ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Mención en Sistemas de Información

PARALELIZACIÓN DE ALGORITMOS DE MINERÍA DE TEXTOS CON HADOOP

PARALLELIZATION OF TEXT MINING ALGORITHMS USING HADOOP

Realizado por
Elena Carrasco Barrios
Tutorizado por
Ismael Navas Delgado
Co-tutorizado por
José Francisco Aldana Montes
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Octubre de 2014

Fecha defensa:
El Secretario del Tribunal

Resumen: Este Trabajo Fin de Grado (TFG) tiene como objetivos paralelizar algoritmos de minería de textos para poder permitir su ejecución con una gran cantidad de textos en el menor tiempo posible y con usuarios concurrentes, y la creación de un modelo de datos RDF con las anotaciones generadas por el algoritmo en los documentos. La paralelización se ha realizado siguiendo la filosofía *MapReduce*. En la fase del *mapper* se realiza la ejecución del algoritmo de minería de textos sobre el texto de entrada y se genera el modelo RDF asociado a ese texto. La fase del *reducer* se encarga de unir todos los modelos RDF que hagan referencia a textos de un documento en un único modelo global. El resultado de la ejecución de este programa son pares <nombre del documento, modelo RDF>. Para cumplir con el segundo objetivo se ha desarrollado otra aplicación que une todos los modelos generados por el programa anterior en un solo modelo. El desarrollo del sistema se ha realizado usando Java SE y las tecnologías Apache Hadoop, Gate y Apache Jena. En este trabajo se expondrán un sistema capaz de paralelizar algoritmos de minería de textos desarrollados en GATE y crear el modelo RDF correspondiente a las anotaciones generadas a partir de los textos, las conclusiones alcanzadas a raíz de este trabajo y algunas propuestas de trabajos futuros.

Palabras claves: Paralelización, Algoritmo de minería de textos, MapReduce, Algoritmo de reconocimiento de nombres de entidades, modelo RDF

Abstract: This Degree Thesis (TFG) aims to parallelization algorithms of text mining in order to allow them to run with a lot of texts as quickly as possible and with concurrent users, and the creation of a RDF data model with the annotations generated by the algorithms in the documents. The parallelization has been made following the filosophy of the MapReduce. In the phase of the mapper, the text mining algorithm is run on the input text and the RDF model associated at this text is generated. The phase of the reducer aims to put together all the RDF models that refer to a document in a single global model. Once run this program, pairs are obtained: <document name, model RDF>. To achieve the second objective, another application, that

unites all models generated by the above program in a single model, has been developed. The development of the system has been done using Java SE and Apache Hadoop, Gate and Apache Jena technologies. In this work will be exposed a system able to parallelize text mining algorithms developed in Gate and create the RDF model corresponding to the generated annotations from texts, the conclusions reached as a result of this work and some proposals for future work.

Keywords: Parallelization, Text mining algorithm, MapReduce, Named-entity recognition algorithm, RDF model

Índice

1. **Introducción** 1
   - 1.1 Motivación y objetivos 2
   - 1.2 Metodología 4
   - 1.3 Estructura de la memoria 5
2. **Estado del Arte** 7
   - 2.1 Minería de textos 7
   - 2.2 Procesamiento del Lenguaje Natural y NER 10
   - 2.3 Big Data y Paralelización de algoritmos 13
   - 2.4 Cloud Computing 19
   - 2.5 RDF y Web Semántica 20
3. **Tecnologías y herramientas utilizadas** 23
4. **Desarrollo** 25
   - 4.1 Elección y configuración de la arquitectura del sistema 25
   - 4.2 Análisis, diseño y construcción del sistema 29
     - i. Aspectos básicos 29
     - ii. Incrementos y etapas de la fase 30
     - iii. Implementación y funcionamiento del sistema 32
   - 4.3 Pruebas del sistema y análisis de los resultados 51
     - i. Pruebas del sistema 51
     - ii. Análisis de los resultados 52
   - 4.4 Consideraciones 55
5. **Conclusiones y trabajos futuros** 57
   - 5.1 Conclusiones 57
   - 5.2 Trabajos futuros 58
6. **Bibliografía** 61
7. **Anexos Técnicos** 65
   - 7.1 Manual de usuario 65
   - 7.2 Resultados del sistema usando ANNIE 66
   - 7.3 Ficheros utilizados para el análisis 67

1. Introducción

Este Trabajo Fin de Grado pertenece a la línea *Gestión y Análisis de Datos*. En dicha línea se busca, por un lado, la creación de aplicaciones capaces de gestionar y analizar datos, y por otro lado, el desarrollo de programas que sirvan para poder usar las anteriores aplicaciones con un gran volumen de datos, convirtiéndose así en técnicas de *Big Data*.

El TFG presentado a continuación se englobaría en la segunda parte, ya que su finalidad es paralelizar algoritmos de minería de textos para permitir su ejecución con una gran cantidad de textos en el menor tiempo posible y con usuarios concurrentes. Dichos algoritmos han sido desarrollados por el proyecto de investigación que está asociado a la línea general, Bioledge (*BIO knowLEDGe Extractor and Modeller for Protein Production*) [1], por lo que el ámbito de este Trabajo es, en esencia, biológico, pero la implementación del algoritmo de paralelización permite que pueda ser utilizado por otros algoritmos de minería de textos de cualquier ámbito que estén desarrollados bajo una arquitectura determinada. La paralelización consiste en una serie de fases, que se muestran en el diagrama de bloques representado en la Figura 1, en la que se simula una ejecución válida del programa.

Primero se distribuye el trabajo a realizar en las máquinas (nodos) seleccionadas siguiendo el criterio de la localización de los documentos a procesar y se distribuye los documentos en dichas máquinas. A continuación, cada documento (*input*) se trocea en varios *records*, es decir, fragmentos del documento (fase *splitting*). Tras ello, en la fase *mapping* se ejecuta el algoritmo de minería de textos a cada *record*, y se genera un modelo RDF a partir de las anotaciones generadas por dicho algoritmo. Esas anotaciones almacenan información en el propio documento de la palabra encontrada y la entidad a la que se refiere, entre otras. Luego se reúnen las que tengan la misma clave (en este caso el nombre del documento) de cada ejecución (*shuffling*) y en la fase del *reducing* se unen todos los modelos de un documento concreto.

El resultado final de la paralelización son pares <*documento, modelo*>. Por ello se ha desarrollado un programa para unir todos esos modelos en un único modelo general que los centralice, sin perder información.

La tecnología utilizada para la programación de la aplicación es Java Standard, siguiendo el modelo MapReduce. Para lo cual, se ha hecho uso del framework Hadoop para realizar la paralelización. Además de estas tecnologías, también se han usado las librerías de GATE, ya que los algoritmos de minería de textos han sido desarrollados bajo esta arquitectura, y Apache Jena para generar los modelos RDF de las anotaciones generadas por dichos algoritmos.

En el diseño de la aplicación se ha utilizado la metodología UML. La programación se ha desarrollado a través del IDE Eclipse Juno y la instalación de Hadoop se ha realizado en máquinas con SO Linux Red Hat.

1.1 Motivación y objetivos

El objetivo de este Trabajo Fin de Grado es paralelizar algoritmos de minería de textos. Como veremos en el capítulo 2, la minería de textos está adquiriendo cada vez más importancia para las empresas al permitir la extracción y el análisis de la información contenida de forma implícita en grandes volúmenes de textos a través del reconocimiento de nombres de entidades (NER), como pueden ser la marca de un coche o el nombre de una célula. Esto ha supuesto una gran ventaja, entre otros, en el ámbito científico, en el que se generan una gran cantidad de artículos al año, lo cual provoca que sea prácticamente imposible que de forma manual se pueda extraer la información de todos ellos. El problema de estos algoritmos es que son demasiado lentos, especialmente si tienen que procesar de manera secuencial miles de documentos, como en el caso de los artículos científicos. Por ello, a partir de este inconveniente surge el presente TFG, en el que se da la posibilidad de paralelizar la ejecución de algoritmos de tipo NER de forma eficiente.

A título personal, este Trabajo ha servido para el conocimiento y aprendizaje de la minería de textos y sus fases, así como los algoritmos NER y, mayormente, la paralelización de algoritmos, además de las tecnologías que se utilizan actualmente en dichos ámbitos. Aspectos que apenas se mencionan en el Grado pero que, gracias al TFG, han podido ser abarcados.

Con respecto a los objetivos, el objetivo general de este Trabajo Fin de Grado es tratar la paralelización de algoritmos de minería de textos que se desarrollen sobre la plataforma GATE, usando para ello la metodología de programación *MapReduce* de Hadoop.

Concretando, los objetivos principales son permitir el uso de dichos algoritmos en entornos con varios usuarios concurrentes y mejorar el rendimiento, especialmente en lo que a tiempo total de cómputo se refiere, de los algoritmos de minería de textos al paralelizarlos. Las anotaciones generadas por dichos algoritmos se almacenarán en un modelo de datos tipo RDF para que, combinándolas con otras tecnologías en otros posibles TFG, sean usados para ciertas aplicaciones en la Web semántica.

Todos estos objetivos se subdividen en los siguientes:

- Creación de un programa *MapReduce* que distribuya y ejecute los algoritmos en varios nodos.
- Creación de modelos RDF con las anotaciones generadas por las aplicaciones de minería de textos durante el proceso *MapReduce*, y su posterior unión en un modelo final para globalizarlo.
- Análisis y comparación del tiempo total de procesamiento de los algoritmos de minería de textos al ejecutarlos con y sin paralelización.
- Análisis de la pérdida de información debido a que, en teoría, al trocear los documentos de entrada se pierde el contexto.

