permite contextualizar el problema.

En el Capítulo 2 se detallan los antecedentes, conceptos y tecnologías necesarias para comprender el problema y la solución.

En el Capítulo 3 se resume el diseño y describe el razonamiento detrás de las decisiones tomadas, además se habla sobre el *pipeline* desarrollado y la arquitectura de la API.

En el Capítulo 4 se habla en detalle sobre el preprocesamiento que se realizó para preparar los datos de CatWISE2020, además se profundiza sobre los formatos de archivos seleccionados y su estructura.

En el Capítulo 5 se explica de forma detallada la implementación de la API, abordando así el formato de las *requests* y sus parámetros, y su formato de salida. También se proporcionan ejemplos de cómo se puede consultar a la API y cómo funciona el *pipeline* para hacer las consultas.

En el Capítulo 6 se indican específicamente los experimentos realizados, en conjunto con los datos utilizados en cada uno. Además se muestran y explican los resultados obtenidos utilizando tablas y gráficos para comparar los resultados.

En el Capítulo 7 se da conclusión al trabajo realizado en esta memoria. Se describen los objetivos cumplidos, y los que no se lograron completamente. Sumado a esto se sugieren posibles opciones para realizar trabajos futuros basados en el desarrollo de esta memoria.

Capítulo 2

Estado del Arte

En este capítulo se presentan los temas relevantes para comprender la solución que se desarrolla a lo largo de este trabajo y para entender el problema abordado.

2.1. CDS X-Match

CDS X-Match es el servicio externo del cual ALeRCE depende actualmente para el proceso de *crossmatch*, el cual a su vez es dependiente del Centro de Datos Astronómicos de Estrasburgo (*Centre de Données astronomiques de Strasbourg*, o CDS). A través de una interfaz web, este servicio permite a los usuarios utilizar algoritmos para cruzar datos astronómicos de una manera eficiente con catálogos o listas de objetos muy grandes (con cerca de mil millones de filas). Este servicio cuenta con tres sub-servicios:

- VizieR [13]: Servicio de catálogos de objetos astronómicos obtenidos de distintos *surveys*. El agente de alertas trabaja específicamente con el Wide-field Infrared Survey Explorer (WISE) [20], una misión de la NASA lanzada el año 2010 que escaneó el cielo completo, obteniendo imágenes de 750.000.000 de objetos, dentro de los cuales se encuentran galaxias, estrellas y asteroides.
- SIMBAD [19]: Base de datos que suministra datos básicos, identificaciones cruzadas, bibliografía y medidas para objetos astronómicos.
- Datos subidos por los usuarios[1]: Se pueden subir tablas como VOTable[2], FITS[3] o CSV a través de la interfaz web.

CDS tiene a disposición varias implementaciones para Hierarchical Equal Area isoLatitude Pixelisation (HEALPix) [10], el cual es un algoritmo para dividir una esfera en secciones de

[1] `http://cdsxmatch.u-strasbg.fr/xmatch/doc/table-upload.html`

[2] Formato basado en XML respaldado por la International Virtual Observatory Alliance

[3] Flexible Image Transport System, permite el almacenamiento, transmisión y procesamiento de datos, con formato de arreglos multi-dimensionales (una imagen) o tablas

igual área, y un buscador de catálogos para VizieR en su cuenta de Github[4]. No obstante HEALPix utiliza algoritmos de *crossmatching* para consultas por secciones y no para el cielo completo; esto significa que se tiene que determinar la resolución de una celda y despúes indicar el índice de esta para hacer el *crossmatch*, según lo descrito en la documentación del servicio[5].

Los principales problemas de utilizar el servicio CDS X-Match son:

- Restricción de tiempo: El sistema solo deja hacer consultas que tarden menos de 120 minutos en ejecutarse, de lo contrario arroja un error.
- Control sobre los datos: No se tiene control sobre los datos de VizieR, es decir, no se pueden modificar o actualizar. Además se desconoce la frecuencia con la cual se actualizan los datos almacenados en VizieR, el cual depende de CDS.
- Disponibilidad: Debido a que es un servicio alojado en otro país, no se puede manipular eficazmente si el sistema se cae o si llega a existir algún tipo de inconveniente, lo cual paraliza el funcionamiento del *pipeline*.
- Costos: Debido a que es un servicio externo y además se encuentra en otro país puede llegar a ser muy costoso en términos de red ya que es un servicio en tiempo real y con demanda.

2.2. Índices espaciales

Los índices espaciales se utilizan en procesos como el *crossmatching*. Éste es un proceso que consiste en identificar un objeto dentro de uno o más catálogos, los cuales se generan a través de diferentes sondeos astronómicos. El *crossmatching* consiste en cruzar la información contenida en las listas según las coordenadas aproximadas del cuerpo celeste; este cruce permite a los astrónomos realizar un seguimiento de los objetos, facilitando el análisis de las variaciones en sus características a lo largo del tiempo.

Para realizar el *crossmatch* se utilizan consultas de *conesearch*. Para realizar el *conesearch* de manera más eficiente y rápida, se pueden utilizar *hashes* espaciales, los cuales son columnas de una tabla que utilizan estructuras de datos con métodos de búsqueda eficientes. Dentro de los *hashes* espaciales, existen dos opciones de especial relevancia: Geohash [17] y S2 [14]. Estos *hashes* dividen la tierra en celdas con un *hash* único determinado por las Latitudes y Longitudes geográficas; Geohash trabaja dividiendo un plano en 2 dimensiones, mientras que S2 divide una esfera. En ambos casos la Latitud y la Longitud corresponden a las coordenadas ecuatoriales utilizadas en astronomía, Latitud como Declinación y Longitud como Ascensión Recta.

Geohash es una estructura de datos espaciales que subdivide el espacio en celdas con forma de grilla y asigna un identificador único en base32 a una celda utilizando un alfabeto

[4] `https://github.com/cds-astro`
[5] `http://cdsxmatch.u-strasbg.fr/xmatch/doc/xmatch-area.html`

propio como se ve en la Figura 2.1. Esta división es recursiva y por cada nivel de profundidad el plano es dividido en 32 partes como se ve en la Figura 2.2. Las divisiones son de forma recursiva hasta una profundidad máxima de 12.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

Figura 2.1: Alfabeto en base 32 de Geohash. Fuente: `https://en.wikipedia.org/wiki/Geohash`.

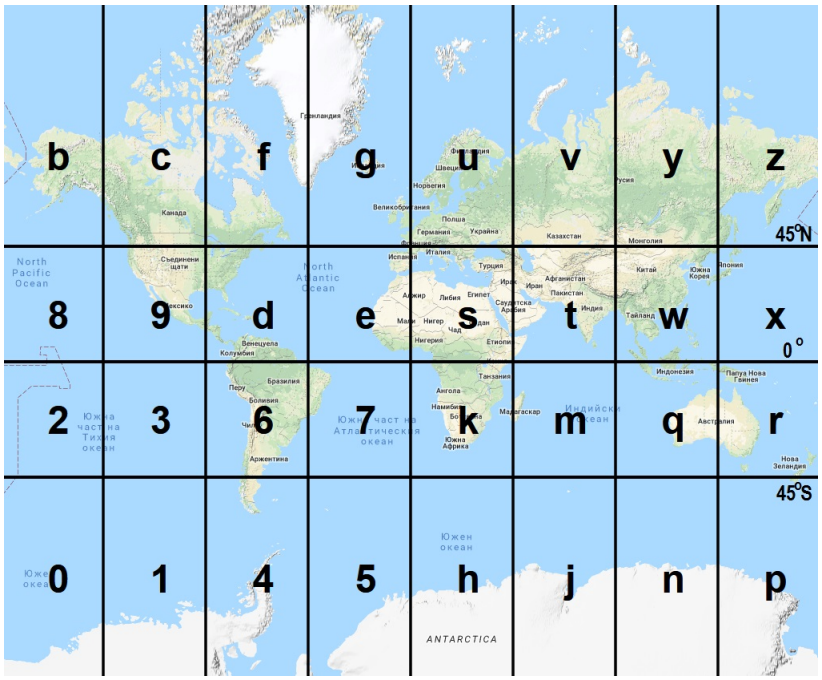


Figura 2.2: División en celdas con identificadores únicos utilizando una profundidad de 1 en Geohash. Fuente: `https://petrov.free.bg/academic/publication/geohash-eas-modified-geohash-geocoding-system-equal-area-spaces/`.

Por otro lado, Google S2[6] divide la Tierra utilizando líneas geodésicas a partir de la proyección de un cubo sobre la esfera terrestre como se ve en la Figura 2.3. Luego puede tener hasta 31 niveles de profundidad, los cuales conllevan a más subdivisiones, como se ve en la Figura 2.4. Cada cara del cubo proyectado contiene una curva de Hilbert[7] en su interior, quedando así en total 6 curvas de Hilbert unidas por los extremos, generando así una única curva que atraviesa toda la esfera; por cada subdivisión (nivel extra de profundidad) se divide cada "cuadrado" en 4 más pequeños y se agrega resolución a la curva.

[6] `https://s2geometry.io/devguide/s2cell_hierarchy.html`.
[7] `https://en.wikipedia.org/wiki/Hilbert_curve`

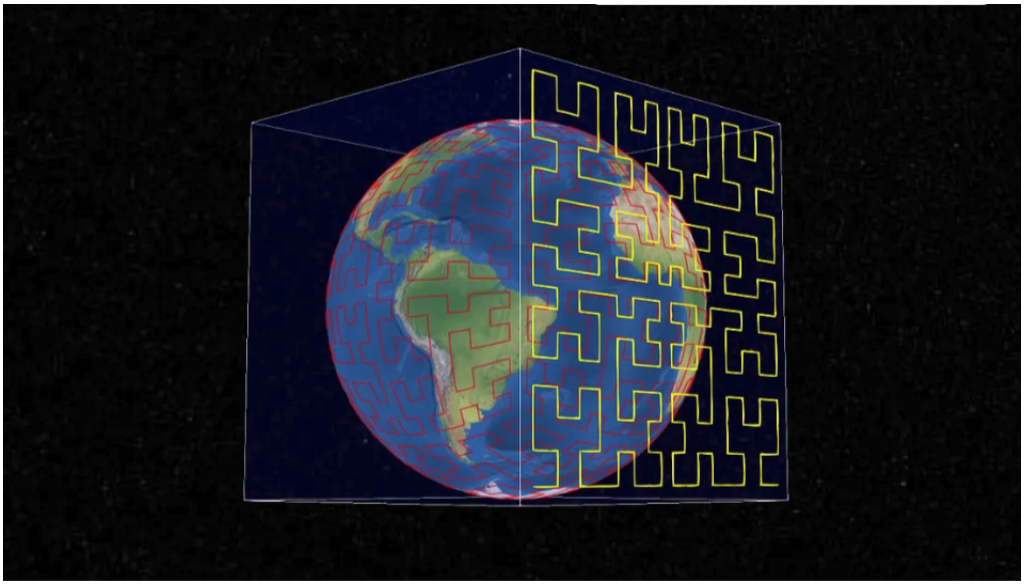


Figura 2.3: Proyección del cubo sobre la esfera terrestre, con una curva de Hilbert en cada cara del cubo. Fuente: `https://medium.com/sidewalk-talk/s2-cells-and-space-filling-curves-keys-to-building-better-digital-map-tools-for-cities-a312aa5e2f59`.

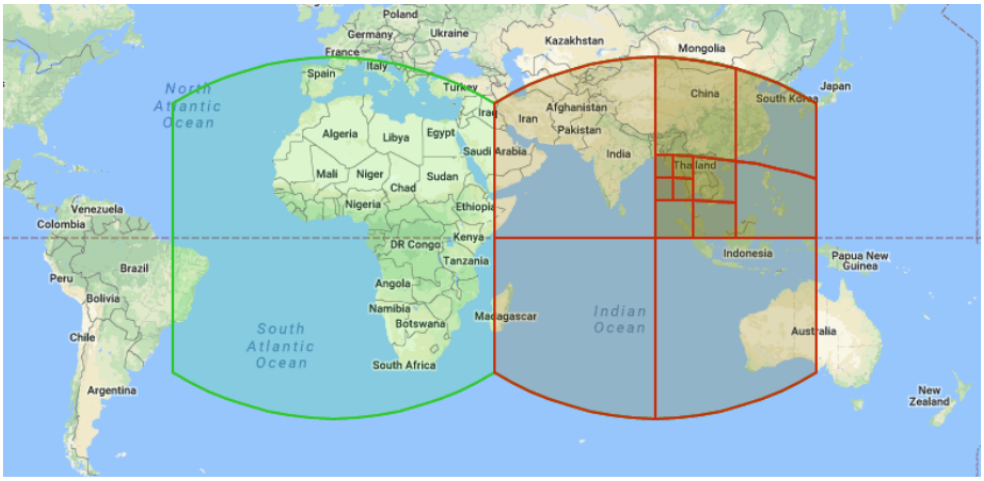


Figura 2.4: División en celdas utilizando 4 líneas geodésicas. Fuente: `https://s2geometry.io/devguide/s2cell_hierarchy.html`.

Esto permite a Google S2 maximizar la cercanía de referencias[8], lo que posibilita un mejor uso de los recursos junto con mejorar la indexación espacial.

2.3. AXS

Astronomy eXtensions for Spark (AXS) [23] es un framework escalable y *open-source* para el análisis de datos astronómicos, hecho en Apache Spark [22], un motor usado ampliamente

[8] `https://en.wikipedia.org/wiki/Locality_of_reference`

en la industria para el procesamiento masivo de datos. AXS permite realizar todo el procesamiento desde un mismo framework de trabajo, es decir, simplifica el proceso de cargar catálogos, pre-procesarlos y las posteriores operaciones sobre estos datos, sin la necesidad de migrar a otro sistema.

AXS es un *wrapper* que funciona sobre Spark SQL [2], por lo que hereda características importantes de Spark, las cuales son[9]:

1. Los *AxsFrames* y *DataFrames* se trabajan de una forma *lazy*, por lo que son evaluados y ejecutados solo al momento de ser llamados en una acción, como un *show* o un *count*.
2. Los *DataFrames* son inmutables. Cuando se realiza una transformación sobre ellos como un *select* o un *where* se obtiene un nuevo *DataFrame*.

Sumado a lo anterior, AXS hereda de Spark la partición de los datos, solo que en este caso el cielo se particiona en forma de rectángulos horizontales con una altura definida; estos rectángulos se llaman Zonas. Los datos son particionados en bloques de archivos Parquet, los cuales contienen parte de los datos. Las zonas se colocan secuencialmente dentro de los bloques y un bloque tiene siempre varias zonas que no son vecinas, ordenadas por la *declination* (DEC) y luego por la *right ascension* (RA), las cuales son coordenadas que definen un punto específico en la esfera terrestre[10]. Esto permite que AXS pueda hacer una búsqueda espacial eficiente y un *crossmatching* rápido entre diferentes catálogos.

AXS además cuenta con 4 métodos específicos para hacer búsqueda de objetos, los cuales son:

- *region*: Busca elementos dentro de un cuadrado, el cual se especifica con 2 puntos en el cielo, con sus coordenadas RA y DEC.
- *cone*: Realiza una búsqueda de elementos dentro de un cono, del cual se entrega un punto central y un radio.
- *histogram*: Dada una función específica, los objetos se agrupan según corresponda conforme el valor resultante de dicha función.
- *histogram2d*: Realiza lo mismo que *histogram* pero en dos dimensiones.

Los principales problemas de utilizar AXS son:

1. Se encuentra en una versión temprana de desarrollo, y no ha sido actualizada.
2. No se han hecho pruebas de velocidad con el método de búsqueda *cone*.
3. Algunos de los cálculos que tiene están incorrectos.
4. Su implementación es difícil, ya que no cuenta con documentación suficiente sobre las dependencias.

[9] `https://axs.readthedocs.io/en/latest/introduction.html`
[10] `https://voyages.sdss.org/es/preflight/locating-objects/ra-dec/`

2.4. MongoDB

En un trabajo de memoria previo, se desarrolló un sistema de *crossmatching* en ALeRCE que está implementado en Python, utilizando MongoDB [1], ya que éste permite hacer búsquedas geográficas similares a las de *conesearch* con solo unos ajustes.

Para este sistema se implementó primero una función que aplica una conversión de unidades utilizando los parámetros de WGS 84[11] considerando que la Tierra se asemeja más a un elipsoide que a una esfera.

Para el *conesearch* se aprovecha la presencia de los índices espaciales para optimizar la búsqueda con GeoJSON[12]; para esta búsqueda se utiliza además el operador *maxDistance*, el cual utiliza metros como unidad de medida, por lo que se debe aplicar la función anterior para transformar el radio de búsqueda a metros.

