UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Implementación de Apache Spark para análisis de proteínas

HÉCTOR PATRICIO CASTILLO VÁSQUEZ

Profesor Guía: RENZO ANGLES

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
Julio, 2018

CONSTANCIA

La Dirección del Sistema de Bibliotecas a través de su encargado Biblioteca Campus Curicó certifica que el autor del siguiente trabajo de titulación ha firmado su autorización para la reproducción en forma total o parcial e ilimitada del mismo.

Curicó, 2022

Dedicado a mi madre y hermanos.

AGRADECIMIENTOS

Agradecimientos a mi familia, la cual ha formado parte fundamental de mi vida en todo este proceso, en especial a mi madre y hermanos. Agradezco también a mis profesores de la universidad, asi como a todo el personal que día a día se dedica a crear un ambiente pleno para nuestro crecimiento. Es necesario también mencionar a mi profesor guía, el Sr. Renzo Angles que me apoyó desde el primer momento y me guió a través de todo el proceso de construcción de la memoria. También a otros profesores a los cuales agradezco mucho como son el Profesor Edgardo Ortiz, el cual me enseñó no solo contenidos si no también lecciones de vida. Agradezco también al profesor Matthew Bardeen por haber sido un amigo y un profesor dentro de todos estos años como estudiante, así como a todos los profesores del departamento de ciencias de la computación. Agradezco también a mis compañeros y amigos los cuales de una u otra forma me han ayudado a crecer. Además agradezco al profesor de matemáticas del colegio San José de Parral, don Carlos Norambuena por ser una gran ayuda tanto profesional y emocional en el momento de elección de una carrera.

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

RESUMEN

Este proyecto se trata sobre el uso de tecnologías distribuidas para resolver problemas relacionados con la manipulación de datos masivos y complejos. El problema específico abordado es el de AFAL2, una aplicación web que permite analizar datos de proteínas, los cuals se obtienen del Protein Data Bank (PDB).

AFAL2 permite consultar tres tipos de frecuencias. Además, este sistema funciona sobre una base de datos relacional, hecha en PostgreSQL y no tiene el mejor tiempo de respuesta frente a operaciones complejas. Además considerando que el Protein Data Bank crece constantemente, esto puede ser un problema a futuro.

Para la investigación realizada fue necesario analizar la base de datos de AFAL2 y extraer las consultas que permiten responder a las frecuencias para luego replicar estas consultas usando usando una herramienta de programación distribuida, específicamente un framework llamado Apache Spark. Finalmente se realizaron pruebas para comparar la efectividad del algoritmo en Spark versus la base de datos hecha en PostgreSQL.

Para el desarrollo de los algoritmos que responen a las frecuencias de AFAL2, se usó el modelo de programación Map Reduce, el cual consiste en dos operaciones, la primera es el map que separa los datos en los distinos nodos que componen el sistema distribuido para dividir el trabajo y más tarde unirlos en la etapa de Reduce. Replicar las consultas requirió el uso de varias operaciones Map y Reduce. Para probar la efectividad de los algoritmos con respecto a las consultas sql se realizaron varias pruebas en distinos ambientes, entre los que destacan notebook con PostgreSQL, notebook con Spark, y un cluster de Amazon EMR con Spark. En cuanto a los resultados, si bien no fueron los más favorables para Spark, se puede concluir que la tasa de crecimiento de tiempo de respuesta con respecto a un aumento en la cantidad de datos es mucho menor en Spark que en PostgreSQL, por lo que si en algún momento el Protein Data Bank crece lo suficiente, se podrán encontrar problemas lo suficientemente grandes como para aprovechar el potencial de Spark.

1. Introducción

En este capítulo se define el problema, la propuesta de solución y los objetivos con los que el proyecto plantea ejecutar la misma. Además se define la hipótesis y los resultados obtenidos después de la ejecución de la posible solución.

1.1. Definición del Problema

Hoy en día existen muchos sistemas que trabajan con datos masivos, desde los buscadores de internet como Google, los sistemas de predicción meteorológica y las grandes redes sociales como Facebook hasta cualquier sistema que requiera el almacenamiento de muchos datos.. Tal es el caso de AFAL2, un sistema web que permite analizar datos de proteínas. Esta aplicación web tiene la funcionalidad de buscar patrones estructurales en una base de datos de proteínas, creada con datos obtenidos del Protein Data Bank. El principal problema de AFAL2 es que se ejecuta sobre una base de datos relacional (PostgreSQL) y su tiempo de respuesta es muy elevado. Además los datos son obtenidos del Protein Data Bank, el cual se actualiza y crece cada día, por lo tanto se cree que la base de datos quedará obsoleta en algún momento, dándole la oportunidad a una tecnología más escalable.

1.2. Propuesta de solución

Lo que se pretende es comprobar la eficiencia de la base de datos relacional de AFAL2 contra un sistema distribuido. Esta base de datos está implementada en PostgreSQL y será sometida a pruebas para posteriormente compararada con un algoritmo de programación distribuida implementado en Apache Spark.

1.3. Hipótesis

Un algoritmo desarrollado en Apache Spark puede responder a una consulta en menor tiempo que una base de datos relacional implementada en postgresql.

1.4. Objetivos

Objetivo General: Implementar una plataforma basada en Apache Spark, que permita consultar y analizar datos de proteínas de la misma forma que lo hace AFAL2.

Objetivos específicos.

- Describir los datos del Protein Data Bank usados por AFAL2.
- Diseñar un modelo para almacenar los datos para que puedan ser leídos por Apache Spark.
- Identificar las consultas de AFAL2.
- Desarrollar las consultas de AFAL2 en Spark.
- Comparar la eficiencia de Map reduce en Apache Spark versus base de datos relacional en Postgresql.

1.5. Resultados.

La hipótesis se rechaza, ya que actualmente Spark no es más rápido que postgresql para resolver las consultas de AFAL2. Sin embargo se cree que en un futuro Spark lo superará, ya que tiene una tasa de aumento de tiempo con respecto al aumento de datos mucho menor a la de postgresql. Es decir, si se produce un gran aumento en la cantidad de datos, el tiempo de respuesta en posgresql aumentará en proporción a ese aumento, mientras que el aumento en Spark podría ser mucho menor. Con el crecimiento adecuado en el conjunto de datos, es posible que el uso de Spark sea factible en algún momento.

2. Contexto

En este capítulo se presentan todos los antecedentes para la comprensión del proyecto. Tales como el Protein Data Bank, las estructuras moleculares estudiadas como las proteínas, aminoácidos y ligandos. Además se incluyen las tecnologías distribuidas que se usaron para implementar la solución. Entre ellas Apache Hadoop, Apache Spark y el concepto de Map Reduce. Finalmente se presentan trabajos relacionados con el uso de tecnologías distribuidas para el análisis de proteínas.

2.1. Protein Data Bank y sus estructuras

2.1.1. Protein Data Bank

El Protein Data Bank(PDB) es un repositorio que almacena estructuras biológicas macromoleculares, tales como proteínas y ácidos nucleicos. Se encuentra bajo el dominio público, y cualquier persona puede hacer uso de los datos que se encuentran en él. Fue creado en 1971 en Brookhaven National Laboratory bajo el mando de Walter Hamilton con el objetivo de entregar informaciń a estudiosos interesados en visualizar estructuras de proteínas[2]. Originalmente contaba con siete estructuras, pero gracias a los avances tecnológicos en biología estructural, como son la cristalografía o resonancia magnética nuclear, actualmente cuenta con más de 135000 proteínas, lo cual demuestra un crecimiento exponencial [1].

Cada estructura macromolecular en el PDB es identificada con una clave de cuatro carácteres alfanuméricos que forman una clave única para cada proteína, ADN, ARN o híbridos. Estos archivos contienen datos de la secuencia de aminoácidos y ligandos, además contiene los átomos que forman las moléculas antes mencionadas, junto a sus átomos y coordenadas espaciales.

2.1.2. Proteínas y sus componentes

Las proteínas son biomoléculas o macromoléculas presentes en los seres vivos. Estas cumplen diferentes funciones, ya sean estructurales, enzimáticas, reguladoras, metabólicas, defensivas (ya que crean los anticuerpos), entre otras. Estas estructuras están formadas por una o varias cadenas de aminoácidos. Los aminoácidos son moléculas de tamaño pequeño formadas por un átomo de carbono central, y átomos de un grupo amino, carboxilo, hidrógeno y un grupo R [4].
Los ligandos son moléculas pequeñas o iones que interactúan con las proteínas de forma transitoria y reversible. Esta interacción permite a los organismos responder a cambios en el ambiente, lo que convierte en esto en una parte fundamental para la vida. Una misma proteína puede unirse con varios ligandos diferentes. La unión ligando-proteína produce un cambio conformacional en la proteína y por ende puede influenciar en sus funciones.

Una proteína está formada por cadenas de aminoácios, en la figura 2.1 se puede observar que los círculos representan los aminoácidos y forman una cadena entre ellos, mientras que los ligandos son externos a la cadena. Es importante mencionar que todos los componentes están separados por una distancia, la cual está medida em Amstrong y se usa para definir un radio de revisión en las consultas de la aplicación web, AFAL2.

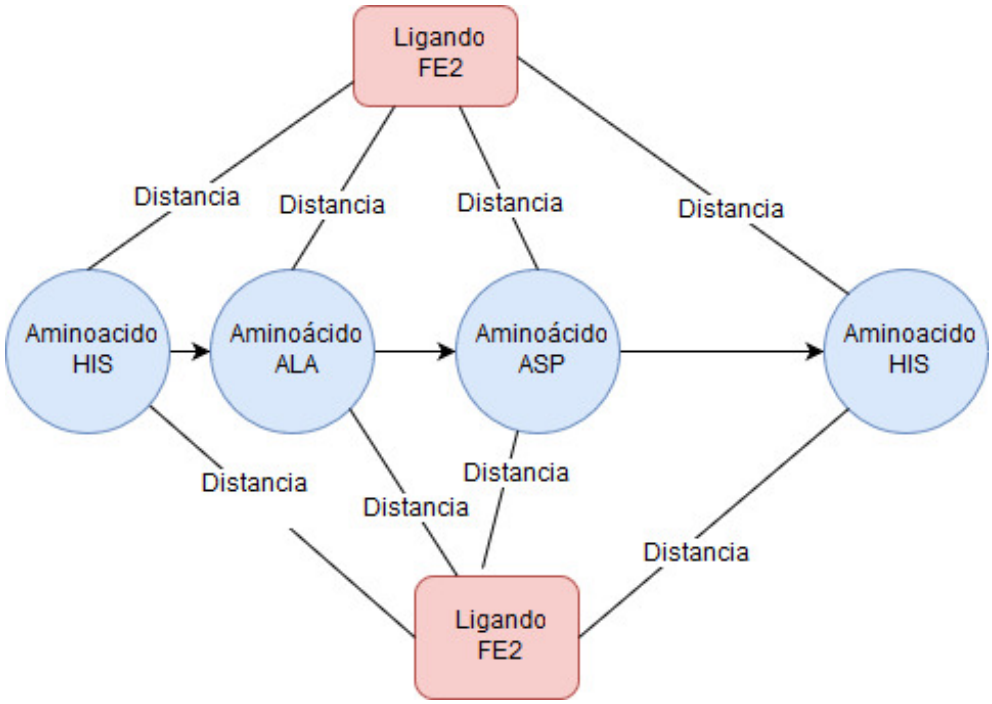

Figura 2.1: Ejemplo de una proteína a nivel molecular, y la relación entre aminoácidos y ligandos.