1.2 Metodología

La metodología usada para el desarrollo de la aplicación ha sido, en esencia, el modelo Iterativo e Incremental [2]. Las fases generales del desarrollo fueron las siguientes:

1. Estudio y configuración de la arquitectura del sistema.
2. Análisis de los requisitos, diseño e implementación de la aplicación.
3. Pruebas de sistemas y análisis de los resultados.

En la primera fase se ha realizado el estudio del estado del arte, las diferentes tecnologías utilizadas, los algoritmos a paralelizar, y la instalación y configuración de la arquitectura del sistema. En la segunda fase tuvo lugar las etapas de análisis de los requisitos, diseño e implementación del algoritmo para la paralelización y el de la unión de todos los modelos generados por el programa anterior. La última fase se corresponde con la realización de las pruebas necesarias al sistema para testearlo y verificar su correcto funcionamiento, junto con el análisis de los resultados obtenidos, que fueron mencionados en el apartado 1.1.

Para la segunda fase se realizaron una serie de incrementos, que se dividían en tres etapas. Estas etapas se explican a continuación:

- Estudio y análisis de los requisitos de dicho incremento: Se definían los requisitos que se iban a implementar en el incremento en cuestión, lo que requería que se hiciera un pequeño estudio del estado del arte en el que estaba englobado.
- Diseño e implementación de los requisitos: Se diseñaban y construían los códigos necesarios para el desarrollo de los requisitos escogidos.
- Pruebas e integración con el resto del sistema: Primero se realizaban pruebas unitarias al código implementado. Una vez pasadas de manera satisfactoria se integraba con el resto del sistema desarrollado y se realizaban nuevas pruebas para comprobar el correcto funcionamiento al estar integrado.

En el capítulo 4 se contará con más detalle los incrementos realizados durante el desarrollo de la aplicación.

1.3 Estructura de la memoria

Para finalizar este capítulo se explica la estructura de la memoria, que se divide en capítulos y apartados. En los apartados anteriores se explica la motivación para la elección y realización de este TFG, junto con los objetivos, es decir, el sistema que se va a construir y los análisis que se van a realizar. En el apartado 1.2 se describe la metodología seguida durante el desarrollo del proyecto.

En el capítulo 2 se detalla el estado del arte y los fundamentos teóricos esenciales para el desarrollo del sistema y en el capítulo 3 se listan las tecnologías y herramientas utilizadas para la construcción de las aplicaciones. El capítulo 4 detalla el trabajo realizado durante las fases del desarrollo, junto con la explicación de las partes del código más relevantes. En el capítulo 5 se exponen las conclusiones alcanzadas a partir del desarrollo del sistema y el análisis de los resultados, así como algunas ideas de futuros TFGs relacionados. Para acabar, en el capítulo 6 se lista la bibliografía usada para la realización del TFG y en el capítulo 7 se incluyen todos los materiales que no han tenido cabida en el cuerpo de la memoria, como el manual de usuario o los resultados de ejecuciones del sistema.

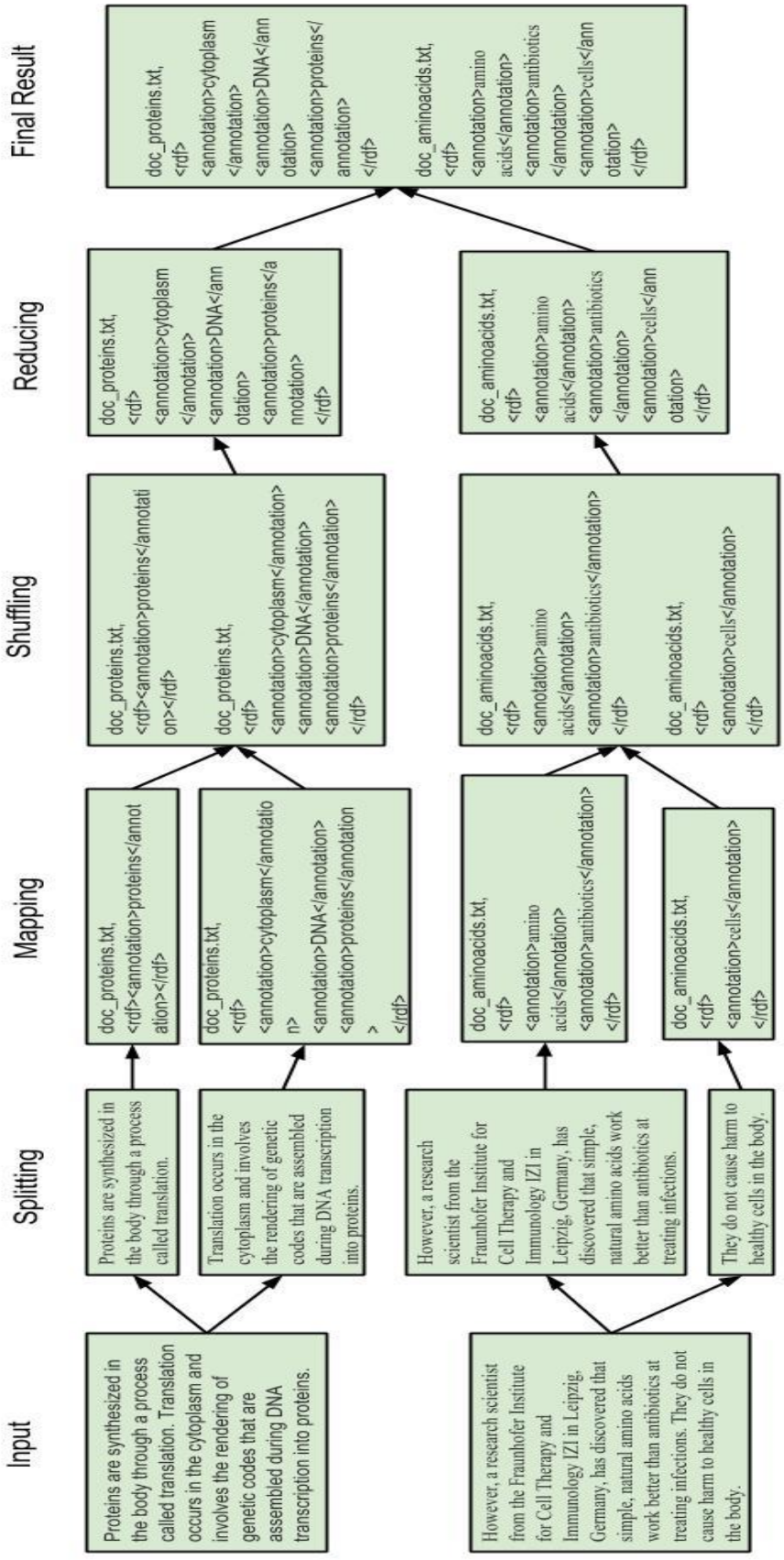


Figura 1. Fases del algoritmo MapReduce desarrollado

2. Estado del Arte

En esta sección se describe el estado del arte en el que se comprenden las aplicaciones realizadas. Con este apartado se pretende situar al lector en el contexto de este trabajo, resaltando la importancia de paralelizar algoritmos, mayormente los de minería de textos, y explicando los conceptos claves.

2.1 Minería de textos

En la actualidad hay una gran cantidad de textos digitalizados y casi la totalidad de ellos se encuentran en el mayor almacén de información a día de hoy, Internet. Esto nos da una serie de ventajas, como pueden ser que los documentos sean actualizados y no se deterioren con el tiempo, la posibilidad de revisión por múltiples usuarios, la facilidad en el acceso, la compartición mediante redes sociales o correo electrónico, y el análisis de los textos, entre otros.

El análisis de los textos es vital para poder descubrir información y nuevo *conocimiento* a partir de los datos no estructurados que proporcionan un conjunto de textos. De manera más detallada, los algoritmos de minería de textos [3] extraen, de forma automatizada y rápida, información no trivial o desconocida (hechos, relaciones) que se halla de forma implícita en los textos a través de la identificación de patrones y correlaciones de los términos que contienen. Por este motivo, se utilizan bastante estas técnicas en ámbitos de investigación y de negocios, usándose ese conocimiento, por ejemplo, para el apoyo a la toma de decisiones estratégicas. Otro ejemplo puede ser la identificación de entidades biológicas (células, virus, proteínas específicas) que pueden servir para la clasificación de artículos científicos.

Este tipo de algoritmos está cobrando cada vez más importancia, ya que facilitan el análisis y procesamiento de cualquier tipo de información de y para cualquier ámbito, lo cual se demuestra en las predicciones que se han realizado hace unos años. En dichas predicciones se auguraba que tendrían un

gran impacto comercial ya que, además de que la mayor parte de la información (sobre el 80%) es no estructurada y se encuentra almacenada en textos [4], según un informe publicado por McKinsey Global Institute [5], la minería de textos y datos tendría el potencial de generar 250 mil millones de euros anuales en la economía europea si a los investigadores se les permitiera hacer un uso completo de ellas. Esta predicción tiene bastante fundamento, no sólo porque gran parte de la información esté en documentos, sino también por el creciente y cada vez más consolidado mercado de las herramientas de minería de textos. Entre estas herramientas se destacan:

- **Autonomy** [6], un software que ofrece herramientas para realizar minería de textos, clustering y categorización a través de la búsqueda y el procesamiento de texto a partir de datos estructurados e información no estructurada.
- **SAS Text Analytics** [7], otro software que se utiliza para descubrir patrones y tendencias en los textos, muy usado tanto en la minería de textos como en el análisis de sentimientos o la extracción de contenidos.
- **AeroText** [8], herramienta o biblioteca con aplicaciones de minería de textos para realizar análisis de contenido para textos en diversos idiomas.