Este sistema cuenta con una API con los siguientes métodos:

- */catalog*: Retorna una lista con los nombres de los catálogos.
- */crossmatch*: Recibe el radio del cono a buscar y un archivo de texto con las coordenadas de los puntos a buscar; opcionalmente puede recibir los parámetros *limit* para limitar la cantidad de objetos retornados de la búsqueda y *catalog* para definir en qué catálogo se hará la búsqueda. Retorna el tiempo que tardó en buscar, el tiempo que tardó en generar el archivo de resultado, la cantidad de objetos encontrados y finalmente los objetos.

Los problemas de utilizar esta solución son los siguientes:

1. Actualmente solo funciona con CatWISE.
2. Los tiempos de consulta son mejores que CDS X-Match para lotes pequeños, pero para consultas más grandes el sistema tiende a empeorar los tiempos de consulta.
3. Cuenta con una diferencia en los objetos devueltos, que ronda el 1 % respecto con CDS X-Match.

2.5. Apache Sedona

Apache Sedona [21] es un framework que extiende los sistemas de computación distribuida Apache Spark y Apache Flink, añadiendo *Spatial Datasets* y *Spatial SQL*, los cuales de forma eficiente y distribuida, cargan, procesan y analizan grandes volúmenes de datos espaciales a través de las máquinas. En la Figura 2.5 se puede apreciar la arquitectura de Apache Sedona;

[11] `https://confluence.qps.nl/qinsy/latest/en/world-geodetic-system-1984-wgs84-29855173.html`

[12] `https://www.mongodb.com/docs/manual/geospatial-queries/#geojson-objects`

se puede ver que esta es capaz de utilizar múltiples fuentes de datos y formatos, además de tener soporte tanto con Spark, como con Apache Flink, que es bastante parecido a Spark, solo que éste puede trabajar con *streaming* de datos de forma nativa.

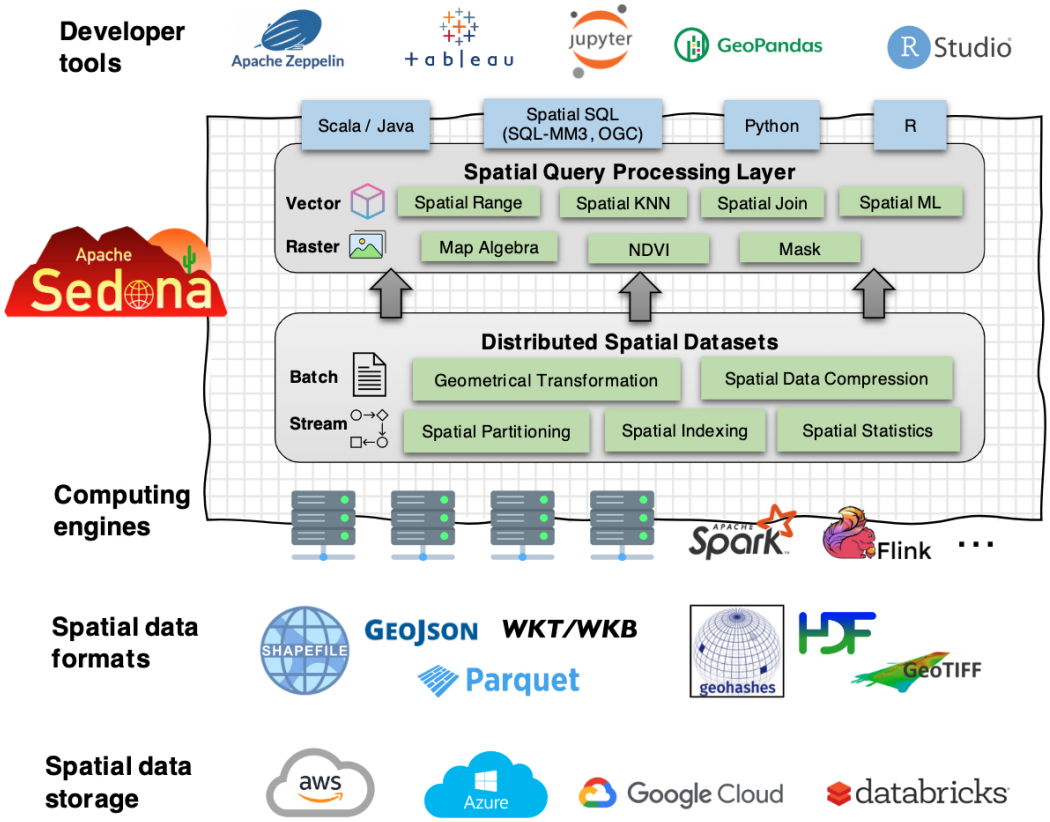


Figura 2.5: Arquitectura de Apache Sedona. Fuente: `https://sedona.apache.org/`.

Sedona es más rápido y utiliza menos memoria en general en comparación a otras soluciones de análisis espacial en Spark, esto gracias a que cuenta con 3 capas principales:

1. *Apache Spark Layer*: Permite realizar las funcionalidades básicas de Spark como cargar o guardar datos, además de poder ocupar todas las operaciones de los *RDD*[13].
2. *Spatial RDD Layer*: Es una extensión de los *RDD* de Spark que puede soportar objetos espaciales y geométricos; con esta extensión se pueden hacer operaciones geométricas entre los *SRDDs*[14], por ejemplo, intersecciones o superposiciones.
3. *Spatial Query Layer*: Esta capa se encarga de ejecutar algoritmos de procesamiento de consultas espaciales, tales como *Spatial Range*, *Join* y *KNN query* en los *SRDDs*.

Apache Sedona también permite crear índices espaciales usando las estructuras *R-tree* y *Quad-tree*, los cuales mejoran considerablemente el procesamiento de datos espaciales en cada partición de los *SRDD*.

[13]Resilient Distributed Dataset
[14]Spatial Resilient Distributed Datasets

Con *SRDD* se pueden hacer las operaciones mencionadas en la *Spatial Query Layer*, además de poder crear los índices distribuidos.

Las dos APIs principales para interactuar con Apache Sedona utilizando PySpark son:

1. SQL: Se utilizan funciones espaciales de Sedona para SQL, las cuales permiten hacer operaciones de *join* y filtrar.
2. RDD (core): Se utilizan funciones y métodos de *SRDDs* para crear, hacer consultas y construir los índices.

Los principales beneficios de utilizar Apache Sedona son:

1. Es capaz de distribuir los índices espaciales automáticamente a través de las particiones de los SRDDs.
2. Cuenta con la implementación de dos índices(Geohash y S2).
3. Es de fácil instalación.
4. Permite tanto hacer consultas con figuras, como con los vecinos más cercanos.
5. Está siendo constantemente actualizado.

Los principales problemas de utilizar Apache Sedona son:

1. Se encuentra en una versión temprana de desarrollo.
2. No tiene una implementación específica para el *conesearch*.
3. En caso de utilizar *SRDDs* hay que transformar las coordenadas ICRS[15] a WGS84, ya que utiliza distancia euclidiana para los cálculos.
4. Es difícil de probar, ya que está muy optimizado para utilizarse de forma distribuida, por lo que es bastante más lento cuando se prueba sin un cluster.

2.6. Astroide

Astroide [3], que proviene de *ASTROnomical In-memory Distributed Engine*, es un software de código abierto, distribuido bajo la licencia GPL V2. Está bajo el soporte de la *University of Versailles Saint-Quentin (UVSQ)* y el *Centre National d'Etudes Spatiales*.

Astroide es un servidor de datos distribuidos creado para manejar grandes volúmenes de datos astronómicos y enfocado en procesamiento y consultas de datos. Éste está diseñado como una extensión de Apache Spark y toma en consideración las peculiaridades de los datos y consultas de los catálogos astronómicos y las *surveys*.

Astroide contiene las siguientes extensiones sobre Spark:

[15]International Celestial Reference System

- Soporte de indexación y partición de datos astronómicos. Combina la partición de datos con la técnica de indexación adaptada a datos astronómicos HEALPix, con el fin de poder disminuir el tiempo de consulta.
- Operadores astronómicos. Propone una forma eficiente y escalable de algoritmos de *conesearch*, *KNN search*, *crossmatch* y *KNN join*.
- Acceso a datos de alto nivel. Soporta acceso a datos y consultas usando el lenguaje de consultas astronómicas más común: *Astronomical Data Query Language (ADQL)*[16].
- Optimización de consultas espaciales.

Los beneficios de Astroide son:

- Cuenta con un *wrapper* en Python para hacer *conesearch*.
- Soporta indexación espacial con HEALPix.

Las desventajas de Astroide son:

- La última actualización fue el 2019, por lo que puede estar desactualizado.
- Utiliza una versión antigua de Spark, HEALPix y ADQL.

[16] `https://www.ivoa.net/documents/ADQL/`

Capítulo 3

Solución

En este capítulo se explica de forma general el problema que motiva este trabajo y la solución propuesta junto con las tecnologías utilizadas.

3.1. Descripción del problema

ALeRCE actualmente utiliza una API (*Application Program Interface*, o interfaz de programación de aplicaciones) llamada CDS Xmatch para resolver consultas de *crossmatch* sobre catálogos existentes. Esta API depende de un centro de investigación ubicado en Francia, por lo que no se tiene control sobre los datos almacenados y la disponibilidad de la plataforma, tal como fue mencionado en la Sección 2.1.

3.2. Solución propuesta

La solución propuesta consiste en dejar ejecutando una API en un servidor local de ALeRCE; esta API debe ser capaz de ejecutar consultas, en específico, que se pueda realizar *conesearch* sobre el catálogo CatWISE2020 con distintos radios en arcosegundos o arcominutos. Además, la API debe recibir datos de las alertas de ZTF y en un futuro, de LSST, y proporcionar resultados necesarios para que sean utilizados en el resto del flujo del trabajo de ALeRCE. Por otro lado, además de la API, se desarrollarán funciones de preprocesamiento para el catálogo CatWISE2020. Por último se creará un archivo de CatWISE2020 optimizado para este catálogo y tipo de consultas. En el futuro, este procedimiento podrá replicarse para añadir nuevos catálogos a la API. Para comprobar que el código esté correctamente escrito y que cumpla con las normas PEP8, se utilizará `pre-commit` junto con los *hooks black*, *flake8*, *isort*, *trailing-whitespace*, *mixed-line-ending* y *check-added-large-files*. Además, se usará GitHub para el control de versiones. Por último, la solución estará debidamente probada tanto en rendimiento como en correctitud de los datos entregados; para esto se utilizará Pandas y Matplotlib con el fin de hacer los cálculos de rendimiento y poder graficar las comparaciones entre los distintos métodos de Sedona y CDS X-Match.

La implementación de la solución propuesta se divide en dos etapas principales, las cuales serán abordadas en detalle en los próximos capítulos.

La primera etapa consiste principalmente en el preprocesamiento de los datos; en esta etapa se analizará la estructura de los datos y se definirá una metodología para determinar cúal es la configuración óptima de cada método en términos de rendimiento y correctitud, utilizando una versión inicial de la *pipeline* y la API.

La segunda etapa se centrará en la consolidación de la API en conjunto con sus *endpoints* correspondientes. Se realizarán ajustes a las funciones de *crossmatch* en caso de ser requerido para poder alcanzar la correctitud necesaria y se realizarán comparaciones entre los métodos utilizando su mejor configuración.

Las herramientas que serán utilizadas durante el desarrollo de este trabajo son las siguientes:

- **Apache Spark**: Motor de computación distribuida. Se escogió por su capacidad de paralelizar los procesos, lo que permite optimizar el preprocesamiento de datos; éste es la base sobre la cual se utiliza Apache Sedona y tiene *plugins* para poder utilizarse con Python.
- **Apache Sedona**: Framework que extiende las funcionalidades de Apache Spark, añadiendo funciones para procesar y analizar grandes volúmenes de datos geoespaciales utilizando la API de *SRDDs* y la API de *SQL*; en este caso se escogió usar la API de *SQL*. Se eligió Sedona debido a su velocidad para realizar algoritmos de *join* utilizando geometrías; en este caso se utiliza para buscar puntos en CatWISE2020 usando geometrías circulares.
- **Docker**: Se utiliza Docker para empaquetar la aplicación y facilitar el despliegue y el desarrollo futuro en los servidores de ALeRCE. Docker se encarga de *containerizar* la aplicación junto a las dependencias de esta en un contenedor aislado; se utilizó ya que garantiza la portabilidad y la fácil replicación de la aplicación.
- **Python**: Lenguaje de programación de alto nivel. Se utiliza debido a su amplia gama de bibliotecas y su facilidad para la comprensión y ejecución del código. Durante el desarrollo del trabajo, se emplean las siguientes bibliotecas:
  - **PySpark**: Conector de Apache Spark para Python que permite utilizar todo el potencial de Apache Spark junto con la facilidad de ejecutar código en Python.
  - **Apache Sedona Spark**: Conector de Apache Sedona para utilizar con PySpark que permite hacer uso de las funciones de Apache Sedona sobre PySpark, permitiendo así hacer consultas y *joins* geoespaciales sobre dataframes de PySpark.
  - **FastAPI**: Framework web para desarrollo de APIs REST. Se optó por este por su facilidad y rapidez para implementar los *endpoints*.
  - **Requests**: Permite enviar solicitudes HTTP de manera sencilla y amigable; se utilizó para probar el funcionamiento de la API y realizar los experimentos durante el desarrollo de la solución.

- **Pandas**: Permite el procesamiento y manipulación de datos tabulares. Se empleó para retornar los resultados de las consultas de Sedona junto con la optimización de Apache Arrow, además de utilizarse para procesar y analizar los experimentos.
- **Apache Arrow**: Implementación de datos columnares en memoria que es agnóstico al lenguaje en el cual se utiliza y permite lecturas desde distintos lenguajes; en este caso se usó para obtener los resultados desde Spark en una tabla de Pandas de manera eficiente para ser retornados por la API.
- **Uvicorn**: Permite la creación de un servidor ASGI[1] y se escogió por su facilidad de implementar un servidor.

Es posible implementar la solución propuesta gracias a estas herramientas, las cuales proporcionan la base tecnológica que permite el procesamiento eficiente de los datos, acceso a los archivos y disponibilización de la API en el servidor local de ALeRCE. Su integración y correcto uso resultan cruciales para lograr los objetivos planteados en este trabajo; esto se puede ver en la Figura 3.1.

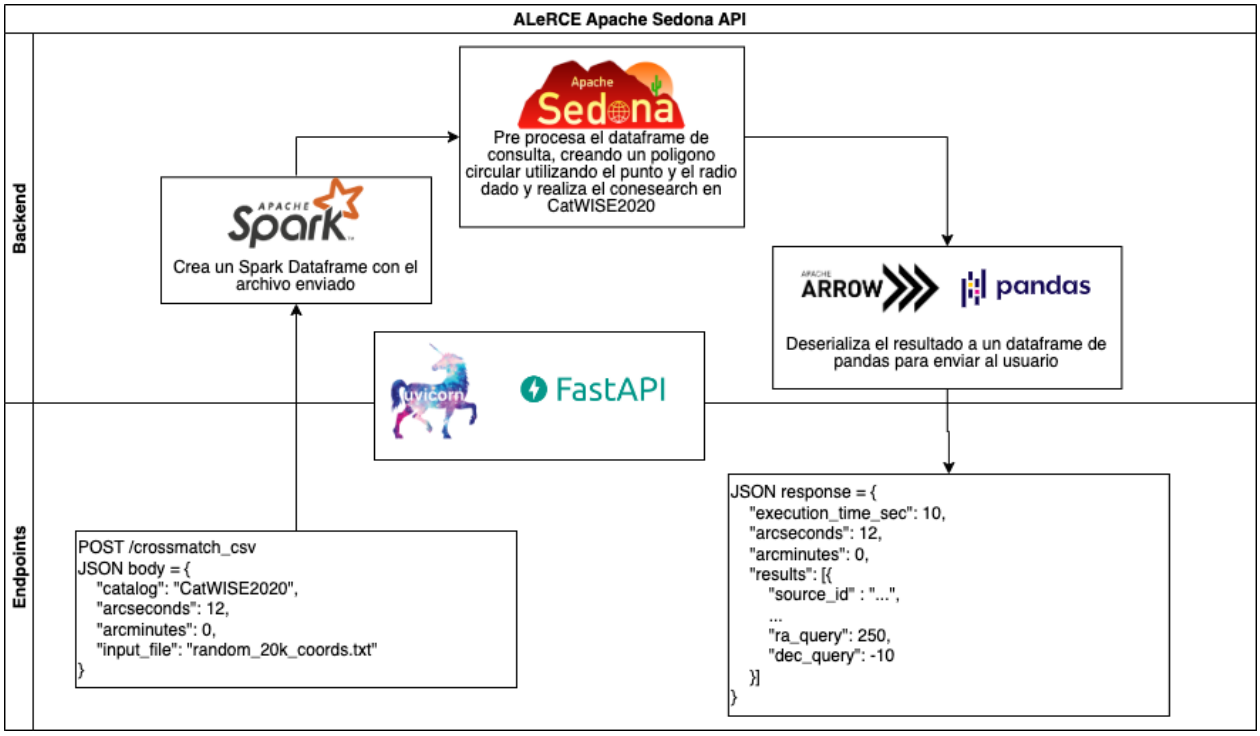


Figura 3.1: Diagrama de la arquitectura del servicio de crossmatch.