2.2. AFAL2

AFAL2 es una aplicación web que permite la búsqueda de información estructural detallada de proteínas, mediante el uso de datos obtenidos del PDB. AFAL2 permite al usuario seleccionar un ligando en particular, con lo que procede a analizar todas las estructuras del PDB que contengan esa molécula. Posteriormente se muestran los resultados del análisis, mostrando las las estructuras cercanas al ligando (aminoácidos presentes en la unión) y que cumplen con los criterios indicados. Estos criterios pueden ser especificados usando una serie de filtros que incluyen la familia de proteínas, organismo fuente y criterios de exclusión cristalográfica como la distancia entre el ligando y los aminoácidos.

Para analizar proteínas en AFAL2 usuario puede elegir cualquiera de los filtros que aparecen en la figura 2.2.

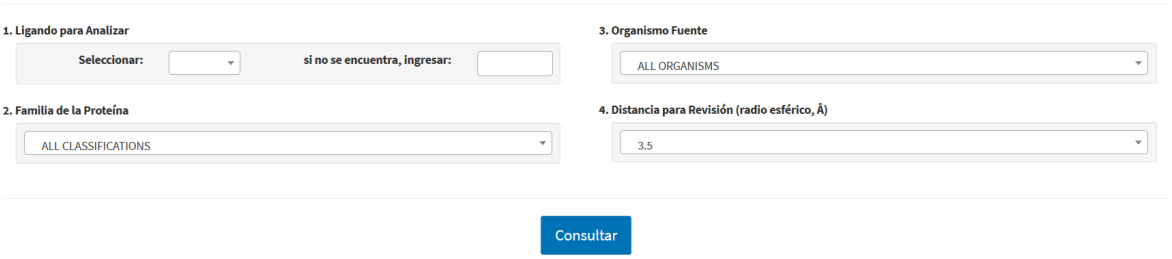

Figura 2.2: Parametros de entrada de AFAL2.

- Ligando: Es el ligando que AFAL2 analiza, el usuario debe seleccionar uno de los ligandos existentes. Todas las operaciones de AFAL2 requieren este parámetro, por lo que es obligatorio. Además, existe la opción de seleccionar un ligando alternativo, en caso de no encontrar el seleccionado.
- Familia de la Proteína: Es la familia en la cual se clasifica la proteína, esta información se obtiene de los archivos del PDB. Este filtro es opcional.
- Organismo Fuente: Es el organismo del cual se obtiene la proteína, al igual que la familia de la proteína, es un filtro opcional.
- Distancia de revisió n o radio esférico: Es el radio de cercanía al ligando dentro del cual se analizará. Es un parámetro obligatorio (3.5 por defecto).

AFAL2 entrega como respuesta tres tipos de resultados, la frecuencia específica, frecuencia general, y la ocurrencia.

- La Frecuencia Específica agrupa y cuenta el número total de apariciones de cada aminoácido dentro del radio esférico entregado por el usuario (ver figura 2.3).
- La Frecuencia General cuenta los aminoácidos en el radio esférico entregado por el usuario, pero a diferencia de la frecuencia específica, solo considera una aparición de cada aminoácido por proteína (ver figura 2.4).

- La Ocurrencia tiene que ver con la cantidad de apariciones del ligando buscado, dentro de las proteínas. La ocurrencia responde con el número de proteínas en las que aparece un ligando cierto número de veces (ver figura 2.5).

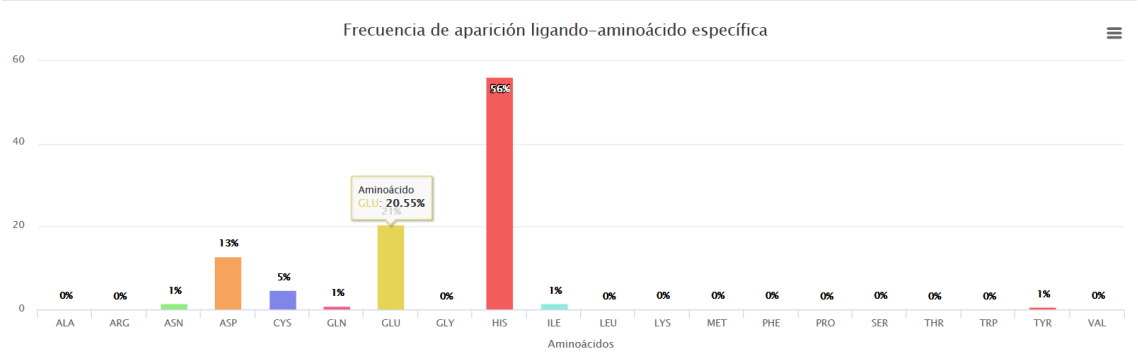

Figura 2.3: Frecuencia Específica en AFAL2

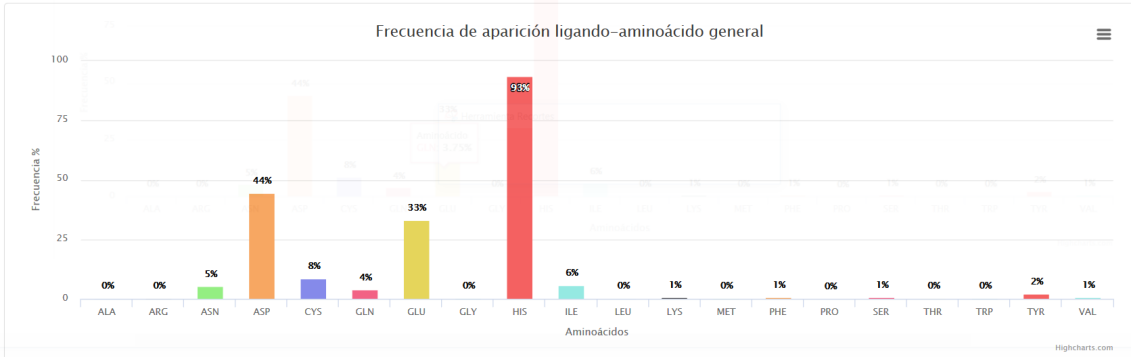

Figura 2.4: Frecuencia General en AFAL2

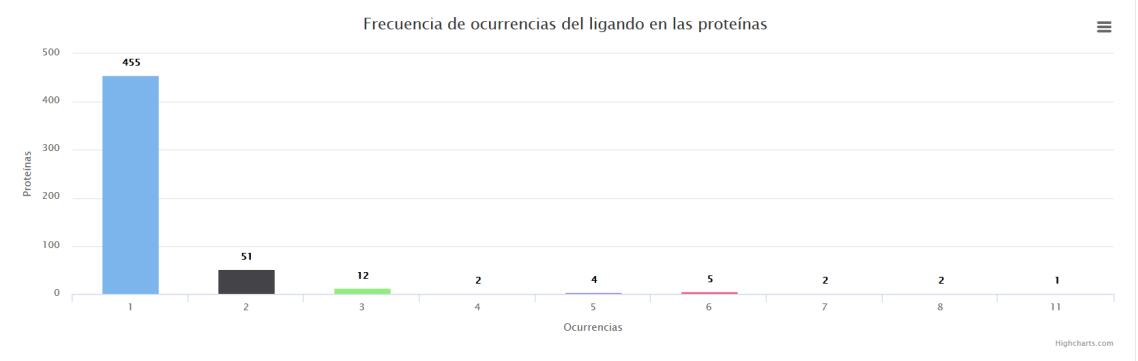

Figura 2.5: Ocurrencia en AFAL2

2.3. Apache Hadoop

Apache Hadoop es la herramienta principal usada en este proyecto, es un framework de código abierto que permite soportar aplicaciones distribuidas, usando su sistema de archivos distribuido (HDFS) y su modelo de programación MapReduce. Un Distributed File System (DFS), es un sistema que maneja archivos o carpetas a través de varios computadores o servidores en un cluster, y que permite a los usuarios acceder a estos de manera transparente. En pocas palabras tiene la misma función de los sistemas de archivos convencionales, con la diferencia de que los archivos están distribuidos y replicados en muchas máquinas pero organizan la información a través de la red, de tal forma que el usuario percibe que todo está siendo realizado por una sola máquina.

Hadoop usa su propio sistema de archivos distribuido, el Hadoop DistributedFile System (HDFS), este sistema de archivos está escrito en Java, es escalable y portátil. HDFS funciona con múltiples nodos, donde cada nodo corresponde a una máquina del cluster. Esto significa que cada máquina aporta parte de su disco para almacenar cualquier tipo de datos. Siendo el Namenode la pieza central, que tiene el conocimiento de que está almacenado en los demás nodos y aprueba las peticiones. Los nodos que se dedican únicamente al almacenamiento son llamados Datanodes. En cuanto a la resistencia a fallas del HDFS, los Datanodes se encuentran replicados, pero no es posible replicar el Namenode, por lo que si este falla, el sistema de archivos dejará de funcionar. Para evitar esto también existe un Secondary Namenode que es un servicio presente en otra máquina que genera checkpoints periódicamente para realizar restauraciones, pero no reemplaza el namenode. Si lo que se requiere es una alta disponibilidad (High availability), es necesario recurrir al servicio Backup Namenode, mediante el cual, un usuario puede cargar una imagen del Namenode en otro equipo [11].

2.4. Map Reduce

MapReduce es un modelo de programación usado por Apache Hadoop que permite el procesamiento de grandes cantidades de datos de manera distribuida y paralela [8]. Como su nombre lo indica, cuenta con dos tareas principales, Map y Reduce. Primero se realiza el trabajo de Map donde un grupo de datos es leído y procesado

para obtener una salida parcial, la cual consiste en tuplas <key, value>. Esta salida es recibida por el trabajo Reduce, el cual toma estas tuplas y las agrupa según su key creando la salida final. El ejemplo más común de MapReduce es un programa llamado "WordCount". Este programa cuenta las palabras presentes en varios archivos de texto y entrega como salida cada palabra con su respectiva cantidad. Podemos observar un ejemplo del funcionamiento de MapReduce en la figura 2.6. La entrada son archivos de texto plano que contienen palabras separadas por un espacio. Es posible identificar que los datos se distribuyen y cada nodo se encarga de la porción de datos que le corresponde; para esto crea un diccionario de la forma $< key, value >$ donde cada palabra es la clave y 1 es el valor. Estas tuplas son agrupadas por el clave para finalmente sumar cada valor y obtener el resultado final (etapa de reducing).

