Una semántica formal para Apache Spark en lógica de reescritura

Mateo Sanabria Ardila

Director
Dr. Camilo Rocha

Programa de Matemáticas
Escuela Colombiana de Ingeniería *Julio Garavito*
Bogotá, Colombia
Julio, 2017

Resumen

Este documento presenta una semántica ejecutable para Apache Spark en lógica de reescritura. Apache Spark es un entorno de trabajo de código abierto diseñado para el procesamiento de datos que provee una interfaz de programación de aplicaciones para la manipulación de grandes volumenes de datos. La semántica ejecutable de Apache Spark se hace disponible a través de una especificación formal en Maude siendo esta especificación ejecutable y con la cual se puede estudiar el comportamiento dinámico del agendador de trabajos de Apache Spark.

Agradecimientos

Quiero agradecer al profesor Camilo Rocha por guiarme en el desarrollo de este proyecto, por el tiempo y la dedicación que tuvo para mi. Le agradezco por todo lo que aprendí de él y por su excelente disposición a la hora de enseñar. Así mismo, agradezco a Miguel Romero y Sergio Ramírez por su tiempo y colaboración.

A mis padres Marina y Edilberto, a lo cuales les debo la vida, les agradezco por su inconmensurable esfuerzo para formar la persona que soy hoy, de la cual me siento muy orgulloso.

A la vida, a todo lo que he aprendido y a todos lo que me han acompañado en ella; en particular a Tatiana quien es una gran amiga y que siempre a confiado en mi.

Índice general

Resumen ii

Agradecimientos iii

Capítulo 1. Introducción 1

Capítulo 2. Trabajo relacionado 3

Capítulo 3. Preliminares 5
- 3.1. Lógica de reescritura 5
- 3.2. Maude 6

Capítulo 4. Spark 8
- 4.1. Configuración inicial de Spark 9
- 4.2. RDD: Resilient Distributed Datasets 10
- 4.3. DAG: Directed Acyclic Graph 13
- 4.4. Piscinas 14
- 4.5. Asignación dinámica de recursos 15

Capítulo 5. Especificación formal 17
- 5.1. Configuracion basada en objetos 17
- 5.2. Modelamiento formal de etapas 18
- 5.3. Moldeamiento formal de DAG 19
- 5.4. Moldeamiento formal de piscinas 20
- 5.5. Moldeamiento formal de ejecutores 21
- 5.6. Moldeamiento formal del agendador 21
- 5.7. Modelamiento formal de las transiciones 22

Capítulo 6. Casos de estudio 26
- 6.1. Simulación 26
- 6.2. Exploración de estados 27

Capítulo 7. Conclusión y trabajo futuro 32

Apéndice A. Especificación formal en Maude 33
- A.1. 2-TUPLE 33
- A.2. 6-TUPLE 33

A.3. DAG 35
A.4. EXECUTER 36
A.5. LESFTIST-HEAP 36
A.6. POOL 38
A.7. QUEUE 39
A.8. SCHEDULER 39
A.9. STAGE 40
A.10. SPARK 41

Apéndice B. Ejemplos casos de estudio 51
B.1. EJEM1 51
B.2. EJEM2 53

Bibliografía 55

Capítulo 1

Introducción

El procesamiento de grandes volumen de datos, muchas veces en tiempo real, se ha convertido en uno de los pilares del desarrollo de la ciencia. Países y empresas utilizan los datos para, por ejemplo, dar respuesta a necesidades de la comunidad, tomar decisiones de manera óptima o aprovechar una oportunidad en el mercado. Hoy en día grandes compañías dependen en gran medida de su capacidad para analizar e inferir eficazmente información a partir de grandes volúmenes de datos. Para ello, se suelen implementar redes de computadores (clusters) con el fin de tener mayor capacidad de procesamiento, distribuyendo la carga a través de toda la red. Apache Spark [**1**] surge como una plataforma para agendar y ejecutar procesos sobre grandes redes de computadores y manipular grandes cantidades de datos.

Spark presenta ciertas facilidades para la asignación de recursos a través del cluster donde se encuentra configurado. Por defecto, Spark realiza agendamiento de trabajos con el metodo FIFO; esto es, el primer trabajo adquiere prioridad sobre todos los recursos disponibles siempre que existan tareas asociadas al trabajo que se está ejecutando. Cuando este termina, el segundo trabajo adquiere prioridad, y el proceso contiúa así. Sin embargo, con el agendamineto mediante el algoritmo FIFO, los trabajos con un tiempo de ejecución largo retrasa los trabajos con corto tiempo de ejecución en la cola de agendamiento, Por esta razón, desde Spark 0.8, es posible configurar un tipo de agendamiento en el cual los recursos del cluster pueden ser compartidos entre los trabajos dependiendose la prioridad, este es el algoritmo FAIR. Así pues, Spark soporta agrupamientos de trabajos en objetos llamados piscinas que pueden ser configurados de forma tal que los trabajos asociados a ellos tengan mayor o menor prioridad sobre otros trabajos. El algoritmo FAIR permite que ciertos usuarios o trabajos importantes tengan los recursos suficientes, idealmente.

Este documento presenta una semántica en lógica de reescritura que formaliza el agendamiento de trabajos en Spark. La semántica en lógica

de reescritura de Spark es una teoría ejecutable [**11**] cuya especificación puede ser simulada y analizada algoritmicamente en el sistema Maude [**4**]. Este es un sistema y lenguaje de programación declarativo, de especificación formal y verificación formal. La semántica en lógica de reescritura de Spark hace parte de un esfuerzo para brindar técnicas algorítmicas para razonar sobre de las propiedades especiales de los sistemas concurrentes. La semántica ejecutable presentada en este documento puede ser utilizada para comprobar propiedades del sistema de agendamiento de trabajos de Spark, comprobar los diferentes estados alcanzables desde una configuración en particular y comparar ejecuciones entre los diferentes algoritmos de agendamiento de Spark.

El documento esta organizado de la siguiente manera :

- El Capítulo 2 presenta un breve resumen de algunos trabajos similares a este.
- El Capítulo 3 incluye algunos conceptos requeridos para la lectura del documento.
- El Capítulo 4 incluye un resumen del funcionamiento de Apache Spark.
- El Capítulo 5 presenta el modelamiento de estados y transiciones para Apache Spark.
- El Capítulo 6 presenta algunos casos de estudio y sus respectivos análisis algorítmicos.
- El Anexo A incluye los módulos de sistema y funcionales escritos en sintaxis de Maude para modelar el agendamiento de trabajos de Apache Spark.
- El Anexo B incluye los ejemplos implementados en los casos de estudio.

Capítulo 2

Trabajo relacionado

Se han encontrado muy pocos trabajos que realicen modelaciones formales de Apache Spark. En [**9**] se desarrolló un modelo ejecutable y configurable llamado *stans for Spark Streamin Processing* (SSP) para modelar y evaluar cargas de trabajo ejecutadas en Spark Streaming, una herramienta de Apache Spark que soporta procesamiento en tiempo real. SSP es modelado en una plataforma formal, diseñada para desarrollar y analizar sistemas en software concurrentes distribuidos llamada ABS [**16**] es ejecutable y además es orientado a objetos. El modelamineto de SSP está relacionado con la adecuada implementación de la configuración de Spark para un efectivo procesamiento de datos debido a que una inadecuada configuración conlleva a sobre-aprovisionamiento de recursos o puede causar bajo rendimiento. SSP abstrae el funcionamiento clave de Spark Streamning, lo cual permite hacer comparaciones de rendimiento con Spark Streaming en varios escenarios. Según los autores SSP es el primer modelo ejecutable y configurable para Spark Streaming.

Por otra parte el enfoque que se presenta en [**3**] es algo diferente pues se preocuparon por el determinismo en las ejecuciones de las programas en Spark. Un programa en Spark puede producir diferentes resultados para el mismo conjunto de datos en diferentes ejecuciones, debido a un tipo de operaciones sobre las RDDs llamadas *combinators*. Han desarrollado un librería Haskell llamada `purespark`. Lo cual permite expresar en un lenguaje claro programas de Spark, dejando a un lado los lenguajes tradicionales de programación de esta plataforma. Con base en la especificación los autores de [**3**] han podido identificar condiciones necesarias y suficientes para que los *combinators* puedan generar comportamientos deterministas. Los autores también mencionan que su trabajo es el primero en proveer una especificación formal para *combinators* de Spark. En este trabajo, el enfoque es sobre la semántica de los algoritmos de agendamiento y no se tienen en cuenta detalles como los *combinators*.

Desde otro punto de vista en [7] los autores estuvieron interesados en desarrollar herramientas para poder verificar cuando dos programas de Spark son equivalentes y también pueden producir contra ejemplos en caso contrario. En ese trabajo se desarrollo un modelo simple de Spark, SparkLite, un lenguaje de programación en el cual las funciones definidas por el usuario están expresadas sobre una teoría decidible.

Los trabajos anteriormente presentados son muy recientes y dos de ellos mencionan que son el primer trabajo en hacer un modelamiento formal para partes de Spark. Esto, combinado con la importancia y los requerimientos del mercado para el análisis de grandes cantidades de datos motiva el desarrollo de una semántica formal para Apache Spark en lógica de reescritura.

Capítulo 3

Preliminares

3.1. Lógica de reescritura

Lógica de reescritura [**12**] es una lógica de cambios concurrentes. Las reglas de la lógica de reescritura son patrones generales para acciones básicas que pueden ocurrir concurrentemente con otras acciones en un sistema concurrente. Por lo tanto, la lógica de reescritura permite razonar sobre cambios complejos en un sistema, teniendo en cuenta que los cambios corresponden a las acciones básicas axiomatizadas por las reglas de reescritura.

3.1.1. Teoría ecuacional. Una *signatura* ordenada por tipos Σ es una tupla $\Sigma = (S, \leq, F)$ con un conjunto parcialmente ordenado de tipos $(S, \leq)$ y un conjunto de símbolos de función F. La relación binaria $\equiv_{\leq}$ denota la relación de equivalencia generada por $\leq$ sobre S y su extensión a cadenas en S^*.

La colección de variables X es una familia S-indexada $X = \{X_s\}_{s \in S}$ de conjuntos de variables disyuntos con cada X_s infinito contable. $T_\Sigma(X)_s$ es el *conjunto de términos de tipo* s y el *conjunto de términos simples de tipo* s se denota por $T_{\Sigma,s}$. Las expresiones $\mathcal{T}_\Sigma(X)$ y $\mathcal{T}_\Sigma$ denotan las correspondientes álgebras de Σ-términos ordenadas por tipos.

Una Σ-*ecuación* es una pareja $t = u$ con $t \in T_\Sigma(X)_{s_t}$, $u \in T_\Sigma(X)_{s_u}$ y $s_t \equiv_{\leq} s_u$. Una Σ-*ecuación condicional* es una ecuación condicional $t = u$ **if** γ con $t = u$ una Σ-ecuación y γ una conjunción finita de Σ-ecuaciones. Un *tipo al tope* en Σ es un tipo $s \in S$ tal que si $s' \in S$ y $s \equiv_{\leq} s'$, entonces $s' \leq s$.

Una *teoría ecuacional* es un par (Σ, E), donde Σ es una signatura y E es una colección finita de ecuaciones, posiblemente condicionales. Una teoría ecuacional $\mathcal{E} = (\Sigma, E)$ induce la relación de congruencia $=_{\mathcal{E}}$ sobre $T_\Sigma(X)$ definida por $t =_{\mathcal{E}} u$, con $t, u \in T_\Sigma(X)$, sí y sólo sí $\mathcal{E} \vdash t = u$ por las reglas de deducción para lógica ecuacional ordenada por tipos

en [**12**], sí y sólo sí, en [**12**] $t = u$ es válido en todos los modelos de $\mathcal{E}$.

Las expresiones $\mathcal{T}_{\Sigma/E}(X)$ y $\mathcal{T}_{\Sigma/E}$ corresponden a las algebras de cocientes inducidas por $=_{\mathcal{E}}$ sobre las algebras de términos $\mathcal{T}_{\Sigma}(X)$ y $\mathcal{T}_{\Sigma}$. El algebra $\mathcal{T}_{\Sigma/E}$ es llamado el *algebra inicial* de (Σ, E).