3.3. Trabajos previos

Se ha realizado previamente un trabajo por Alejandra Alarcón en su memoria titulada "Servicio de Crossmatching de Objetos Astronómicos" [1], donde desarrolló una API para

[1] *Asynchronous Server Gateway Interface:* `https://asgi.readthedocs.io/en/latest/`

abordar esta problemática utilizando MongoDB como sistema de base de datos junto con índices geoespaciales para mejorar la velocidad en las consultas.

Comparando la API con CDS se concluyó que la API desarrollada presenta un pequeño porcentaje de diferencia en los puntos devueltos como resultados, aproximadamente del 1 %, principalmente en los bordes de las consultas. Esta discrepancia puede resolverse ampliando los radios de las consultas para incluir los objetos de los bordes. Además, en pruebas de velocidad de respuesta, esta API resultó más eficiente que CDS para muestras de datos pequeñas con radios menores a 8 arcosegundos pero menos eficiente en otros casos.

Sin embargo, se plantea la pregunta de porqué es necesario investigar otros motores de búsqueda. Una de las razones radica en resolver las diferencias en los resultados, ya que, aunque sean de aproximadamente el 1 %, podrían tener impactos imprevistos. Asimismo, es crucial explorar alternativas que optimicen los tiempos de respuesta para lograr el mejor sistema posible. Por último, la API desarrollada por Alarcón no abordó radios de 1 arcominuto, lo cual representa una necesidad a veces en el contexto de ALeRCE. Siguiendo la tendencia de los resultados de la API de MongoDB, los tiempos para consultas de 1 arcominuto alcanzarían valores sobre los 450 segundos para 50.000 puntos e incluso podrían ser mucho más elevados.

Capítulo 4

Preprocesamiento

En este capítulo se describe con mayor detalle la primera etapa del trabajo, que consiste en el preprocesamiento de los datos que componen al catálogo a utilizar, así como también la construcción de la base de datos que lo almacena.

4.1. Catálogo CatWISE2020

CatWISE2020 es un catálogo de todo el cielo que recolecta los datos de WISE y NEOWISE en las longitudes de onda de 3,4 μm y 4,6 μm [4]. Este catálogo tiene 2 versiones: una preliminar que usa datos entre los años 2010 y 2016, y CATWISE2020 la cual añade dos años más de objetos observados.

CatWISE2020 contiene 2.232.515.025 objetos entre *catalog* y *reject*, cuya diferencia radica en que el primero debe cumplir las siguientes condiciones[1]:

1. Ser "primario" (es decir, estar en la sección donde esa fuente está más lejos de los bordes) en su sección de la grilla[2].
2. Tener una relación señal/ruido (o SNR, por signal to noise ratio en inglés) igual o mayor a 5 para W1[3] sin artefactos identificados (un valor de 0 para "ab_flags" en la parte de W1 (caracter izquierdo) de la columna 178[4]).

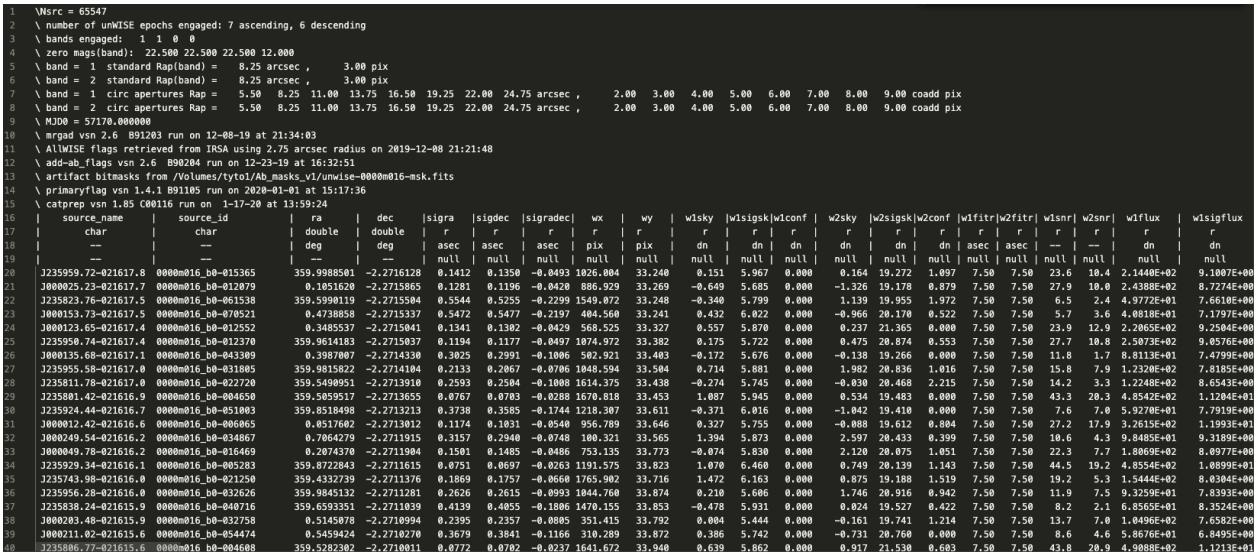
En caso de que no cumpla con lo anterior, se puede incluir si cumple con:

1. Tener SNR ≥ 5 para W2[5] sin artefactos identificados (un valor de 0 para "ab_flags" en la parte de W2 (caracter derecho) de la columna 178).

[1] `https://portal.nersc.gov/project/cosmo/data/CatWISE/2020README.txt`
[2] Según el sistema de coordenadas ecuatorial
[3] La banda 1 de 3,4 μm.
[4] `https://portal.nersc.gov/project/cosmo/data/CatWISE/2020cwcat.sis20200318.txt`
[5] La banda 2 de 4,6 μm.

Sin embargo, para poder garantizar la correctitud de los resultados descritos en el Capítulo 6, solo se utilizaron los archivos etiquetados como *catalog* que representan a 1.890.715.640 objetos observados por WISE y NEOWISE.

El conjunto de datos original, al cual se le llamará "raw", está subdividido en 36.481 archivos de los cuales la mitad son *catalog* y la otra mitad son *reject*. Cada archivo raw tiene un formato `.tbl` y viene con una compresión `.gz`, lo que significa que son archivos ASCII comprimidos con formato de tabla como el que se muestra en la Figura 4.1. Las tablas con *catalog* tienen 187 columnas mientras que las con *reject* tienen 188.



```
\Nsrc = 65547
\ number of unWISE epochs engaged: 7 ascending, 6 descending
\ bands engaged:  1  1  0  0
\ zero mags(band):  22.500 22.500 22.500 12.000
\ band =  1  standard Rap(band) =    8.25 arcsec ,     3.00 pix
\ band =  2  standard Rap(band) =    8.25 arcsec ,     3.00 pix
\ band =  1  circ apertures Rap =    5.50   8.25  11.00  13.75  16.50  19.25  22.00  24.75 arcsec ,     2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00 coadd pix
\ band =  2  circ apertures Rap =    5.50   8.25  11.00  13.75  16.50  19.25  22.00  24.75 arcsec ,     2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00 coadd pix
\ MJD0 = 57170.000000
\ mrgad vsn 2.6  B91203 run on 12-08-19 at 21:34:03
\ AllWISE flags retrieved from IRSA using 2.75 arcsec radius on 2019-12-08 21:21:48
\ add-ab_flags vsn 2.6  B90204 run on 12-23-19 at 16:32:51
\ artifact bitmasks from /Volumes/tyto1/Ab_masks_v1/unwise-0000m016-msk.fits
\ primaryflag vsn 1.4.1 B91105 run on 2020-01-01 at 15:17:36
\ catprep vsn 1.85 C00116 run on  1-17-20 at 13:59:24
|    source_name     |    source_id          |   ra      |   dec     |sigra   |sigdec  |sigradec|   wx   |   wy   | w1sky  |w1sigsk|w1conf |  w2sky  |w2sigsk|w2conf |w1fitr|w2fitr| w1snr| w2snr|  w1flux   |  w1sigflux  |
|        char        |       char            |  double   |  double   |   r    |   r    |    r   |   r    |   r    |    r   |   r   |   r   |    r    |   r   |   r   |   r  |   r  |   r  |   r  |     r     |      r      |
|         --         |        --             |    deg    |    deg    |  asec  |  asec  |  asec  |  pix   |  pix   |   dn   |   dn  |   dn  |    dn   |   dn  |   dn  | asec | asec |  --  |  --  |     dn    |      dn     |
|         --         |        --             |    --     |    --     |  null  |  null  |  null  |  null  |  null  |  null  |  null |  null |   null  |  null |  null | null | null | null | null |    null   |     null    |
J235959.72-021617.8  0000m016_b0-015365       359.9988501  -2.2716128  0.1412  0.1350  -0.0493 1026.004  33.240    0.151   5.967   0.000    0.164  19.272  1.097   7.50   7.50   23.6   10.4  2.1440E+02   9.1007E+00
J000025.23-021617.7  0000m016_b0-012079         0.1051620  -2.2715865  0.1281  0.1196  -0.0420  886.929  33.269   -0.649   5.685   0.000   -1.326  19.178  0.879   7.50   7.50   27.9   10.0  2.4388E+02   8.7274E+00
J235823.76-021617.5  0000m016_b0-061538       359.5990119  -2.2715504  0.5544  0.5255  -0.2299 1549.072  33.248   -0.340   5.799   0.000    1.139  19.955  1.972   7.50   7.50    6.5    2.4  4.9772E+01   7.6610E+00
J000153.73-021617.5  0000m016_b0-070521         0.4738858  -2.2715337  0.5472  0.5477  -0.2197  404.560  33.241    0.432   6.022   0.000   -0.966  20.170  0.522   7.50   7.50    5.7    3.6  4.0818E+01   7.1797E+00
J000123.65-021617.4  0000m016_b0-012552         0.3485537  -2.2715041  0.1341  0.1302  -0.0429  568.525  33.327    0.557   5.870   0.000    0.237  21.365  0.000   7.50   7.50   23.9   12.9  2.2065E+02   9.2504E+00
J235950.74-021617.4  0000m016_b0-012370       359.9614183  -2.2715037  0.1194  0.1177  -0.0497 1074.972  33.382    0.175   5.722   0.000    0.475  20.874  0.553   7.50   7.50   27.7   10.8  2.5073E+02   9.0576E+00
J000135.68-021617.1  0000m016_b0-043309         0.3987007  -2.2714330  0.3025  0.2991  -0.1006  502.921  33.403   -0.172   5.676   0.000   -0.138  19.266  0.000   7.50   7.50   11.8    1.7  8.8113E+01   7.4799E+00
J235955.58-021617.0  0000m016_b0-031805       359.9815822  -2.2714104  0.2133  0.2067  -0.0706 1048.594  33.504    0.714   5.881   0.000    1.982  20.836  1.016   7.50   7.50   15.8    7.9  1.2320E+02   7.8185E+00
J235811.78-021617.0  0000m016_b0-022720       359.5490951  -2.2713910  0.2593  0.2504  -0.1008 1614.375  33.438   -0.274   5.745   0.000   -0.030  20.468  2.215   7.50   7.50   14.2    3.3  1.2248E+02   8.6543E+00
J235801.42-021616.9  0000m016_b0-004650       359.5059517  -2.2713655  0.0767  0.0703  -0.0288 1670.818  33.453    1.087   5.945   0.000    0.534  19.483  0.000   7.50   7.50   43.3   20.3  4.8542E+02   1.1204E+01
J235924.44-021616.7  0000m016_b0-051003       359.8518498  -2.2713213  0.3738  0.3585  -0.1744 1218.307  33.611   -0.371   6.016   0.000   -1.042  19.410  0.000   7.50   7.50    7.6    7.0  5.9270E+01   7.7919E+00
J000012.42-021616.6  0000m016_b0-006065         0.0517602  -2.2713012  0.1174  0.1031  -0.0540  956.789  33.646    0.327   5.755   0.000   -0.088  19.612  0.804   7.50   7.50   27.2   17.9  3.2615E+02   1.1993E+01
J000249.54-021616.2  0000m016_b0-034867         0.7064279  -2.2711915  0.3157  0.2940  -0.0748  100.321  33.565    1.394   5.873   0.000    2.597  20.433  0.399   7.50   7.50   10.6    4.3  9.8485E+01   9.3189E+00
J000049.78-021616.2  0000m016_b0-016469         0.2074370  -2.2711904  0.1501  0.1485  -0.0486  753.135  33.773   -0.074   5.830   0.000    2.120  20.075  1.051   7.50   7.50   22.3    7.7  1.8069E+02   8.0977E+00
J235929.34-021616.1  0000m016_b0-005283       359.8722843  -2.2711615  0.0751  0.0697  -0.0263 1191.575  33.823    1.070   6.460   0.000    0.749  20.139  1.143   7.50   7.50   44.5   19.2  4.8554E+02   1.0899E+01
J235743.98-021616.0  0000m016_b0-021250       359.4332739  -2.2711376  0.1869  0.1757  -0.0660 1765.902  33.716    1.472   6.163   0.000    0.875  19.188  1.519   7.50   7.50   19.2    5.3  1.5444E+02   8.0304E+00
J235956.28-021616.0  0000m016_b0-032626       359.9845132  -2.2711281  0.2626  0.2615  -0.0993 1044.760  33.874    0.210   5.606   0.000    1.746  20.916  0.942   7.50   7.50   11.9    7.5  9.3259E+01   7.8393E+00
J235838.24-021615.9  0000m016_b0-040716       359.6593351  -2.2711039  0.4139  0.4055  -0.1806 1470.155  33.853   -0.478   5.931   0.000    0.024  19.527  0.422   7.50   7.50    8.2    2.1  6.8565E+01   8.3524E+00
J000203.48-021615.9  0000m016_b0-032758         0.5145078  -2.2710994  0.2395  0.2357  -0.0805  351.415  33.792    0.004   5.444   0.000   -0.161  19.741  1.214   7.50   7.50   13.7    7.0  1.0496E+02   7.6582E+00
J000211.02-021615.6  0000m016_b0-054474         0.5459424  -2.2710270  0.3679  0.3841  -0.1166  310.289  33.872    0.386   5.742   0.000   -0.731  20.760  0.000   7.50   7.50    8.6    4.6  5.8676E+01   6.8495E+00
J235806.77-021615.6  0000m016_b0-004608       359.5282302  -2.2710011  0.0772  0.0702  -0.0237 1641.672  33.940    0.639   5.862   0.000    0.917  21.530  0.603   7.50   7.50   43.8   20.9  4.9088E+02   1.1213E+01
```

Figura 4.1: Ejemplo de archivo TBL.

Primero, se tuvo que preprocesar cada archivo para dejarlos en una estructura más simple de trabajar. Se creó un programa en Python para leer cada archivo y eliminar las líneas que comenzaban con caracteres especiales, los cuales pueden ser "\" o "|". Una vez que se eliminaron estas líneas, se creó una tabla, se concatenaron los archivos y se generó un único CSV[6] para poder trabajar con los *SRDDs* de Sedona. En paralelo se procesó cada tabla por separado y se almacenaron en formato Parquet[7] para poder crear los tres formatos de tablas finales con los cuales se va a trabajar, los cuales son índices de S2[8], formato GeoParquet de Sedona[9,10] y por último en Parquet. Esto se realizó utilizando PySpark, que es un conector de Spark que sirve para aprovechar una de las más grandes ventajas de Spark que es el procesamiento en paralelo de código abierto orientado a la velocidad de cómputo y a la tolerancia a fallos. PySpark utiliza su propia implementación de *Dataframe* basado en el de Pandas, pero adaptado para trabajar con los *Resilient Distributed Dataframe* de Spark.

Una vez generadas las tablas, se procedió a seleccionar solo las columnas más importantes que utiliza ALeRCE basándose en el trabajo previo realizado por Alarcón [1]; gracias a esto

[6] *Comma-Separated Values*

[7] `https://parquet.apache.org/`

[8] `https://s2geometry.io/`

[9] `https://geoparquet.org/`

[10] `https://sedona.apache.org/latest-snapshot/api/sql/Optimizer/#push-spatial-predicates-to-geoparquet`

se pudo disminuir el número de columnas desde 186 a 20, cuyas descripciones se detallan en la Tabla 4.1.

Tabla 4.1: Nombres de columnas y descripciones de los datos de CatWISE2020 utilizados en el trabajo.