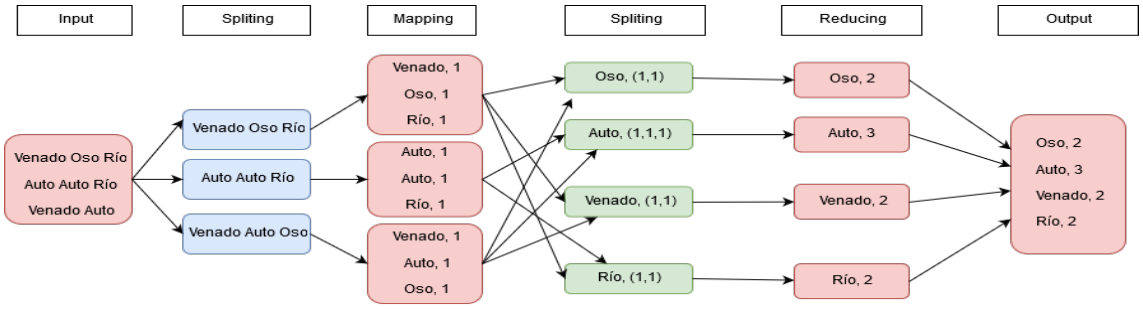

Figura 2.6: Ejemplo de Map Reduce

2.5. Apache Spark

Apache Spark es un rápido y ligero motor para el procesamiento de grandes cantidades de datos. Apache Spark puede correr sobre Hadoop y permite desarrollo tanto en Scala, Python y Java. A diferencia de Hadoop, Spark permite programar de forma más sencilla y amigable porlo que es recomendable utilizarlo para implementar Hadoop en AFAL2[5].

La principal característica de la programación en Spark, es la creación de objetos del tipo RDD. Un RDD (Resilient Distributed Dataset) es el tipo de dato abstracto básico de la programación en Apache Spark, el cual permite a los programadores crear datasets y almacenarlos en memoria de forma que el programa sea tolerante a

fallas[12].

Las clases y funciones más importantes en Spark son las siguientes

- JavaRDD$< Object >$: Es la clase que representa los "RDD", permite almacenar objetos del tipo String, Integer, Float, incluso arreglos.
- JavaPairRDD$< Object, Object >$: Esta clase también permite crear "RDD", con la diferencia de que estos RDD son diccionarios que almacenan tuplas de la forma $< Key, Value >$.
- map(): Esta función permite transformar un RDD, es decir permite recorrerlo y ejecutar una misma función para cada fila de este. Siempre retorna un RDD.
- mapToPair(): Este método es un map para el objeto JavaPairRDD, permite ejecutar una misma función para cada valor de cada tupla en el diccionario.
- flatmap(): Es similar a la función map, con la diferencia de que puede retornar una lista de elementos o un iterador y no solo uno.
- reduceByKey(): Este método se usa para reducir las claves de un RDD que tenga almacenado un diccionario, para esto se agrupan las claves iguales y se realiza una operación con los valores de cada una .
- collect(): Sirve para recuperar datos desde las clases propias de Spark y almacenarlos en clases nativas de Java como las clases "List","ArrayList", entre otras.

Al igual que en Hadoop, el ejemplo básico en Spark es el WordCount, cuyo código es el siguiente:

```
1  SparkSession spark = SparkSession
2        .builder()
3        .appName("JavaWordCount")
4        .getOrCreate();
5
6  JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
7  JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACE.split(s)).iterator());
8  JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));
9  JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);
10 List<Tuple2<String, Integer>> output = counts.collect();
```

Cómo se puede observar en el código, el primer paso es leer el archivo de texto con la función "spark.readd()", resultando un RDD como el de la figura 2.7.A. Luego se ejecuta una funcion "flatMap"para convertir cada palabra en una fila del RDD, como se puede observar en la figura 2.7.B. Una vez que cada palabra es una fila del RDD, se crea un nuevo RDD, pero esta vez es un diccionario de la forma "$< key, value >$"donde cada palabra es una clave y el valor de cada una es 1 (ver figura 2.7.C). Finalmente se ejecuta un método "reduceByKey"sobre el RDD que contiene el diccionario, este método agrupa las claves idénticas y suma los valores de cada una, obteniendo el número total de apariciones de cada palabra, tal y como se puede apreciar en la figura 2.7.D.

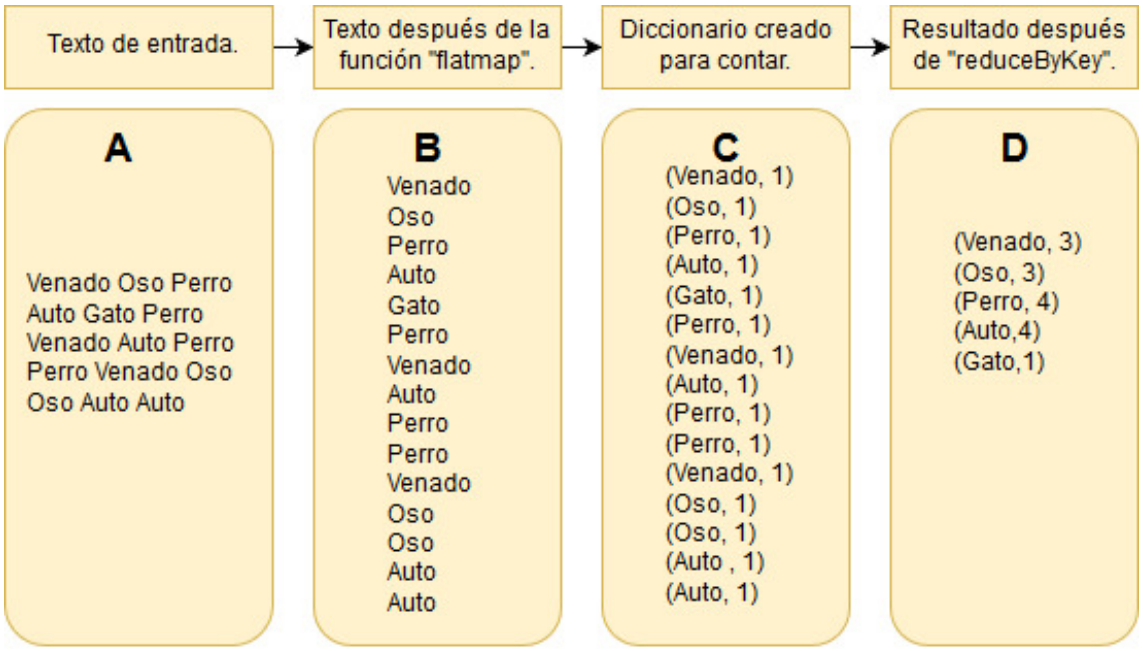

Figura 2.7: Funcionamiento de WordCount en Spark

2.6. Trabajo relacionado

Hay diversos proyectos que han relacionado Apache Hadoop con PDB, la principal característica de ellos, es que usan los archivos tal y como son obtenidos desde el sitio. En cambio en este proyecto los archivos son preprocesados y se obtienen nuevos archivos con características más deseables. A continuación se presentan algunos

trabajos relacionados al procesamiento de proteínas en sistemas distribuidos.

Scalable data analytics of the pdb with the macromolecular transmission format (MMTF) and Big data Technologies[7]. Un pequeño poster que introduce el concepto de las tecnologías distribuidas y el análisis de proteínas. usando los archivos .mmtf, los cuales a diferencia de los archivos pdb y mmcif, son específicos para el análisis de estructuras en 3d, especialmente para su modelado. A diferencia de los otros formatos, este tiene un costo menor de almacenamiento, pero es más lento de procesar. Además se presentan las ventajas de la herramienta Mmtf-Spark, que es un framework que trabaja a base de Spark y contiene funciones que facilitan el procesamiento de los archivos mmtf.

Hadoop: Solution for Big Data Challenges in Bioinformatics and its prospective in India[10]. Este paper habla sobre la importancia del big data en la bioinformática, no sólo en el análisis de proteínas, si no en muchas aplicaciones, tales como el estudio de genes. Además relata las ventajas de usar hadoop y por qué es una buena solución en problemas de big data, y como algunas empresas en la India lo han comenzado a utilizar.

SpeedDB: Fast structural protein searches [6]. SpeedDB es una herramienta web que permite el análisis de proteínas, lo cual se realiza mediante consultas SQL. Estas consultas vienen previamente cargadas, y el usuario tiene la opción de crear consultas nuevas. Cabe mencionar que speedDB implementa paralelismo en sus funciones.
Este paper también habla de los distintos tipos de interacciones que hay entre los componentes de una proteína, como los enlaces iónicos, de hidrógeno y otros. Además de la metodología que debe usarse para trabajar con este sistema, comenzando por el diseño de una base de datos (creada a partir de archivos mmCIF y PDB).

Finalmente este paper contiene resultados de investigaciones previas, donde se justifica el por qué debería usarse este software.

An Overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics[9]. Muy parecido a uno de los papers an-

teriores pero con mucho más detalle, habla de la gran cantidad de datos que generan las aplicaciones en Bioinformática y la necesidad de paralelizar el trabajo. Explica lo que es Hadoop y lista varias aplicaciones hechas en Hadoop, tales como HBase. Además habla del uso de esta en varias aplicaciones de Bioinformática.

PH2: An Hadoop-based framework formining structural properties[3]. PH2 es una herramienta basada en sql y hadoop, que se usa para extraer la información estructural de proteínas existente en PDB. Este paper dice que la estructura de una proteína tiene relación con sus propiedades. Esta herramienta consta de varias partes y permite el diseño de una base de datos y el análisis de estas estructuras, con una interfaz de usuario transparente.

3. Analizando Proteínas con Spark

El capítulo 3 contiene los detalles del funcionamiento de AFAL2 y su base de datos. Además contiene la descripción de los algoritmos desarrollados en Spark para obtener las frecuencias de AFAL2 de forma distribuida.

3.1. Funcionamiento de AFAL2.

Tal y como se dijo en el capítulo 2, AFAL2 permite analizar proteínas, calculando tres tipos de frecuencias, cada una se obtiene de un método diferente, los cuales se explican a continuación.