Como vemos, cada vez hay un mayor auge y una alta necesidad en usar estos algoritmos que, generalmente, realizan una serie de tareas (*tasks*) importantes. Algunas de ellas suelen requerir ejecutarse en un cierto orden ya que necesitan la salida de otra. Algunas de las tareas más importantes de la minería de textos [3] se presentan a continuación:

- Recuperación de Información. Recolección de los textos para el análisis. Estos ficheros se obtienen desde la Web, sistemas de ficheros o bases de datos. Generalmente a los algoritmos de minería de textos se les pasa como argumento un *corpus*, es decir, un conjunto estructurado de textos.

- Procesamiento del lenguaje natural (NLP). El procesamiento del lenguaje natural, al igual que el reconocimiento de nombres de entidades, se verá con más detenimiento en el siguiente apartado. Aun así, se destaca que el objetivo de los algoritmos NLP es procesar los textos por su sentido, no como un mero archivo binario.

- Reconocimiento de nombres de entidades (NER). Se identifican nombres de entidades en el texto y se clasifican bajo una categoría predefinida, como pueden ser personas, organizaciones, lugares, dinero, fechas. Algunos algoritmos de este tipo usan desambiguación del sentido de las palabras para poder clasificarlas con la mayor probabilidad de éxito. Por ejemplo, la palabra *Apple* puede hacer referencia a una compañía o a una fruta, así que dependiendo del contexto del documento se clasificará de una forma u otra.

- Reconocimiento de patrones de entidades identificadas. A partir de expresiones regulares y patrones permite reconocer entidades estructuradas, como cuentas bancarias, direcciones de correo electrónico, números de teléfono, etc.

- Correferencia. En esta tarea se identifican frases nominales que se refieren a un mismo objeto. Por ejemplo, *"Amino acids work better than antibiotics at treating infections. They do not cause harm to healthy cells in the body".*

- Extracción de relaciones. Identificación de las relaciones entre entidades y otros elementos de los textos.

- Análisis de sentimientos. Trata de analizar y transformar en datos los sentimientos y las emociones humanas a partir de textos. Se le conoce también como minería de opiniones [9].

- Análisis cuantitativo del texto. Engloba las técnicas que tratan de buscar el modelo estilístico o el significado de un texto a partir de la extracción de relaciones semánticas o gramaticales entre palabras. Estas técnicas son muy usadas en las ciencias sociales, especialmente en el ámbito psicológico [10].

2.2 Procesamiento del Lenguaje Natural y NER

En el apartado anterior se ha mencionado que el procesamiento del lenguaje natural y el reconocimiento de nombres de entidades son tareas con gran peso en la minería de textos. El procesamiento del lenguaje natural nació como una rama de la Inteligencia Artificial (IA) con el objetivo de estudiar la generación y comprensión automática del lenguaje natural [11]. Más adelante, se expandieron sus tareas hasta cubrir los tres objetivos actuales:

- Interfaces en lenguaje natural. Consiste, básicamente, en poder comunicarse verbalmente con las máquinas. No se refiere a un simple reconocimiento de voz, sino dar órdenes hablando de manera cotidiana y escuchando las respuestas.

- Procesamiento de textos. Se refiere tanto a la recuperación de información como a la extracción de datos significativos de los textos. En este aspecto es donde se enfoca la minería de textos.

- Traducción automática (TA). Es la traducción de textos por ordenador. Consiste en pasar de un lenguaje fuente a otro lenguaje, a veces pasando por una representación intermedia. El principal problema de los algoritmos de TA es la diferencia entre lenguajes, que pasa desde el orden de las palabras hasta el uso de expresiones propias de un determinado idioma.

Independientemente de los objetivos, los algoritmos NPL se estructuran en las siguientes fases:

1. Análisis morfológico: El análisis de las palabras para extraer raíces, unidades léxicas compuestas, entre otros.
2. Análisis sintáctico: El análisis de la estructura sintáctica de la frase usando la gramática de la lengua.
3. Análisis semántico: Obtención del significado de la frase y resolución de ambigüedades.
4. Análisis pragmático: El análisis del significado de las palabras según el contexto.
5. Planificación de la frase: Se planifica la estructura de la frase para expresar el significado adecuado.
6. Generación de la frase: La creación de la frase según la estructura que se ha especificado en la fase anterior.

Estas fases se pueden descomponer hasta englobar frases de un texto. Uno de los objetivos del procesamiento del lenguaje natural es el procesamiento de texto y en él se usa una de las técnicas de la minería de textos, el reconocimiento de nombres de entidades.

El reconocimiento de nombres de entidades (NER) es especialmente útil para procesar textos, ya que su objetivo principal es clasificar nombres en unas categorías que se hayan definido antes, como nombres de personas, organizaciones, localizaciones, cantidades, formatos de tiempo, entre otros.

Los algoritmos de minería de textos, y en concreto los de tipo NER, funcionan de la siguiente manera: primero dividen el texto en párrafos y sentencias (y las anotan como tal), luego dividen cada oración en *tokens*. Una vez se han generado los *tokens*, se etiqueta cada uno o un conjunto de ellos bajo la categoría que corresponda.

Por ejemplo, suponiendo que se tiene la categoría "*Fruit*" con todas las frutas especificadas en ella y la frase "*I eat an apple*", el algoritmo generaría una anotación como la siguiente:

```
I eat an [annotation: Fruit]apple[/annotation]
```

El algoritmo comprobaría si algún elemento de las categorías predefinidas se encuentra en la frase, y, en el caso de que sea así, la etiqueta con la categoría correspondiente.

Como vemos, el funcionamiento de los algoritmos de tipo NER es bastante fácil de comprender, al igual que el problema que deben afrontar. Siguiendo con el ejemplo, en el caso en el que existiera la categoría "*Organization*" junto a la anterior, ¿cómo sabe el algoritmo a qué categoría pertenece? Para solucionarlo hay muchas aproximaciones. Los algoritmos más sencillos etiquetan el nombre con la primera categoría en la que se puedan encuadrar. Los más sofisticados hacen uso de modelos probabilísticos o de la IA para solucionar la desambiguación del significado de las palabras [12]. Para no entrar en muchos detalles, se mencionará que se usan un conjunto de datos para alimentar esos modelos y, en el caso de que se tenga en cuenta el contexto de la frase o del texto, se incluyen también las relaciones entre ciertas palabras. De este modo, se ponderan los posibles significados que tiene la palabra en cuestión y se categoriza según los resultados.

En el ejemplo anterior, si tenemos un modelo en el que se relacione verbos con las palabras podemos deducir que "*apple*" se refiere a una comida, ya que el verbo "*eat*" no se utiliza cuando se habla de empresas. Por tanto, la categoría "*Fruit*" tendrá más ponderación que la categoría "*Organization*", ya que las frutas se pueden comer. Así pues, la palabra "*apple*" se anotaría con dicha categoría. Se podría llegar a la misma conclusión si tuviésemos corpus de gran tamaño con el mismo o parecido contexto, lo que haría que la categoría "*Fruit*" tuviera de base una ponderación superior.

Los algoritmos NER están proliferando en ámbitos relacionados con la investigación, especialmente en sectores biomédicos. Este interés tiene su fundamento en la gran cantidad de textos y artículos científicos que existen sobre el tema y en su objetivo principal, poder anotar artículos y textos para la extracción de información a través de la identificación de entidades o para la clasificación de documentos. Algunos algoritmos que se han desarrollado en este ámbito son **Genetag** [13], un algoritmo basado en NER que etiqueta

genes y proteínas, y algoritmos NER que etiquetan entidades relacionadas con enfermedades [14].

Para el desarrollo de algoritmos de minería de textos existen muchas herramientas y arquitecturas de desarrollo. Una de las herramientas más extendidas es la arquitectura GATE (*General Architecture for Text Engineering*) [15]. GATE es una infraestructura de código libre (*open source software*) usada para desarrollar y desplegar componentes software que procesan textos en lenguaje humano, es decir, es un entorno de desarrollo para algoritmos de procesamiento del lenguaje natural y minería de textos.

GATE proporciona un sistema de extracción de textos, que utiliza NER, llamado ANNIE [16]. Este sistema es usado para el desarrollo del algoritmo de paralelización, por lo que se explicará con más detalle en el capítulo 4.

2.3 Big Data y Paralelización de algoritmos

En el apartado anterior se ha mencionado que los algoritmos NER son muy utilizados en el ámbito científico principalmente para la clasificación de textos y la extracción de información. Las preguntas que cabrían hacerse es de cuántos documentos estamos hablando y si es posible extraer información de todos ellos.

A partir del estudio de Bo-Christer Björk [17] podemos responder a las preguntas anteriores. Al año se publican muchos artículos científicos (alrededor de 1.5 millones), por lo que el volumen de datos es inmanejable, resultando casi imposible que un humano sea capaz de extraer conocimiento a partir de ese volumen.