Nombre de la columna	Descripción
source_name	designación hexasegimal de la fuente
source_id	tile name código de procesamiento índice wphot
ra	ascensión recta (ICRS)
dec	declinación (ICRS)
sigra	incertidumbre en ra (arcsec)
sigdec	incertidumbre en dec (arcsec)
w1mag	magnitud de apertura estándar con corrección aplicada
w1sigm	incertidumbre de la magnitud de apertura estándar, banda 1
w1flg	apertura estándar flag, banda 1
w2mag	magnitud de apertura estándar con corrección aplicada
w2sigm	incertidumbre de la magnitud de apertura estándar, banda 2
w2flg	standard aperture flag, banda 2
w1k	índice k de Stetson para variabilidad [12]; banda 1
w1mlq	$-log(Q)$, $Q = 1 - P$(chi-square); banda 1
w2k	índice k de Stetson para variabilidad [12]; banda 2
w2mlq	$-log(Q)$, $Q = 1 - P$(chi-square); banda 2
PMRA	movimiento en ra
PMDec	movimiento propio en dec
sigPMRA	incertidumbre en PMRA
sigPMDec	incertidumbre en PMDec

4.2. Base de datos

En primera instancia se trabajó con archivos CSV, esto debido a que una de las limitaciones de Sedona al trabajar con *typed SRDDs* es que la función para crear un *typed SRDD* conserva las columnas con información correspondiente a la geometría principal y únicamente cuando el archivo es un CSV se conservan las columnas con información que no corresponde a la geometría principal; con otros formatos se perdían estas columnas. Luego se utilizó una API distinta a los *SRDD* ya que en estos últimos, en específico el *PointRDD* y el *CircleRDD*, fueron deprecados, por lo que se decidió utilizar la API de Sedona SQL. Junto con esto se ocuparon dos tipos de archivos distintos: los Parquet y los GeoParquet[11]. En primer lugar se utilizaron dos variantes de los archivos Parquet donde en una se encontraba el catálogo CatWISE2020 en formato Parquet y en la otra se encontraba el mismo catálogo junto con una columna extra que indicaba en qué celda de S2 se ubicaba cada objeto. Por otra parte, se usaron los GeoParquet, los cuales son archivos Parquet pero optimizados para trabajar con

[11] `https://geoparquet.org/`

datos geoespaciales; esto lo logra almacenando en la metadata de cada archivo información sobre los Geohashes que contenía cada uno de estos.

Una de las principales ventajas de utilizar GeoParquet en vez de Parquet en Sedona es la posibilidad de hacer *filter pushdown*, el cual consiste en filtrar archivos tan solo leyendo su metadata. En este caso se utilizó para que en lugar de cargar el archivo e ir filtrando punto por punto, se pudieran descartar los archivos dependiendo del cuadrante del Geohash en el cual se encontraba y así utilizar menos recursos.

4.2.1. Estructura básica de los archivos

Como se mencionó en la sección anterior, Sedona es capaz de utilizar distintos formatos y estructuras de archivos, por lo cual acá entramos un poco en profundidad sobre cómo se ven las columnas principales de cada uno de los archivos:

- CSV: La columna principal es la primera, la cual debe ser un *string* con la geometría de un punto generado con los datos de RA y DEC de cada objeto. Es importante que la columna con la geometría sea la primera ya que el CSV se tiene que crear sin *headers* y la función de la API de *SRDDs* `PointRDD` lee la primera columna del archivo como la geometría.
- Parquet: La columna principal es `geom`, la cual es una geometría de un punto creada con el constructor `ST_Point(ra, dec)`[12].
- GeoParquet: Utiliza la columna `geom`, la cual es la geometría de un punto creada con el constructor `ST_Point(ra, dec)` y la columna de nombre `geohash`[13] la cual se crea con la función `ST_GeoHash(ST_Point(ra, dec))`; esta última columna se añade como una *bbox* en la metadata del archivo Parquet.
- Parquet con S2: Las columnas principales son `geom` y `cellId`. La primera es la geometría de un punto creada con el constructor `ST_Point(ra, dec)`[14], mientras que la segunda es un *hash* creado utilizando `ST_S2CellIDs(ST_Point(ra, dec))`, el cual en este caso, asigna a cada punto un único *hash* debido a que cada punto sólo puede estar en un único cuadrante.

Al guardar los archivos CSV, estos se deben ordenar por los Geohash para así aprovechar al máximo la capacidad del *filter pushdown*.

4.3. Indexación espacial

En una base de datos, un índice es una estructura de datos que mejora la velocidad de las operaciones de búsqueda y recuperación de información a cambio de una mayor cantidad de

[12] `https://sedona.apache.org/1.4.1/api/sql/Constructor/#st_point`
[13] Sedona necesita que exista una columna con nombre `geohash` al momento de guardar un GeoPArquet
[14] `https://sedona.apache.org/1.4.1/api/sql/Constructor/#st_point`

escrituras y mayor uso de almacenamiento. Generalmente se usan estructuras con tiempos de búsqueda eficientes[15] como *B+ trees*, árboles balanceados y *hashes*.

Sedona en este caso cuenta con dos índices geoespaciales para poder tener búsquedas y *join* más eficientes en consultas con coordenadas geográficas. El primero es un *R-Tree*, el cual agrupa objetos cercanos en el rectángulo más pequeño; éste genera un árbol balanceado de búsqueda. Por otro lado tenemos el *Quadtree* que es un árbol de 2 dimensiones en el que cada nodo tiene exactamente 4 hijos, siendo así una división recursiva de 4. En este caso se utilizarán los dos índices para poder compararlos y escoger el mejor.

Para crear un índice espacial en las distintas APIs de Sedona, se tienen que seguir los siguientes pasos:

- Sedona typed *SRDDs*: Para agregar un índice a un *SRDD*, primero se tiene que leer un archivo utilizando una función de Sedona, y luego armar un índice sobre este, ya sea *Quadtree* o *R-Tree*.
- Sedona SQL: Para armar un índice basta con cambiar el *SparkConfig*, y agregar las siguientes opciones.

```
spark = SparkSession.builder.appName(app_name).getOrCreate()
spark.conf.set("sedona.global.index","true")
spark.conf.set("sedona.global.indextype", "quadtree") # rtree
```

Una vez terminado el preprocesamiento, se tenían una serie de catálogos de CatWISE2020 creados con los distintos métodos mencionados anteriormente, en el siguiente capítulo se abordará el cómo se realizaban consultas de *crossmatch* y también cómo se creó la API para realizar las consultas.

[15]Un tiempo eficiente se considera que sea menor a $O(n)$ en promedio

Capítulo 5

Implementación de la solución

En el siguiente capítulo, se proporciona una descripción detallada de la configuración de la implementación del servicio de *crossmatching*, y de la interfaz, la cual se integrará al *pipeline* de ALeRCE.

Para la implementación de esta interfaz, se tomó en cuenta las funciones realizadas para las pruebas de *crossmatching* que se exploran en profundidad en el Capítulo 6.

En general, todo el código ha sido desarrollado utilizando Python como lenguaje de programación principal, lo cual proporciona una alta flexibilidad y facilidad de uso.

Este capítulo se divide en dos secciones: la primera aborda de manera exhaustiva las funciones para generar el *pipeline* de *crossmatching*, mientras que la segunda se centra en la implementación de las funciones dentro de una API, la cual fue desarrollada utilizando FastAPI.

5.1. Crossmatching

Esta sección del capítulo se puede separar en dos subsecciones principales: la preparación de la consulta y la ejecución de la consulta de *conesearch*.

5.1.1. Preparación de la consulta

Originalmente se utilizaba la API de *SRDDs* de Sedona para realizar el *conesearch*, pero luego se descartó la idea por varias razones:

- Los *typed SRDD* fueron deprecados debido a problemas con la precisión de las consultas, lo que podría llegar a traer problemas en un futuro. Además los *SRDD* genéricos demostraban ser muy lentos en comparación con otros métodos.

- Realizar un *conesearch* preciso tiene una complejidad elevada, debido a que las funciones de intersección entre *SRDDs* calcula la distancia de manera cartesiana. Dado lo anterior, para crear un `Point SRDD` de CatWISE2020 se requiere inicialmente transformar las coordenadas del catálogo desde ICRS al sistema de coordenadas WGS84, para luego al momento de crear el `Point SRDD` transformarlas nuevamentente desde WGS84 a la proyección de Mercator pues en esta última es posible el cálculo de distancias de forma cartesiana. Este proceso también se realizó con los puntos de consulta.
- Al utilizar una proyección cartesiana(proyección de Mercator), se pierden datos. Esto se refleja al momento de realizar consultas sobre ciertas declinaciones(latitud) donde los puntos ya no existen, es decir, la información se perdió.

Debido a lo mencionado anteriormente, se decidió trabajar utilizando Sedona SQL, en específico métodos que involucran a los GeoParquet y a los índices de S2. Para esto, el primer paso consiste en cargar el catálogo de datos en una tabla de Spark y persistirla en disco, ya que como se puede ver en el Capítulo 6, persistir la tabla en disco favorece a que las consultas se ejecuten más veloz. El segundo paso consiste en hacer una consulta de *warm-up* para poder persistir la tabla en disco[1].

Una vez se tiene la tabla persistida en disco, se puede comenzar a hacer consultas en la aplicación. El *pipeline* procesa una lista de coordenadas astronómicas, representadas por RA(Ascensión Recta) y Dec(Declinación) en el sistema de referencia ICRS y con unidades en grados. Cada elemento de esta lista es un diccionario que tiene como llaves RA y Dec, y los valores respectivos en grados. Además se proporcionan entradas para modificar el radio de la consulta, la cual puede ser en arcosegundos y/o arcominutos, valores que luego son transformados a grados.

El diccionario con los puntos a consultar es cargado a un *Spark DataFrame*. Se deben transformar las coordenadas de RA pues estas pueden variar desde (0° a 360°), sin embargo, su análogo, la latitud, se mueve en los valores de (−180° a 180°). Para que RA se ajuste a este rango, se requiere restar 180 grados a sus valores. Por otro lado, se crea una circunferencia en cada punto, utilizando el radio transformado a grados como *buffer*; para esto se utiliza la función `ST_Buffer(`*`geom, radius`*`)`. Para los índices de S2 se tiene que hacer un paso extra, el cual es crear los índices para los puntos de la consulta utilizando la misma profundidad que se utilizó al momento de crear el catálogo en la *pipeline* de preprocesamiento; esto se hace con la función `ST_S2CellIDs(`*`geom, depth`*`)`. Por último se crea una vista temporal de una tabla en Spark con la función *`sparkDataFrame`*`.createOrReplaceTempView(`*`table_name`*`)`.

5.1.2. Ejecutar la consulta

Para ejecutar la consulta, se utiliza Sedona SQL con PySpark. Estas librerías permiten la conexión a un cluster de Sedona Spark y crear una sesión que permitirá realizar consultas. Para realizar la consulta se generaron tres consultas en Sedona SQL: una para utilizar de *baseline* en los archivos Parquets, otra para los GeoParquets y finalmente para los índices de

[1]Esto se debe a la *Lazy evaluation* de Spark

S2. La consulta para los Parquet y GeoParquet son idénticas, solo que cambia la tabla a la cual apuntan. La consulta es la siguiente:

```
SELECT
    CAST(c.geom AS STRING) as point,
    CAST(p.query_point AS STRING) as query_point,
    ...
FROM
    catwise c, points p
WHERE
    ST_Intersects(c.geom, p.geom)
```

Se seleccionan las columnas de interés a retornar, en este caso se muestran las primeras dos para fines de demostración. Estas columnas se castean a tipos de datos existentes en PyArrow con el fin de aprovechar la optimización para transformar un *DataFrame* de Spark a un *DataFrame* de Pandas. Una limitación de lo anterior es que las columnas de tipo `Geometry` de Sedona no se encuentran en la optimización de Apache Arrow, por lo que se deben transformar a formato `string`. Por otro lado, se selecciona primero el catálogo y luego el set de consulta, esto debido a que Sedona construye el índice en la primera tabla del `FROM`; por último se realiza el filtrado utilizando la función `ST_Intersects`, la cual retorna `True` si la geometría de la derecha de la función intersecta a la de la izquierda.

Para el caso de la consulta con los índices de S2, se agrega un paso extra, el cual consiste en añadir un filtro por la celda de S2 en la cual la geometría se encuentra, quedando así la consulta:

```
SELECT
    CAST(c.geom AS STRING) as point,
    CAST(p.point AS STRING) as query_point
    ...
FROM
    catwise AS c, points AS p
WHERE
    c.col = p.cellId AND ST_Intersects(p.geom, c.geom)
```

Finalmente, para obtener los resultados y hacer la ejecución de la consulta, se utiliza la función `sparkDataFrame.toPandas()`, la cual transforma el Spark DataFrame a una tabla de Pandas; en este caso se activó la configuración de Spark[2] para poder utilizar la optimización de Apache Arrow. El resultado final consiste en una lista de diccionarios, cada uno conteniendo la información por punto encontrado en el catálogo que cumple con los requerimientos del *crossmatch*.

5.2. API

Para desarrollar la API se utilizó la biblioteca FastAPI[3], donde se habilitaron distintos *endpoints*, los cuales se describen a continuación.

[2] `spark.sql.execution.arrow.pyspark.enabled`
[3] `https://fastapi.tiangolo.com/`

5.2.1. Lista de catálogos (método GET)

Retorna una lista con los nombres de los catálogos disponibles para consulta. Esta información es obtenida mediante el *endpoint* de "/catalog". Este método no requiere de ningún parámetro por parte del usuario; la lista que retorna sirve para poder ver qué catálogos están siendo utilizados en el *endpoint* de *crossmatch*.

Un ejemplo de respuesta es la siguiente:

```
[
    "catwise2020_s2_8",
    "catwise2020_geohash_6"
]
```

5.2.2. Crossmatch con output JSON (método POST)

Este *endpoint* ofrece una lista de los objetos resultantes del proceso de *crossmatching* dada una lista de posiciones y un nombre de catálogo en específico. El *endpoint* está disponible a través de la URL "/equijoin_refined". Se envía una lista de diccionarios con la información de cada punto, y la salida es un diccionario con formato JSON, los cuales se explicarán más adelante. Los parámetros necesarios para este *endpoint* son los siguientes:

- *arcseconds* (`float`): Distancia en arco segundos que se utilizará para realizar el *conesearch*. La aplicación fue probada con radios desde 1 arcosegundo hasta 60 arcosegundos.
- *arcminutes* (`float`): Distancia en arcominutos que se utilizará para realizar el *conesearch*. La aplicación fue probada con radios de 1 arcominuto. El valor en arcominutos se suma a la entrada de arcosegundos, por ejemplo, una consulta con 30 arcosengundos y 1 arcominuto corresponde a una consulta de 1,5 arcominutos o 90 arcosegundos.
- *points* (`list[dict[str, float]]`): El conjunto de ubicaciones para realizar el *crossmatch*. Esta estrategia se emplea para optimizar la eficiencia del servicio, ya que se evita enviar una posición por solicitud y permite que Spark paralelice las consultas. A continuación se muestra un ejemplo de la lista de diccionarios a enviar:

  ```
  [
      {"ra": 53.3667744, "dec": -85.5547224},
      {"ra": 262.40749148, "dec": -22.20264135},
      {"ra": 254.32098384, "dec": 37.03822843},
      {"ra": 97.23693453, "dec": -25.08230158},
      {"ra": 157.35843221, "dec": -71.17841342}
  ]
  ```

- *catalog* (`str`): Corresponde al nombre del catálogo astronómico a utilizar para el proceso de *crossmatch*. La API restringe a solo utilizar los catálogos que aparecen en el endpoint "/catalog".

Aquí se muestra un ejemplo de una solicitud ocupando la librería requests[4] de Python:

[4] `https://requests.readthedocs.io/en/latest/`

```
points = [
    {"ra": 53.3667744, "dec": -85.5547224},
    {"ra": 262.40749148, "dec": -22.20264135},
    {"ra": 254.32098384, "dec": 37.03822843},
    {"ra": 97.23693453, "dec": -25.08230158},
    {"ra": 157.35843221, "dec": -71.17841342}
]
body = {
    "points": points,
    "arcseconds": 60,
    "arcminutes": 0,
    "catalog": "catwise2020_s2_8"
}
start_time = time.time()


r = requests.post("http://127.0.0.1:8000/equijoin_refined", json=body)
```

La respuesta obtenida al realizar una solicitud a este *endpoint* corresponde a un diccionario con formato JSON, el cual está conformado por los campos a continuación:

- *count* (`int`): La cantidad total de objetos resultantes al agregar todas las consultas.
- *result* (`str`): Es un diccionario con formato JSON, el cual corresponde a los objetos resultantes del *crossmatch*. Sumado a esto se agregan las coordenadas Ra y Dec correspondientes al objeto de entrada para el cual se encontró una coincidencia.