Las Σ-ecuaciones están divididas en un conjunto A de axiomas estructurales (tales como asociatividad, conmutatividad y/o identidad) y el conjunto E de ecuaciones.

3.1.2. Teoría de reescritura. Una *teoría de reescritura* es una tupla $\mathcal{R} = (\Sigma, E, R)$ con una teoría ecuacional $\mathcal{E}_{\mathcal{R}} = (\Sigma, E)$ y un conjunto finito de Σ-reglas R. Una *teoría de reescritura al tope* es una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, tal que cada regla $t \to u$ **if** $\gamma \in R$ es tal que $l, r \in T_{\Sigma}(X)_s$ para algún $s = [s]$ en Σ, $l \notin X$, y no hay operadores en Σ que tengan al tipo s como argumento.

Una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ induce la relación de reescritura $\to_{\mathcal{R}}$ sobre $T_{\Sigma}(X)$ definida por $t \to_{\mathcal{R}} u$, con $t, u \in T_{\Sigma}(X)$, sí y sólo sí una demostración de reescritura de un paso de $\mathcal{R} \vdash t \to u$ puede ser obtenida por las reglas de deducción para teorías de reescritura ordenadas por tipos en [**2**], sí sólo sí, en [**2**] $t \to u$ es válido en todos los modelos de $\mathcal{R}$.
La expresión $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \to^*_{\mathcal{R}})$ denota el modelo de alcanzabilidad inicial de $\mathcal{R} = (\Sigma, E, R)$ [**2**], donde $\to^*_{\mathcal{R}}$ representa la clausura transitiva-reflexiva de $\to_{\mathcal{R}}$.

3.2. Maude

Maude [**4**] es un lenguaje declarativo. Un programa en Maude es una teoría lógica y un cálculo en Maude es una deducción lógica que utiliza los axiomas especificados en la teoría o el programa, según corresponde, bajo el sistema deductivo de la lógica de reescritura [**11**].
Maude cuenta con dos tipos de módulos: funcional y de sistema.

3.2.1. Módulos funcionales. Los módulos funcionales corresponden a teorías ecuacionales y definen tipos de datos y operaciones sobre estos tipos de datos.

Los módulos funcionales de Maude suponen la propiedad de que la ecuaciones son consideradas reglas de simplificación que se usan únicamanete de izquierda a derecha y su repetida aplicación reduce un

término a su forma canónica, la cual no depende del orden en el que se usen las ecuaciones. Este tipo de simplificación canónica es posible si la teoría ecuacional asociada a un módulo funcional es Church-Rosser [**6**] y terminante [**10**].

Los módulos funcionales pueden contener: ecuaciones con o sin atributos, pertenencias y operadores con sus respectivos atributos. Las ecuaciones y pertenencias pueden ser condicionales o incondicionales. Un módulo funcional se define con la palabra clave `fmod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos funcionales se refiere al lector a [**4**].

3.2.2. Módulos de sistema. Un módulo de sistema especifica una teoría de reescritura. Una teoría de reescritura contiene tipos de datos, clases de equivalencia de tipos de datos, operadores, y ecuaciones, pertenencias y reglas de reescritura, las cuales pueden ser condicionales. De lo anterior se puede afirmar que toda teoría de reescritura tiene una teoría ecuacional subyacente. Un módulo de sistema se declara con la palabra clave `mod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos de sistema se refiere al lector a [**4**].

El conjunto de Σ-reglas R es coherente con respecto a las ecuaciones E módulo A, si Maude puede ejecutar un módulo de sistema admisible [**4**] usando la estrategia de primero simplificar un término t a su forma E/A-canónica y luego usar una regla con R módulo A para lograr el efecto de reescribir con R módulo $E \cup A$.

Capítulo 4

Spark

Apache Spark es un entorno de trabajo de código abierto diseñado para el procesamiento de grandes volúmenes datos. Provee una interfaz de programación de aplicaciones para la manipulación de datos a gran escala, distribuyendo la carga de trabajo sobre varios nodos. Spark fue creado en la Universidad de Berkeley en California. Mantiene las características de tolerancia a fallos y escalabilidad lineal (si la cantidad de datos aumenta, es posible añadir más capacidad de procesamiento para reducir el tiempo de ejecución) de sus predecesores, como Apache Hadoop, pero hace un cambio de paradigma añadiendo en su ejecución DAGs (del inglés, Directed Acyclic Graph) que representan el plan lógico de ejecución y RDDs [**17**] encargados del almacenamiento de información.

Apache Spark cuenta con una una amplia librería que lo hace más complejo que sus predecesores. En Spark es posible realizar trabajos en cálculos de grafos (GraphX) y procesamiento de datos en en tiempo real, entre otros. También es posible escribir aplicaciones en lenguajes de programación como Java, Scala, Python, R y SQL.

Una aplicación enviada a Spark es transformada en un DAG que permite dividir la ejecución en una serie de etapas, este DAG crea dependencias entre etapas generando un orden lineal en el cual las etapas deben ser ejecutadas. Al interior de cada etapa se producen continuas transformaciones de conjuntos de datos (RDD) distribuidos a través del clúster. Cada trasformación genera un nuevo conjunto de datos y queda ligado a los conjuntos de datos que lo generaron. Cuando se produce una falla o una perdida de información en un conjunto de datos es posible solucionar dicho problema repitiendo la ejecución de la ultima instrucción que falló.

Una aplicación en Apache Spark consta de un `SparkContex` y un código con el que el usuario especifica transformaciones de RDDs para alcanzar un resultado esperado. En este documento un trabajo será

una aplicación enviada al clúster manejado por Spark. Los trabajos podrán estar en espera para ser ejecutados, en ejecución o ejecutados correctamente. También existe la posibilidad de que un trabajo falle y no se haya podido terminar.

4.1. Configuración inicial de Spark

Cada clúster que es manejado por Spark debe ser configurado antes de empezar a funcionar, existen muchos atributos que pueden ser modificados, Spark provee de varias formas para configurar el sistema:

- Propiedades Spark: controla la mayoría de los parámetros de la aplicación y puede establecerse utilizando SparkConf o mediante propiedades del sistema Java; este tipo de configuraciones se hacen para cada trabajo en especifico.
- Variables de entorno: se pueden usar para establecer configuraciones por máquina, como la dirección IP, a través del script `conf/spark-env.sh` en cada nodo, este tipo de configuración es global y influyen sobre el comportamiento de todo el sistema.

Algunas de las posibles propiedades que pueden ser cambiadas dentro de la configuración. se presentan a continuación algunos ejemplos aunque la mayoría cuentan por defecto con un valor predeterminado:

- `spark.executor.memory` Especifica la cantidad de memoria usada por ejecutor.
- `spark.shuffle.service.enabled` Habilita el servicio de mezcla externo. Este servicio conserva los archivos aleatorios escritos por los ejecutores para que los ejecutores puedan eliminarse de forma segura. Esto debe habilitarse si `spark.dynamicAllocation.enabled` es true.
- `spark.executor.cores` Controla el numero de núcleos por ejecutor.
- `spark.default.parallelism` Informa sobre el número predeterminado de particiones en RDD devuelto por transformaciones como `join`, `reduceByKey` y `parallelize` cuando el usuario no lo establece.
- `spark.scheduler.maxRegisteredResourcesWaitingTime` Especifica la cantidad máxima de tiempo para esperar a que se registren los recursos antes de que comience la programación.
- `spark.scheduler.mode` El modo de programación entre trabajos enviados al mismo SparkContext (FIFO/FAIR).
- `spark.blacklist.enabled` Si se establece en true se evita que Spark programe tareas en ejecutores que han sido puestos en la lista negra debido a demasiadas fallas en la tarea.

- `spark.task.cpus` Parámetro que informa sobre el número de núcleos para asignar para cada tarea.
- `spark.task.maxFailures` Número de fallas de una tarea en particular antes de renunciar al trabajo.

Un *nodo* o *ejecutor* es la entidad encargada de ejecutar una sección de un trabajo del procesamiento de datos. Un nodo consta de una cantidad de núcleos de CPU y una cantidad específica de memoria; estas características son definidas en `SparkContex` y pueden cambiar significativamente el tiempo de ejecución de los trabajos en el clúster al cambiar sus valores entre configuraciones.

4.2. RDD: Resilient Distributed Datasets

Las RDDs son estructuras de datos distribuidas y tolerantes a fallas, que permiten a los usuarios control en sus particiones de datos para optimizar la asignación de datos y la manipulación usando un amplio conjunto de operadores. Las RDDs solo pueden ser creadas mediante operaciones deterministas en datos de almacenamiento estable o en otros RDDs; las RDDs suelen estar almacenadas en memoria RAM, lo que les permite una velocidad de operación alta. Dichas operaciones son llamadas *transformaciones*, llamaremos *RDD padre* a la RDD a la cual se le aplica la transformación y *RDD hija* a las RDD resultante de dicha transformación. Las RDDs son clasificadas por las dependencias entre sus RDDs predecesoras. De esta forma las transformaciones sobre RDDs se pueden clasificar como:

- Transformaciones de tipo Narrow: Cada partición de la RDD padre es usada una única vez para generar una partición dentro del la nueva RDD hija asociada a la transformación. Estas transformaciones son más eficientes que su contra parte puesto que en este tipo de transformación no es necesaria la comunicación con los demás nodos de trabajo (que contiene las particiones restantes de la RDD) pues toda la información necesaria esta contenida en cada partición, esto mismo permite que en caso de falla las RDD generadas por una operación de este tipo se recupere más rápido.

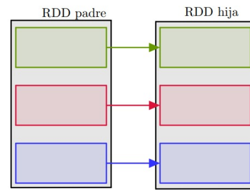

FIGURA 4.1. Transformacion Narrow

- Transformaciones de tipo Wide: Cada partición de la RDD padre puede ser usada varias veces para la creación de las nuevas particiones en la RDD hija asociada a la transformación. Este tipo de transformaciones son menos eficientes que su contra parte a causa de que es necesario la comunicación entre los nodos de trabajo que contienen las particiones de la RDD.

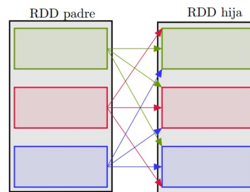

FIGURA 4.2. Transformacion Wide

Algunos ejemplos de transformaciones definidas en Spark:

- Wide
  - `distinct([numTasks]))`: Devuelve un nuevo conjunto de datos que contiene los distintos elementos del conjunto de datos de origen.
  - `groupByKey([numTasks])`: Cuando se llama a un conjunto de datos de pares (K, V), devuelve un conjunto de datos de pares (K, repetciones <V>).
  - `cogroup(otherDataset, [numTasks])`: Cuando se invoca a datasets de tipo (K, V) y (K, W), devuelve un conjunto de datos de (K, (Iterable <V>, Iterable <W>)) tuplas. Esta operación también se llama groupWith.
- Narrow
  - `map(func)`: Devuelve un nuevo conjunto de datos distribuidos formado al pasar cada elemento de la fuente a través de una función func.
  - `filter(func)`: Devuelve un nuevo conjunto de datos formado seleccionando los elementos de la fuente en la que el func devuelve true.

- `flatMap(func)` : Similar al mapa, pero cada elemento de entrada se puede asignar a 0 o más elementos de salida (por lo que func debería devolver un secuencia en lugar de un solo elemento).

Es común que al finalizar una transformación la cantidad de particiones de la RDD hija cambien con respecto a su RDD padre. Esto esmas común en transformaciones de tipo Wide. En las transformaciones de tipo Narrow esto suele suceder cuando alguna partición se ha quedado sin información relevante para el trabajo. Existen transformaciones, por ejemplo join que pueden ser de tipo Narrow o de tipo Wide. En la figura 4.3 se muestra un ejemplo en el que sucede esto porque goupBY organiza las particiones por letras en la RDD C, mientras que en la RDD E no están organizadas por letras. Usualmente las transformaciones solo pertenecen a una única clase.