Los algoritmos de minería de textos automatizan ese proceso. Aun así es difícil procesar una cantidad tan elevada de documentos, tanto por el tiempo para procesarlos todos como por los recursos para realizarlo. El tratamiento de grandes cantidades de información es un problema que trata de solucionar el Big Data [18]. El término *Big Data* se usa para hacer referencia a las

aplicaciones o a los algoritmos que son capaces de gestionar un elevado conjunto de datos que un software habitual no podría manipular, como terabytes o petabytes de datos en un único *data set*, es decir, un único conjunto de datos.

Este es el motivo por el que las tecnologías del Big Data son muy usadas, ya que en bastantes sectores se requiere hacer uso de muchos datos, como en el sector empresarial con las técnicas de *Business Inteligence*, entre otros. Grandes empresas como son Microsoft, SAP y Oracle proporcionan herramientas de Big Data para la organización y la manipulación de grandes volúmenes de datos a una velocidad de procesamiento bastante razonable.

Uno de los sistemas que están cobrando fuerza en este ámbito es Hadoop [19], de Apache Software Foundation. Hadoop es un *framework* de software que permite el procesamiento distribuido de grandes conjuntos de datos a través de clústeres usando el modelo de programación *MapReduce*. En Hadoop, un clúster tiene un nodo *master* y uno o varios nodos *slaves* [20]. El nodo *master* almacena los metadatos de los nodos *slaves* y tiene o puede tener los siguientes servicios:

- *Namenode*: Gestiona el espacio de nombres del sistema de ficheros. Mantiene la estructura del *filesystem* y los metadatos. Además, conoce los datanodes en los que se almacenan los bloques de un determinado fichero.
- *Secondary namenode*: Copia del namenode principal. Se activa cuando el principal cae o falla por algún motivo.
- *Datanode*: Almacena y recupera bloques cuando lo pide el *namenode* o los clientes (cuando se accede al sistema de ficheros). También informan al *namenode* de los bloques que almacenan.
- *Jobtracker*: Coordina y divide en tareas el *job* (una aplicación de Java, por ejemplo) que se está ejecutando.
- *Tasktracker*: Ejecuta las tareas en las que se ha dividido el *job*.

Los nodos *slaves* almacenan los datos y ejecutan los trabajos de procesamiento. Sólo tienen dos servicios:

- *Datanode*
- *Tasktracker*

En la Figura 2 se observa toda la estructura explicada anteriormente. En el capítulo 4, concretamente en el primer apartado, se profundizará en la arquitectura de Hadoop y su funcionamiento. Sin embargo, destacar que Hadoop tiene su propio sistema de ficheros llamado HDFS (*Hadoop Distributed File System*) [21] que está diseñado específicamente para almacenar ficheros de gran tamaño de forma distribuida. Esos ficheros se dividen en bloques (de 64 MB cada uno) y se distribuyen por los nodos con el rol *datanode* del clúster. HDFS incluye replicación de bloques para facilitar la recuperación en caso de fallo de un nodo.

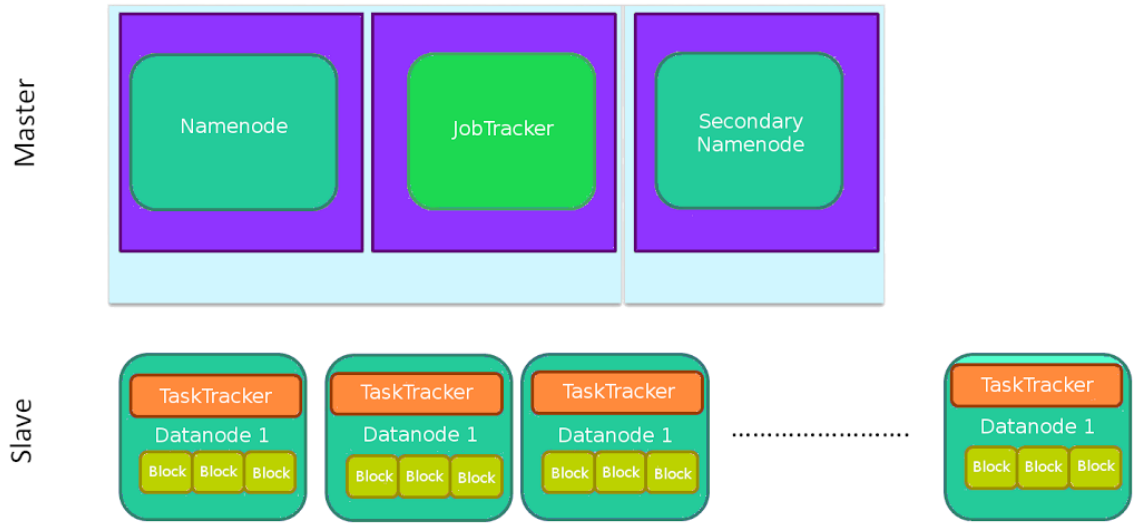


Figura 2. Estructura de los nodos en Hadoop

Hadoop es una solución ideal para el problema que habíamos mencionado antes. Hay una gran cantidad de textos y artículos científicos, cuyo procesamiento es imposible debido al tamaño del corpus de entrada a los algoritmos de minería de textos. Por ello, la solución propuesta por la filosofía de Hadoop es básicamente la paralelización de esos algoritmos,

distribuyéndolos entre los diferentes nodos *slaves* según la cercanía a la que están los ficheros a procesar y ejecutándolos. Los algoritmos de minería de textos, o de cualquier otro tipo, deben estar implementados siguiendo la filosofía *MapReduce*, definiendo al menos las dos clases principales, *Mapper* y *Reducer*.

El *Mapper* es la función encargada de mapear las claves. A partir de pares <clave, valor> devuelve una lista de pares <clave, valor>.

$$Map\ (key1, value1) \rightarrow list\ (key2, value2)$$

El *Reducer* es la función de reducción. A partir de una clave y una colección de valores genera una lista de valores.

$$Reduce\ (key2, list(value2)) \rightarrow list(value3)$$

El funcionamiento del MapReduce se entiende de manera más intuitiva en la Figura 3, en el que se muestra la paralelización de un algoritmo para contar palabras. El texto de entrada se divide en pequeños fragmentos llamados *records*, que son pequeños trozos de un texto, y cada uno es el *input* de un map. En la fase del *mapper* se crea un par <clave, valor> cada vez que encuentra una palabra, siendo la clave la propia palabra y el valor 1. En esta fase el objetivo es simplemente anotar cada palabra que aparece, sin sumar el número de veces que aparece la misma. Eso se realiza en la fase del *reducer*, al cual en cada ejecución le llega una determinada palabra y una lista de valores (1) por cada vez que lo han encontrado los *mappers*.

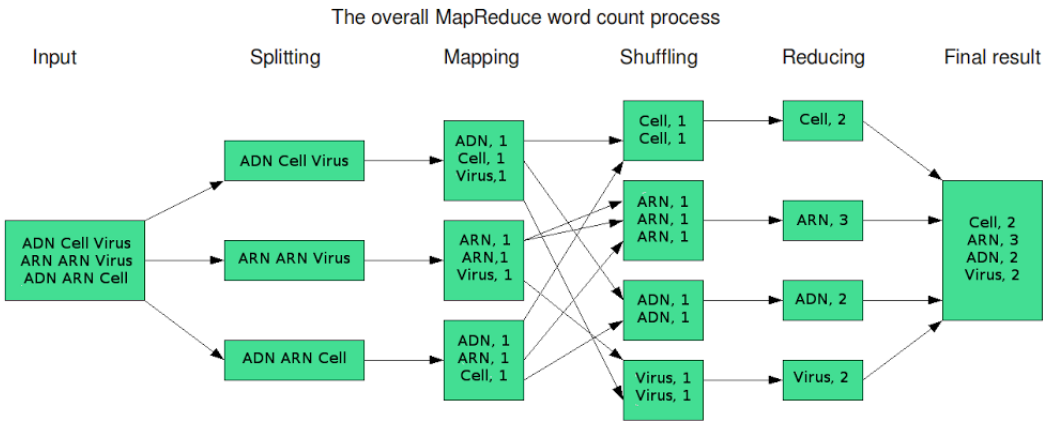


Figura 3. Ejecución algoritmo MapReduce

Como podemos suponer, la implementación es mucho más sencilla que haciendo todo el proceso de una vez, y mucho más rápida a la hora de la ejecución. El *mapper* se encarga de almacenar cada palabra que aparece y el *reducer* recorre la lista de valores que se le pasa como argumento y los suma uno a uno, escribiendo el par <palabra, suma total>.

Una vez visto cómo funciona una aplicación basada en *MapReduce*, a continuación se explicará cómo es el proceso que sigue Hadoop para llevar a cabo la paralelización de esas aplicaciones. De manera general, el proceso de paralelización es el siguiente:

- La aplicación, implementada siguiendo la filosofía *MapReduce*, crea un *job* que será gestionado por un nodo *master* (*jobtracker*).
- El *jobtracker* iniciará los *tasktrackers* convenientes, según su proximidad a los bloques. Como se dijo antes, los ficheros de entrada se dividen en bloques.
- Empieza la primera fase, el *mapping*. Cada *tasktracker* ejecuta el mapper y el *combiner* si está implementado. El *combiner* se utiliza para disminuir la salida generada de los *maps*, así que suele tener la misma implementación que los *reducers*.