Un ejemplo de respuesta se presenta a continuación:


```
{
    "count": 1,
    "result": {
        "point": {"0": "POINT␣(-86.34861976151569␣55.36992907522)"},
        "query_point": {"0": "POINT␣(-86.34854015␣55.370185219999996)"},
        "source_name": {"0": "J154128.82-862055.0"},
        "source_id": {"0": "2398m864_b0-076810"},
        "ra": {"0": "235.36992907522"},
        "dec": {"0": "-86.34861976151569"},
        "sigra": {"0": "0.1444"},
        "sigdec": {"0": "0.1601"},
        "w1mag": {"0": "16.77"},
        "w1sigm": {"0": "0.04"},
        "w1flg": {"0": "3"},
        "w2mag": {"0": "16.67"},
        "w2sigm": {"0": "0.14"},
        "w2flg": {"0": "3"},
        "pmra": {"0": "0.13548"},
        "pmdec": {"0": "-0.10615"},
        "sigpmra": {"0": "0.0595"},
        "sigpmdec": {"0": "0.0667"},
        "w1k": {"0": "0.90987"},
        "w2k": {"0": "0.92343"},
        "w1mlq": {"0": "4.07"},
        "w2mlq": {"0": "1.02"}
```

```
    }
}
```

La respuesta indica el número de resultados. Cada resultado hace referencia a un cuerpo astrónimico y está asociado con un documento anidado de JSON que contiene toda la información clave que el grupo de ALeRCE necesita para su *pipeline*.

Capítulo 6

Experimentos y resultados

Para poder validar la utilidad del nuevo servicio con los objetivos de ALeRCE, se realizaron diversas pruebas para primero encontrar la mejor configuración de nuestra solución, y luego evaluar el tiempo de ejecución en comparación con CDS. Por otro lado, es de extrema importancia comprobar la precisión de los resultados, es decir, poder garantizar que los objetos detectados están realmente dentro del radio de búsqueda al realizar el *crossmatch*. Para mejorar la velocidad de las consultas, se implementaron índices y persistencia en disco para los catálogos.

La máquina en la cual se cargaron los catálogos y se realizaron los experimentos es un servidor con 4 CPUs y 96 cores. Cada CPU es un procesador Intel(R) Xeon(R) Gold 5220R con 24 núcleos. La memoria RAM instalada es de 376 GB. Para el almacenamiento, se utilizaron discos de estado sólido (SSD), teniendo disponible 56TB. Dentro de este servidor se utilizó un contenedor de Docker con una imagen de openjdk:8-slim de base, donde se instaló Python, Spark y Sedona; este contenedor se ejecutaba con volumen compartido con el servidor, de tal manera que no se tuvieran que copiar los datos de los catálogos a la imagen y que así fuese más portable. Este servidor corresponde al servidor de ALeRCE, donde se montará el pipeline completo en el futuro.

6.1. Explicación de experimentos y datos utilizados

Con el fin de realizar los experimentos, se crearon distintas muestras del firmamento, las cuales se describen a continuación:

1. 20.000 coordenadas seleccionadas aleatoriamente.
2. 50.000 coordenadas seleccionadas aleatoriamente.
3. 30.000 coordenadas de objetos de CatWISE2020.
4. 450 coordenadas en el plano galáctico de la Vía Láctea.
5. 180 coordenadas en el plano perpendicular al plano galáctico.

6. 1.000 coordenadas a un máximo de 5° del Ecuador.
7. 1.000 coordenadas a un máximo de 5° de los polos.

Para el primer y segundo conjunto de datos, se utilizó un random de Python para generar valores aleatorios en pares Ra/Dec. Los valores de Ra varían entre 0° y 360°, mientras que los valores de Dec pueden variar entre -90° y 90°, abarcando así toda la esfera de coordenadas.

Por otro lado, el tercer conjunto de datos se generó utilizando una consulta de Spark SQL con un `orderBy` de manera aleatoria y después un `LIMIT` de 30.000 puntos; de esta manera se asegura que los puntos escogidos son aleatorios; posteriormente se almacenaron solo los valores de Ra y Dec de los objetos.

Para el cuarto y quinto conjunto se trabajó con Astropy generando coordenadas SkyCoord bajo el *frame* galáctico. Posteriormente las coordenadas fueron transformadas a ICRS, que corresponde al *frame* utilizado por las coordenadas de CatWISE2020. Estas coordenadas galácticas están conformadas por dos valores: longitud `l` cuyo valor varía entre 0° y 360°, y la latitud `b`, cuyo rango es entre $-90°$ y 90°. Para la cuarta muestra, se fijó el valor de `b` en cero y los valores de `l` se separaron en intervalos de 0,8 grados. Por otro lado, para la quinta muestra, se fijó el valor de `l` en cero y se varió el valor de `b` entre el rango $-90°$ a 90°, con una separación de 1°.

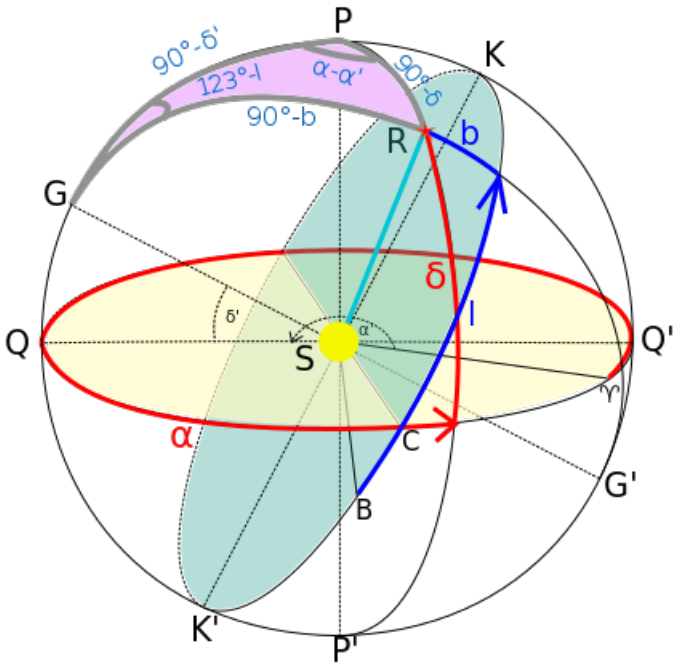


Figura 6.1: Transformación de coordenadas galácticas ecuatoriales. Fuente: `https://commons.wikimedia.org/wiki/File:Equatorial_galactic_coordinates_transformation.svg`.

En la Figura 6.1 se muestra un diagrama que representa el sistema de coordenadas galáctico y el sistema de coordenadas ecuatorial. El plano amarillo corresponde al Ecuador

celestial y el verde al plano galáctico. La porción resaltada en rosado se conoce como el tercer triángulo astronómico y se utiliza para convertir estos sistemas usando trigonometría.

Los primeros tres conjuntos fueron seleccionados de esta forma para realizar una comparación respecto a los resultados obtenidos por Alejandra Alarcón en su memoria "Servicio de Crossmatching de objetos Astronómicos" [1], quien utilizó cantidades similares en el caso de la primera, segunda y tercera muestra; la cuarta y quinta muestra se construyeron a partir de lo descrito en su memoria.

Por último, el sexto y séptimo conjunto fueron elaborados usando *random*, y restringiendo que los valores de Ra se encontrasen a no más de 5° respecto del Ecuador o los polos. Este conjunto tenía como objetivo realizar una comparación de los resultados que se pueden obtener realizando *crossmatch* en una proyección del cielo en un plano en dos dimensiones pues éstas pueden presentar dificultades en los cálculos de las distancias de objetos cercanos a los polos. Estas dificultades surgen ante la representación de un objeto de naturaleza esférica en un plano, lo que causa distorsiones sobre las distancias en los polos. Es de vital importancia tener en cuenta esta distorsión al momento de utilizar mapas celestes o realizar mediciones precisas de las posiciones de objetos en las regiones polares. Por lo general se suelen utilizar distintas técnicas para compensar esta distorsión, por ejemplo, utilizar una proyección adaptada a una esfera o realizar correcciones adicionales a las coordenadas que se utilizan.

Con estos conjuntos como entrada, se realizaron consultas en CDS X-Match y el servicio desarrollado en este trabajo para los radios de 1, 2, 4, 8, 16, 32 y 60 arcosegundos. Los resultados de estas consultas se detallarán en profundidad en las secciones siguientes. En específico, se realizaron experimentos para cada radio, utilizando las dos metodologías de indexación trabajadas, GeoHash y Google S2; en particular para el primer experimento se utilizó el 50 % de los datos de CatWISE2020. Por otro lado, utilizando la mejor configuración de cada método se experimentó con los niveles de partición con ayuda de la función `repartition` de Spark, esto debido a que en Sedona se recomienda tener entre 2 a 8 veces la cantidad de particiones del dataset[1], en este caso se experimentó con los siguientes factores para el número de particiones: $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4 y 8 sobre el catálogo de CatWISE2020 completo. Por último, sobre el número de particiones que se comporte mejor, se realizarán 3 experimentos adicionales, correspondientes al uso de los índices de Sedona, en los cuales hay tres casos, R-Tree, Quad-Tree y sin indexación, utilizando el catálogo de CatWISE2020 completo.

Los resultados de estos experimentos nos dieron conclusiones sobre la mejor configuración de nuestra solución. Para concluir los experimentos, se hizo una comparación entre la mejor configuración encontrada para nuestra solución y CDS X-Match sobre el catálogo completo.

[1] `https://sedona.apache.org/1.4.1/tutorial/Advanced-Tutorial-Tune-your-Application/?h=partitions#be-aware-of-spatial-rdd-partitions`

6.2. Resultados

6.2.1. Test de configuraciones

Para obtener la mejor configuración de cada método, se realizaron experimentos evaluando el tiempo de ejecución. El primer experimento consiste en comparar la velocidad de consulta utilizando distintas profundidades, tanto en los Geohashes como en los índices de S2; para esto se hicieron 10 pruebas para 1, 2, 4, 8, 16, 32 y 60 arcosegundos, y se midió el tiempo total de ejecución.

Una vez realizados los experimentos, se escogió la mejor profundidad para cada uno de los métodos y se procedió a poner a prueba la velocidad de las consultas dados distintos factores de particiones; los factores utilizados fueron $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4 y 8, estos factores se multiplican por la cantidad de particiones recomendada por Spark. Por último se probó el rendimiento utilizando las 3 configuraciones distintas de los índices que utiliza Sedona para las consultas de *join*, las cuales son: R-Tree, Quad-Tree y sin índice.

La primera prueba de profundidad fue realizada con un sample aleatorio del 50 % de los objetos de CatWISE2020 como catálogo y un set de 20.000 puntos aleatorios; se usa una muestra de los datos pues, para este experimento, es necesario re-indexar los datos para cada profundidad diferente, a la escala a la cual se quiere trabajar, tendrá un costo muy grande generar todos los índices necesarios sobre los datos completos. Las siguientes pruebas, la de particiones e índices fueron realizados con CatWISE2020 completo, y con el mismo set de 20.000 puntos aleatorios.

Parquet (sin índices)

Se intentó realizar experimentos de consultas de *conesearch* en un catálogo de CatWISE2020 almacenado en un archivo Parquet y sin índices creados, con el fin de comparar qué tanto mejoraron los tiempos según la ausencia o presencia de índices en una consulta. Sin embargo, ningún experimento se pudo ejecutar ya que se generaba una consulta muy grande para Spark y como resultado arrojaba un error explicando que el proceso se había quedado sin memoria disponible.

GeoParquet (Geohash)

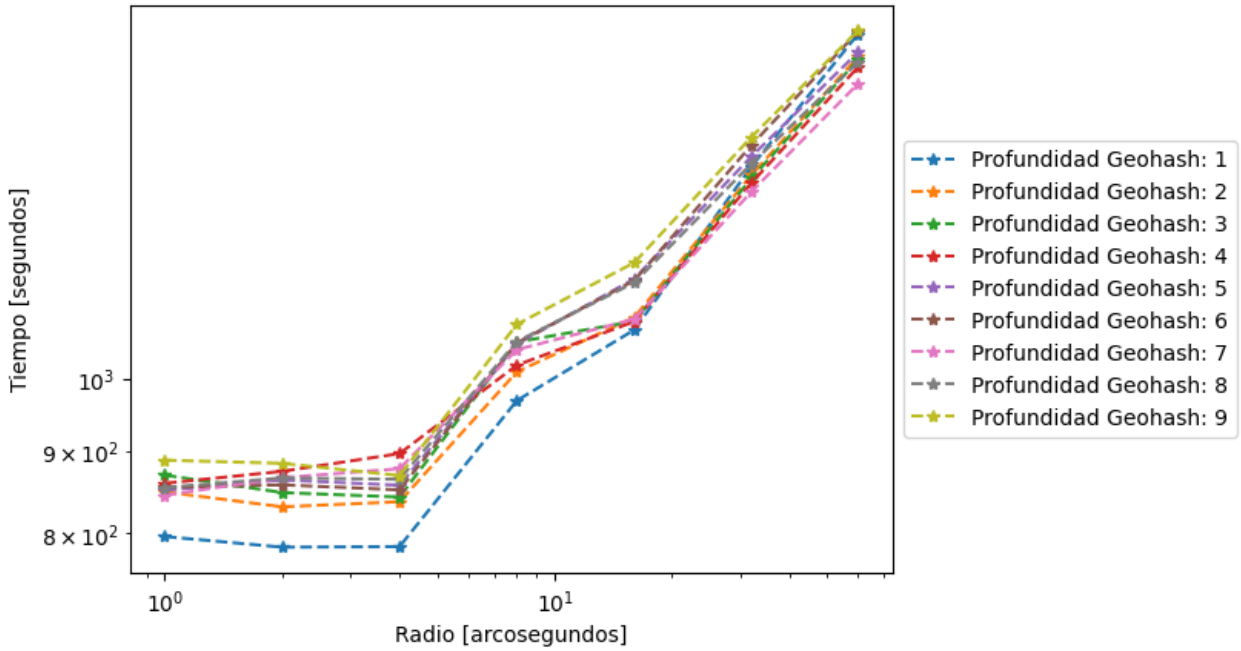


Figura 6.2: Gráficos de los tiempos de *crossmatch* usando los GeoParquet, consultando 20 mil puntos y utilizando el 50 % de los datos de CatWISE2020.

Como se observa en la Figura 6.2, los tiempos de consulta al utilizar GeoParquets son altos, esto se debe a que la estructura de GeoParquets fue creada para optimizar el uso de recursos al momento de leer una gran cantidad de archivos; solo se utilizaron alrededor de 40 cores, 100 GB de memoria RAM de 300 GB disponibles, y un bajo uso del disco. Esto se logra haciendo *filter pushdown* sobre la metadata del archivo y solo leyendo los que se encuentran en las celdas del archivo consultado. Una de las principales hipótesis de por que su rendimiento no es óptimo es que está diseñado para optimizar recursos y que el formato se encuentra en una fase temprana de desarrollo, por lo que es posible que no pueda soportar bien una alta cantidad de archivos grandes.

Debido a estos resultados preliminares se decidió no continuar experimentando con los GeoParquet. La razón detrás de esto es que se encuentran muy lejos del rendimiento esperado por ALeRCE, sumado a esto los GeoHashes son una división en dos dimensiones de la Tierra, por lo que eventualmente se podrían generar resultados inconsistentes en las pruebas de correctitud.

Google S2 (equi-join)

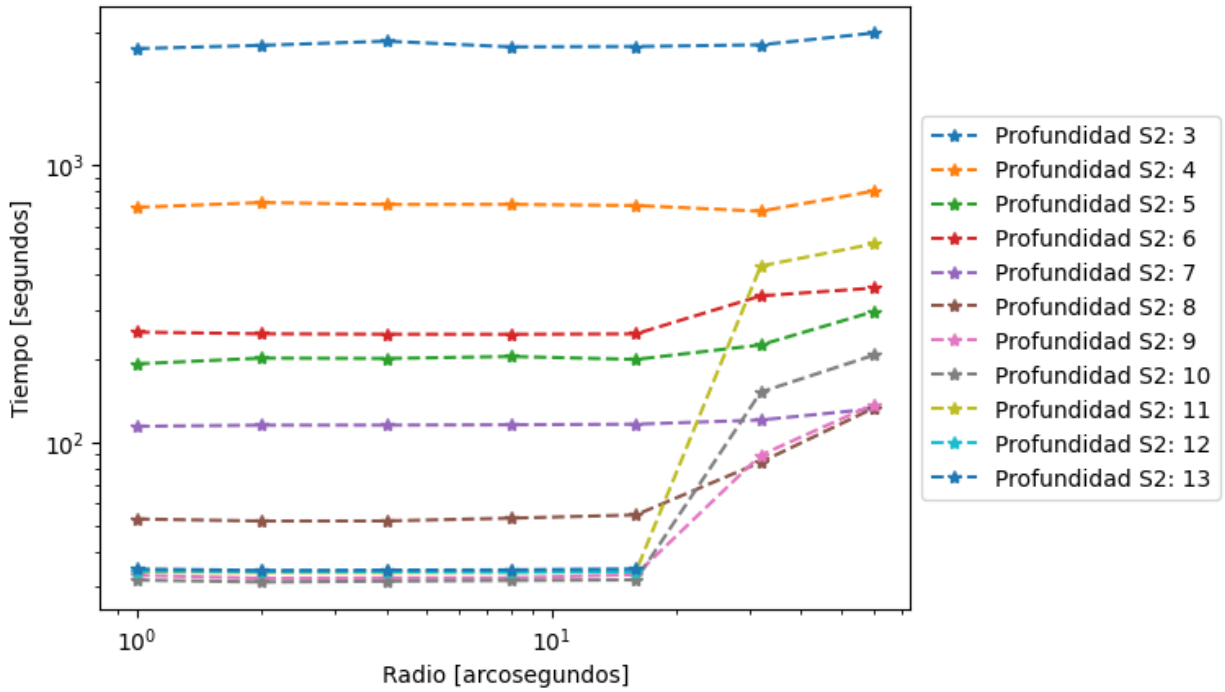


Figura 6.3: Gráficos de los tiempos de *crossmatch* usando los índices de Google S2, consultando 20 mil puntos y utilizando el 50 % de los datos de CatWISE2020.