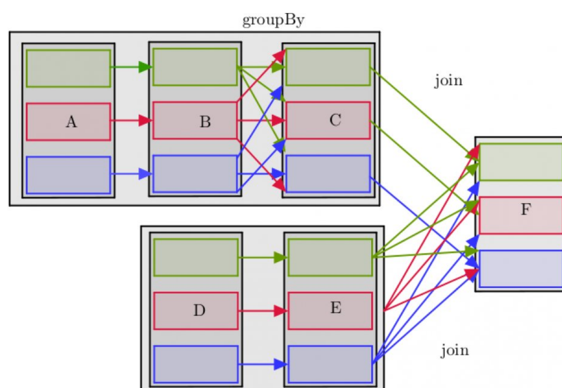

FIGURA 4.3. Ejemplo transformación tipo Wide y Narrow

El usuario puede controlar dos aspectos sobre las RDDs:

- Persintence: El usuario puede indicar que RDDs serán reutilizadas y podrá almacenarlas, esto con el objetivo de hacer la computación más efectiva.
- Partitioning: El usuario también puede definir en cuantos y en cuales nodos quiere que se particione una RDD.

Como se mencionó anteriormente, las transformaciones son operaciones que definen una nueva RDD. Se llamara *acción* a una operación que inicia un cálculo sobre una RDD para calcular valores distintos a una RDD o escribe datos en un almacenamiento. Algunos posibles ejemplos de acciones sobre RDDs son:

- Acciones

- `count()`: Devuelve la cantidad de elementos en el conjunto de datos.
- `first()`: Devuelve el primer elemento del conjunto de datos.
- `takeSample(withReplacement, num, [semilla])`: Devuelve una matriz con una muestra aleatoria de elementos num del conjunto de datos, con o sin reemplazo, opcionalmente especificando previamente una semilla de generador de números aleatorios.

4.3. DAG: Directed Acyclic Graph

Un DAG es un grafo finito dirigido sin ciclos en donde cada nodo representa una RDD, cada arco representa una operación (transformaciones o acciones) entre RDDs. En Spark el usuario codifica el trabajo en una secuencia de comandos definendo implícitamente un DAG. En general, un trabajo en Spark consiste en:

1. Leer el origen de los datos y escribir la primera RDD; dicha RDD puede estar distribuida a través de todo el clúster en diferentes particiones.
2. Realizar una serie de modificaciones a los datos; cada vez que se aplica una operación sobre una RDD se crea una nueva.
3. Materializar el resultado; en general, guardándolo en memoria.

El DAG de cada trabajo permite especificar de manera secuencial cómo serán ejecutadas las transformaciones. Esto es posible pues el DAG permite dividir la ejecución en etapas. Cada etapa contiene *tareas*, siendo una tarea la ejecución de una operación sobre una **única** partición de la RDD.

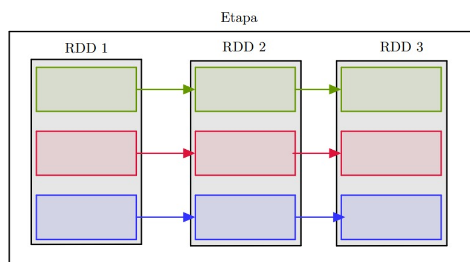

FIGURA 4.4. Ejemploe de Etapa

Por defecto, un único nodo es el encargado de ejecutar una tarea. Para que un trabajo en Spark se ejecute en el menor tiempo posible, se deben asignar la misma cantidad de nodos de trabajo como particiones hay en la primer RDD, bajo la suposición de que ningún otro

RDD tiene un mayor número de particiones si este fuera el caso; en el transcurso de la ejecución, el trabajo pedirá más nodos de trabajo. Es posible cambiar esto en la definición de SparkContex y configurar Spark de tal forma que sea más de un nodo el encargados de ejecutar una única tarea con el propósito de finalizar la ejecución en menor tiempo. Pero esto supone contar con un alto numero de nodos de trabajo.

El criterio de creación de etapas se basa en el tipo de transformaciones que se deban ejecutar. Esto es, una etapa nueva se crea siempre que se deba hacer una trasformación de tipo wide. Por ejemplo, obsérvese el siguiente trabajo sencillo, el cual dado un archivo de texto, entrega un nuevo archivo de texto que contiene parejas (palabra,repeticiones) que expresan el numero de veces que dicha palabra se repite en el archivo [**15**]:

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Para este trabajo, el DAG generado sería, suponiendo que el archivo se ha dividido en tres particiones:

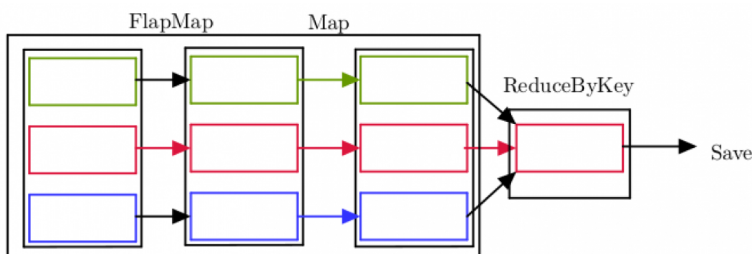

Figura 4.5. Flujo de ejecución representado en un DAG

4.4. Piscinas

Si se tienen varios usuarios que envían trabajos a un clúster manejado por Spark, la configuración más básica permite que cada trabajo enviado sea ejecutado usando todos los recursos disponibles y los trabajos pendientes se archiven en una cola. De esta manera el primer trabajo en entrar será el primero en ser ejecutado, ordenamiento según algoritmo FIFO. Spark permite un mejor aprovechamiento de los

recursos, pues es posible definir grupos en los cuales se agrupan los trabajos de los usuarios dependiendo del nivel de importancia que se les da. Estos grupos, llamados piscinas, permiten asignar una importancia (peso) y una cantidad mínima de núcleos de CPU para cada piscina. Además, al interior de cada piscina se puede escoger si se desea realizar un agendamiento mediante un algoritmo FIFO o FAIR. La creación de piscinas se hace mediante un archivo XML en el cual se especifica las características de cada piscina y el nombre, a continuación de una configuración para las piscinas `administración` y `laboratorio`:

```
<allocations>
  <pool name="administración">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>3</minShare>
  </pool>
  <pool name="laboratorio">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>8</minShare>
  </pool>
</allocations>
```

En esta configuración, en el caso tal que todos los recursos del clúster estén siendo ocupados, si hay más ejecutores disponibles, entonces estos se le asignarán con mayor prioridad a los trabajos pendientes pertenecientes a la piscina `laboratorio`. En esta configuración se esperarse que los trabajos en esta piscina acaben más rápido que cualquier otro trabajo asignado en las otras piscinas, pues tiene valores de `minShare` y `weight` más elevados. Ademas, cuando se tienen varias piscinas, es posible definir cómo se realiza el el agendamiento entre estas, esto se hace desde la configuración del clúster.

4.5. Asignación dinámica de recursos

Esta configuración está por defecto inactiva, pero si se establece `spark.dynamicAllocation.enabled` en `true` desde la configuración del clúster, este servicio quedará activado. La asignación dinámica de recurso permite remover ejecutores que han sido asignados a trabajos que han pasado más de `spark.dynamicAllocation.executorIdleTimeout` segundos (característica necesariamente definida en la en la configuración del clúster si la asignación dinámica esta activa) inactivos para que otros trabajos los puedan utilizar.

Así mismo esta configuración permite entregar por rondas más ejecutores a trabajos que lleven más de `spark.dynamicAllocation.schedulerBacklogTimeout` segundos con tareas pendientes y después de esto el trabajo realizara solicitudes cada `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` segundos. Además la cantidad de ejecutores asignados cada ronda aumentara exponencialmente con respecto a la ronda anterior. Esto es, en la primera ronda se asigna un ejecutor luego dos,cuatro,ocho y así sucesivamente en las siguientes rondas.

Según la documentación oficial de Spark la entrega exponencial de recursos tiene doble motivación. La primera es a corto plazo un trabajo debe hacer petición de recursos cautelosamente pues puede que con pocos ejecutores sea capaz de cumplir la demanda. En segundo lugar, el trabajo debería poder aumentar el uso de sus recursos de manera oportuna en caso de que se necesiten muchos ejecutores. Para finalizar, note que la eliminación de ejecutores debe hacerse cautelosamente pues puede existir en memoria datos importares para la ejecución de algún otro proceso.

Capítulo 5

Especificación formal

Este capítulo presenta una especificación formal del agendamiento de trabajos en Spark usando la sintaxis de Maude [**5**]. La especificación formal consta de varios módulos funcionales que definen la sintaxis requerida. Dichos módulos incluyen, entre otros, la definición de etapas, grafos dirigidos acíclicos, piscinas, nodos de ejecución, etc. Se han definido reglas de reescritura que permiten modelar el comportamiento dinámico de una configuración de Spark, por ejemplo, cuando al interior de una etapa se acaban tareas, cuando se asignan ejecutores a un trabajo, etc.

5.1. Configuracion basada en objetos

Para el desarrollo de la especificación se ha usado el modulo predefinido CONFIGURATION de Maude que soporta el modelamiento de sistemas basados en objetos. Los términos del `sort Configuration` consisten en una colocación de objetos y mensajes. Este tipo de organización es asociativo y conmutativo; cada configuración puede ser pensada como un conjunto de objetos (definidos en el `sort Object`) que describen los posibles estados del sistema. Los identificadores de los objetos se materializan por elementos del `sort Oid`; dichos identificadores en este caso serán números naturales (`subsort Nat <Oid`). El estado de un objeto es descrito por elementos del `sort Attribute` o `AttributeSet` si son varios.

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet  [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet [ctor
   assoc comm id: none] .

  sorts Oid Cid Object Msg Portal Configuration .
  subsort Object Msg Portal < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
```

```
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration [ctor
   config assoc comm id: none] .
  op <> : -> Portal [ctor] .
endm
```

5.2. Modelamiento formal de etapas

Como se mencionó en el Capítulo 4, un trabajo es ejecutado en Spark dividiéndolo en etapas. Estas deben iniciarse en un orden especifico, pues cada etapa genera dependencias con otras etapas posteriores. Es importante observar cuántas y en qué estado se encuentran las tareas al interior de una etapa. Al interior de una etapa cada tarea sufre transformaciones. A esta transformación de una tarea se denomina `pipe`:

```
op pipe[_]:_____     : Nat Nat Bool Bool Bool Bool -> Attribute
     [prec 20].
```

Los dos argumentos de tipo natural especifican sobre el identificador del pipe y qué tarea del pipe que se está ejecutando, respectivamente. Los cuatro argumentos de tipo Booleano informan sobre: la actividad del pipe, si tiene un ejecutor asignados, si esta en un fallo. En el módulo `STAGE` se define la inicialización de una etapa con ayuda de la operación `init-stage`:

```
  op  init-stage: Nat Nat Nat Nat Nat  -> Object.

  eq init-stage(ID, DN, PI, MT, PN)
   = < ID: stage | petition: false,
                   pool’: PI,
                   pipe-cnt: PN,
                   pipe-end: 0 ,
                   DAG: DN,
                   maxtask: MT,
                   $init-stage(PN, none) >.
```

Las variables `ID,DN,PI` de tipo `Nat` sirven de etiqueta para identificar el nombre de la etapa (esto permite utilizar el orden usual de los números naturales para la ejecución de etapas, esto es, la etapa `0` de un trabajo será la primera en ejecutarse y seguirá la etapa `1` así hasta acabar todas la etapas del trabajo), el nombre del DAG al cual pertenece la etapa (esto es importante dado que cuando hay varios DAGs presentes en la configuración es necesario diferenciar entre las etapas de cada uno,

pues cada DAG tiene una petición de recursos especifica) y por último el nombre de la piscina a la cual está asociado el DAG y por ende del trabajo.

Las variables PN,MT permiten conocer la cantidad de pipes que posee cada etapa y el máximo de tareas, que puede tener estas etapas. Específicamente se considera como una cota máxima pues existe una remota posibilidad de acabar la ejecucion de un pipe en un numero menor de tareas. El atributo petition permite saber si la etapa asociada ya realizó petición de ejecutores. Por último, la operación $init-stage(PN, none) permite recurrir sobre PN para la creación de PN pipes:

```
op $init-stage: Nat AttributeSet -> AttributeSet.
eq $init-stage(0, AtS')
   = AtS'.
  eq $init-stage(s(PN), AtS')
   = $init-stage(PN, (AtS',  pipe[s(PN)]: 0 true false false
    false ) ).
```

Observe que un pipe se inicializa en la tarea 0 y en estado activo. Pasa a estar inactivo cuando ha llegado a la tarea máxima o cuando termina correctamente antes de alcanzar este máximo.

5.3. Moldeamiento formal de DAG

Como se vió en la Sección 5.2 cada etapa está relacionada con un DAG, el cual es el encargado de informar en qué etapa se encuentra la ejecución de dicho trabajo, y puede conocer cuántos ejecutores tiene disponibles y cuántos de ellos está utilizando. Esta especificación se encuentra en el módulo funcional DAG donde se define los DAGs como objetos que se pueden crear con la operación init-DAG y que tienen los siguientes atributos:

```
op init-DAG: Nat Nat Nat -> Object.
  eq init-DAG(ID,SN,PN)
   = < ID: DAG | finish: false,
                 pool: PN,
                 current: 0,
                 last: s(SN),
                 pipe-pets: 0,
                 exe-asig: 0,
                 exe-cnt: 0  >.
```

La variable `ID` es la etiqueta identificadora del DAG, `SN` identifica la cantidad de etapas asociadas a este DAG, mientras que `PN` identifica la piscina a la cual pertenece el DAG. Estos son los únicos atributos necesarios para iniciar un DAG. El atributo `current` es de tipo natural que informa sobre la etapa que se está ejecutando, siempre se empieza por la etapa 0. El atributo `pipe-pets` es de tipo natural e informa sobre las peticiones de ejecutores realizadas por las etapas, `exe-cnt` y `exe-asig` son atributos de tipo natural e informan sobre los ejecutores disponibles para el DAG y los ejecutores asignados, respectivamente. Finalmente, `finish` es un atributo con un valor Boolenao que informa si el DAG a terminado su ejecución, es decir, cuando `current` tiene valor `SN` .

5.4. Moldeamiento formal de piscinas

Una piscina permite asignar valores de prioridades a los trabajos, dándoles preferencia en el agendamiento a los trabajos asociados a ellas. En el módulo funcional `POOL` se define la operación `init-pool` para la iniciación de una piscina. Los atributos necesarios para iniciar una piscina son: `IP` que es la etiqueta de la piscina, `W` que identifica el peso de la piscina (i.e, su prioridad), `MS` que identifica el valor de `minShare` y `QD` que es una cola de naturales representando etiquetas de un DAG, con este atributo se puedén conocer todos los DAGs asociados a la piscina:

```
op  init-pool: Nat Nat Nat Queue{Nat} -> Object.

eq init-pool(ID, W, MS, QD  )
 = < IP: pool | waiting-reque: emp,
                cnt-exe: 0,
                weight: W,
                min-share: MS,
                running-task: 0,
                wait: QD,
                executing: empty,
                finished: empty,
                DAG-pets: emp >.
```

El objeto creado para representar las piscinas tiene como atributos `running-taks` que es un atributo de tipo natural que permite conocer cuántas tareas se están ejecutando simultáneamente al interior de la piscina, sin importar si son de diferentes DAGs. Los atributos `DAG-pets`

y `waiting-reque` son colas cuyos elementos son tuplas `(PP,DN)` definidas en el modulo funcional `2-TUPLE`, tales que la cola `DAG-pets` guarda las peticiones `PP` y el DAG que ha echo dicha petición; si las peticiones se entregan completamente se borran de la cola `DAG-pets` pero si no se completa todas la peticiones la cuota faltante se guarda en `waiting-reque`.

El agendamiento de peticiones al interior de una piscina, puede ser mediante el algoritmo `FIFO` o `FAIR`. Para el algoritmo `FAIR`, los trabajos asociados a una piscina tienen todos los mismos pesos, `minshare` y `runnigTaks`. Es decir, Los trabajos que pertenecen a la misma piscina tendrán los mismos parámetros y en este caso el algoritmo `FAIR` reduce el agendamiento de trabajos a un ordenamiento por el valor de la etiqueta. Por otro lado, si el algoritmo `FIFO` está activado para el ordenamiento de peticiones al interior de una piscina, cuando en la ejecución de un trabajo se libera un ejecutor y existen trabajos pendientes de asignación al interior de la misma piscinas, dicho ejecutor es asignado al siguiente trabajo en la cola de peticiones. Es por esto que se ha decidido en la especificación hacer el agendamiento de peticiones al interior de una piscina por el algoritmo `FAIR`

5.5. Moldeamiento formal de ejecutores

En las configuraciones de Spark es necesario saber cuántos ejecutores se tienen disponibles. Esto es importante para realizar la entrega de ejecutores a los trabajos que estén pendientes por ejecución. Para esto en el módulo funcional `EXECUTER` se ha definido la operación `init-exec`:

```
op init-exec          :  Nat Nat Nat -> Object   .
eq init-exec(NE,CE,TT)
   = < NE: exec | core-cnt: CE, time-per-task: TT >.
```

De esta manera `NE` srive como etiqueta e indica la cuenta de cuantos ejecutores disponibles. Es decir, si `NE` es 0 entonces no existen ejecutores disponibles en la configuración. Los atributos `core-cnt` y `time-per-task` son de tipo natural y permiten implementar diferentes configuraciones de tiempo por tarea o de cantidad de núcleos por tarea, respectivamente.

5.6. Moldeamiento formal del agendador

Se ha especificado un objeto llamado `scheduler` definido en el módulo funcional `SCHEDULER`. Este tiene un atributo `pools-petitions`

encargado de guardar las peticiones para posteriormente entregar ejecutores cuando hayan disponibles al primer elemento de la cola. Para la creación del objeto se define la operación `init-scheduler`:

```
op init-scheduler    : Nat -> Object.
eq init-scheduler( IS )
   = < IS: scheduler | pools-petitions: empty >.
```

El atributo `pools-petitions` es una cola de prioridad tal que sus elementos son seis-tuplas de la forma (RT,WE,MS,IP,ID,PP); las variables hacen referencia a los atributos mencionados en las secciones anteriores. Es decir, `IP` y `ID` son las etiquetas de la piscina y el DAG que están haciendo petición, `RT,WE,MS` son los atributos de la piscina y `PP` es la cantidad de peticiones. La cola de prioridades se define en el módulo funcional `LESFTIST-HEAP`, dicho módulo paramétrico en un orden parcial estricto. El parámetro asociado a estas colas de prioridades fue el sort `6-Tuple` para el cual se define el orden `<` que coincide con el algoritmo FAIR para el agendamiento de piscinas en Spark, este algoritmo se ha modelado segun se muestra en [**8**].

5.7. Modelamiento formal de las transiciones

Las transiciones concurrentes del sistema están modeladas por dieciocho reglas de reescritura en la sintaxis de Maude. Las reglas de reescritura aplican sobre toda la configuración Spark es decir son, *topmost* [**14**] y permiten cambiar entre los diferentes estados alcanzables una configuración inicial. En el módulo funcional `SPARK` se define el `sort Sys`, con la intención de incñuir toda la configuración de objetos entre dos corchetes.

```
sort Sys.
op {_}: Configuration -> Sys.
```

Debido a las propiedades del sort `Configuration` las dos configuraciones presentadas a continuación son equivalentes. Esto resulta ideal para la especificación de las reglas de reescritura pues se puede hacer *maching* especificando el estado deseado y el orden de los objetos al interior de la configuración resultara irrelevante. Además, el orden de los objetos en el estado inicial de la configuración no afecta en el conjunto de estados alcanzables desde este:

```
{< 5: exec | core-cnt: 10,time-per-task: 10 >
   < 99: scheduler | pools-petitions: empty >}
{< 99: scheduler | pools-petitions: empty >
   < 5: exec | core-cnt: 10,time-per-task: 10 >}
```

Algunas de las reglas de reescritura se especifican a continuación, el resto de ellas se encuentran referenciadas en el A en el módulo SPARK:

```
rl[petition-stage2DAG2pool]:
{
<0:stage|petition:false,pool':PN,pipe-cnt:PC,DAG:DN',ATS>
<DN':DAG|finish:false,pipe-pets:0,pool:PN,ATS'>
<PN:pool|DAG-pets:HP,ATS''>
Con}
=>
{
<0:stage|petition:true,pool':PN,pipe-cnt:PC,DAG:DN',ATS>
<DN':DAG|finish:false,pipe-pets:PC,pool:PN,ATS'>
<PN:pool|DAG-pets:((PC,DN')HP),ATS''>
Con}.
```

La regla `[petition-stage2DAG2pool]` realiza las peticiones de ejecutores desde la primera etapa del DAG `DN'`, esto añadiendo a la cola de peticiones de la piscina la información de la cantidad de pipes de la primera etapa (PC) y el identificador del DAG al que está asociada la primera etapa del trabajo (DN'). Ademas, el atributo `petition` es actualizado a `true` indicando que la etapa ha realizado la petición a la piscina .

```
rl[petition-pool2sched]:
{
<PN:pool|DAG-pets:HP',weight:WE,min-share:MS,
                           running-task:RT,waiting-reque:HP,ATS
   >
<IS:scheduler|pools-petitions:HS>
Con}
=>
{
<PN:pool|DAG-pets:deletefirst(HP'),weight:WE,min-share:MS,
                             running-task:RT,waiting-reque:(
   first(HP')HP),ATS>
<IS:scheduler|pools-petitions:insert((RT,WE,MS,PN,DN(first(HP')
   ),PP(first(HP'))),HS)>
Con}.
```

La regla `[petition-pool2sched]` solicita ejecutores desde una piscina al agendador. Esto lo hace añadiendo la petición al scheduler. La función `insert` utiliza el agendamiento FAIR para ubicar la petición

en la posición adecuada de la cola de prioridades del `scheduler`. Además, dicha petición queda registrada en la cola de requerimientos de la piscina hasta que dicha petición sea asignada.

```
crl[ asignation-sche2DAG]:
{
<IS:scheduler|pools-petitions:T(N,(RT,WE,MS,PN,DI,PP'),HS,HS'')
    >
<PN:pool|running-task:RT,waiting-reque:HP,ATS>
<DI:DAG|finish:false,pipe-pets:PP',exe-cnt:EC,pool:PN,ATS''>
<NE:exec|core-cnt:CE,time-per-task:TT>
Con}
=>
{
<IS:scheduler|pools-petitions:merge(HS,HS'')>
<PN:pool|running-task:(RT+PP'),(waiting-reque:deletefirst(HP)),
    ATS>
<DI:DAG|finish:false,pipe-pets:0,exe-cnt:(EC+PP'),pool:PN,ATS
    ''>
<remove(NE,PP'):exec|core-cnt:CE,time-per-task:TT>
Con}

if  NE>=PP'.
```

La regla `[asignation-sche2DAG]` entrega ejecutores, si hay disponibles, al DAG con más prioridad en el agendador:

```
crl[advance-pipe]:
{
<CR:stage|petition:true,pool':PN,DAG:DI,maxtask:MT,pipe[BC]:M
    true true false,ATS>
<DI:DAG|finish:false,current:CR,pool:PN,ATS''>
Con}
=>
{
<CR:stage|petition:true,pool':PN,DAG:DI,maxtask:MT,pipe[BC]:s(M
    ) true true false,ATS>
<DI:DAG|finish:false,current:CR,pool:PN,ATS''>
Con}

if  s(M)<MT
.
```

La regla [advance-pipe] permite hacer avances al interior de una etapa. Cada avance de una etapa es la terminación de a lo más MT tareas.