- Cuando todos los tasktrackers terminan comienza la siguiente fase, el *reducing*. En esta fase se inician los *tasktrackers* necesarios, que ejecutarán el *reducer* usando como entrada la salida de los *mappers*. Cuando terminan su ejecución mueven los resultados de la ejecución al sistema de ficheros de Hadoop, es decir, al HDFS.

En la figura anterior se puede observar que los bloques de entrada se dividen en *splits*, y esos son la verdadera entrada de los *tasktrackers*. Como ya se verá en el capítulo del Desarrollo, por defecto en las aplicaciones que se desarrollan en Hadoop se divide la entrada en trozos más pequeños según una cierta lógica, como por ejemplo un salto de línea. De esta forma las ejecuciones de los *maps* son mucho más rápidas y evita las pérdidas de tiempo y de recursos en el caso de que falle uno de ellos y se tenga que volver a reiniciar. Esta forma de recuperarse del error es posible gracias a la separación del *jobtracker* y del *tasktracker*, de tal forma que si un *tasktracker* falla, el *jobtracker* se encargaría de eliminar dicho *tasktracker* y crear otro en un nodo disponible del clúster para que ejecute el trabajo que estaba haciendo el primero.

Las ventajas de la paralelización se han ido mencionando a lo largo de la explicación de este apartado y de los anteriores. Éstas se resumen en la mayor velocidad de procesamiento frente a la ejecución secuencial y, desde el punto de vista de la autora de este TFG, el mecanismo de recuperación ante posibles errores. Este mecanismo se expande a varios niveles, por ejemplo a nivel de los *tasktrackers* en el que, como se ha mencionado anteriormente, se evita que falle la ejecución entera del programa reiniciando sólo la tarea que estaba realizando dicha parte. Para acabar este apartado, destacar la gestión interna que implementa Hadoop para evitar la pérdida de información: la replicación de bloques entre *datanodes*. Gracias a la replicación de los bloques se evita que se pierdan datos en el caso de que un *datanode* caiga, y por otro lado facilita que el *jobtracker* pueda seleccionar un nodo tipo *tasktracker* que no tenga mucha sobrecarga de trabajo, lo que mejoraría la concurrencia de varios usuarios.

2.4 Cloud Computing

Las ventajas que proporciona el *Big Data* junto con la paralelización de los algoritmos han quedado patentes en el anterior apartado. Según el análisis realizado por Ryan Merriman [22], el tiempo de ejecución de un algoritmo de procesamiento del lenguaje natural paralelizado usando Hadoop en un clúster de 20 máquinas está comprendido en alrededor de una octava parte de la ejecución del mismo algoritmo en modo *standalone*, es decir, sin paralelización. Este hecho ha llevado a las empresas a hacerse con clústeres de un tamaño considerable, ya que con clústeres de dos nodos no se obtienen los resultados anteriores, lo que puede generar dificultades.

Entre esas dificultades, las más destacables son el precio para adquirirlos y configurarlos así como los gastos que se generan, incluyendo el desaprovechamiento de los recursos y las limitaciones en el caso de querer ampliar el clúster de forma rápida.

El *cloud computing* [23] es un paradigma que surgió para dar solución a estas dificultades proporcionando servicios de computación a través de Internet. Usando este paradigma, las empresas podrían tener clústeres en la nube, lo que supone una gran ventaja a la hora de monitorizar el clúster gracias a las apps destinadas a ello. Otra ventaja a destacar es la posibilidad de pagar por el uso o los recursos que realmente se estén utilizando por lo que, además de abaratar costes, se aumenta la optimización de los recursos. Por otro lado, la mayoría de las empresas dedicadas a ofrecer estos servicios permiten la escalabilidad en el caso de que se requiera, y lo contrario también, junto con la gestión de las actualizaciones automáticas para que no afecten negativamente a los recursos de TI.

Gracias a la expansión de la Web y a los beneficios que aporta, el *cloud computing* está en alza, sobre todo en los sectores en los que se requiere *Big Data*. Los algoritmos de *Big Data* requieren procesar una gran cantidad de documentos, por lo que el *cloud computing*, tal y como se ha visto, es la solución más adecuada en términos de coste y escalabilidad [24]. Este énfasis por el *cloud computing* se refleja en empresas internacionalmente reconocidas

que están ofreciendo servicios de cloud. Ejemplos de estos servicios son IBM Cloud [25] y Amazon Web Services (AWS) [26].

2.5 RDF y Web Semántica

Las anotaciones que se generan por los algoritmos desarrollados en GATE están incrustadas en el documento procesado, por consiguiente, a menos que se transforme por ejemplo en XML, no se pueden observar en el fichero. Para poder obtener esas anotaciones se suelen utilizar modelos de datos enfocados a los metadatos, como el RDF, para almacenarlos y, también, para posibles usos en la Web Semántica.

El RDF (*Resource Description Framework*) [27] es un modelo estándar basado en XML para el intercambio de datos en la Web basado en tripletas. Estas tripletas tienen la siguiente estructura:

```
Sujeto – Predicado – Objeto
```

- Sujeto: El recurso. Cualquier objeto web identificado mediante una URI, generalmente hace referencia a un objeto de un repositorio de datos libre, como DBpedia [28].
- Predicado: También conocido como propiedad. Son rasgos o aspectos del recurso. Expresa la relación entre el sujeto y el objeto.
- Objeto: El valor concreto de ese recurso para el predicado dado. Pueden ser literales o recursos (sujetos).

Este modelo de datos tiene una estructura bastante sencilla, de ahí que sea extensamente utilizado por ser tan flexible para la estructuración de la información [29]. Los modelos RDF, como se muestra en la Figura 4, son grafos acíclicos dirigidos que representan la estructura anterior. Como se puede observar, estos modelos representan de forma bastante acertada y sencilla el conocimiento, de ahí que sea muy usado en redes semánticas y ontologías.

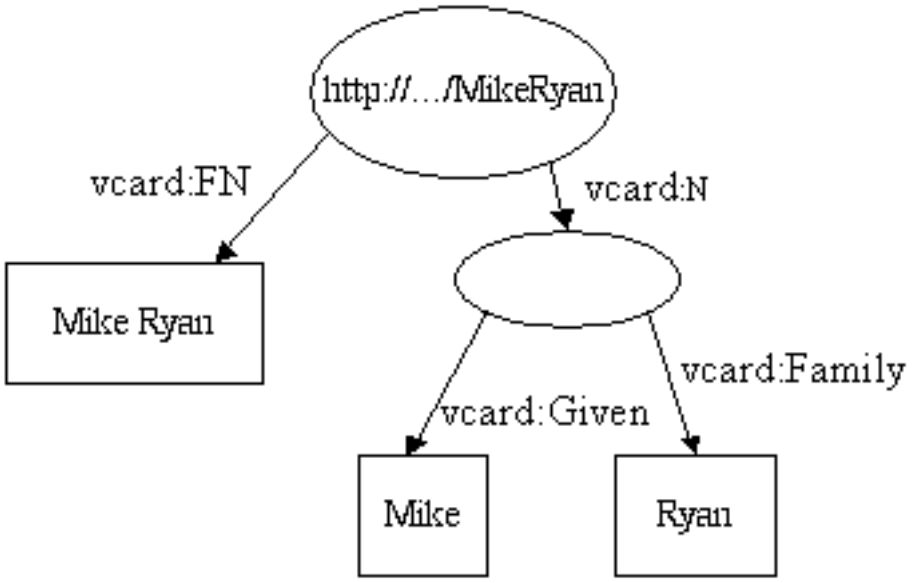


Figura 4. Modelo RDF en Jena

El framework Jena, de Apache Software Foundation, es utilizado especialmente para el desarrollo de aplicaciones basadas en ontologías y proporciona una API bastante potente para el desarrollo de modelos RDF.

Además de lo explicado anteriormente, el RDF está cada vez más extendido en la actualidad por ser una de las piedras angulares de la Web Semántica. La Web Semántica [30] es un concepto impulsado por Tim Berners-Lee con el objetivo de mejorar Internet al permitir que los ordenadores sean capaces de entender e interpretar la información almacenada en la nube. Hoy en día los buscadores colocan la información en una lista, sin llegar a comprender el significado de esa información. Por ello, la Web Semántica trata de dar un paso más, formando una red de documentos e información inteligente, con contenido semántico o atributos que lo determinen, de forma que los buscadores puedan interpretarla, haciendo sus consultas más inteligentes y precisas. Las diferencias entre la forma de estructurar la información en la Web actual y en la Web Semántica se observan en la Figura 5.