En la Figura 6.3 se presentan los resultados de comparar diferentes niveles de profundidad con Google S2; se puede observar que, en primer lugar, no se pueden ejecutar consultas en todos los niveles y para todos los radios; por ejemplo, para las profundidades 1 y 2, no se terminó de ejecutar ninguna consulta. Después de los 30 minutos de ejecución, esta se interrumpió de forma manual, debido a que al tener celdas tan grandes a las cuales consultar, se hacía imposible cargar una en un *executor* y esto causaba *swapping* al quedarse sin memoria principal.

Por otro lado, para las profundidades 12 y 13, fue imposible ejecutar consultas más allá de los 16 arcosegundos, debido a que la consulta se quedaba sin memoria princial disponible para ciertos *executors* de Spark. Para solucionar esto se trató de repartir el posible *data skew* que había, pero no se tuvo éxito. Luego se comenzaron a hacer pruebas sobre los puntos a consultar y se encontró que el Spark *DataFrame* utilizado para crear los círculos en las consultas crece de manera cuadrática al aumentar el radio de la consulta y de manera exponencial al momento de aumentar la profundidad, esto pues al incrementar el radio de consulta, el círculo generado contiene más celdas en su interior, y cada celda es representada como una fila dentro del *DataFrame* de consulta. Este comportamiento se veía potenciado al momento de consultar profundidades mayores, ya que se generan celdas más pequeñas y es más probable que el radio de la consulta involucre más celdas.

Se puede concluir que la mejor profundidad es de 9, ya que tiene tiempos bajos tanto en radios pequeños como en más grandes. Durante este experimento, la utilización de memoria

RAM de Spark llegó a alrededor de 200 GB de los 300 GB asignados, usando además los 96 cores al 100 % durante la ejecución. Por último la utilización del disco fue alta.

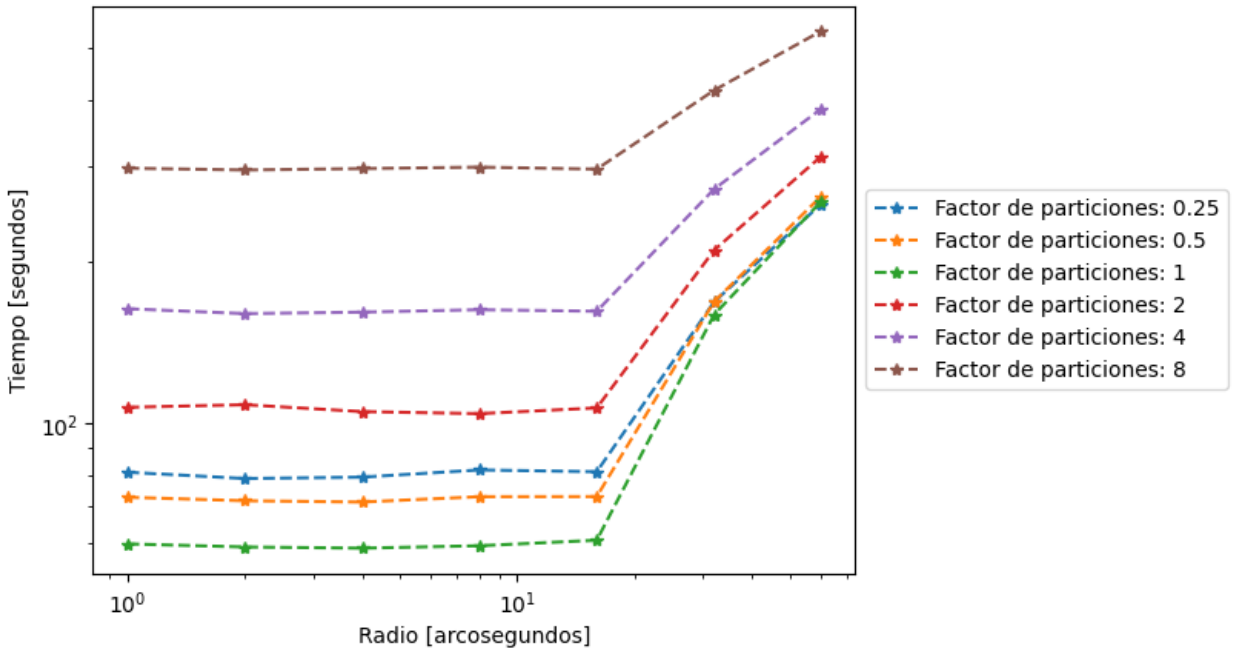


Figura 6.4: Gráficos de los tiempos de *crossmatch* usando distintos factores de particiones, consultando 20 mil puntos y utilizando el 100 % de los datos de CatWISE2020.

Una vez se encontró la profundidad ideal de los índices de S2 se realizaron experimentos sobre la cantidad de particiones en el *DataFrame* estos experimentos se hicieron tomando la cantidad por defecto de particiones de Spark sobre el *DataFrame* y se multiplicaron por un factor, el cual se mueve dentro de los valores de $\frac{1}{4}$ hasta 8. Lo anterior se realizó con el 100 % de los datos disponibles en CatWISE2020 debido a que al tener más datos, el número de particiones varía y con este experimento se quería recrear lo mejor posible las condiciones finales de la consulta de *crossmatch*.

Como se observa en la Figura 6.4, el mejor factor para las particiones es el número por defecto, así que se logró confirmar la recomendación de Spark. Una de las razones es que al ser un *equi-join*, utilizar un algoritmo de *Hash Join* hace que esta operación sea sensible a la cantidad de datos a los cuales se consulta, por lo tanto, si el número de particiones aumenta demasiado el rendimiento decae, debido a que se tienen que consultar un número mayor de bloques más pequeños y cada lectura de bloque viene con una penalización ya que están almacenados en el disco. Por otro lado, disminuir la cantidad de particiones implica recorrer bloques más grandes y como consecuencia no se logra una paralelización ideal pues cada *executor* de Spark tiene que recorrer más datos con recursos limitados. Se puede confirmar que la configuración por defecto un balance con buen rendimiento.

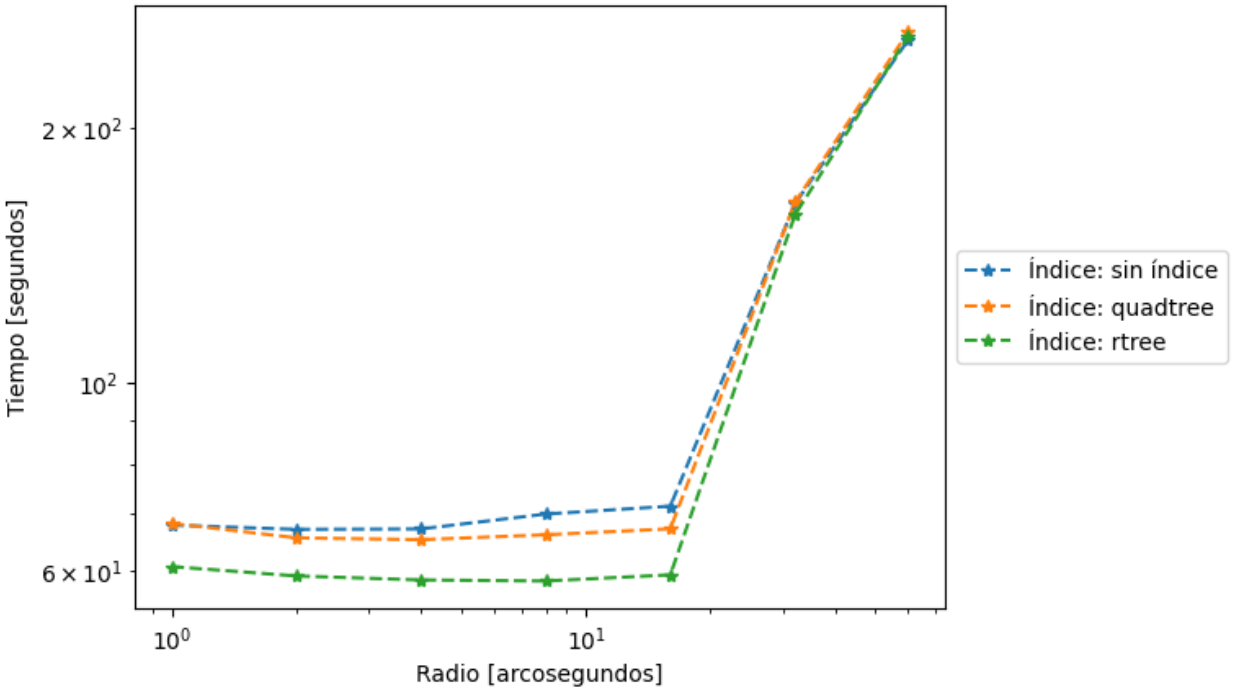


Figura 6.5: Gráficos de los tiempos de *crossmatch* usando distintos índices, consultando 20 mil puntos y utilizando el 100 % de los datos de CatWISE2020.

El último experimento realizado para determinar la mejor configuración del método de *equi-join* con los índices de Google S2 consistió en probar el rendimiento utilizando los distintos índices que Sedona dispone para optimizar las operaciones de *Join*. Dentro de las opciones se encuentran las siguientes: Quad-Tree, R-Tree y sin índice. Como se puede apreciar en la Figura 6.5 el índice de R-Tree logra unos tiempos alrededor de un 16 % más bajos en casos de radios menores a 16 arcosegundos, para luego igualar el rendimiento en radios mayores. Esto se puede deber a la naturaleza de los R-Tree, que generan un árbol balanceado con los rectángulos más pequeños que contienen una cantidad n arbitraria de datos que le permite al R-Tree ser muy eficiente en consultas de búsqueda de puntos cercanos. Por ejemplo, si se encuentra una concentración de puntos, el R-Tree generará un solo rectángulo que contenga esos datos, mientras que el Quad-Tree subdividirá esa misma área en 4 partes balanceadas recursivamente, lo que puede provocar que se necesiten más ciclos para obtener una serie de puntos en un cuadrante pequeño.

6.2.2. Correctitud

Para comprobar la precisión de las consultas de *crossmatch*, se compararon los resultados obtenidos entre la solución de Sedona y CDS X-Match. Para esta comparación, se contaron los objetos resultantes de cada consulta y luego se verificó si los objetos retornados por la solución de Sedona existían en lo retornado por CDS X-Match. Para esta verificaciónb, se utilizó un código escrito en Python, el cual utilizando la librería Pandas hacía un *Outer Join* entre los resultados de CDS y del método propio, utilizando la columna `source_id` de los objetos resultanes como llave. De esta forma se podían identificar los objetos que coincidian

entre los resultados de ambas soluciones, los objetos que solo se encontraban en los resultados de CDS, pero no en los resultados del método de Sedona, y finalmente los objetos que se encontraban en los resultados de la solución propia, pero no se encontraban en los resultados de CDS.

Se utilizaron los archivos CSV que proporciona CDS y el sistema propio, descubriendo que los objetos resultantes en las consultas del sistema propio se encontraban en su totalidad dentro de los objetos retornados por CDS. Sin embargo también se obtuvo una gran cantidad de objetos no retornados por el sistema propio, sobre todo cuando los puntos a consultar se encuentran cerca de los polos.

Coordenadas de objetos aleatorios cerca de los polos

Tabla 6.1: Conteo de coincidencias entre el método implementado y el de CDS X-Match para la muestra de mil puntos aleatorios a una distancia máxima de 5 grados de los polos.

Radio (arcosegundos)	Coincidencias CDS	Coincidencias Sedona	% de Coincidencia
1	10	2	20.00
2	45	3	6.67
4	185	10	3.51
8	729	36	4.94
16	2.928	121	4.13
32	11.659	494	4.24
60	40.332	1.737	4.36

Tomando en cuenta los resultados de la Tabla 6.1, se puede notar una clara baja de coincidencias cuando los puntos se encuentran cercanos a los polos; de hecho se puede ver que solo se obtiene aproximadamente un 4 % de coincidencias entre los métodos, lo que puede explicarse en cómo Sedona maneja la información, ya que realiza una proyección en el plano de un círculo y busca los puntos que se encuentran dentro pero el problema es que el círculo no está escalando correctamente en conjunto con la proyección del plano en dos dimensiones, lo que genera que el radio de consulta sea menor a lo que debiera ser, es por esto que todos los puntos que se retornan en el método propio se encuentran en CDS X-Match, pero no todos los de CDS se encuentran en los resultados del método de Sedona.

Coordenadas de objetos aleatorios cerca del Ecuador

Como se ve en la Tabla 6.2, cuando la consulta es cerca del Ecuador se ve una notable mejora en el número de coincidencias, pero sigue sin tener exactamente las mismas que CDS X-Match, esto se debe a que por más pequeño que sea el ángulo de diferencia respecto del Ecuador, el método desarrollado va a contar con una pequeña discrepancia debido a la distorsión al momento de proyectar la Tierra en un plano y no ajustar el radio de la consulta acorde con esta deformación.

Tabla 6.2: Conteo de coincidencias entre el método implementado y el de CDS X-Match para la muestra de mil puntos aleatorios a una distancia máxima de 5 grados del Ecuador.

Radio (arcosegundos)	Coincidencias CDS	Coincidencias Sedona	% de Coincidencia
1	15	15	100.00
2	45	44	97.78
4	157	156	99.36
8	607	599	98.68
16	2.410	2.395	99.38
32	9.809	9.753	99.43
60	34.518	34.263	99.26

Coordenadas de objetos en el plano galáctico

Tabla 6.3: Conteo de coincidencias entre el método implementado y el de CDS X-Match para la muestra de puntos en el plano galáctico.

Radio (arcosegundos)	Coincidencias CDS	Coincidencias Sedona	% de Coincidencia
1	15	13	86.67
2	42	31	73.81
4	126	103	81.75
8	489	349	71.37
16	1.911	1.393	72.89
32	7.754	5.753	74.19
60	27.022	20.008	74.04

Como se observa en la Tabla 6.3, hay una gran densidad de objetos astronómicos, por lo que se ponía a prueba la precisión que tiene el sistema propio sobre las distancias de los objetos. Se puede notar que existe un error de aproximadamente de un 28 % respecto con los resultados de CDS, lo que nos indica que el método desarrollado en Sedona está consultando radios más pequeños de lo esperado.

Coordenadas de objetos en el plano perpendicular al galáctico

Tomando en consideración los resultados de la Tabla 6.4, se puede ver una diferencia respecto a los resultados obtenidos en la Tabla 6.3. Esta diferencia se debe a que en el plano perpendicular se encuentra una menor concentración de objetos respecto del plano galáctico; los puntos al estar a una distancia mayor y en menor concentración, hacen que la imprecisión de Sedona al momento de consultar no sea tan acentuada como en el caso del plano galáctico.

Posiciones de objetos seleccionados de manera aleatoria

En la Tabla 6.5 se aprecian los resultados del experimento. Uno de los principales aspectos que llama la atención es que Sedona tiene una coincidencia de alrededor del 58 % de los

Tabla 6.4: Conteo de coincidencias entre el método implementado y el de CDS X-Match para la muestra de puntos en el plano perpendicular al plano galáctico.

Radio (arcosegundos)	Coincidencias CDS	Coincidencias Sedona	% de Coincidencia
1	2	2	100.00
2	10	8	80.00
4	21	18	85.71
8	135	107	79.26
16	532	463	87.03
32	1.984	1.745	87.95
60	6.905	5.965	86.39

puntos retornados con respecto a X-Match. Cabe destacar que todos los puntos retornados en la consulta del sistema propio se encuentran en los resultados de X-Match.