3.1.1. Frecuencia específica.

La frecuencia específica es el número de veces que cada aminoácido aparece dentro de la esfera de selección de un ligando, es decir, cuyo radio de cercanía con el ligando, sea igual o menor a la distancia indicada por el usuario.

La frecuencia específica se calcula con la siguiente fórmula:

$$f(a) = \frac{C_e * 100}{R} \tag{3.1}$$

Ce = Número de apariciones de los aminoácidos (conteo total de veces por proteína), dentro de la esfera de selección del ligando.
R = Número de ocurrencias del ligando en el total de proteínas, que cumplen con los filtros de familia de proteínas y organismo fuente.

3.1.2. Frecuencia General.

La frecuencia general indica el número de veces que cada aminoácido aparece dentro de la esfera de selección de un ligando. Se considera solo una aparición de cada aminoácido por proteína.

$$f(a) = \frac{C_g * 100}{R} \tag{3.2}$$

Cg = Número de apariciones de los aminoácidos (conteo una vez por proteína), dentro de la esfera de selección del ligando.
R = Número total de proteínas que contienen el ligando, y que cumplen con los filtros de familia de proteína y organismo fuente.

3.1.3. Ocurrencia.

La ocurrencia indica el número de protenas respecto a la cantidad de apariciones del ligando en ellas. Es decir, en cuántas proteínas el ligando se repite X veces, donde X es el número de apariciones diferentes del ligando buscado en una misma proteína. Por ejemplo, el ligando FE2 aparece una vez en 439 proteínas, dos veces en 52 proteínas y tres veces en 10 proteínas. Además las apariciones contadas están condicionadas por los filtros de búsqueda.

3.2. Base de datos de AFAL2.

AFAL2 funciona con una base de datos relacional, hecha en PostgresSQL, la cuál está formada por las tablas presentes en la figura 3.1.

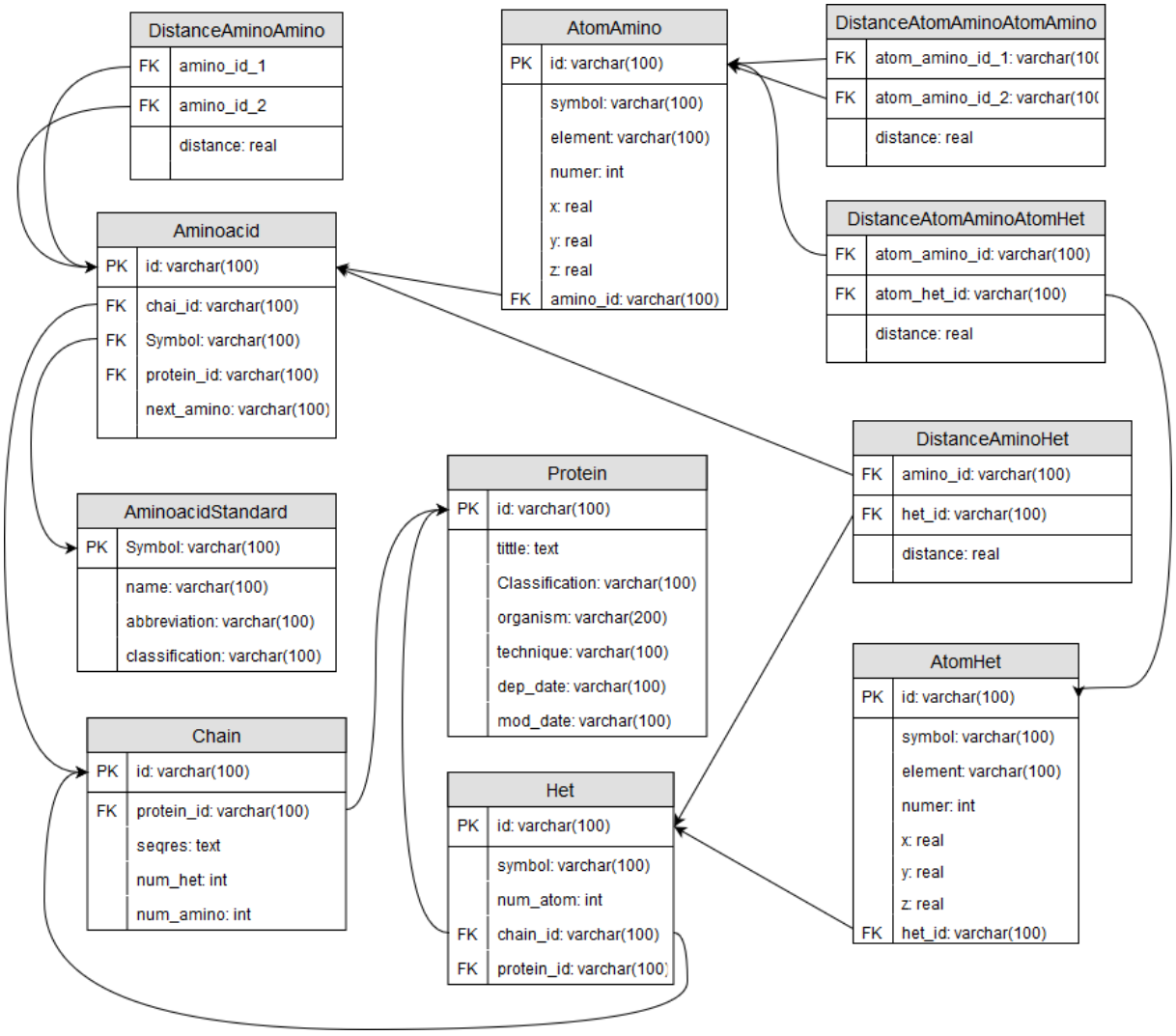

Figura 3.1: Esquema de base de datos de AFAL2

Tablas de la base de datos de AFAL2.

- Protein: Representa la proteína, que es la estructura más general. Tiene como atributos el nombre de la proteína ("tittle"), el id, el cual es único y es obtenido del propio Protein Data Bank, clasificación, organismo fuente, técnica de extracción y las fechas de modificación y publicación del archivo .pdb de la proteína.

- Chain: Es la cadena "A"de cada proteína, y de la cual se puede obtener la secuencia de aminoácidos que componen dicha proteína. Permite además con-

sultar la cantidad de aminoácidos presentes en la cadena.

- Aminoacid: Esta tabla representa los aminoácidos presentes en la cadena de una proteína. Contiene un id compuesto por un número único en la proteína, el id de la cadena y el id de la proteína. Cabe mencionar que en una misma cadena pueden existir muchas moléculas que compartan el mismo símbolo, pero todas se diferencian por el id.
- AminoacidStandard: Es una tabla de búsqueda que almacena las nombres de todos los aminoácidos. Existe con el objetivo de reducir el tamaño de la tabla Aminoacid.
- Het: Esta tabla almacena los ligandos presentes en cada proteína, además de su id único, tiene como atributos el id de la cadena y el de la proteína a la cual pertenece.
- AtomAmino: Esta tabla contiene todos los átomos de los cuales se forma cada aminoácido presente en cada proteína. Contiene un id propio formado por un número único en el aminoácido y el id del mismo.
- AtomHet: Esta tabla contiene todos los átomos que forman cada ligando presente en la proteína. Contiene un id propio formado por un número único en el aminoácido y el id del mismo.
- DistanceAminoAmino: Este tabla corresponde a la distancia entre un aminoácido y los demás aminoácidos en la cadena. Para calcularla, se usan los átomos carbono alpha (CA) de cada aminoácido.
- DistanceAtomAmino: Corresponde a la distancia de cada átomo de cada aminoácido en la cadena con respecto a todos los demás átomos de otros aminoácidos en la cadena. Para calcularla se usan las coordenadas atómicas de cada uno.
- DistanceAtomHet: Corresponde a la distancia de cada átomo de cada ligando en la cadena con respecto a todos los demás átomos de otros ligandos en la cadena. Para calcularla se usan las coordenadas atómicas de cada uno.
- DistanceAminoHet: Es la distancia de cada aminoácido presente en la cadena con respecto a cada ligando presente. Se obtiene calculando la distancia entre

el átomo carbono alpha de cada aminoácido contra el átomo centro de masa de cada ligando.

3.2.1. Consultas SQL de AFAL2

Para cada una de las frecuencias descritas anteriormente, AFAL2 genera una consulta SQL, la cual es ejecutada en PostgreSQL. A continuación se presentan estas consultas.

Consulta SQL para la Frecuencia General.

```
SELECT abbreviation, COUNT(distinct(protein_id)) AS frecuencia
FROM( SELECT abbreviation, sub1.id as protein_id FROM
        (SELECT id FROM "Protein"
        WHERE classification='classification1' and organism='organism1') AS sub1

INNER JOIN "het" ON "het".protein_id = sub1.id
INNER JOIN "DistanceAminoHet" ON "het".id = "DistanceAminoHet".het_id
INNER JOIN "Aminoacid" ON "Aminoacid".id = "DistanceAminoHet".amino_id
INNER JOIN "AminoacidStandard" ON "AminoacidStandard".symbol="Aminoacid".symbol

WHERE distance <=7 AND het.symbol = 'FE2'
) AS sub2

GROUP BY abbreviation
ORDER BY abbreviation
```

Consulta SQL para la Frecuencia Específica.

```
SELECT abbreviation, COUNT(Distinct(amino_id)) AS frecuencia
FROM(SELECT abbreviation, sub1.id, DistanceAminoHet.amino_id
        FROM (SELECT id FROM Protein
        WHERE classification='clasificacion1'
        and organism='organismo1') AS sub1
        INNER JOIN "het"
        ON "het".protein_id = sub1.id
        INNER JOIN "DistanceAminoHet"
        ON "het".id = "DistanceAminoHet".het_id
        INNER JOIN "Aminoacid"
        ON "Aminoacid".id = "DistanceAminoHet".amino_id
        INNER JOIN "AminoacidStandard"
        ON "AminoacidStandard".symbol = "Aminoacid".symbol
        WHERE distance <=7 AND het.symbol = 'FE2') AS sub2
GROUP BY abbreviation
ORDER BY abbreviation
```

Consulta SQL para la Ocurrencia

```
SELECT ocurrencia, COUNT(ocurrencia)
FROM
(SELECT sub1.protein_id, count(distinct(sub1.id))
AS ocurrencia FROM
(SELECT id, protein_id FROM "het"
INNER JOIN "DistanceAminoHet" ON het_id = "het".id
WHERE symbol = 'FE2' and distance <=7) AS sub1
JOIN "Protein" on "Protein".id = sub1.protein_id
GROUP BY sub1.protein_id
) AS sub2
GROUP BY ocurrencia
ORDER BY ocurrencia
```

3.3. Desnormalización de la Base de Datos

Para responder de manera más eficiente se creó una nueva tabla que contiene todas las columnas necesarias para responder las tres consultas, sin la necesidad de consultar otras tablas, lo cual mejora el tiempo de respuesta a cambio de aumentar el costo de almacenamiento (ver figura 3.2).