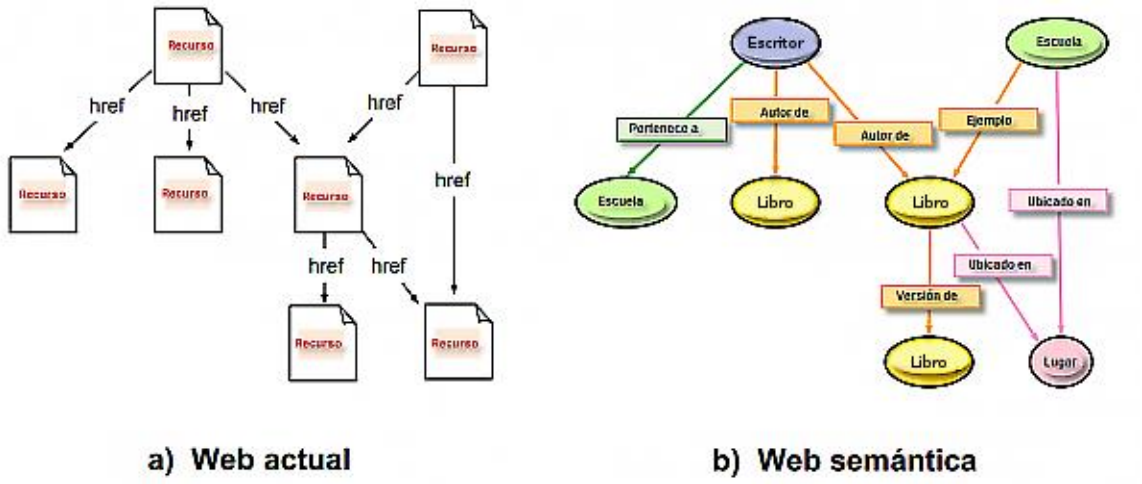


Figura 5. Comparación Web actual y Web Semántica

3. Tecnologías y herramientas utilizadas

Para el desarrollo de las dos aplicaciones se ha utilizado Java Standard versión 1.6, usando las librerías y los frameworks que se listan a continuación:

- Apache Hadoop [19]: Framework de software que permite almacenar y procesar grandes conjuntos de datos, propios del *Big Data*, a bajo coste y elevada escalabilidad. Es un proyecto de código libre desarrollado en Apache. Este framework es la tecnología principal de la aplicación desarrollada para la paralelización.

- GATE [15]: Suite de herramientas en Java desarrolladas en la Universidad de Shelffied para el desarrollo de cualquier tipo de tareas de procesamiento del lenguaje natural. Los algoritmos a paralelizar han sido implementados usando esta arquitectura.

- Apache Jena [31]: Framework Java de la fundación Apache para construir aplicaciones basadas en ontologías. Este framework, más concretamente el API RDF, se ha usado en este TFG para la escritura de las anotaciones generadas por los algoritmos de minería de textos.

- Maven [32]: Herramienta desarrollada por Apache para la gestión de proyectos Java a través del uso de un *Project Object Model* (POM). En ese POM se describe el proyecto que se va a crear y sus dependencias internas y externas. La característica principal de Maven y el motivo por el que se ha utilizado para el desarrollo de la aplicación principal de este Trabajo Fin de Grado es por la facilidad que proporciona para utilizar jars externos. Simplemente hay que indicarlos en el POM y Maven se encarga de buscarlos en un repositorio y descargarlos.

4. Desarrollo

A lo largo de este apartado se detallará el trabajo realizado durante las fases seguidas para el desarrollo del proyecto.

4.1 Elección y configuración de la arquitectura del sistema

En esta fase se realizó gran parte del estudio del estado del arte, ya que no se tenían los conocimientos mínimos sobre el ámbito del TFG. Una vez adquiridos dichos conocimientos, se procedió a la creación de dos máquinas virtuales para la instalación del framework Apache Hadoop. El SO usado en ambas máquinas es Red Hat de 64 bits, concretamente la versión 6.5 de CentOS, por que las versiones superiores daban problemas con el framework y esta versión era la más estable. Por otro lado, se descartó el uso de un sistema operativo Windows porque el framework estaba más enfocado a sistemas operativos Linux, es decir, originalmente no tenía soporte para Windows y, en las versiones de Apache Hadoop inferiores a la 2.x.x, era más tedioso instalarlo en este SO que en Linux. En las versiones 2.2.x ya se ofrece soporte nativo para Windows [33] (Server, Vista y 7).

En las dos máquinas virtuales se instaló la versión 2.2.0 de Apache Hadoop, conocido como YARN (*Yet Another Resource Manager*). YARN [34], también llamado MapReduce2, es una mejora de las versiones anteriores de Hadoop para evitar cuellos de botella con muchos nodos. Esto lo consigue separando las tareas del *jobtracker* en dos demonios:

- *Resource Manager*: Gestiona el uso de los recursos del clúster. El nodo con este rol también se encarga de planificar los contenedores de los *node managers*. Se considera un rol de un nodo.
- *Application Manager*: Gestiona el ciclo de vida de las aplicaciones que se ejecutan en el clúster. Decide dónde se ejecutan los *tasks* del *job* MapReduce (en la misma JVM o en otras) y los coordina. No se

considera un rol propiamente dicho, porque se ejecuta en un *node manager* dentro del container que genera.

Al separar las tareas del *jobtracker* en dos nodos se consigue recuperarse de posibles errores sin hacer que caiga la aplicación entera, tal y como ocurría en el caso del *jobtracker*. Cabe destacar que en YARN el *tasktracker* se le conoce como *Node Manager*, aunque se le amplía su funcionalidad. Un *node manager* crea un *container* con el *application manager* (en ese caso el nodo se consideraría parte de los masters) o un *container* para los *tasks* JVM, que pueden ser *map tasks* o *reduce tasks* (en ese caso se considera como esclavo). En la Figura 7 se muestra de forma más clara el funcionamiento de Hadoop usando YARN.

Otra de las ventajas de YARN es la compatibilidad con las versiones anteriores de Hadoop, por lo que se pueden usar tanto los comandos nuevos como los antiguos y ejecutar aplicaciones desarrolladas en esas versiones.

La instalación de Apache Hadoop se realizó de manera manual para tener un control total de lo que se estaba realizando e instalando y, por otro lado, para adquirir experiencia a la hora de configurar software. Primero se instaló el framework de forma pseudo-distribuida en los dos nodos y, una vez comprobado que ambos funcionaban de forma independiente, se procedió a configurarlos para que funcionaran de manera distribuida, asignando una de las máquinas con el rol de maestro (*master*) y la otra con el rol de esclavo (*slave*) e iniciando los servicios correspondientes al rol de cada una (dichos servicios se explicaron en el apartado 2.3, pero cambiando el *jobtracker* y el *tasktracker* por el *resource manager* y el *node manager*). Existen formas más sencillas y automáticas para instalar y configurar Hadoop, tales como el CDH de Cloudera [35] o el HDP de Hortonworks [36]. Estas soluciones son ideales para un clúster de un tamaño considerable, ya que instalan Hadoop en todas las máquinas y configuran los distintos roles y servicios en cada una de ellas según la elección del usuario.

Una vez instalado y configurado, se realizó el estudio de las fases que conforman una aplicación que implemente la filosofía *MapReduce*, en concreto el framework Hadoop. A su vez se probaron el ejemplo más simple de un algoritmo *MapReduce*, el *wordcount* (contador de palabras), y el ejemplo del libro de referencia sobre Hadoop [37], que trata de un algoritmo que devuelve la temperatura más elevada de cada año. El primer algoritmo funciona tal y como se mostró en la Figura 3 y se explicó en el capítulo 2. El segundo algoritmo es un poco más complejo, pero fácil de comprender. El *mapper* extraía y devolvía los datos relevantes de los *inputs* de entrada del NCDC (National Climatic Data Center), es decir, el año y la temperatura frente a todos los demás datos que había medido una determinada estación en un momento determinado. Al *reducer* le llega una lista de todos los valores asociados a una clave determinada, que en este caso es el año. Por lo tanto, el *reducer* es el que realmente realiza la búsqueda de la temperatura máxima, recorriendo la lista de valores para un año (la clave) concreto. La implementación de este problema sería bastante más complicada si no se siguiera esta filosofía, sobre todo cuando haya que procesar una gran cantidad de documentos.

El siguiente y último paso de esta fase fue el estudio de la arquitectura GATE y del algoritmo que se iba a paralelizar. Sin embargo, dicho algoritmo no estaba listo, así que para el desarrollo de la aplicación se utilizó uno de los algoritmos que proporcionaba la herramienta GATE y que se asemejaba lo suficiente al de Bioledge, ANNIE [16].

ANNIE es un sistema de extracción de textos basado en NER. Como la mayoría de los algoritmos desarrollados en GATE, este sistema se estructura en una serie de componentes que forman un pipeline. A dichos componentes se les conoce como *processing resources* y, en el caso de ANNIE, los más representativos son los siguientes:

- *Document Reset*: Este recurso limpia el documento de anotaciones, devolviéndolo al estado original. Se usa en el caso de que el documento se haya procesado anteriormente por el mismo pipeline o por otro.

- *Tokeniser*: Se encarga de dividir el texto en tokens, diferenciando números, palabras y signos de puntuación.
- *Gazetteer*: Identifica nombres de entidades basándose en listas estáticas.
- *Sentence Splitter*: Divide el texto en sentencias.
- *Part of Speech Tagger*: Crea anotaciones de cada palabra o símbolo en forma de etiquetas. Para ello usa un diccionario y un conjunto de reglas. También existe el *Semantic Tagger*, que se basa en el lenguaje JAPE [38] (Java Annotation Patterns Engine) y se encarga de crear las anotaciones identificando las diferentes entidades y el tipo al que pertenecen (Person, Location, Money, etc.).
- *OrthoMatcher*: Agrega las relaciones de identidad existentes entre las entidades identificadas por el *Tagger*, de tal forma que si encuentra dos entidades iguales y una de ellas se ha clasificado como "*Unknown*" se vuelve a clasificar con el mismo tipo que se haya asignado a la otra entidad. Esto se realiza para mejorar la correferencia.