En la documentación de la implementación de *equi-join* con los índices de S2[2], se menciona que el método puede no obtener buenos resultados si los puntos se encuentran en los polos o en el antimeridiano, lo que puede explicar el 42 % de diferencia que tienen ambos métodos, correspondiente a la probabilidad de un punto a encontrarse cerca de alguna de estas zonas.

Tabla 6.5: Conteo de coincidencias entre el método implementado y el de CDS X-Match para la muestra de 50 mil puntos aleatorios.

Radio (arcosegundos)	Coincidencias CDS	Coincidencias Sedona	% de Coincidencia
1	536	314	58.58
2	2.091	1.206	57.68
4	8.334	4.870	58.44
8	33.238	19.559	58.85
16	133.030	77.912	58.57
32	528.778	309.580	58.55
60	1.860.330	1.088.808	58.53

Resumen

Tomando en cuenta estos descubrimientos, se puede concluir que los resultados obtenidos por el método desarrollado no han superado la prueba de precisión. No se logró obtener los mismos resultados que CDS, siendo una de las principales razones la forma en que Sedona trabaja con las coordenadas pues este framework realiza una proyección en el plano y genera un círculo de igual radio para todas las latitudes, lo que causa que al momento de consultar cerca de los polos retorne una cantidad considerablemente menor de puntos. Cabe destacar que todos los puntos retornados por el método propio se encontraban dentro de los resultados de CDS, lo que nos puede indicar que con un pequeño factor de ajuste a los radios de las consultas basado en la latitud de cada punto se podría obtener la cantidad correcta de resultados. En la siguiente sección, se harán experimentos para comparar los tiempos de ejecución entre la solución desarrollada y CDS X-Match.

[2] `https://sedona.apache.org/1.4.1/api/sql/Optimizer/?h=eq#s2-for-distance-join`

6.2.3. Eficiencia de procesar las consultas

Con el propósito de comparar la velocidad de consulta entre el método desarrollado en este trabajo y CDS X-Match, se realizó una medición del tiempo promedio al realizar una consulta con un total de 10 muestras por cada radio consultado. Para esto se utilizó la API desarrollada y la API de CDS X-Match, a la cual se puede consultar en el siguiente enlace: `http://cdsxmatch.u-strasbg.fr/xmatch/api/v1/sync`. Si bien la API de CDS cuenta con una interfaz web, se decidió no utilizarla debido a que la medición de tiempos resultaba ser imprecisa en cuanto a los tiempos de ejecución de las consultas e incluso para las consultas que demoran menos de 1 segundo, solo mostraba "$< 1s$", sin mayor precisión ni detalles. Para obtener mediciones precisas se decantó por usar la librería *requests* de Python para consultar a la API de CDS X-Match[3].

Esta sección busca dar respuesta a las siguientes preguntas:

- ¿Cómo se comportan las mediciones de tiempo a medida que aumenta el radio?
- ¿Cómo se comportan los tiempos al aumentar o disminuir la cantidad de coordenadas consultadas respecto a CDS X-Match?
- ¿Cómo se comparan los tiempos de consulta promedio con respecto a CDS X-Match al momento de utilizar la mejor configuración del sistema propio?

Posiciones de objetos seleccionados de manera aleatoria

50 mil objetos aleatorios

Los primeros conjuntos de coordenadas puestos a prueba fueron el de 50 mil coordenadas y el de 20 mil coordenadas generadas aleatoriamente. En este caso se compararon los tiempos de consulta para radios que varían desde 1 arcosegundo hasta 60 arcosegundos. El tiempo de ejecución se consideró desde que se enviaba el *request* hasta el momento en que se terminaba de escribir el archivo CSV; los resultados obtenidos se pueden observar en las Figuras 6.6 y 6.7.

Se puede ver que el tiempo de consulta para el método desarrollado en este trabajo se mantiene constante en radios menores a 32 arcosegundos, pero luego estos tiempos aumentan de forma más abrupta que los de CDS; para las consultas de 60 arcosegundo el método propio se demora en promedio 168 segundos para procesar los 50 mil objetos, mientras que CDS solo se demora 57 segundos.

[3] `http://cdsxmatch.u-strasbg.fr/xmatch/doc/API-calls.html`

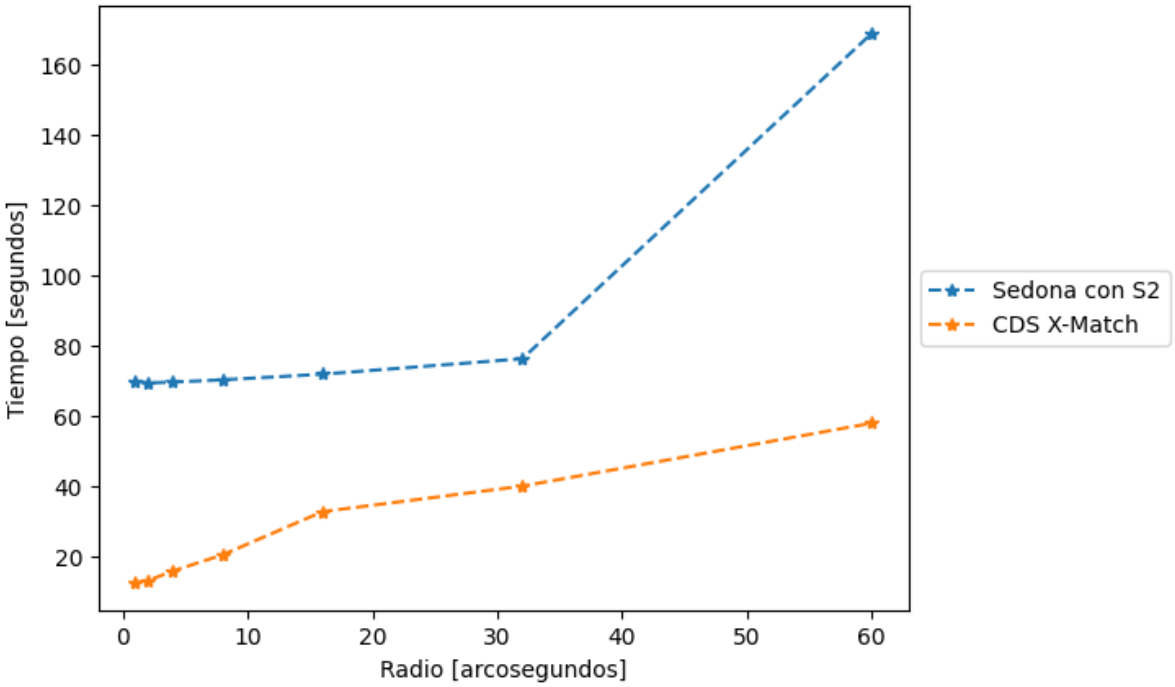


Figura 6.6: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando 50 mil puntos.

20 mil objetos aleatorios

Como podemos observar en la Figura 6.7, el tiempo de consulta para el método desarrollado en este trabajo se mantiene constante en radios menores o iguales a 16 arcosegundos, pero luego estos tiempos aumentan de una forma más abrupta que los de CDS; para las consultas de 60 arcosegundo el método propio se demora en promedio 256 segundos para procesar 20 mil objetos, mientras que CDS solo demora 88 segundos. Esto se puede deber al aumento de celdas de S2 al consultar pues a mayor radio más celdas se consultan, por lo tanto más filas existen en el *DataFrame* de puntos de consulta.

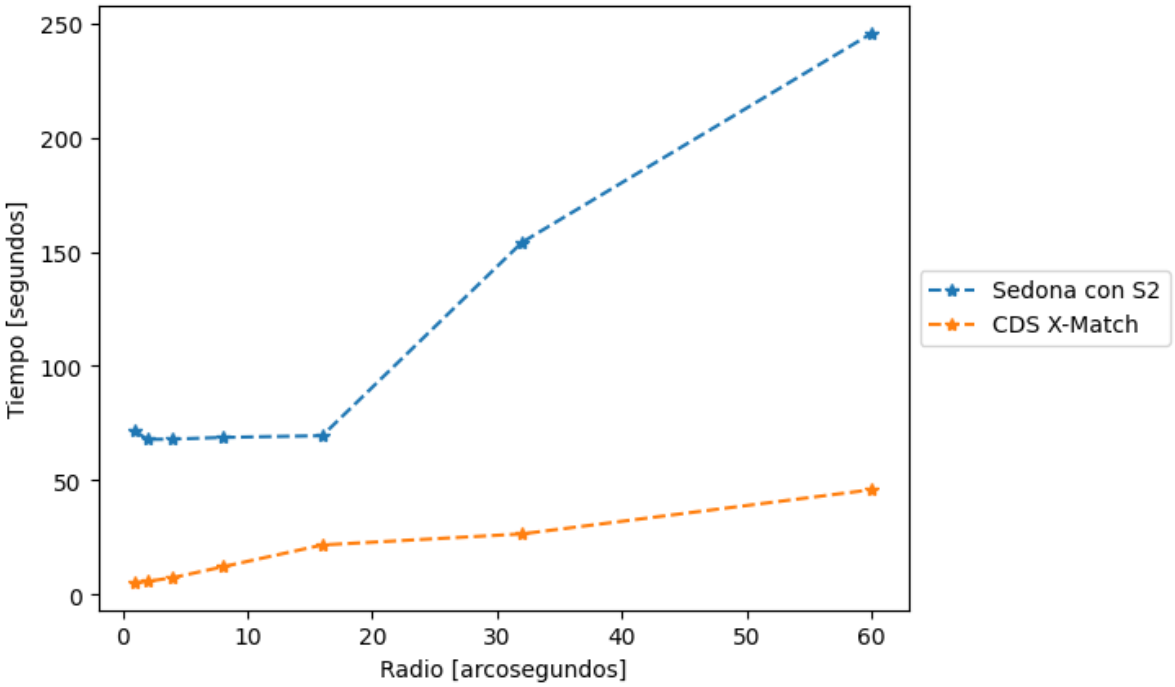


Figura 6.7: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando 20 mil puntos.

Objetos en regiones particulares

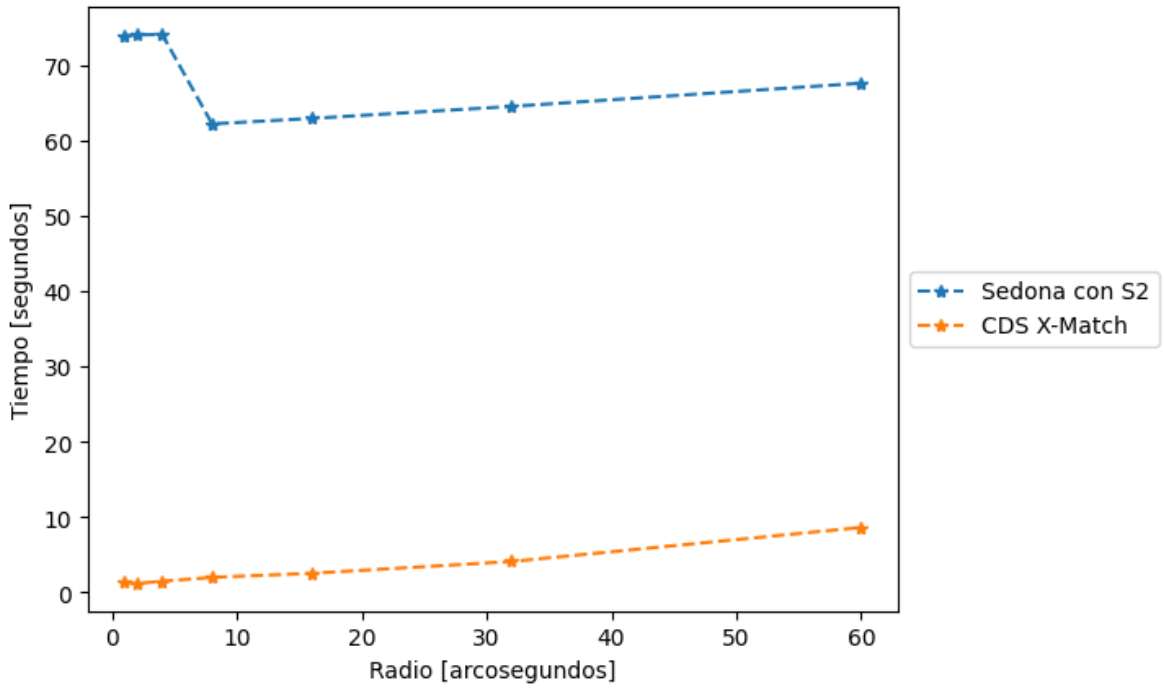


Figura 6.8: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando 1.000 puntos que están máximo a 5 grados del ecuador.

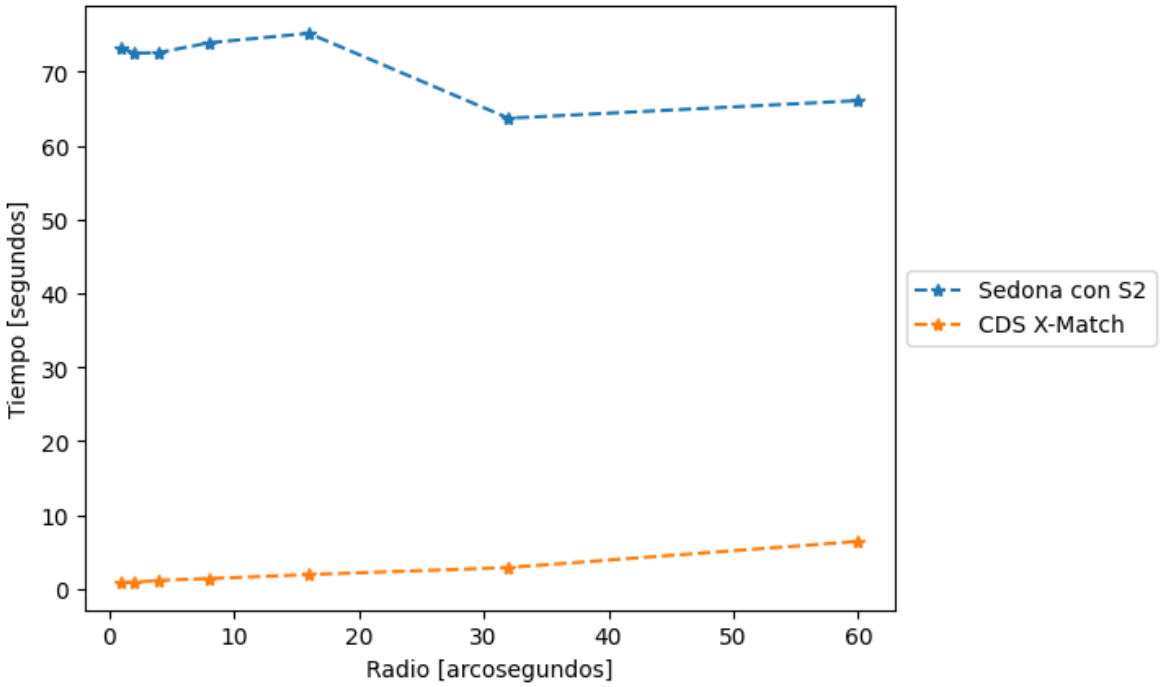


Figura 6.9: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando puntos en el plano galáctico.

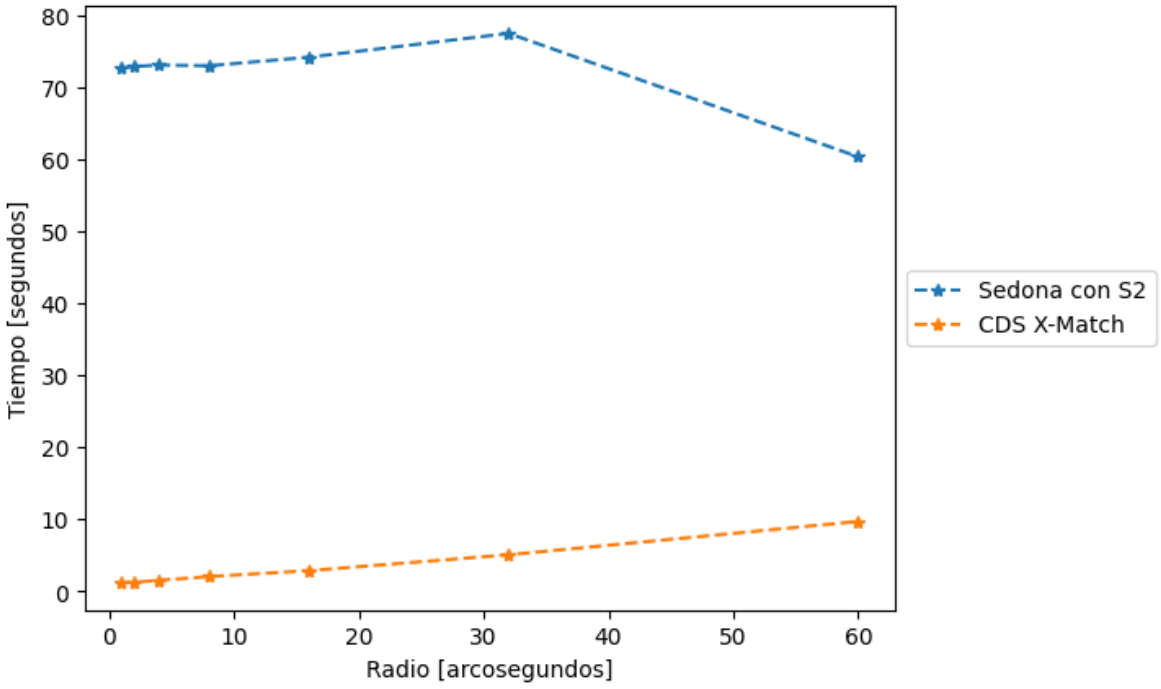


Figura 6.10: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando 1.000 puntos que están máximo a 5 grados de los polos.