```
rl[finish-DAG]:
{
<DI:DAG|finish:false,current:SN,last:SN,pipe-pets:N,exe-asig:PE
    ,exe-cnt:K,ATS>
<SN:stage|pipe-end:PE,pipe-cnt:PE,ATS'>
<NE:exec|core-cnt:CE,time-per-task:TT>
Con}
=>
{
<DI:DAG|finish:true,current:SN,last:SN,pipe-pets:0,exe-asig:0,
    exe-cnt:0,ATS>
<(NE+K):exec|core-cnt:CE,time-per-task:TT>
<SN:stage|pipe-end:PE,pipe-cnt:PE,ATS'>
Con}.
```

La regla [finish-DAG] finaliza un DAG y por ende el trabajo asociado a este. Cuando se finaliza un DAG se liberan los ejecutores que este estaba ocupando y el atributo finish se actualiza a true.

Capítulo 6

Casos de estudio

En este capitulo se utilizan diferentes herramientas de Maude como los son los comandos `rewrite` y `search` para ejecutar diferentes configuraciones de Apache Spark. Esto con el objetivo de entender el comportamiento dinámico de las diferentes trazas de ejecución de una configuración, comprobar la utilidad de algoritmo de agendamiento FAIR y buscar comportamientos deseables de configuraciones de Spark. El lector es referido a [4] para mas detalle sobre las herramientas de Maude.

6.1. Simulación

Es posible usar el comando `rewrite` para simular un posible comportamiento de ejecución partiendo de un estado inicial de una configuración de Spark. Por ejemplo, en la siguiente configuración, que comprende una piscina con un único DAG compuesto por dos etapas y e 5 ejecutores disponibles:

```
rewrite in SPARK: {
  < 5: exec | core-cnt: 10,time-per-task: 10 >
  < 999: scheduler | pools-petitions: empty >
  < 1: stage | petition: false,pipe-end: 0,pool': 0,pipe-cnt:
    1,dag: 0,
        maxtask: 400,pipe[1]: 0 true false false false >
  < 0: stage | petition: false,pool': 0,dag: 0,maxtask: 200,
    pipe-end: 0,pipe-cnt: 2,
        pipe[1]: 0 true false false false,pipe[2]: 0 true false
     false false >
  < 0: dag | finish: false,pool: 0,current: 0,last: 2,pipe-pets
    : 0,exe-cnt: 0,exe-asig: 0 >
  < 0: pool | weight: 2,min-share: 3,cnt-exe: 0,running-task:
    0,wait: 0,executing: empty,
        finished:empty,dag-pets: emp,waiting-reque: emp >
                  }.
```

```
rewrites: 1825 in 164ms cpu (167ms real) (11128 rewrites/second
   )
result Sys: {
  < 0: stage | pool': 0,petition: true,dag: 0,maxtask: 200,pipe
    -cnt: 2,pipe-end: 2,
        pipe[1]: 200 true true false false,pipe[2]: 200 true
    true false false >
  < 0: dag | finish: true,pipe-pets: 0,pool: 0,current: 2,last:
     2,exe-cnt: 0,exe-asig: 0 >
  < 0: pool | weight: 2,min-share: 3,cnt-exe: 0,running-task:
    0,wait: empty,
        executing: empty,finished: 0,dag-pets: emp,waiting-
    reque: emp >
  < 1: stage | pool': 0,petition: true,dag: 0,maxtask: 400,pipe
    -cnt: 1,
        pipe-end: 1,pipe[1]: 400 true true false false >
  < 5: exec | core-cnt: 10,time-per-task: 10 >
  < 999: scheduler | pools-petitions: empty >
           }
```

Se observa que el resultado de la simulación es el esperado, pues las etapas acabaron sus tareas correctamente. Note que en la etapa 0 cada pipe realizó 200 tareas y en la etapa 1 el único pipe realizó las 400 tareas especificadas por el atributo `maxtask`. Atributos como `pools-petitions`, `wait` y `executing` terminaron en estado `empty` indicando que no quedaron peticiones pendientes y que no quedaron tareas en estado de ejecución, respectivamente. Ademas, el argumento `finish` de la piscina 0 indica que el DAG `0` fializo su ejecución y el número de ejecutores disponibles es el mismo que en el estado inicial.

6.2. Exploración de estados

Si se tiene un número mayor de DAGs y etapas en la configuración inicial, el comando `rewrite` no brinda suficiente información sobre el comportamiento dinámico del agendamiento pues solo nos muestra el resultado de una posible ejecución. Es por esto que resulta interesante utilizar el comando `search` para comprobar si es posible alcanzar ciertos estados con características especificas partiendo de un estado inicial, para mas información sobre el comando `search` se remite al lector a [**4**].

Considere la configuración presentada en el Apéndice B (B.1) en la cual encuentran dos piscinas, cada una con propiedades diferentes, la piscina `3` tiene tres DAGs asociados (`30,31,32`), mientras que la piscina

4 tiene dos dags asociados (40,41). Además, inicialmente se encuentran 13 ejecutores disponibles para toda la configuración de Spark.

Uno de los propósitos de la búsqueda es mostrar la evolución de la configuración cuando los trabajos asignados a la piscinas han terminado. El resultado esperado es que deben existir estados en los cuales todos los trabajos de una piscina ocupen los 13 ejecutores disponibles. Los comandos empleados para la búsqueda se muestran a continuación. Por cuestiones de espacio y comodidad en la lectura, en las siguientes búsquedas se referencia la configuración del Apéndice B (B.1) como EJEM1:

```
search  in SPARK:
 EJEM1
=>*
{ CON:Configuration
  < 4: pool | finished: (40,41), Ats:AttributeSet >
  < 0: exec | core-cnt: 10, time-per-task: 10 >
}
.
search  in SPARK:
  EJEM1
=>*
{ CON:Configuration
  < 3: pool | finished: (32,30,31), Ats:AttributeSet >
  < 0: exec | core-cnt: 10, time-per-task: 10 >
}
.
```

Con un tiempo límite de treinta minutos, los comandos de búsqueda no arrojaron algún resultado debido al problema inherente de la explosión de estados. Para exponer la complejidad de la ejecución, se presentan diferentes búsquedas con variaciones en la profundidad hasta encontrar un estado con las condiciones dadas. Los resultados de las diferentes exploraciones se resumen en el cuadro 1:

Profundidad	Tiempo de ejecución (Min)	Estados	Reescrituras
10	0.5	148502	3612935
11	2.6	420310	11453270
12	23.1	1121380	34177039
13	-	-	-

Cuadro 1. Datos de búsquedas.

Es parte del trabajo futuro aplicar técnicas de reducción de estados para ser capaces de realizar busquedas eficientes atravez de la especificación. Por ejemplo, se explorar como se puede comportar el agendador cuando los DAGs `30,32,41` han finaliado su ejecucion, también se exploro si es posible que existan peticiones pendientes del DAG `30` de un ejecutor mientras que el DAG `31` tiene una cuenta pendiente de un ejecutor. Teniendo en cuenta esta ultima configuracion se realizo una nueva búsqueda añadiendo dos condicones; no existiesen ejecutores disponibles y la ultima etapa del DAG `40` hubiera acabado, note que esta ultima condición permite que el DAG `40` tenga asignado ejecutores o por el contrario ya se hayan entregado a la siguiente etapa. Luego de la ejecución de estos ejemplos se excedió el tiempo limite sin encontrar resultados.

```
search  in SPARK:
  EJEM1
=>*
{ CON:Configuration
  < 41: dag | finish: true, Ats:AttributeSet >
  < 32: dag | finish: true, Ats':AttributeSet >
  < 30: dag | finish: true, Ats':AttributeSet >
}
.
search  in SPARK:
  EJEM1
=>*
{ CON:Configuration
  < 999: scheduler |
     pools-petitions: T(1,(0,2,6,3,30,1),T(1,(0,2,6,3,31,1),
    empty,empty),empty) >}.

.
search  in SPARK:
  EJEM1
=>*
{ CON:Configuration
  < 999: scheduler |
     pools-petitions: T(1,(0,2,6,3,30,1),T(1,(0,2,6,3,31,1),
    empty,empty),empty) >
  < 0 : exec | core-cnt: 10, time-per-task: 10 >
  < 0: stage | pool': 4,dag: 40, pipe[8]: 50 true true false
    false, Ats:AttributeSet >
```

```
}
.
```

En contra parte a los resultados obtenidos con la configuración del Apéndice B (B.1), reduciendo la configuración (B.2) y ejecutando nuevamente el experimento donde se buscaban los estados tales que el DAG 40 hubiera terminado su ejecución y se hubiese informado de su estado a la piscina asociada a el, se encontró que hay 1251 estados que satisfacen las condiciones especificadas:

```
search  in SPARK:
  EJEM2
=>*
{CON:Configuration
< 4: pool | finished: (40),Ats:AttributeSet >
}
.
```

Algunas de estas soluciones nos permiten observar estados donde los trabajos de la piscina 4 han sido ejecutados y el agendador tiene peticiones de los otros dos trabajos de la piscina 3. En la especificación esto se representa como sigue:

```
< 999:
    scheduler | pools-petitions: T(1,(0,2,6,3,32,2),T
    (1,(0,2,6,3,30,2),empty,empty),empty) >
```

Un resultado interesante se obtine con la configuración EJEM2 (B.2), la cual se implemento con el objetivo de estudiar si es posible que el trabajo asociado a la piscina 4 acabe dejando algún tipo de requerimiento pendiente al agendador:

```
search  in SPARK:
  EJEM2
=>*
{CON:Configuration  < 4: pool | Ats:AttributeSet,finished: 40 >
     < 999: scheduler | pools-petitions:  T(P:Nat,M,5,2,4,40,N),
    H:Heap{6-Tup},G:Heap{6-Tup}) >}.

No solution.
states: 580216  rewrites: 12483279 in 904292ms cpu (904392ms
    real) (13804 rewrites/second)
```

En conclusión, dada una ejecución arbitraria, siempre que los DAGs asociados a la piscina 4 hayan finalizado, en el agendador no existirán

requerimientos pendientes asociados a esta piscina. Este resultado confirma el buen uso del agendamiento de tareas de tipo FAIR, pues la prioridad de la piscina 4 es mayor que la de la piscina 3, es decir en el agendamiento FAIR la piscina 4 tiene prioridad sobre la piscina 3 en cualquier estado.

Capítulo 7

Conclusión y trabajo futuro

La lógica de reescritura es una lógica para especificar sistemas concurrentes la cual, al ser combinada con Maude y sus herramientas, facilita la especificación formal de sistemas como Apache Spark y la exploración de estados alcanzables partiendo de un estado inicial. Este proceso, aunque limitado por la gran cantidad de estados iniciales y alcanzables, es automático y en algunos casos, efectivo.

Se comprobó el buen uso del agendamiento de tareas de tipo FAIR el cual permite priorizar trabajos según la importancia asignada a ellos por las piscinas que los contienen. Además se comprobó que la verificación algorítmica para ejecuciones complejas de configuraciones de Apache Spark puede llegar a ser en exceso compleja debido a la gran cantidad de diferentes interacciones que pueden suceder en el agendamiento y ejecución de los trabajos al interior de una configuración, generando una expolición exponencial de estados.

Como trabajo futuro se propone ejecutar ejemplos reales en un ambiente de Apache Spark. Dichos ejemplos deberán ser ejecutados por varios usuarios o haciendo uso de las piscinas de Spark con el objetivo de utilizar el agendamiento FAIR, y buscar su traza de ejecución usando la especificación propuesta en este documento para comprobar veracidad de la misma. Así mismo, se propone extender la especificación de tal forma que sea posible contemplar fallos aleatorios en la ejecución y añadir tiempo real a la ejecución para poder comparar rendimientos entre el algoritmo FIFO y el algoritmo FAIR.

Finamente el excesivo crecimiento en la cantidad de estados alcanzables a partir de una configuración inicial, supone el uso de técnicas de reducción de estados como abstracciones ecuaciones [**13**].

Apéndice A

Especificación formal en Maude

Este anexo incluye los módulos funcionales 2-TUPLE, 6-TUPLE, DAG EXECUTER, SCHEDULER, POOL, LESFTIST-HEAP, STAGE y el módulos de sistema: SPARK.

A.1. 2-TUPLE