Para asimiliar ANNIE con el algoritmo que se tenía pensado paralelizar en un futuro se sustituyó el *Gazetteer* por el *LKB Gazetteer* [39]. Este *gazetteer* es más potente que el de ANNIE, ya que no se basa únicamente en listas estáticas, sino también en repositorios semánticos, ya sean diccionarios estáticos o dinámicos. Ejemplos de repositorios que usa es DBpedia [28], accediendo a su información mediante el lenguaje SPARQL [40].

Cualquier algoritmo desarrollado en GATE necesita para ejecutarse unos *inputs*, conocidos como *Language Resources*. Estos recursos se componen de corpus y documentos, pero el algoritmo sólo acepta como entrada los corpus, que estarán compuestos de uno o más documentos.

Para ejecutar un algoritmo desarrollado en GATE desde cualquier lenguaje de programación basta con exportar dicho algoritmo a zip (o tenerlo en un directorio) y crear un programa, por ejemplo en Java, con los siguientes pasos:

1. Iniciar Gate.
2. Crear un corpus vacío.
3. Establecer las rutas de los plugins y de la aplicación.
4. Cargar la aplicación en GATE y asignarle el corpus.
5. Crear documentos GATE a partir de los documentos y añadirlos al corpus.
6. Ejecutar la aplicación.

Para el caso de ANNIE, la versión 7.1 de GATE provocaba errores a la hora de ejecutarse desde Java. Sin embargo, con la versión 7.0 no había ningún problema.

4.2 Análisis, diseño y construcción del sistema

En la fase anterior se ha mostrado los puntos que se debe seguir para la ejecución de un algoritmo desarrollado en GATE desde cualquier lenguaje de programación. En esta fase se explicará el resto del análisis y diseño del sistema y su implementación.

i. Aspectos básicos

El desarrollo se realizó sobre el sistema operativo CentOS 6.5 usando el IDE Eclipse Juno [41]. El sistema está compuesto de dos proyectos:

- TFG_ParallelizationApp: La aplicación con más peso en el sistema. Se encarga de la paralelización de los algoritmos desarrollados en GATE usando MapReduce. También genera los modelos RDF de cada documento procesado por el algoritmo de GATE. Para su funcionamiento se debe ejecutar el jar a través de Hadoop.

- TFG_UnionModels: Aplicación secundaria que ofrece más funcionalidad al sistema. Se ejecuta independiente de Hadoop y, por

tanto, de la paralelización. Esta aplicación une en un único modelo global todos los modelos RDF generados por la primera aplicación.

La estructura de ambos modelos se refleja en la Figura 6.

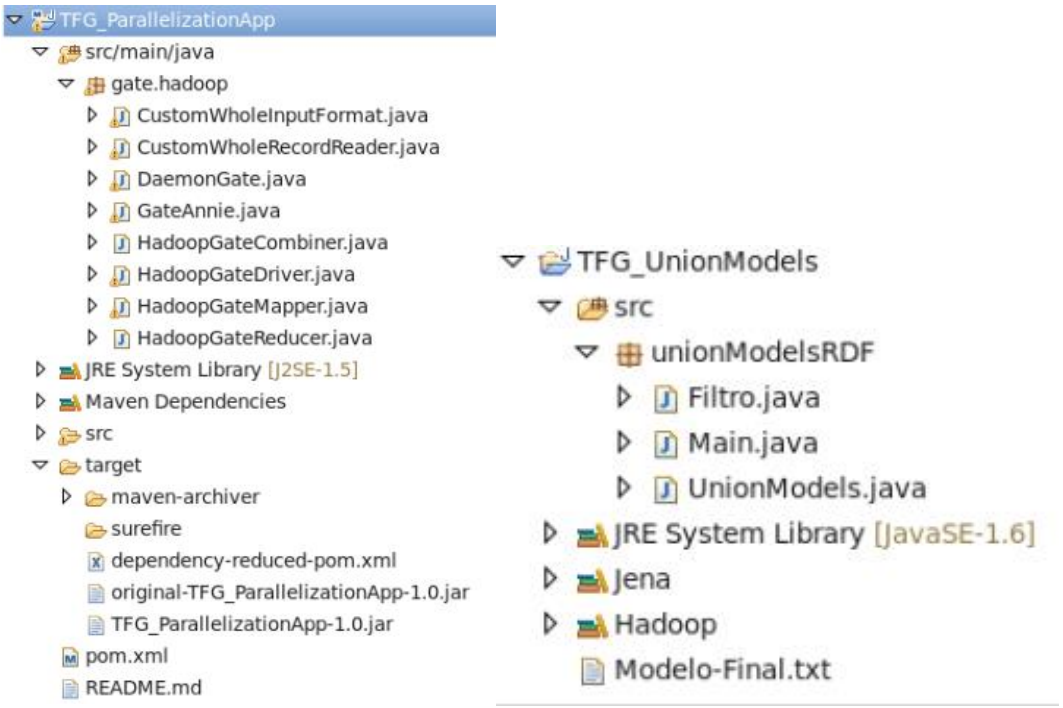


Figura 6. Estructura de las aplicaciones desarrolladas

ii. Incrementos y etapas de la fase

A continuación se describen los incrementos realizados durante el desarrollo de esta fase. Destacar que en la fase anterior se probó la ejecución en Hadoop de ejemplos *MapReduce* desarrollados en Java. Por ello, en las etapas de esta fase se empezó el propio desarrollo de ambas aplicaciones.

1. Primera versión TFG_ParallelizationApp: Esta primera versión consistía en el desarrollo de un algoritmo MapReduce capaz de iniciar GATE y ejecutar el algoritmo ANNIE, escribiendo en un fichero las anotaciones que generaba el algoritmo.

2. Segunda versión TFG_ParallelizationApp: En esta versión se implementaba otro de los requisitos, la generación de modelos RDF. Para ello, se sustituyó el código del *MapReduce* para que en el *mapper* se creara el modelo RDF del *input* correspondiente (las anotaciones del procesamiento de un párrafo de un documento) y el *reducer* unía todos los modelos generados por el *mapper* de un documento en concreto. La salida de la ejecución en Hadoop es un fichero con pares <nombreDocumento, modelo RDF>.

3. TFG_UnionModels: Desarrollo de la aplicación encargada de unir todos los modelos RDF de documentos distintos en un único modelo.

4. Tercera versión TFG_ParallelizationApp: Implementación de un demonio encargado de gestionar GATE, es decir, su inicialización y la limpieza del corpus. De esta forma se libera al *mapper* de realizar dicho trabajo. En esta versión también se desarrolló el código necesario para que Hadoop no troceara los documentos, es decir, no dividiera el documento en una serie de pequeños *records*.

Cada uno de estos incrementos se dividía en varias etapas:

- Selección de los objetivos y requisitos a implementar en el incremento. En todos los incrementos, excepto en el primero, se tenían en cuenta también unos objetivos de corrección o mejoras del código ya desarrollado.
- Priorización y planificación de los objetivos y requisitos. La corrección de errores tenía la máxima prioridad.
- Diseño e implementación de los requisitos.
- Realización de pruebas (*testing*). Primero se realizaban pruebas unitarias al código implementado, y luego se realizaban las pruebas al sistema completo. Durante esta etapa se detectaban errores que se tenían en cuenta en el siguiente incremento.

iii. Implementación y funcionamiento del sistema

En este apartado se explicará el funcionamiento del sistema, junto con su implementación.

En el diagrama de arquitectura representado en la Figura 7 se resume el funcionamiento básico del sistema completo. Para ejecutar una aplicación con YARN se debe utilizar la consola de comandos e introducir el siguiente comando:

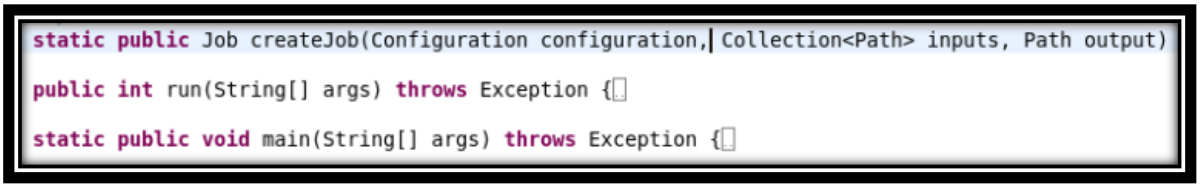
```
yarn jar application.jar MainClass inputs output
```

- *Application.jar*: Aplicación MapReduce que va a ejecutar Hadoop.
- *MainClass*: Clase que crea el *job* y lo lanza.
- *Inputs*: Carpeta o documentos de entrada a la aplicación. Se deben encontrar en el sistema distribuido de Hadoop (HDFS).
- *Output*: Carpeta con los resultados de la ejecución. Se crea en el HDFS durante la ejecución, por lo que no debe existir anteriormente.

El nodo maestro se encargará de controlar dicho *job* y dividirlo en *tasks*, mientras que los nodos esclavos ejecutarán las *tasks* que les correspondan, comenzando de esta forma el proceso MapReduce.

Para el caso de este sistema, hay que indicar en el comando el jar de la aplicación TFG_ParallelizationApp y su clase que actúa de principal, HadoopGateDriver, incluyendo los documentos o directorios de entrada y el directorio de salida. En el apartado 7.1 se indica cómo se ejecuta.