DistanceAminoHetMod	
FK	amino_id: varchar(100)
	amino_abbreviation: varchar(100)
FK	het_id: varchar(100)
	het_symbol: varchar(100)
	distance: real
	protein_id: varchar(100)
	classification: varchar(100)
	organism: varchar(100)

Figura 3.2: Tabla creada para desnormalizar la base de datos de AFAL2.

Como la base de datos fue desnormalizada, fue necesario modificar las consultas. El resultado de esto es el siguiente:

Frecuencia General desnormalizada.

```
SELECT amino_abbreviation,
COUNT(distinct protein_id)AS Frecuencia
FROM distanceaminohetmod
WHERE het_symbol='FE2' and distance<=7
GROUP BY amino_abbreviation
ORDER BY amino_abbreviation
```

Frecuencia Específica desnormalizada.

```
SELECT amino_abbreviation,
COUNT(distinct amino_id)AS Frecuencia
FROM distanceaminohetmod
WHERE het_symbol='FE2' and distance<=7
GROUP BY amino_abbreviation
ORDER BY amino_abbreviation
```

Ocurrencia desnormalizada.

```
SELECT ocurrencia, COUNT(ocurrencia) AS frecuencia
FROM
(SELECT protein,COUNT(DISTINCT het_id)AS ocurrencia FROM
 (SELECT het_id, protein_id as protein FROM
 distanceaminohetmod WHERE het_symbol='FE2'
  and distance<=7
 ) AS sub1
GROUP BY protein
)AS sub2
GROUP BY ocurrencia
ORDER BY ocurrencia
```

3.4. Pre-procesamiento de archivos.

3.4.1. Conversión de archivos para Spark.

Para llenar la base de datos de AFAL2, se obtienen los datos presentes en el Protein Data Bank, para esto se consultan específicamente los datos de los archivos ".pdb", los cuales se convierten en archivos ".csv"de tal forma que sus columnas sean compatibles con la base de datos relacional. Los archivos ".pdb"si bien no son los más ligeros, pueden pueden ser fácilmente consultados mediante librerias presentes en Java y Python.

Para implementar Spark es necesario convertir los archivos con la extensión ".csv", los cuales contienen información para todas las tablas de la base de datos, pero en el caso de Spark basta con usar un sólo archivo.

Los archivos de entrada tienen la forma representada en en la figura 3.3, y contienen todos los datos necesarios para el cálculo las consultas. De izquierda a derecha las columnas de los archivos se traducen en los siguientes términos:

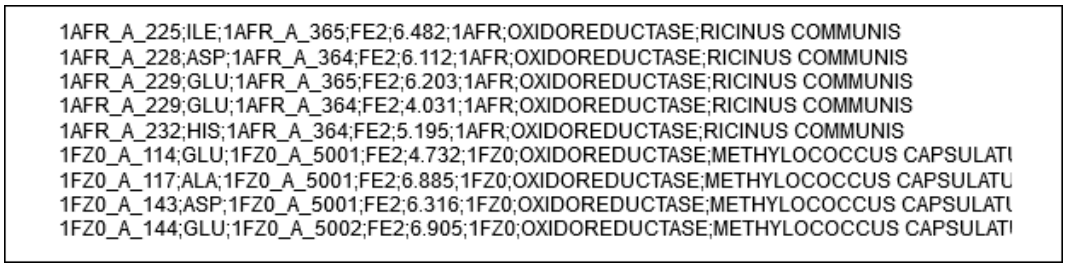

Figura 3.3: Archivos de entrada

- Id del aminoácido: Es el identificador único que distingue a cada aminoácido, está formado por el Id de la proteína, seguido de la cadena a la cual pertenece en esa proteína y finalmente un número entero (que lo hace único). Por ejemplo en la primera línea

- Símbolo del aminoácido: Es el símbolo con el que se representa a un aminoácido de manera universal, es único para cada aminoácido. Por ejemplo en la línea 1 el id es 1AFR_A_255, lo que se traduce en que es el aminoácido 255 de la cadena A de la proteína 1AFR.

- Símbolo del ligando: Es el identificador único que distingue a cada ligando en una proteína, está formado por el Id de la proteína, seguido de la cadena a la cual pertenece en esa proteína y finalmente un número entero (que lo hace único). Por ejemplo en la línea 1 el id del ligando es 1AFR_A_365, por lo que es el ligando número 365 de la cadena A de la proteína 1AFR.

- Proteína: Es el id de la proteína, es único para cada proteína.

- Clasificación: Es la familia de la proteína, como es un parámetro que entrega el usuario, es necesario incluir esta información en los archivos.

- Organismo Fuente: Es el organismo fuente del que se obtiene cada proteína, es también un parámetro entregado por el usuario, por lo que esta información

debe estar en los archivos.

- Distancia: Es la distancia atómica por la que se separan el aminoácido y el ligando de cada fila. También es un parámetro entregado por el usuario por lo que es necesario incluirla.

3.5. Cálculo de Frecuencia General en Spark.

Las consultas SQL que responden a cada frecuencia de AFAL2 fueron replicadas en Spark, y se obtuvieron los siguientes algoritmos:

```
1  JavaRDD<String> lineas = spark.read().textFile(args[0]).javaRDD();
2  JavaRDD<String[]> lineasAux = lineas.map(s->s.split(";"));
3  lineasAux= lineasAux.filter(s-> Float.parseFloat(s[4])<=distancia);
4  lineasAux= lineasAux.filter(s-> s[3].equals(ligandoBuscado));
5  if(args.length > 4)
6  {
7          lineasAux= lineasAux.filter(s-> s[6].equals(args[4]));
8  }
9  if(args.length > 5)
10 {
11         lineasAux=lineasAux.filter(s-> s[7].equals(args[5]));
12 }
13 JavaRDD<String> aminos=lineasAux.map(s->s[1]+";"+s[5]);
14 JavaPairRDD<String, Integer> aminosParciales = aminos.mapToPair(s -> new Tuple2<>(s, 1));
15 JavaPairRDD<String, Integer> cuentaParcial = aminosParciales.reduceByKey((i1, i2) -> i1=1);
16 JavaPairRDD<String[],Integer> cuentaParcialAux=cuentaParcial.mapToPair(s->new Tuple2<>(s._1.split(";"), s._2));
17 JavaPairRDD<String, Integer> cuentaFinal = cuentaParcialAux.mapToPair(s->new Tuple2<>(s._1[0],s._2));
18 JavaPairRDD<String, Integer> frecGeneral = (cuentaFinal.reduceByKey((i1,i2)->i1+i2)).sortByKey();
19 List<Tuple2<String, Integer>> output = frecGeneral.collect();
```

Primero se crea el objeto del tipo **"JavaRDD"** llamado **"lineas"**, con el objetivo de almacenar el archivo de texto línea por línea, siendo cada línea una fila del RDD. Luego se separan las columnas y se almacenan dentro de otro RDD (línea 2). Además para filtrar las filas que cumplan con los parámetros ingresados por el usuario (ligando, clasificación, organismo, distancia), se hace uso de varios métodos **"filter"** (ejecutados sobre la variable **"lineasAux"** a partir de la línea 3). Una vez filtradas las líneas se aislan las columnas que contienen la proteína y el símbolo de tres letras del aminoácido, tal y como se puede observar en la figura 3.4.A. Una vez se tienen aisladas las dos columnas, se crea un **"JavaPairRDD"** que almacena

un diccionario de la forma $< \text{“}proteina; aminoacido\text{”}, 1 >$ donde la proteína y el aminoácido actúan como clave y 1 es el valor (ver figura 3.4.B. Luego se hace uso de un método **“reduceByKey”** que agrupa las claves idénticas y asigna el valor 1 a cada clave diferente (ver figura 3.4.C). Después de la primera reducción se crea un nuevo diccionario de la forma $< \text{“}aminoacido\text{”}, 1 >$ y se vuelve a reducir por clave, esta vez se agrupan las claves idénticas pero los valores se suman, obteniendo como resultado el objeto “frecuenciaGeneral”, tal y como se aprecia en la figura 3.4.E.

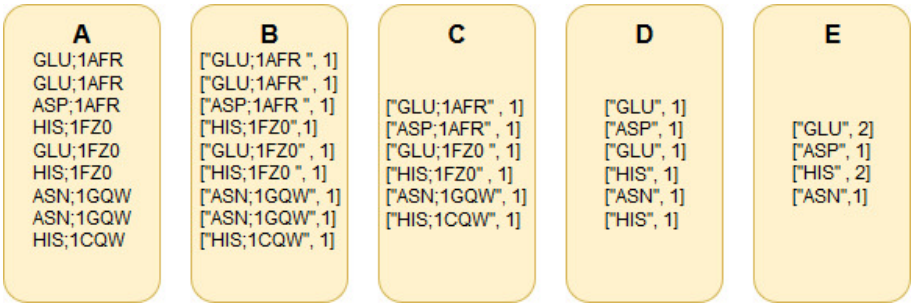

Figura 3.4: Ejemplo del proceso de la Frecuencia General en Spark

3.6. Cálculo de Frecuencia Específica en Spark.

Para implementar la frecuencia específica se utilizó el siguiente código en Spark.

```
1  JavaRDD<String> lineas = spark.read().textFile(args[0]).javaRDD();
2  JavaRDD<String[]> lineasAux = lineas.map(s->s.split(";"));
3  lineasAux= lineasAux.filter(s-> Float.parseFloat(s[4])<=distancia);
4  lineasAux= lineasAux.filter(s-> s[3].equals(ligandoBuscado));
5  if(args.length > 4)
6  {
7          lineasAux= lineasAux.filter(s-> s[6].equals(args[4]));
8  }
9  if(args.length > 5)
10 {
11         lineasAux=lineasAux.filter(s-> s[7].equals(args[5]));
12 }
13 JavaRDD<String> aminos=lineasAux.map(s->s[1]);
14 JavaPairRDD<String, Integer> cuentaParcial = aminos.mapToPair(s->new Tuple2<>(s,1));
15 JavaPairRDD<String, Integer> cuentaTotal = cuentaParcial.reduceByKey((i1, i2) -> i1 + i2);
16 cuentaTotal = cuentaTotal.sortByKey();
17 List<Tuple2<String, Integer>> output = cuentaTotal.collect();
```

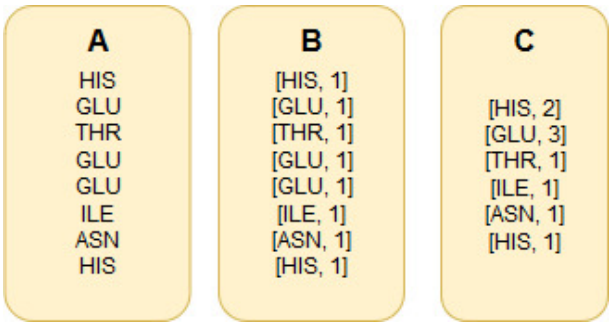

Figura 3.5: Ejemplo del proceso de la Frecuencia Específica en Spark.