```
fmod 2-TUPLE is
  sort 2-Tuple .
  pr NAT .
  op ((_,_)) : Nat Nat -> 2-Tuple [ctor] .
  op PP_ : 2-Tuple -> Nat .
  op DN_ : 2-Tuple -> Nat .
  eq PP(A:Nat, B:Nat) = A:Nat .
  eq DN(A:Nat, B:Nat) = B:Nat .
  op _<_ : 2-Tuple 2-Tuple -> Bool .
  vars A B C D : Nat .
  eq  (A,B) < (C,D) = if A < C then true else false fi .

endfm
```

A.2. 6-TUPLE

```
fmod 6-TUPLE is
  sort 6-Tuple .
  pr NAT .
  pr RAT .
  pr CONVERSION .
  op compare     : Rat Rat -> Rat .
  op ((_,_,_,_,_,_)) : Nat Nat Nat Nat Nat Nat -> 6-Tuple [ctor
   ] .
  op rt_         : 6-Tuple -> Nat .
  op weigh_      : 6-Tuple -> Nat .
  op min-share_  : 6-Tuple -> Nat .
  op PI_         : 6-Tuple -> Nat .
```

```
op DI_         : 6-Tuple -> Nat .
op PP_         : 6-Tuple -> Nat .
ops TTW MSR    : 6-Tuple -> Rat .
op _<_         : 6-Tuple 6-Tuple -> Bool .

eq rt( A:Nat , B:Nat , C:Nat, D:Nat , E:Nat , F:Nat )
 = A:Nat .
eq weigh( A:Nat , B:Nat , C:Nat, D:Nat, E:Nat , F:Nat )     =
  B:Nat .
eq min-share( A:Nat , B:Nat , C:Nat, D:Nat, E:Nat , F:Nat ) =
  C:Nat .
eq PI( A:Nat , B:Nat , C:Nat, D:Nat, E:Nat , F:Nat )        =
  D:Nat .
eq DI( A:Nat , B:Nat , C:Nat, D:Nat, E:Nat , F:Nat )        =
  E:Nat .
eq PP( A:Nat , B:Nat , C:Nat, D:Nat, E:Nat , F:Nat )        =
  F:Nat .


vars R R' : Rat .
var PO PO' : 6-Tuple .

eq TTW( PO ) = (rt( PO ) / max( min-share(PO),1)) .
eq MSR( PO)  = rt( PO ) / weigh( PO )  .

ceq compare( R , R') =  1   if R < R' .
ceq compare( R , R') = -1   if R' < R .
ceq compare( R , R') =  0   if R = R' .



ceq PO < PO' = true
if rt( PO ) < min-share( PO ) /\  not(rt( PO' ) < min-share(
 PO' ))    .
ceq PO < PO' = false
if rt( PO' ) < min-share( PO' ) /\  not( rt( PO ) < min-share
 ( PO ) ) .


ceq PO < PO' = true
if   rt( PO ) < min-share( PO )  /\   (rt( PO' ) < min-share(
  PO' ))
```

```
  /\   compare( compare( TTW(PO) , TTW(PO') ),  compare( MSR(PO
   ) , MSR(PO') ))   < 0
  .
  ceq PO < PO' = false
  if  rt( PO ) < min-share( PO )  /\   (rt( PO' ) < min-share(
   PO' ))
  /\   not(compare( compare( TTW(PO) , TTW(PO') ),  compare(
   MSR(PO) , MSR(PO') ))   < 0)
  .
  ceq PO < PO' = false
  if  not(rt( PO ) < min-share( PO )) /\   not(rt( PO' ) < min-
   share( PO' ))
  /\   not (compare( compare( TTW(PO) , TTW(PO') ),  compare(
   MSR(PO) , MSR(PO') ))   < 0)
  .
  ceq PO < PO' = true
  if  not(rt( PO ) < min-share( PO )) /\   not(rt( PO' ) < min-
   share( PO' ))
  /\   (compare( compare( TTW(PO) , TTW(PO') ),  compare( MSR(
   PO) , MSR(PO') ))   < 0)
  .
endfm
```

A.3. DAG

```
fmod DAG is
  pr BOOL .
  pr NAT .
  inc CONFIGURATION .
  subsort Nat < Oid .

  op dag : -> Cid .
  op finish:_              : Bool -> Attribute .
  op pipe-pets:_          : Nat  -> Attribute .
  op pool:_               : Nat -> Attribute .
  op current:_            : Nat -> Attribute .
  op last:_               : Nat -> Attribute .
  op exe-cnt:_            : Nat -> Attribute .
  op exe-asig:_           : Nat -> Attribute .
  vars ID SN PN : Nat .

  op init-dag : Nat Nat Nat -> Object .
```

```
  eq init-dag(ID, SN, PN)
   = < ID : dag | finish: false ,
                  pool: PN ,
                  current: 0,
                  last: s(SN),
                  pipe-pets: 0 ,
                  exe-asig: 0 ,
                  exe-cnt: 0  > .
endfm
```

A.4. EXECUTER

```
fmod EXECUTER is
  pr NAT .
  pr INT .
  inc CONFIGURATION .
  sort Nat? .
  subsort Nat < Oid .
  subsort Nat < Nat? .
  op remove(_,_)               :  Int Int     -> Int      .
  op exec                   :      -> Cid        .
  op core-cnt:_             :  Nat -> Attribute .
  op time-per-task:_        :  Nat -> Attribute .
  op init-exec              :  Nat Nat Nat -> Object    .
  vars N M CE NE  TT : Nat .
  eq remove(N,M) = (N - M) .
  eq init-exec(NE,CE,TT)
   = < NE : exec | core-cnt: CE , time-per-task: TT > .
endfm
```

A.5. LESFTIST-HEAP

```
fmod LESFTIST-HEAP{X :: STRICT-TOTAL-ORDER} is
  pr INT .
  protecting NAT .
  sort Heap{X} NeHeap{X} .
  subsort NeHeap{X} < Heap{X} .
  op remove'(_,_)               :  Int Int     -> Int      .
  op empty : -> Heap{X} .
  op T(_,_,_,_) : Nat X$Elt Heap{X} Heap{X} -> NeHeap{X} .

  vars L L' R R' : Heap{X} .
```

```
vars E E'       : X$Elt .
vars Ra Ra' N M    : Nat .
eq remove'(N,M) = (N - M) .

op isEmpty : Heap{X} -> Bool .
eq isEmpty(empty) = true .
eq isEmpty(T(Ra,E,L,R)) = false .

op rank : Heap{X} -> Nat .
eq rank(empty)
 = 0 .
eq rank(T(Ra,E,L,R))
 = Ra .

op makeT : X$Elt Heap{X} Heap{X} -> NeHeap{X} .
eq makeT(E,L,R)
 = if rank(L) >= rank(R)
   then T(rank(R) + 1,E,L,R)
   else T(rank(L) + 1,E,R,L)
   fi .
op merge : Heap{X} Heap{X} -> Heap{X} .
eq merge(empty, L)
 = L .
eq merge(L, empty)
 = L .
eq merge(T(Ra,E,L,R),T(Ra',E',L',R'))
 = if (E < E' or E == E')
   then makeT(E,L,merge(R,T(Ra',E',L',R')))
   else makeT(E',L',merge(T(Ra,E,L,R),R'))
   fi .

op insert : X$Elt Heap{X} -> NeHeap{X} .
eq insert(E,L)
 = merge(T(1,E,empty,empty),L) .

op findMin : NeHeap{X} -> X$Elt .
eq findMin(T(Ra,E,L,R))
 = E .

op deleteMin : NeHeap{X} -> Heap{X} .
eq deleteMin(T(Ra,E,L,R))
 = merge(L,R) .
```

```
endfm
```

A.6. POOL

```
load queue.maude
load leftistHeap.maude
load 2-tuple
view 2-Tup from TRIV to 2-TUPLE is
  sort Elt to 2-Tuple .
endv
fmod POOL is
  pr NAT .
  pr INT .
  pr SET{Nat} .
  pr QUEUE{Nat} .
  pr QUEUE{2-Tup} .
  inc CONFIGURATION .
  subsort Nat < Oid .

  op pool                       :      -> Cid .
  op weight:_                   : Nat -> Attribute .
  op min-share:_                : Nat -> Attribute .
  op cnt-exe:_                  : Nat -> Attribute .
  op running-task:_             : Int -> Attribute [prec 20] .
  op wait:_                     : Set{Nat} -> Attribute .
  ops executing:_ finished:_ : Set{Nat}   -> Attribute .
  op dag-pets:_                 : Queue{2-Tup} -> Attribute .
  op waiting-reque:_            : Queue{2-Tup} -> Attribute .

  vars ID W MS DI : Nat .
  var  QD : Set{Nat} .
  vars AtS AtS' : AttributeSet .

  op  init-pool : Nat Nat Nat Queue{Nat} -> Object .

  eq init-pool(ID, W, MS , QD  )
   = < ID : pool | waiting-reque: emp ,
                   cnt-exe: 0,
                   weight: W,
                   min-share: MS,
                   running-task: 0,
                   wait: QD ,
```

```
                   executing: empty,
                   finished: empty,
                   dag-pets: emp > .
endfm
```

A.7. QUEUE

```
fmod QUEUE{X :: TRIV} is
  protecting NAT .
  sorts  NeQueue{X} Queue{X} .
  subsort X$Elt < NeQueue{X} < Queue{X} .

  op emp :                -> Queue{X}   .
  op _ _ : X$Elt Queue{X} -> NeQueue{X} [id: emp ] .
  ops first  deletefirst   : Queue{X}       -> X$Elt .
  op l                     : Queue{X}       -> Nat    .

  var S : X$Elt . var Queue : Queue{X} .
  eq l( emp ) = 0 .
  eq l( S ) = 0 .
  eq l( S Queue ) = 1 + l( Queue )   .
  eq first( S ) = S .
  eq first( S Queue  ) = first ( Queue ) .
  eq deletefirst( S )
   = emp .
  eq deletefirst( S Queue )
   = S deletefirst( Queue ) .
endfm
```

A.8. SCHEDULER

```
load leftistHeap.maude
load 6-tuple.maude
view 6-Tup from STRICT-TOTAL-ORDER to 6-TUPLE is
  sort Elt to 6-Tuple .
endv


fmod SCHEDULER is
  pr NAT .
  pr LESFTIST-HEAP{6-Tup} .
  inc CONFIGURATION .
```

```
  subsort Nat < Oid .
  op scheduler          :  -> Cid .
  op pools-petitions:_  :  Heap{6-Tup} -> Attribute .
  op init-scheduler     : Nat -> Object .
  var IS : Nat .
  eq init-scheduler( IS )
   = < IS : scheduler | pools-petitions: empty > .

endfm
```

A.9. STAGE

```
fmod STAGE is
  pr NAT .
  pr BOOL .
  inc CONFIGURATION .
  subsort Nat < Oid .
  op stage             :                      -> Cid .
  op pool':_           : Nat                  -> Attribute .
  op petition:_        : Bool                  -> Attribute .
  op dag:_             : Nat                  -> Attribute .
  op maxtask:_         : Nat                  -> Attribute .
  op pipe-cnt:_        : Nat                  -> Attribute .
  op pipe-end:_        : Nat                  -> Attribute .
  op pipe[_]:_____     : Nat Nat Bool Bool Bool Bool ->
   Attribute [prec 20] .
  vars ID MT PN DN PI : Nat .
  vars AtS AtS' : AttributeSet .