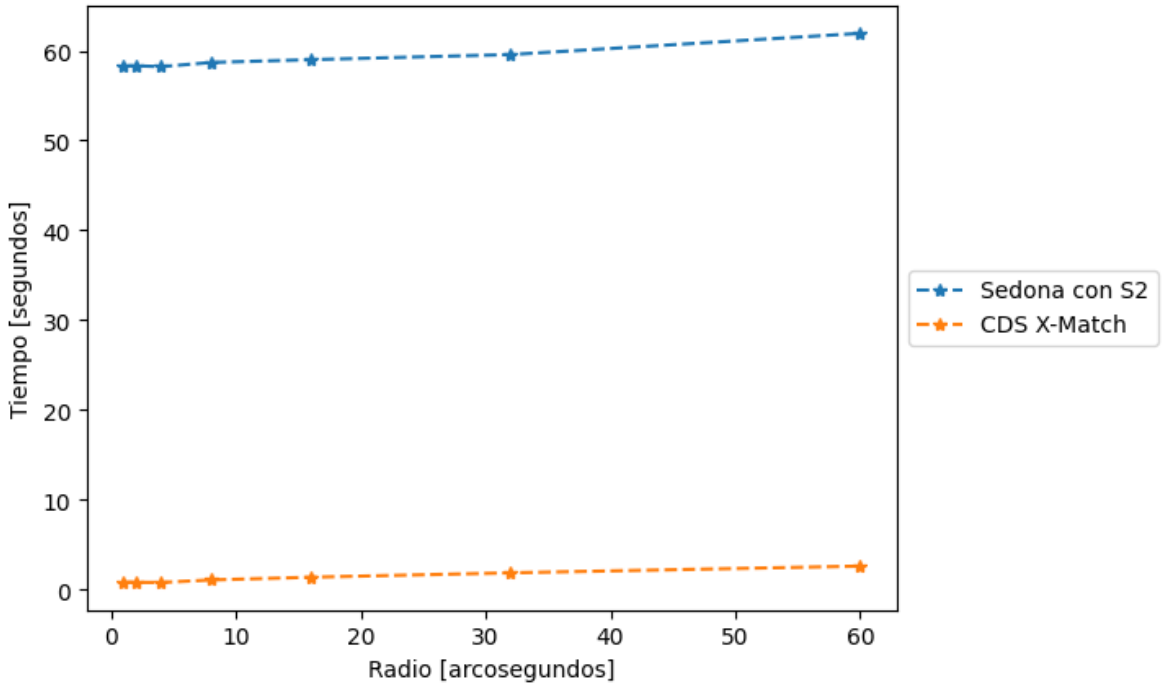


Figura 6.11: Gráficos de los tiempos de *crossmatch* entre el método desarrollado y CDS X-Match, consultando puntos perpendiculares al plano galáctico.

En las Figuras 6.8, 6.9, 6.10 y 6.11, con los resultados para los objetos cercanos al Ecuador, en el plano galáctico, cercanos a los polos y perpendiculares al plano galáctico respectivamente, se ve una clara diferencia entre el método desarrollado y CDS X-Match, esto se explica ya que el método desarrollado no es lo más eficiente al momento de consultar una cantidad pequeña de puntos, pues al ser un algoritmo de *Hash Join* tiene la penalización de recorrer todos los puntos del catálogo consultado, lo que trae como consecuencia que al hacer consultas pequeñas llega tardar lo mismo que hacer una consulta grande. Lo anterior se ejemplifica en las figuras mencionadas al comparar las consultas de 20 mil y 50 mil puntos aleatorios.

A partir de estos resultados se puede notar que dentro de consultas más pequeñas Sedona no parece competir contra CDS, el sistema desarrollado es ineficiente para este tipo de consultas pequeñas. Sin embargo para consultas de mayor tamaño hay una tendencia a aproximarse en los tiempos, pero aún así está lejos del rendimiento de CDS X-Match.

Estos resultados responden las preguntas planteadas al inicio de la subsección, por ejemplo, dependiendo de la cantidad de puntos a consultar, el radio parece no afectar los tiempos de consulta del sistema desarrollado pero si se aumenta la cantidad de puntos a consultar y el radio simultáneamente, entonces el tiempo de consulta aumenta exponencialmente. Por último y como se mencionó anteriormente, el tiempo de consulta del sistema propio es en todos los casos bastante más elevado que los de CDS X-Match, efectuando que la solución desarrollada no sea una gran competidora con CDS.

Algo importante a destacar es que existe un amplio trabajo de optimización que se puede hacer aún, por ejemplo, se pueden utilizar dos índices de S2 con profundidades distintas y así usar el de mayor profundidad para pre-filtrar puntos a consultar y el segundo optimizando

la búsqueda de puntos en específico.

Discusión General

La información mostrada a través de los distintos experimentos sugiere que el método desarrollado en Sedona está lejos del rendimiento de CDS; el trabajo desarrollado en ninguno de los experimentos llevados a cabo mostró el rendimiento ni la precisión esperada. A pesar de que falta hacer más trabajo de ajuste de consultas para incrementar la precisión, también se debe considerar el factor de tiempo para hacer las consultas, algo que es posible aún de optimizar. Sin embargo, cabe destacar que esta es una conclusión preliminar, y que requiere de más pruebas para tener una conclusión definitiva. No obstante, estos resultados son un indicador relevante para inferir que Sedona no sería la mejor opción para desarrollar el servicio de *crossmatch*. Sería beneficioso e interesante compartir estos resultados con el equipo de desarrollo de Apache Sedona, ya que este *framework* se encuentra en constante mantenimiento y actualizaciones, tanto de rendimiento como de precisión[4].

Con todos los resultados presentados a lo largo de este trabajo, se pueden listar las siguientes conclusiones obtenidas de acuerdo con los resultados de los experimentos:

- Sedona actualmente no parece ser la mejor solución para reemplazar CDS X-Match en la *pipeline* de ALeRCE, esto debido a su forma de calcular las distancias y al rendimiento que posee.
- Se pudo observar que el formato de *GeoParquet* es más eficiente en términos de recursos utilizados pero bastante más lento al momento de procesar consultas.
- El método desarrollado en este trabajo funciona mejor al momento de realizar consultas con una gran cantidad de puntos (sobre los 20 mil puntos), pero no es tan eficiente al momento de consultar una cantidad menor.
- Se encontró una cierta cota mínima de tiempo de consulta que varía según el nivel de profundidad del índice y el tamaño del radio; esto se puede apreciar en la Figura 6.5. En esta figura se puede notar un aumento exponencial del tiempo de consulta a partir de cierto radio; esto sigue la lógica del aumento de área en un círculo dado por la fórmula: $\pi \times radio^2$. Sumado a esta fórmula, también afecta la profundidad de los índices de S2, debido a que la cantidad de celdas crece de manera exponencial dada por la fórmula: $6 \times 4^{profundidad\_S2}$. Luego, al juntar estas dos fórmulas se puede explicar el crecimiento exponencial que se da al probar una cantidad considerable de puntos con un radio de 1 arcominuto.
- Se pudo mejorar los tiempos de consulta encontrando una profundidad ideal de 9 para los índices de S2. Además al utilizar un R-Tree como índice de Sedona mejoraron bastante los tiempos al momento de consultar radios menores a 16 arcosegundos; esto se puede apreciar en la Figura 6.5.

[4]En las últimas semanas agregaron métodos para calcular las distancias de los puntos como si estuviesen sobre una esfera.

Capítulo 7

Conclusión

7.1. Trabajo Realizado

El resultado final de este trabajo fue la creación de un sistema que permite realizar *crossmatching* a través de una API desarrollada utilizando FastAPI; la API está enfocada en el catálogo CatWISE2020 y en su integración con el servidor de ALeRCE. Se realizó un estudio exhaustivo del rendimiento utilizando distintas metodologías y configuraciones, dentro de las cuales destacan los índices de Google S2 y los Geohashes, y sus distintas opciones de profundidad.

Con respecto a los objetivos específicos planteados en la Sección 1.2, se puede decir que se lograron cumplir casi en su totalidad, sumado a esto, los objetivos sirvieron como base para el desarrollo paso a paso del trabajo. Se logró cumplir la parte del objetivo 8 que hace referencia a la cantidad de 350 consultas por segundo para un radio de un arcosegundo, esto haciendo consultas sobre los 50.000 puntos con un tiempo total de 70 segundos, lo que lleva a hacer 714 consultas por segundo en promedio. En el caso del arcosegundo, en donde no existía un mínimo establecido se lograron realizar alrededor de 303 consultas por segundo para el mismo set de 50 mil puntos.

Sobre la correctitud de los resultados, se puede concluir que está lejos de ser ideal, el error va desde el 1 % cuando se consulta cerca del Ecuador, hasta más del 90 % de error cuando se consulta cerca de los polos. Sin embargo no se realizó ningún ajuste para mejorar la precisión de los resultados, los cuales eran objetos que aparecían en CDS pero no en el sistema propio.

En resumen, a pesar de que Sedona está en desarrollo, presenta cambios constantes y que actualmente no parece una buena base para implementar *crossmatch*, se logró un sistema funcional, el cúal tiene un amplio margen de mejora tanto en los tiempos de consulta como en la precisión de los resultados, lo cual si en un futuro se mejoran estos apartados, Sedona se podría convertir en una mejor opción para implementar *crossmatch*.

7.2. Trabajo Futuro y posibles optimizaciones

A pesar de haber logrado los objetivos, hay algunos detalles que se pueden perfeccionar para mejorar tanto la precisión como el rendimiento del sistema, tales como:

- Utilizar un doble índice de Google S2, por ejemplo, crear un catálogo con una columna que contenga los `cellIds` con una profundidad de 4, y otra columna con los `cellIds` con una profundidad de 9. La idea es utilizar los de profundidad 4 como llave de partición[1], lo que generaría alrededor de 1500 particiones, permitiendo descartar una gran parte del catálogo al momento de hacer una consulta con el algoritmo de *Hash Join*.
- Crear una fórmula que permita ajustar el radio de consulta según el valor de Dec (latitud) en el cual se encuentra el punto a consultar, y luego hacer un post-filtrado utilizando la función `ST_DistanceSphere` para filtrar los falsos positivos.
- Realizar experimentos con un cluster de Sedona, debido a que Sedona está optimizado para trabajar con un cluster y esto podría tanto disminuir los tiempos de consulta como hacer una mejor utilización de los recursos.
- Buscar otros índices para optimizar las consultas, por ejemplo, utilizar los índices de H3[2] en Sedona que van a estar disponibles en un futuro[3].
- Buscar otra forma de realizar la consulta que no sea creando un círculo, la cual puede ser: filtrar primero por celdas cercanas y luego ocupar la distancia para terminar de filtrar los puntos de interés.

[1] Se puede utilizar el método de `partitionBy()` de Spark al momento de guardar un archivo.
[2] `https://h3geo.org/`
[3] `https://github.com/apache/sedona/issues/914`

Bibliografía

[1] Alejandra Alarcón. Servicio de crossmatching de objetos astronómicos. Universidad de Chile, Santiago, Chile, 2022.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Mariem Brahem, Karine Zeitouni, and Laurent Yeh. Astroide: A unified astronomical big data processing engine over spark. *IEEE Transactions on Big Data*, 6:477–491, 2020.

[4] Peter R. M. Eisenhardt, Federico Marocco, John W. Fowler, Aaron M. Meisner, J. Davy Kirkpatrick, Nelson Garcia, Thomas H. Jarrett, Renata Koontz, Elijah J. Marchese, S. Adam Stanford, Dan Caselden, Michael C. Cushing, Roc M. Cutri, Jacqueline K. Faherty, Christopher R. Gelino, Anthony H. Gonzalez, Amanda Mainzer, Bahram Mobasher, David J. Schlegel, Daniel Stern, Harry I. Teplitz, and Edward L. Wright. The CatWISE preliminary catalog: Motions from WISE and NEOWISE data. *The Astrophysical Journal Supplement Series*, 247(2):69, apr 2020.

[5] Eric D. Feigelson and G. Jogesh Babu. Big data in astronomy. *Significance*, 9(4):22–25, 2012.

[6] Raphael Finkel and Jon Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974.

[7] F. Förster, G. Cabrera-Vives, E. Castillo-Navarrete, P. A. Estévez, P. Sánchez-Sáez, J. Arredondo, F. E. Bauer, R. Carrasco-Davis, M. Catelan, F. Elorrieta, and et al. The automatic learning for the rapid classification of events (ALeRCE) alert broker. *The Astronomical Journal*, 161(5):242, apr 2021.

[8] Genova, F., Egret, D., Bienaymé, O., Bonnarel, F., Dubois, P., Fernique, P., Jasniewicz, G., Lesteven, S., Monier, R., Ochsenbein, F., and Wenger, M. The cds information hub - on-line services and links at the centre de données astronomiques de strasbourg. *Astron. Astrophys. Suppl. Ser.*, 143(1):1–7, 2000.

[9] David Goodge and Guido van Rossum. Docstring conventions. PEP 257, Python, 2001.

[10] K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759–771, apr 2005.

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[12] Ioannis Karatzas. and Steven E. Shreve. *Brownian Motion and Stochastic Calculus*. Springer, Berlin, 2nd edition, 2000.

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

[14] Christian S. Perone. Google's s2, geometry on the sphere, cells and hilbert curve. *Terra Incognita*, 08 2015.

[15] Philip Protter. *Stochastic Integration and Differential Equations*. Springer, 1990.

[16] Daniel Revuz and Marc Yor. *Continuous martingales and Brownian motion*. Number 293 in Grundlehren der mathematischen Wissenschaften. Springer, Berlin [u.a.], 3. ed edition, 1999.

[17] Iping Supriana Suwardi, Dody Dharma, Dicky Prima Satya, and Dessi Puji Lestari. Geohash index based spatial data model for corporate. In *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 478–483, 2015.

[18] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, Python, 2001.

[19] Marc Wenger, François Ochsenbein, Daniel Egret, Pascal Dubois, François Bonnarel, Suzanne Borde, Françoise Genova, Gérard Jasniewicz, Suzanne Laloë, Soizick Lesteven, et al. The SIMBAD astronomical database-The CDS reference database for astronomical objects. *Astronomy and Astrophysics Supplement Series*, 143(1):9–22, 2000.

[20] Edward L. Wright, Peter R. M. Eisenhardt, Amy K. Mainzer, Michael E. Ressler, Roc M. Cutri, Thomas Jarrett, J. Davy Kirkpatrick, Deborah Padgett, Robert S. McMillan, Michael Skrutskie, S. A. Stanford, Martin Cohen, Russell G. Walker, John C. Mather, David Leisawitz, Thomas N. Gautier, Ian McLean, Dominic Benford, Carol J. Lonsdale, Andrew Blain, Bryan Mendez, William R. Irace, Valerie Duval, Fengchuan Liu, Don Royer, Ingolf Heinrichsen, Joan Howard, Mark Shannon, Martha Kendall, Amy L. Walsh, Mark Larsen, Joel G. Cardon, Scott Schick, Mark Schwalm, Mohamed Abid, Beth Fabinsky, Larry Naes, and Chao-Wei Tsai. The wide-field infrared survey explorer (wise): Mission description and initial on-orbit performance. *The Astronomical Journal*, 140(6):1868–1881, nov 2010.

[21] 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*, SIGSPATIAL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[22] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[23] Petar Zečević, Colin T. Slater, Mario Jurić, Andrew J. Connolly, Sven Lončarić, Eric C. Bellm, V. Zach Golkhou, and Krzysztof Suberlak. AXS: A framework for fast astronomical data processing based on apache spark. *The Astronomical Journal*, 158(1):37, 2019.