Al igual que en la frecuecia general. El primer paso es leer el archivo de texto línea por línea y almacenarlo en un RDD. Luego se ejecutan varias funciones **"filter"** para cumplir con los parámetros entregados por el usuario. Además se eliminan las columnas, excepto el aminoácido, como se ve en la figura 3.5.A. Finalmete se procede a reducir por clave, agrupando las claves idénticas y sumando los valores, usando la función **"reduceByKey"** aplicada al objeto **"cuentaParcial"** (ver figura 3.5.C). Al hacer esto se obtiene un diccionario con todos los aminoácidos agrupados como clave y la cantidad total de apariciones de cada uno como valor.

3.7. Cálculo de Ocurrencia

```
1  linesp= linesp.filter(s-> Float.parseFloat(s[4])<=distancia && s[3].equals(ligando));
2  JavaPairRDD<String, Integer>  ligandosPorProt= ligandosTotales.mapToPair(s -> new Tuple2<>(s, 1));
3
4  ligandosPorProt = ligandosPorProt.reduceByKey((i1, i2) -> i1=1);
5  JavaPairRDD<String[], Integer> apariciones = ligandosPorProt.mapToPair(s->new Tuple2<>(s._1.split("_"),s._2));
6  ligandosPorProt = apariciones.mapToPair(s->new Tuple2<>(s._1[0],s._2));
7
8  ligandosPorProt = ligandosPorProt.reduceByKey((i1, i2) -> i1+i2);
9  JavaPairRDD<Integer,String> cuentaParcial = ligandosPorProt.mapToPair(s->s.swap());
10 JavaPairRDD<Integer,Integer> cuentaFinal=cuentaParcial.mapToPair(s->new Tuple2<>(s._1, 1));
11
12 JavaPairRDD<Integer,Integer> ocurrencia = cuentaFinal.reduceByKey((i1,i2)->i1+i2);
13 List<Tuple2<Integer,Integer>> output=ocurrencia.sortByKey().collect();
```

Para calcular la ocurrencia, después de filtrar las líneas como en las frecuencias anteriores, se crea un diccionario de la forma $< \text{“}ligandoID; proteina\text{”}, 1 >$ (figura 3.6.B), para asegurar contar sólo una vez cada ligando. Luego se reduce por clave para obtener una aparición de cada ligando distinto, como se aprecia en la figura 3.6.C. Después se quita el aminoácido y se deja un diccinoario de la forma $< \text{“}proteina, 1\text{”} >$

Figura 3.6: Ejemplo del proceso de la Ocurrencia en Spark.

(figura 3.6.D). Se reduce por clave nuevamente pero esta vez se suman los valores, obteniendo el total de apariciones del ligando por proteína (figura 3.6.E. Finalmente se crea un nuevo diccionario $<$ "*numerodeaparicionesdelligando*", $1 >$ donde cada clave es el número de apariciones del ligado por cada proteína presente en cada par obtenido en el paso anterior (figura 3.6.F). Finalmente se vuelve a reducir por clave para saber en cuantas proteínas el ligando se repite cada número de veces (ver figura 3.6.G.

4. Experimentos

El objetivo principal del proyecto es probar la eficiencia de la implementación en Spark versus las consultas en PostgreSQL. Se consideraron dos variables, una es el tiempo de ejecución de cada tecnología para saber cuál responde más rápido a las consultas, y la segunda es el aumento de tiempo con respecto a un aumento en la cantidad de datos para saber si en un futuro Spark podría ser más eficiente que PostgreSQL. A continuación se detallan los experimentos realizados para llevar a cabo esta comparación.

4.1. Metodología.

Para conocer si realmente existe una mejora en el tiempo de respuesta de las consultas en Spark con respecto a las consultas SQL se ejecutaron tres consultas de frecuencia en la base de datos PostgreSQL desnormalizada y en Spark. No se incluyó la base de datos normalizada porque se sabe apriori que la base de datos desnormalizada responde más rápidamente.

4.1.1. Escenarios de prueba.

Para las pruebas se consideraron tres conjuntos de datos. El primero es un subconjunto de la base de datos de AFAL2, la cual contiene 42203 proteínas (conjunto 1). El segundo conjunto es la base de datos de AFAL2 y cuenta con 84528 proteínas (conjunto 2). El tercer conjunto es la base de datos de AFAL2 extendida mediante un generador de datos, el cual permitió obtener 141843 proteínas (conjunto 3). Además se eligieron cinco ligandos para las pruebas. Los ligandos fueron recomendados por Mauricio Arenas, profesor co-guía de la presente memoria (SO4, GOL, ZN, MG, CL).

Cada conjunto de datos fue probado en los siguientes ambientes para cada ligando:

- Notebook con PostgreSQL. (Procesador Intel(R) Core(TM) i7-7700HQ, 16 GB de RAM y SSD de 128 GB)
- Notebook con Spark de manera local (Procesador Intel(R) Core(TM) i7-7700HQ, 16 GB de RAM y SSD de 128 GB).
- Cluster de Amazon EMR (3 nodos) con Spark (Procesador 4 vCore, 8 GB de RAM, HDD de 32 GB).

4.2. Resultados.

4.2.1. Comparación de tiempos en la misma base de datos

La prueba preliminar es comparar los tiempos de respuesta entre los ambientes con el mismo conjunto de datos. En la tabla 4.1 De las pruebas de cada consulta se obtuvieron los siguientes tiempos medidos en segundos. Es importante considerar que esta prueba se realizó con el primer conjunto de datos, el cual cuenta con 42203 proteínas. Cabe mencionar que la desventaja de usar Amazon EMR es que los datos son cargados desde el almacenamiento en la cuenta, al igual que el .jar con la implementación de Spark, por lo que el tiempo en cargar los datos es más elevado.

Cuadro 4.1: Tiempos al calcular la Frecuencia General.

Ligandos	PostgreSQL	Notebook	Cluster
SO4	0,301	5,88	14
GOL	0,29	5	14
ZN	0,27	4,72	14
MG	0,225	4,4	13
CL	0,241	4,427	14
Promedio	0,2654	4,8854	13,8

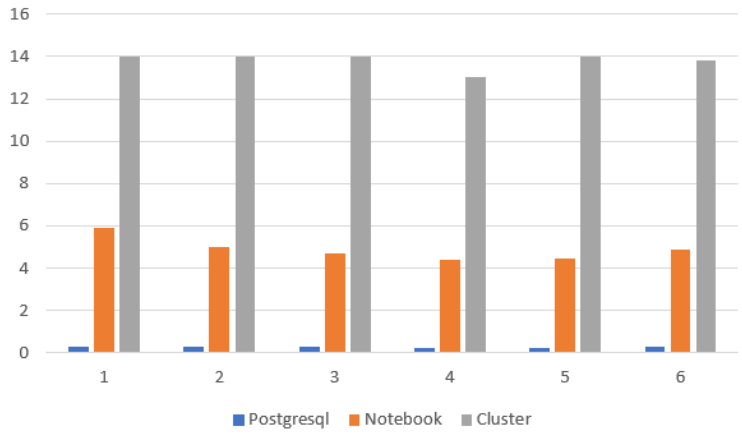

Figura 4.1: Gráfico de los tiempos de respuesta al calcular la Frecuencia General

Cuadro 4.2: Tiempos de respuesta al calcular Frecuencia Específica.

Ligandos	PostgreSQL	Notebook	Cluster
SO4	0,27	4,182	12
GOL	0,326	4,09	13
ZN	0,324	4,125	12
MG	0,23	4,307	12
CL	0,257	4,669	12
Promedio	0,2814	4,2746	12,2

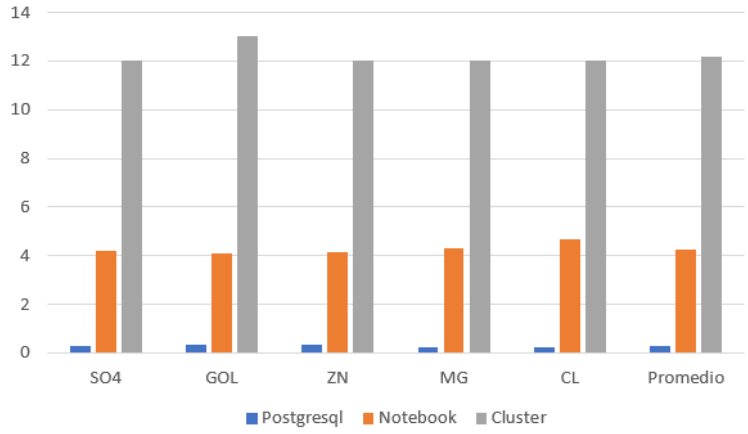

Figura 4.2: Gráfico de los tiempos de respuesta al calcular la Frecuencia Específica.

Cuadro 4.3: Tiempos de respuesta al calcular la Ocurrencia.

Ligandos	PostgreSQL	Notebook	Cluster
SO4	0,289	5,4	14
GOL	0,261	5,82	14
ZN	0,489	5,317	14
MG	0,224	4,627	14
CL	0,25	4,718	12
Promedio	0,3026	5,1764	13,6

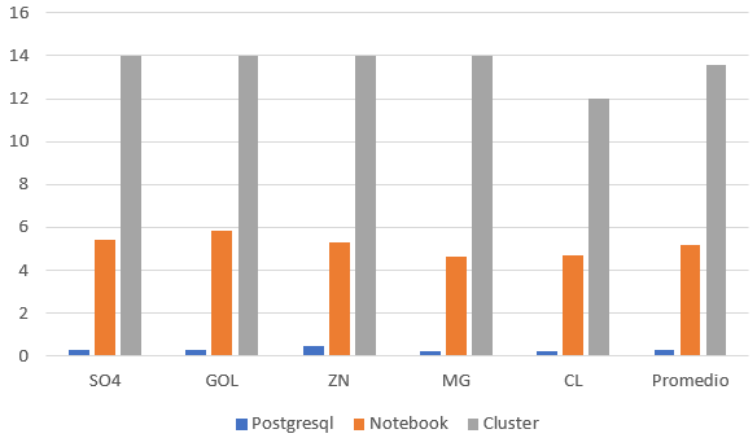

Figura 4.3: Gráfico de los tiempos de respuesta al calcular la Ocurrencia.