  op  init-stage : Nat Nat Nat Nat Nat  -> Object .
  op $init-stage : Nat AttributeSet -> AttributeSet .
  eq init-stage(ID, DN, PI , MT, PN)
   = < ID : stage | petition: false ,
                    pool': PI ,
                    pipe-cnt: PN ,
                    pipe-end: 0   ,
                    dag: DN ,
                    maxtask: MT,
                    $init-stage(PN, none) > .
  eq $init-stage(0, AtS')
   = AtS' .
  eq $init-stage(s(PN), AtS')
```

```
  = $init-stage(PN, (AtS',  pipe[s(PN)]: 0 true false false
   false ) ) .
endfm
```

A.10. SPARK

```
load stage.maude
load executer.maude
load dag.maude
load pool.maude
load scheduler.maude
mod SPARK is
  inc CONFIGURATION .
  pr STAGE .
  pr DAG .
  pr EXECUTER .
  pr POOL .
  pr SCHEDULER .
  pr INT .
  sort Sys .
  op {_} : Configuration -> Sys .
  var HS' : NeHeap{6-Tup} .
  var  HS HS'' : Heap{6-Tup} .
  var HP' : NeQueue{2-Tup} .
  vars HN HP  : Queue{2-Tup} .
  var Con : Configuration .
  vars ED FD QD' : Set{Nat} .
  var WD QD : Queue{Nat} .
  vars K E PP' NE' : NzNat .
  vars N N' M B K' PP'' PE' CR' SN' PE'' EA DI'  TT CC CE DI NE
    CR MS SN DN' PN BC' PC RT LP NP PP PC' WE IS EC BC MT PE  :
    Nat .
  var AT AT' : Attribute .
  vars  ATS ATS' ATS'' ATS''' : AttributeSet .



     rl [petition-stage2dag2pool]:
     { < 0 : stage | petition: false , pool': PN , pipe-cnt: PC
     , dag: DN' , ATS  >
       < DN' : dag | finish: false , pipe-pets: 0 , pool: PN  ,
    ATS' >
```

```
     < PN : pool |  dag-pets: HP , ATS'' >   Con }
=> { < 0 : stage | petition: true , pool': PN , pipe-cnt: PC
  , dag: DN' , ATS >
     < DN' : dag | finish: false , pipe-pets: PC , pool: PN ,
   ATS' >
     < PN : pool |  dag-pets: ( (PC,DN') HP ) , ATS'' > Con }
   .



   rl [petition-pool2sched]:
      { < PN : pool |  dag-pets: HP', weight: WE , min-share:
   MS, running-task: RT, waiting-reque: HP  ,ATS  >
        < IS : scheduler | pools-petitions: HS >  Con }
   => { < PN : pool |  dag-pets: deletefirst(HP'), weight: WE
   , min-share: MS, running-task: RT , waiting-reque: ( first(
  HP') HP )   ,ATS  >
        < IS : scheduler | pools-petitions: insert((RT,WE,MS,
  PN,DN(first(HP')),PP(first(HP')) ),HS) > Con } .


  crl [asignation-sche2dag]:
     { < IS : scheduler | pools-petitions: T(N,(RT,WE,MS,PN,
  DI,PP'),HS,HS'')  >
       < PN : pool | running-task: RT , waiting-reque: HP  ,
  ATS  >
       < DI : dag | finish: false , pipe-pets: PP' ,  exe-cnt
  : EC ,  pool: PN , ATS''  >
       < NE : exec | core-cnt: CE  , time-per-task: TT >
       Con }
  =>
     { < IS : scheduler | pools-petitions: merge(HS,HS'')  >
       < PN : pool |  running-task: (RT ) , (waiting-reque:
  deletefirst(HP) )  ,ATS  >
       < DI : dag | finish: false , pipe-pets: 0 ,  exe-cnt:
  (EC + PP' )   ,  pool: PN , ATS''  >
       < remove(NE,PP') : exec | core-cnt: CE   , time-per-
  task: TT >
       Con }

   if NE >= PP' .
```

```
 crl [asignation-sche2dag1]:
    { < IS : scheduler | pools-petitions: T(N,(RT,WE,MS,PN,
DI,PP'),HS,HS'')  >
      < PN : pool | waiting-reque: HP  ,ATS  >
      < DI : dag | finish: false , pipe-pets: 0 ,  exe-cnt:
 EC ,  pool: PN , ATS''  >
      < NE : exec | core-cnt: CE  , time-per-task: TT >
      Con }
 =>
    { < IS : scheduler | pools-petitions: merge(HS,HS'')  >
      < PN : pool | (waiting-reque: deletefirst(HP) )  ,ATS
  >
      < DI : dag | finish: false , pipe-pets: 0 ,  exe-cnt:
 (EC + PP' )  ,  pool: PN , ATS''  >
      < remove(NE,PP') : exec | core-cnt: CE   , time-per-
task: TT >
      Con }

  if NE >= PP' .


  crl [asignation-sche2dag3]:
     { < IS : scheduler | pools-petitions: T(N,(RT,WE,MS,PN
,DI,PP'),HS,HS'')  >
       < PN : pool | waiting-reque:( (PP',DI) HP ) ,ATS  >
       < DI : dag | finish: false , pipe-pets: 0 ,  exe-cnt
: EC ,  pool: PN , ATS''  >
       < NE' : exec | core-cnt: CE  , time-per-task: TT >
Con }
  =>
     { < IS : scheduler | pools-petitions: merge(HS,HS'')
>
       < PN : pool | waiting-reque:( ( sd(PP',NE'),DI ) HP
) ,ATS  >
       < DI : dag | finish: false , pipe-pets: remove(PP',
NE') ,  exe-cnt: (EC + NE' )  ,  pool: PN , ATS''  >
       < 0 : exec | core-cnt: CE   , time-per-task: TT >
Con }

   if PP' > NE'   .
```

```
 crl [asignation-sche2dag2]:
    { < IS : scheduler | pools-petitions: T(N,(RT,WE,MS,PN,
DI,PP'),HS,HS'')  >
      < PN : pool |  running-task: RT ,  waiting-reque:( (
PP',DI) HP ) ,ATS  >
      < DI : dag | finish: false , pipe-pets: PP' ,  exe-
cnt: EC ,  pool: PN , ATS''  >
      < NE' : exec | core-cnt: CE  , time-per-task: TT >
Con }
 =>
    { < IS : scheduler | pools-petitions: T(N,(RT,WE,MS,PN,
DI, remove'(PP',NE') ),HS,HS'')  >
      < PN : pool |  running-task: ( RT ) , waiting-reque:(
 ( sd(PP',NE'),DI ) HP ) ,ATS  >
      < DI : dag | finish: false , pipe-pets: remove(PP',NE
') ,  exe-cnt: (EC + NE' )  ,  pool: PN , ATS''  >
      < 0 : exec | core-cnt: CE   , time-per-task: TT > Con
 }

  if PP' > NE'   .

  crl [asignation-dag2stage]:
    {< DI : dag | finish: false , exe-asig: EA , exe-cnt:
EC ,  pool: PN , ATS , current: CR    >
     < CR : stage | pool': PN , dag: DI , pipe[N]: 0 true
false false false , ATS' > Con
     < PN : pool | ( wait: (DI,QD') ) , executing: ED ,
running-task: RT , ATS'' >
    }
 =>
    {< DI : dag | finish: false , exe-asig: s(EA) , exe-cnt
: EC ,  pool: PN , current: CR , ATS >
     < CR : stage | pool': PN , dag: DI , pipe[N]: 0 true
true false false , ATS' > Con
     < PN : pool | ( wait: QD' ) , (executing: (DI,ED)) ,
running-task: s(RT) , ATS'' >
    }
 if s(EA) <= EC
  .
```

```
 crl [asignation-dag2stage-1]:
     {< DI : dag | finish: false , exe-asig: EA , exe-cnt:
EC ,  pool: PN , ATS , current: CR    >
      < CR : stage | pool': PN , dag: DI , pipe[N]: 0 true
false false false , ATS' >
      < PN : pool |  (executing: (DI,ED)) ,running-task: RT
 , ATS'' >
      Con
     }
  =>
     {< DI : dag | finish: false , exe-asig: s(EA) , exe-
cnt: EC ,  pool: PN , current: CR , ATS >
      < CR : stage | pool': PN , dag: DI , pipe[N]: 0 true
true false false , ATS' >
      < PN : pool |  (executing: (DI,ED)) ,running-task: s(
RT) , ATS'' >
      Con
     }
  if s(EA) <= EC
  .


 crl [advance-pipe] :
   {
   < CR : stage | petition: true , pool': PN   ,  dag: DI ,
maxtask: MT,  pipe[BC]: M true true false false   , ATS >
    < DI : dag | finish: false , current: CR , pool: PN   ,
ATS'' >
    Con}
 =>
    {
    < CR : stage | petition: true , pool': PN   ,  dag: DI ,
 maxtask: MT,  pipe[BC]: s(M) true true false false , ATS  >
     < DI : dag | finish: false , current: CR , pool: PN   ,
 ATS''  >
     Con }

 if s(M) < MT
 .

 crl [advance-pipe1] :
```

```
    {< PN : pool |  (executing: (DI,ED)) ,running-task: s(RT
) , ATS' >
    < CR : stage | petition: true , pool': PN  , pipe-end:
PE ,  dag: DI , maxtask: MT,  pipe[BC]: M true true false
false  , ATS >
     < DI : dag | finish: false , current: CR , pool: PN  ,
ATS'' >
     Con}
  =>
     {< PN : pool |  (executing: (DI,ED)) ,running-task: (RT
) , ATS' >
     < CR : stage | petition: true , pool': PN  ,  dag: DI ,
 pipe-end: s(PE) , maxtask: MT,  pipe[BC]: s(M) true false
false false , ATS  >
      < DI : dag | finish: false , current: CR , pool: PN  ,
 ATS''  >
      Con }

  if s(M) = MT
  .



  crl [advance-pipe2] :
    {< CR : stage | petition: true , pool': PN  , pipe-cnt:
PC ,  dag: DI , maxtask: MT,  pipe[BC]: M true true false
false  ,  pipe[BC']: 0 true false false false  , ATS >
     < DI : dag | finish: false , current: CR , pool: PN   ,
 exe-asig: EA ,  ATS'' >
     Con}
  =>
     { < CR : stage | petition: true , pool': PN  , pipe-cnt
: PC , dag: DI , maxtask: MT,  pipe[BC]: MT true false false
 false , pipe[BC']: 0 true true false false  , ATS  >
      < DI : dag |  finish: false , current: CR , pool: PN
, exe-asig: EA , ATS''  >
      Con }

  if (PC > EA) /\ (s(M) == MT)
  .
```

```
 rl [advance-stage] :
 {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
    < s(CR) : stage | petition: false , pool': PN  , pipe-
cnt: PE , dag: DI  , ATS''   >
    < DI : dag | finish: false ,  pool: PN , current: CR ,
last: SN, pipe-pets:  0  , exe-cnt: PE , AT >

 Con}
 =>
 { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
   < DI : dag | finish: false ,  pool: PN , current: s(CR)
, last: SN, pipe-pets:  0 , exe-asig: 0 , exe-cnt: PE >
   < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE , dag: DI  , ATS''  >
 Con}
 .
 crl [advance-stage1] :
 {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
    < s(CR) : stage | petition: false , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''   >
    < DI : dag | finish: false , pool: PN , current: CR ,
last: SN, pipe-pets:  N  , exe-cnt: PE  , AT >
    < PN : pool | dag-pets:  HP , ATS' >
 Con}
 =>
 { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
   < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''  >
   < DI : dag | finish: false ,  pool: PN , current: s(CR)
, last: SN, pipe-pets: (N + sd(PE,PE') )  , exe-asig: 0 ,
exe-cnt: PE >
   < PN : pool | dag-pets: ( (sd(PE,PE'),DI) HP) , ATS' >
 Con}
 if PE < PE'
 .
 crl [advance-stage2] :
 {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
```