Aparentemente los resultados no son los mejores para Spark, pero aún hay factores que podrían influir en su efectividad. Tal es la tasa de aumento de tiempo con respecto al aumento de datos, para esto se realizaron pruebas en las que el promedio de tiempo de cada ambiente, se compara según los conjuntos de datos.

4.2.2. Comparación de tiempos en distintos conjuntos de datos

Estos experimentos se hicieron usando el conjunto de datos 1 (muestra de la base de datos de AFAL2), el conjunto de datos 2 que es la base de datos sobre la que se ejecuta AFAL2 y el conjunto de datos extendido. El objetivo de este experimento es obtener una idea de la tasa de crecimiento del tiempo de ejecución de Spark con respecto a un aumento considerable en los datos.

Cuadro 4.4: Promedio de los tiempos de respuesta para cada conjunto de datos.

Ambiente	Conjunto 1	Conjunto 2	Conjunto 3
PostgreSQL	0,2654	0,772	1,18
Notebook	4,8854	5,58	6,508
Cluster	13,8	13,9	14

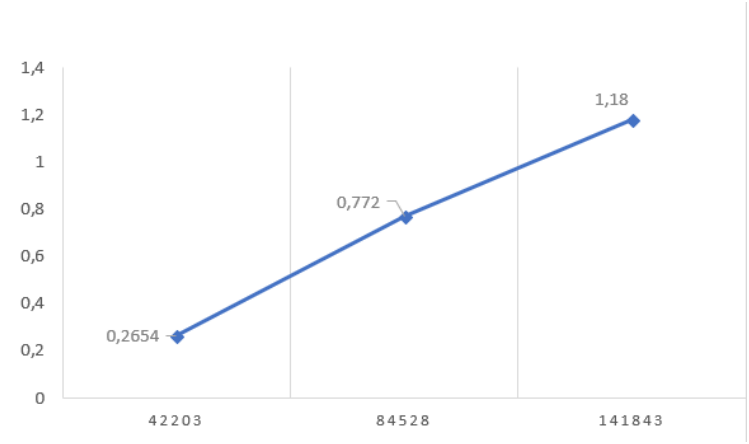

Figura 4.4: Gráfico del aumento de tiempo en PostgreSQL.

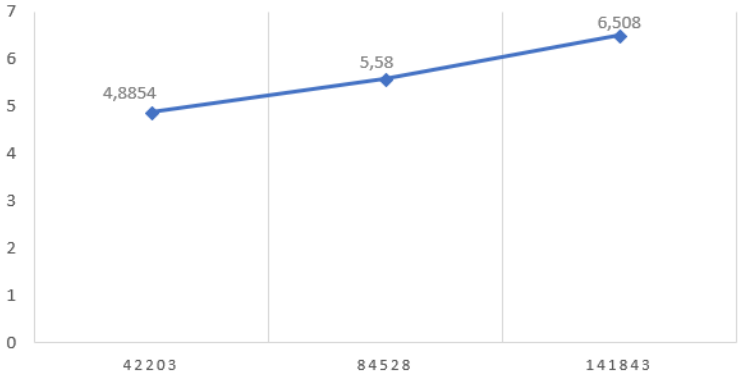

Figura 4.5: Gráfico del aumento de tiempo en Spark sobre un notebook.

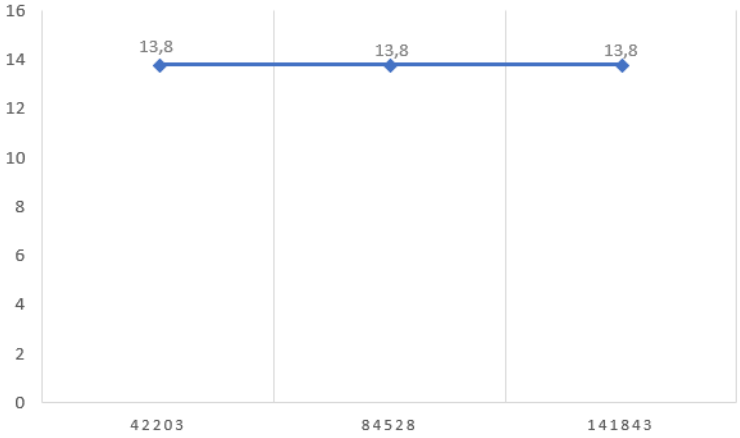

Figura 4.6: Gráfico del aumento de tiempo en Spark sobre un Clúster.

Como se puede observar en la figura 4.4, en una base de datos en PostgreSQL existe un aumento de tiempo casi proporcional al aumento en la cantidad de datos (tasa de aumento de 100 % aproximadamente), por lo tanto podemos suponer que si se aumentara al doble la cantidad de datos, el aumento se vería reflejado en un aumento considerable en el tiempo de consulta. Mientras que en el caso de Spark sobre un notebook (figura 4.5), el aumento es mucho menor, aproximadamente el 20 %. Finalmente el aumento de tiempo en Spark sobre el Cluster de Amazon EMR fue casi despreciable (figura 4.6).

4.3. Conclusiones de los experimentos.

- Para el análisis de Proteínas Spark no supera a PostgreSQL actualmente, teniendo un tiempo de ejecución más alto.
- AFAL2 aún no cuenta con los suficientes datos de proteínas como para probar el poder de un Cluster de Amazon EMR, ya que esta tecnología es sólo accesible de manera remota, por lo que gran parte del tiempo es utilizado en cargar los datos.
- Considerando el aumento exponencial de PDB, se puede pensar que en algún momento existirán las suficientes proteínas como para que Spark sea más rápido que PostgreSQL.

5. Conclusiones

En este capítulo se encuentran las principales conclusiones y aprendizajes obtenidos mediante el desarrollo del proyecto.

5.1. Sobre los objetivos.

- **Describir los datos del Protein Data Bank usados por AFAL2**. Este objetivo se cumplió ya que se realizó un extenso análisis de los datos de las proteínas.
- **Diseñar un modelo para almacenar los datos para que puedan ser leídos por Apache Spark.** Este objetivo se cumplió en su mayoria, ya que a pesar de no contar con un modelo como tal, se desarrolló un archivo .csv específico que puede ser leído por Spark y responder a las necesidades de AFAL2.
- **Identificar las consultas de AFAL2.** Las consultas en PostgreSQL fueron identificadas y aisladas.
- **Desarrollar las consultas de AFAL2 en Spark.** Las consultas fueron desarrolladas en Spark y respondieron con los mismos resultados que las consultas en PostgreSQL para resolver las frecuencias calculadas en AFAL2.
- **Comparar la eficiencia de Map reduce en Apache Spark versus base de datos relacional en PostgreSQL.** Las consultas en Apache Spark fueron comparadas con las consultas en PostgreSQL en varios ambientes y cantidades de datos.

5.2. Comentarios adicionales.

Como parte de este proyecto se puede rescatar que si bien Spark es una herramienta poderosa para el análisis de datos, no es posible una integración completa con aplicaciones web ya existentes, ya que únicamente permite ser ejecutado mediante consola y tiene sus propias herramientas de visualización. Entonces, si AFAL2 quisiera migrar de PostgreSQL, sería necesario una nueva aplicación construida con servicios que permitan sostener Spark. Además es importante mencionar que no es nada fácil desarrollar aplicaciones mediante para Spark, ya que el modelo Map Reduce requiere una programación distinta a la tradicional, además el proceso de ejecución es transparente para el programador, lo cual complica las pruebas y la comparación con otras herramientas.

Para validar completamente la hipótesis de este trabajo, la cual argumentaba que Spark sería más eficiente que PostgreSQL, sería necesario realizar experimentos con mayor cantidad de datos, los cuales no existen aún en el PDB. Además para realizar dichos experimentos sería necesario emplear un Clúster dedicado ya que el servicio de Amazon Web Service de bajo costo no sería suficiente.

Bibliografía

[1] Helen M. Berman, Tammy Battistuz, T. N. Bhat, Wolfgang F. Bluhm, Philip E. Bourne, Kyle Burkhardt, Zukang Feng, Gary L. Gilliland, Lisa Iype, Shri Jain, Phoebe Fagan, Jessica Marvin, David Padilla, Veerasamy Ravichandran, Bohdan Schneider, Narmada Thanki, Helge Weissig, John D. Westbrook, and Christine Zardecki. The Protein Data Bank. *Acta Crystallographica Section D*, 58(6 Part 1):899–907, Jun 2002.

[2] "Frances C. Bernstein, Thomas F. Koetzle, Graheme J.B. Williams, Edgar F. Meyer, Michael D. Brice, John R. Rodgers, Olga Kennard, Takehiko Shimanouchi, and Mitsuo Tasumi". The protein data bank: A computer-based archival file for macromolecular structures. *Journal of Molecular Biology*, 112(3):535 – 542, 1977.

[3] Scott Hazelhurst. Ph2: An hadoop-based framework for mining structural properties from the pdb database. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 104–112, New York, NY, USA, 2010. ACM.

[4] David Baltimore James E. Darnell, Harvey F. Lodish. *Molecular Cell Biology*. US/Data/Medical-Books, 2015.

[5] Andy Konwinski Matei Zaharia, Holden Karau and Patrick Wendell. *Learning Spark*. O'Reilly Media, 2015.

[6] David E. Robillard, Phelelani T. Mpangase, and Scott Hazelhurst. Speedb: Fast structural protein searches, 2015.

[7] Duarte JM Rose PW, Bradley AR. Scalable data analytics of the pdb with the macromolecular transmission format. *ISCB Comm J*, C-28(1321):"poster", June 2017.

[8] Thilina Gunarathne Srinath Perera. *Hadoop MapReduce Cookbook*. PACKT, 2013.

[9] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current, dec 2010.

[10] Akshay Ware, Ganesh Janvale, Faiyaz Shaikh, and Sanjay Harke. Hadoop: Solution for big data challenges in bioinformatics and its prospective in india. 02 2017.

[11] Tom White. *Hadoop: The Definitive Guide*. O'reily, 2015.

[12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

ANEXOS

A. Códigos completos de las consultas en Spark

A.1. Código para la Frecuencia General

```
1  public final class General {
2
3
4            public static void main(String[] args) throws Exception {
5
6              if (args.length < 1) {
7                System.err.println("Uso: General <Archivo>");
8                System.exit(1);
9              }
10             String outputFile=args[1];
11             float distancia =Float.parseFloat(args[3]);
12             String ligandoBuscado = args[2];
13
14             SparkSession spark = SparkSession
15               .builder()
16               .appName("General")
17               .getOrCreate();
18             long lStartTime = System.currentTimeMillis();
19             //Leer archivo y cargarlo como filas en el RDD
20             JavaRDD<String> lineas = spark.read().textFile(args[0]).javaRDD();
21
22             JavaRDD<String[]> lineasAux = lineas.map(s->s.split(";"));
23             /*Con el m\'etodo Filter se recuperan \'unicamente las l\'ineas que cumplen con los par\'ametros.
24              Con el "map" siguiente se procede a aislar unicamente la prote\'ina y el amino\'acido de cada fila.
25              */
26
27
28             lineasAux= lineasAux.filter(s-> Float.parseFloat(s[4])<=distancia);
29             lineasAux= lineasAux.filter(s-> s[3].equals(ligandoBuscado));
30             if(args.length > 4)
```

```
31              {
32                      lineasAux= lineasAux.filter(s-> s[6].equals(args[4]));
33              }
34              if(args.length > 5)
35              {
36                      lineasAux=lineasAux.filter(s-> s[7].equals(args[5]));
37              }
38              JavaRDD<String> aminos=lineasAux.map(s->s[1]+";"+s[5]);
39
40              /*
41               Se crea un diccionario y se le asigna el valor 1 a cada clave compuesta por Prote\'ina + Amino
42               Luego se reduce por la clave y se le asigna un valor 1 a cada Prote\'ina + Amino\'acido diferentes,
43               obteniendo s\'olo 1 aparicion de cada amino\'acido por prote\'ina.
44               */
45              JavaPairRDD<String, Integer> aminosParciales = aminos.mapToPair(s -> new Tuple2<>(s, 1));
46              JavaPairRDD<String, Integer> cuentaParcial = aminosParciales.reduceByKey((i1, i2) -> i1=1);
47
48              JavaPairRDD<String[],Integer> cuentaParcialAux=cuentaParcial.mapToPair(s->new Tuple2<>(s._1.split(";"), s.
49              JavaPairRDD<String, Integer> cuentaFinal = cuentaParcialAux.mapToPair(s->new Tuple2<>(s._1[0],s._2));
50
51              JavaPairRDD<String, Integer> frecGeneral = (cuentaFinal.reduceByKey((i1,i2)->i1+i2)).sortByKey();
52              List<Tuple2<String, Integer>> output = frecGeneral.collect();
53              long lEndTime = System.currentTimeMillis();
54              long executionTime = lEndTime-lStartTime;
55              String time=executionTime+"";
56
57              for (Tuple2<?,?> tuple : output) {
58               System.out.println(tuple._1() + ": " + tuple._2());
59              }
60              frecGeneral.saveAsTextFile(outputFile);
61              spark.stop();
62            }
63  }
64
```

A.2. Código para la Frecuencia Específica

```
1   public final class Frec {
2
3
4     public static void main(String[] args) throws Exception {
5
6
7       float distancia =Float.parseFloat(args[3]);
8       String ligandoBuscado = args[2];
9       String outputFile=args[1];
10
11      SparkSession spark = SparkSession
12        .builder()
13        .appName("Frec")
```

```
14        .getOrCreate();
15
16
17      JavaRDD<String> lineas = spark.read().textFile(args[0]).javaRDD();
18      JavaRDD<String[]> lineasAux = lineas.map(s->s.split(";"));
19      lineasAux= lineasAux.filter(s-> Float.parseFloat(s[4])<=distancia);
20      lineasAux= lineasAux.filter(s-> s[3].equals(ligandoBuscado));
21      if(args.length > 4)
22      {
23              lineasAux= lineasAux.filter(s-> s[6].equals(args[4]));
24      }
25      if(args.length > 5)
26      {
27              lineasAux=lineasAux.filter(s-> s[7].equals(args[5]));
28      }
29      JavaRDD<String> aminos=lineasAux.map(s->s[1]);
30
31      JavaPairRDD<String, Integer> cuentaParcial = aminos.mapToPair(s->new Tuple2<>(s,1));
32      JavaPairRDD<String, Integer> cuentaTotal = cuentaParcial.reduceByKey((i1, i2) -> i1 + i2);
33      cuentaTotal = cuentaTotal.sortByKey();
34      List<Tuple2<String, Integer>> output = cuentaTotal.collect();
35
36      cuentaTotal.saveAsTextFile(outputFile);
37      spark.stop();
38    }
39  }
```

A.3. Código para la Ocurrencia

```
1   public final class Ocur {
2
3
4             public static void main(String[] args) throws Exception {
5
6               if (args.length < 1) {
7                 System.err.println("Usage: Frec <file>");
8                 System.exit(1);
9               }
10
11             /*Se declaran las variables distancia y ligando, que representan el
12              * ligando buscado por el usuario y la esfera de seleccion
13              */
14              float distancia =Float.parseFloat(args[3]);
15              String ligandoBuscado = args[2];
16              String outputFile=args[1];
17              //Se crea la sesi\'on dentro de la cual Spark funcionar\'a.
18              SparkSession spark = SparkSession
19                .builder()
20                .appName("Frec")
21                .getOrCreate();
```

```
22
23              long lStartTime = System.currentTimeMillis();
24              //Este RDD lee el archivo y almacena las lineas de texto como filas.
25              JavaRDD<String> lineas = spark.read().textFile(args[0]).javaRDD();
26
27              //El metodo map separa las columnas del archivo
28              JavaRDD<String[]> lineasAux = lineas.map(s->s.split(";"));
29              //El m\'etodo filter selecciona todas las filas que cumplan con las condiciones.
30              lineasAux= lineasAux.filter(s-> Float.parseFloat(s[4])<=distancia);
31              lineasAux= lineasAux.filter(s-> s[3].equals(ligandoBuscado));
32              if(args.length > 4)
33              {
34                      lineasAux= lineasAux.filter(s-> s[6].equals(args[4]));
35              }
36              if(args.length > 5)
37              {
38                      lineasAux=lineasAux.filter(s-> s[7].equals(args[5]));
39              }
40
41              //Se corta cada fila, dejando solamente la secci\'on que corresponde al id del ligando
42              JavaRDD<String> ligandosTotales=lineasAux.map(s->s[2]);
43              JavaPairRDD<String, Integer>  ligandosPorProt= ligandosTotales.mapToPair(s -> new Tuple2<>(s, 1));
44              //Se crea un diccionario y se le asigna el valor 1 a cada grupo de ligandos
45              ligandosPorProt = ligandosPorProt.reduceByKey((i1, i2) -> i1=1);
46
47              //Los Map siguientes son para aislar la parte del id que pertenece a la prote\'ina.
48              JavaPairRDD<String[], Integer> apariciones = ligandosPorProt.mapToPair(s->new Tuple2<>(s._1.split("_"),s._
49              ligandosPorProt = apariciones.mapToPair(s->new Tuple2<>(s._1[0],s._2));
50
51              //Ahora se procede a reducir para tener las apariciones exactas por prote\'ina de cada ligando
52              ligandosPorProt = ligandosPorProt.reduceByKey((i1, i2) -> i1+i2);
53
54              /*Se intercambian las llaves con los valores, dejando las apariciones como clave
55               y se agrega un 1 como el nuevo valor. Finalmente se vuelve a reducir para obtener la ocurrencia
56              */
57              JavaPairRDD<Integer,String> cuentaParcial = ligandosPorProt.mapToPair(s->s.swap());
58              JavaPairRDD<Integer,Integer> cuentaFinal=cuentaParcial.mapToPair(s->new Tuple2<>(s._1, 1));
59              JavaPairRDD<Integer,Integer> ocurrencia = cuentaFinal.reduceByKey((i1,i2)->i1+i2);
60              List<Tuple2<Integer,Integer>> output=ocurrencia.sortByKey().collect();
61              long lEndTime = System.currentTimeMillis();
62              long executionTime = lEndTime-lStartTime;
63              String time=executionTime+" milisegundos";
64
65              ocurrencia.saveAsTextFile(outputFile);
66
67              spark.stop();
68            }
69
70  }
71
```

A.4. Ejecución de Spark por líneas de comandos.

Para el uso de Spark es necesario escribir las siguientes instrucciones.:

```
spark-submit –class Package.File.Class –master local[4] archivo.jar parámetros....
```

A.5. Ejecución de Spark en un Cluster de Amazon EMR.

Lo primero que hay que hacer es crear un Cluster con Spark, tal y como se observa en la figura A.1.

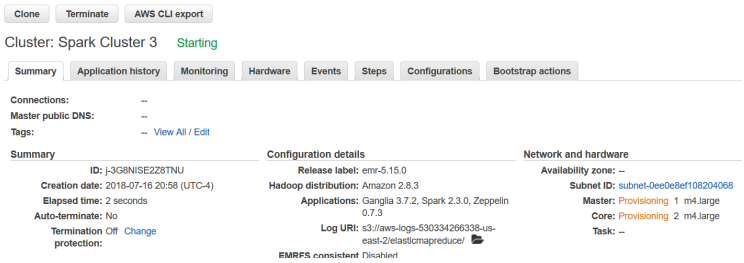

Figura A.1: Vista de un Cluster creado en Amazon EMR.

Para poder ejecutar un .jar propio dentro de un Cluster de Amazon, es necesario agregar un "step.º una etapa de ejecución en el Cluster, para esto se deben llenar los siguientes parámetros:

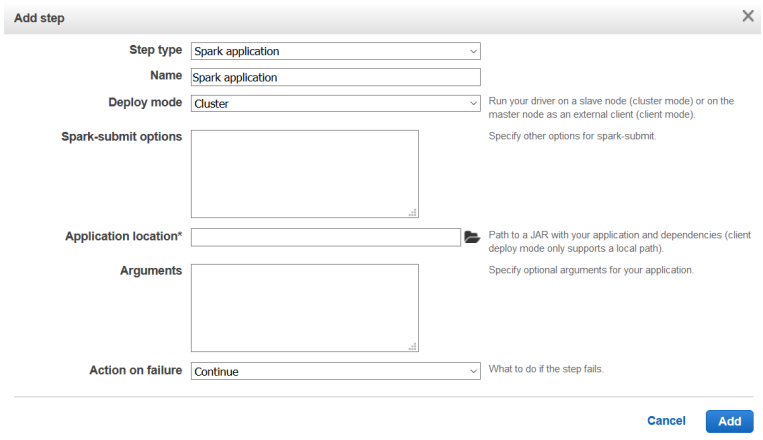

Figura A.2: Parámetros para agregar un step en Amazon EMR.

Para completar los parámetros es necesario que el .jar sea agregado al almacenamiento de la cuenta de Amazon, asi como todos los parámetros. Y de este modo indicar las clases, la carpeta donde se encuentra el .jar y los argumentos para su ejecución.