```
    < s(CR) : stage | petition: false , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''   >
    < DI : dag | finish: false , pool: PN , current: CR ,
last: SN, pipe-pets:  N  , exe-cnt: N'  , AT >
    < PN : pool | dag-pets:  HP , ATS' >
 Con}
 =>
 { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
   < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''  >
   < DI : dag | finish: false ,  pool: PN , current: s(CR)
, last: SN, pipe-pets: (N + sd(PE,PE') )  , exe-asig: 0 ,
exe-cnt: N' >
   < PN : pool | dag-pets: ( (sd(PE,PE'),DI) HP) , ATS' >
 Con}
 if PE < PE'
 .

 crl [advance-stage3] :
 {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
    < s(CR) : stage | petition: false ,  pipe-cnt: PE' ,
pool': PN   , dag: DI  , ATS''   >
    < DI : dag | finish: false ,  pool: PN , current: CR ,
last: SN, pipe-pets:  N  , exe-cnt: N'  , AT >
 Con}
 =>
 { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
   < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''  >
   < DI : dag | finish: false ,  pool: PN , current: s(CR)
, last: SN, pipe-pets: N  , exe-asig: 0 , exe-cnt: N' >
 Con}
 if PE = PE'
 .


 crl [advance-stage4] :
 {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
```

```
    < s(CR) : stage | petition: false , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''   >
    < DI : dag | finish: false , pool: PN , current: CR ,
last: SN, pipe-pets:  0  , exe-cnt: PE , AT >
    < PN : pool | ( waiting-reque: ( (PP,DI') HP) ) , ATS'
>
    < DI' : dag | finish: false , pool: PN , current: CR' ,
 last: SN', pipe-pets:  PP  , exe-cnt: PE'' , AT' >
  Con}

  =>
  { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
   < DI : dag | finish: false ,  pool: PN , current: s(CR)
, last: SN, pipe-pets:  0 , exe-asig: 0 , exe-cnt: PE' >
   < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''  >
   < PN : pool | ( waiting-reque: ( (sd(PP,sd(PE,PE')),DI')
 HP) ), ATS' >
   < DI' : dag | finish: false , pool: PN , current: CR' ,
last: SN', pipe-pets:  sd(PP,sd(PE,PE'))  , exe-cnt: (PE'' +
 sd(PE,PE') ) , AT' >

   Con}
  if PE' < PE  /\ CR' < SN'
  .

  crl [advance-stage5] :
  {  < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS  >
    < s(CR) : stage | petition: false , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''   >
    < DI : dag | finish: false , pool: PN , current: CR ,
last: SN, pipe-pets:  0  , exe-cnt: PE , AT >
    < PN : pool |  waiting-reque:  emp  , ATS' >
    < NE : exec | core-cnt: CE , time-per-task: TT >
  Con}

  =>
  { < CR : stage | petition: true , pipe-end: PE  , pool':
PN  , pipe-cnt: PE , dag: DI , ATS >
```

```
    < DI : dag |  finish: false , pool: PN , current: s(CR)
, last: SN, pipe-pets:  0 , exe-asig: 0 , exe-cnt: PE' >
    < s(CR) : stage |  petition: true  , pool': PN  , pipe-
cnt: PE' , dag: DI  , ATS''  >
    < PN : pool |  waiting-reque: emp , ATS' >
    < (NE + sd(PE,PE') ) : exec | core-cnt: CE , time-per-
task: TT >
    Con}
  if PE' < PE
  .

  rl [finish-dag] :
  {< DI : dag | pool: PN , finish: false ,  current: SN ,
last: s(SN), pipe-pets:  N , exe-asig: N' , exe-cnt: K , ATS
  >
   < SN : stage | pool': PN,dag: DI, pipe-end: PE , pipe-cnt
: PE , ATS' >
   < NE : exec | core-cnt: CE , time-per-task: TT >
   < PN : pool |  (executing: (DI,ED)) , finished: QD' , ATS
'' >

    Con}
  =>
  {< DI : dag |  pool: PN, finish: true , current: s(SN) ,
last: s(SN), pipe-pets:  0 , exe-asig: 0 , exe-cnt: 0 , ATS
 >
  < SN : stage |  pool': PN,dag: DI,pipe-end: PE , pipe-cnt:
 PE , ATS' >
  < (NE + K ) : exec | core-cnt: CE , time-per-task: TT >
  < PN : pool |  (executing: (ED)) , finished: (DI,QD') ,
ATS'' >


  Con}
  .

endm
```

Apéndice B

Ejemplos casos de estudio

B.1. EJEM1

```
{
< 3 : pool | weight: 2,min-share: 6,cnt-exe: 0,running-task: 0,
   wait: (31,32,30) ,
          executing: empty,finished: empty,dag-pets: emp ,
   waiting-reque: emp >
< 30 : dag | finish: false , pool: 3, current: 0, last: 2,
          pipe-pets: 0,exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 3,dag: 30,maxtask: 8,
   pipe-end: 0  ,
         pipe-cnt: 2,pipe[1]: 0 true false false false, pipe
   [2]: 0 true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1 , dag: 30 ,
          maxtask: 4,pipe[1]: 0 true false false false >
< 31 : dag | finish: false , pool: 3, current: 0, last: 4, pipe
   -pets: 0,
          exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 3,dag: 31,maxtask: 10,
   pipe-end: 0  ,
           pipe-cnt: 1,pipe[1]: 0 true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1 , dag: 31,
           maxtask: 400,pipe[1]: 0 true false false false >
< 2 : stage | petition: false , pool': 3,dag: 31,maxtask: 8,
   pipe-end: 0 ,
           pipe-cnt: 1,pipe[1]: 0 true false false false >
< 3 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1 , dag: 31,
            maxtask: 10,pipe[1]: 0 true false false false >
< 32 : dag | finish: false , pool: 3, current: 0, last: 3, pipe
   -pets: 0,
```

```
             exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 3,dag: 32,maxtask: 10,
   pipe-end: 0 ,
            pipe-cnt: 2,pipe[1]: 0 true false false false,pipe
   [1]: 0 true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1 , dag: 32 ,
            maxtask: 400,pipe[1]: 0 true false false false >
< 2 : stage | petition: false , pool': 3,dag: 32,maxtask: 8,
   pipe-end: 0 ,
            pipe-cnt: 1,pipe[1]: 0 true false false false >


< 4 : pool | weight: 5,min-share: 2,cnt-exe: 0,running-task: 0,
   wait: (41,40) ,
         executing: empty,finished: empty,dag-pets: emp ,
   waiting-reque: emp >
< 40 : dag | finish: false , pool: 4, current: 0, last: 3, pipe
   -pets: 0,
         exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 4,dag: 40,maxtask: 56,
   pipe-end: 0 ,pipe-cnt: 8,
           pipe[1]: 0 true false false false, pipe[2]: 0 true
   false false false,
           pipe[3]: 0 true false false false, pipe[4]: 0 true
   false false false,
           pipe[5]: 0 true false false false, pipe[6]: 0 true
   false false false,
           pipe[7]: 0 true false false false, pipe[8]: 0 true
   false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    3 , dag: 40 ,maxtask: 26,
           pipe[1]: 0 true false false false,pipe[2]: 0 true
   false false false,
           pipe[3]: 0 true false false false >
< 2 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    1 , dag: 40 ,maxtask: 4,
            pipe[1]: 0 true false false false >
< 41 : dag | finish: false , pool: 4, current: 0, last: 3, pipe
   -pets: 0,
            exe-cnt: 0,exe-asig: 0 >
```

```
< 0 : stage | petition: false , pool': 4,dag: 41,maxtask: 20,
   pipe-end: 0  ,pipe-cnt: 4,
          pipe[1]: 0 true false false false, pipe[2]: 0 true
   false false false,
          pipe[3]: 0 true false false false, pipe[4]: 0 true
   false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    5 , dag: 41 ,maxtask: 20,
          pipe[1]: 0 true false false false, pipe[2]: 0 true
   false false false,
          pipe[3]: 0 true false false false, pipe[4]: 0 true
   false false false,
          pipe[5]: 0 true false false false >
< 2 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    5 , dag: 41 ,maxtask: 10,
           pipe[1]: 0 true false false false, pipe[2]: 0 true
    false false false,
           pipe[3]: 0 true false false false, pipe[4]: 0 true
    false false false,
           pipe[5]: 0 true false false false >


< 13 : exec | core-cnt: 10, time-per-task: 10 >
< 999 : scheduler | pools-petitions: empty >
}
```

B.2. EJEM2

```
{
< 3 : pool | weight: 2,min-share: 6,cnt-exe: 0,running-task: 0,
   wait: (32,30) ,
     executing: empty,finished: empty,dag-pets: emp , waiting-
   reque: emp >
< 30 : dag | finish: false , pool: 3, current: 0, last: 2,
     pipe-pets: 0,exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 3,dag: 30,maxtask: 2,
   pipe-end: 0,
     pipe-cnt: 2,pipe[1]: 0 true false false false, pipe[2]: 0
    true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1 , dag: 30 ,
     maxtask: 2,pipe[1]: 0 true false false false >
```

```
< 32 : dag | finish: false , pool: 3, current: 0, last: 3, pipe
   -pets: 0,
      exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 3,dag: 32,maxtask: 2,
   pipe-end: 0,
      pipe-cnt: 2,pipe[1]: 0 true false false false,pipe[1]: 0
   true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 3 , pipe-cnt:
    1,
      dag: 32 ,maxtask: 2,pipe[1]: 0 true false false false >
< 2 : stage | petition: false , pool': 3,dag: 32,maxtask: 2,
   pipe-end: 0,
      pipe-cnt: 1,pipe[1]: 0 true false false false >


< 4 : pool | weight: 5,min-share: 2,cnt-exe: 0,running-task: 0,
   wait: (41,40) ,
     executing: empty,finished: empty,dag-pets: emp , waiting-
   reque: emp >
< 40 : dag | finish: false , pool: 4, current: 0, last: 3, pipe
   -pets: 0,
      exe-cnt: 0,exe-asig: 0 >
< 0 : stage | petition: false , pool': 4,dag: 40,maxtask: 2,
   pipe-end: 0  ,
     pipe-cnt: 2,pipe[1]: 0 true false false false, pipe[2]: 0
   true false false false >
< 1 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    3 , dag: 40 ,
     maxtask: 2,pipe[1]: 0 true false false false,pipe[2]: 0
   true false false false,
      pipe[3]: 0 true false false false >
< 2 : stage | petition: false,pipe-end: 0, pool': 4 , pipe-cnt:
    1 , dag: 40,
    maxtask: 2,pipe[1]: 0 true false false false >
< 13  : exec | core-cnt: 10,  time-per-task: 10 >
< 999 : scheduler | pools-petitions: empty >
}
```

Bibliografía

[1] V. Autores. Apache spark. Accedido en 07-18-2018 a `https://github.com/apache/spark`.

[2] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[3] Y.-F. Chen, C.-D. Hong, O. Lengál, S.-C. Mu, N. Sinha, and B.-Y. Wang. An executable sequential specification for spark aggregation. In A. El Abbadi and B. Garbinato, editors, *Networked Systems*, pages 421–438, Cham, 2017. Springer International Publishing.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[5] F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All about maude: A high-performance logical framework. *Lecture Notes in Computer Science*, 4350, 2007.

[6] F. Durán and J. Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *The Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012.

[7] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, and M. Sagiv. Verifying equivalence of spark programs. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, pages 282–300, Cham, 2017. Springer International Publishing.

[8] J. Laskowski. Schedulable pool · mastering apache spark. Accedido en 07-26-2018 a `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-taskscheduler-pool.html`.

[9] J.-C. Lin, M.-C. Lee, I. C. Yu, and E. B. Johnsen. Modeling and simulation of spark streaming. 2018.

[10] S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. *Electronic Notes in Theoretical Computer Science*, 238(3):207–225, 2009.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, Apr. 1992.

[12] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer Berlin Heidelberg, 1998.

[13] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.

[14] C. Rocha, J. Meseguer, and C. A. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Meth. Program.*, 86(1):269–297, 2017.

[15] T. White. *Hadoop: The definitive guide.* .O'Reilly Media, Inc.", 2012.

[16] P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The abs tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer*, 14(5):567–588, Oct 2012.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. pages 2–2, 2012.