HadoopGateDriver La clase principal consta de los siguientes tres métodos:



El primer método que se ejecuta es el `main(String[] args)`, con los argumentos que se pasan en el comando (los *inputs* y el *output*). Este método llama a `run(String[] args)`, que se encarga de extraer los argumentos, crear y ejecutar el *job*, devolviendo un booleano si se ha completado con éxito o no.

El último método es `createJob(Configuration configuration, Collection<Path> inputs, Path output)` y se encarga de crear un *job* dados la configuración con la que se va a ejecutar el *job*, una colección de los documentos de entrada y el directorio de salida por el método explicado anteriormente.

En la creación del *job* se añade a la configuración la dirección URI del algoritmo y se crea el *job*. Una vez creado se añaden los *paths* de los *inputs* y del *output*, el rol de cada clase, es decir, qué clases implementan el *Mapper*, el *Reducer* y/o el *Combiner*, y las clases usadas para el par <clave, valor>, siendo la clase Text de Hadoop. Además, se han especificado las clases para el formato de los *inputs* y del *output*, de tal forma que bastaría con cambiar la clase del formato de los *inputs* (TextInputFormat) para evitar que se troceen los documentos. La clase a cambiar sería WholeInputFormat que es llamada antes de entrar en la fase del *mapping*, por la fase *splitting*.

Tal y como se explicó y mostró en la Figura 3, los documentos de entrada se dividen en diversos *records*. Esta fase se conoce como *splitting*. En las librerías de Hadoop hay clases que implementan esta característica, por lo que se usó la clase TextInputFormat para el sistema a implementar. Esta clase genera *records* por cada salto de línea que encuentra en el texto, que suele corresponderse con los párrafos.

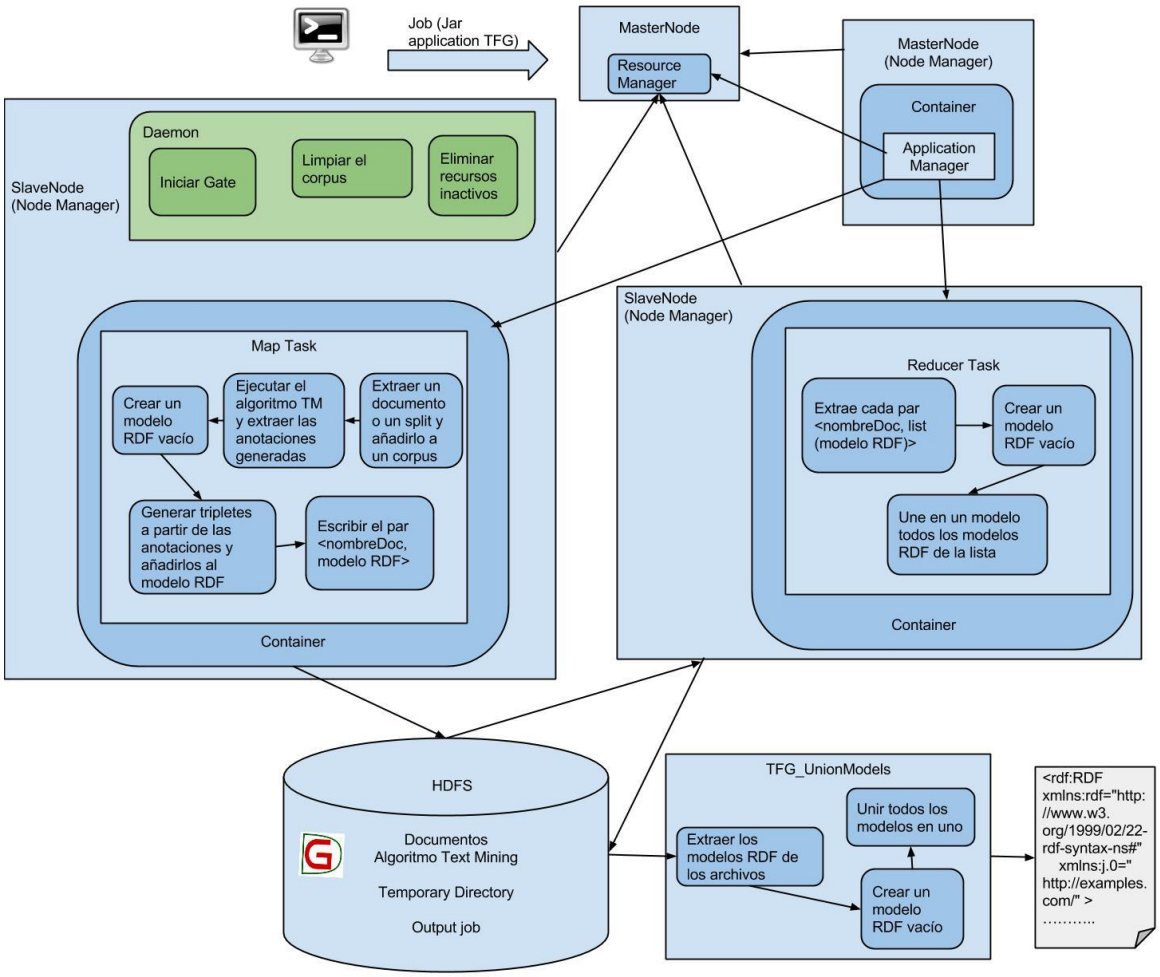


Figura 7. Diagrama de Arquitectura

El problema es que no hay ninguna clase que genere un único *record*, es decir, que no dividiera el documento. Por ello se implementaron las clases `WholeInputFormat` y `WholeRecordReader`, ya que se quería analizar las diferencias de información que puede existir al ir ejecutando el algoritmo de minería de textos con fragmentos del documento, que implica la pérdida del contexto semántico de todo el documento, frente al procesamiento del texto completo en una sola pasada, conservando el contexto.

WholeInputFormat posee dos métodos:

```
public class WholeInputFormat extends FileInputFormat <LongWritable, Text> {
    protected boolean isSplitable (JobContext context, Path file){
public RecordReader<LongWritable, Text> createRecordReader(InputSplit split, TaskAttemptContext context) {
```

El primer método indica si el texto pasado como argumento (`file`) se puede dividir o no. Como esta clase se creó para evitar que se dividiera el texto, siempre devuelve que no lo es. Este método no es suficiente para evitar la generación de varios *records*, ya que hay que crear otra clase que devuelva el *record* correspondiente y como se mencionó anteriormente, no hay clases que implementen esta característica. Esa clase es `WholeRecordReader`.

El segundo método se encarga de crear una instancia de WholeRecordReader e inicializarlo.

WholeRecordReader Clase que devuelve un documento completo como un *record*. Implementa los métodos de la interfaz de la que extiende (RecordReader). Entre dichos métodos se encuentran los *getter* de la clave y el valor que haya extraído, el porcentaje del progreso y la inicialización de las variables para obtener los records, como son `fileSplit`, que es el fichero que se va a dividir, y `conf`, la configuración/contexto del *task* que lo está ejecutando.

Los parámetros para la inicialización se los pasa la clase WholeInputFormat.

El método más importante de la clase es `nextKeyValue()`. Este método devuelve un booleano indicando si hay más *records* o no, así que en este caso solo devuelve *true* cuando no se ha procesado el documento, introduciendo en la variable `value` el contenido completo del documento. En el caso de que ya se haya obtenido el record antes, devuelve `false`.

```
@Override
public void close() throws IOException {

}

public LongWritable getCurrentKey() throws IOException,

@Override
public Text getCurrentValue() throws IOException, InterruptedException {
    return value;
}

@Override
public float getProgress() throws IOException, InterruptedException {
    return procesado ? 1.0f : 0.0f;
}

public void initialize(InputSplit split, TaskAttemptContext context)

 * Método que devuelve el siguiente valor (record), que se corresponde con
 public boolean nextKeyValue() throws IOException, InterruptedException {
```

Los pares *<key, value>* son las entradas al *mapper*, que ejecuta la función map() por cada par que le llega. El key contiene el *offset* del *record* con respecto al texto completo, que en el caso de los *records* generados por la clase WholeRecordReader siempre valdrá 0, ya que solo hay un *record* por documento. El value es el contenido del *record*. Por lo tanto, para el *mapper* que se ha creado, lo único que se necesita es el *value*, el *key* no es necesario.

Así pues, al terminar la etapa de *splitting*, los *records* se envían a los *mappers* correspondientes, dando lugar al comienzo de la etapa del *mapping*.

HadoopGateMapper Clase que implementa el *mapper*. Su función es ejecutar el algoritmo de minería de textos al *record* correspondiente y crear el modelo RDF asociado a las anotaciones generadas por el algoritmo. Los métodos más importantes para realizar esta tarea son los siguientes:

```
 * Método que inicia GateAnnie en el caso de que no exista, creando al
protected void setup(Context context) throws IOException, InterruptedException {

 * Método principal del mapper. Anota el documento y genera el modelo RDF
protected void map(LongWritable offset, Text text, Context context) throws IOException,

protected void cleanup(Context context) throws IOException, InterruptedException {


 * Genera el RDF del documento a partir de las anotaciones de la aplicación
private String generateRDF(Document document, long tamDoc) throws IOException{
```

El método `setup(Context context)` es el primero que se ejecuta en la creación del *mapper* y se encarga de crear, en caso de no existir, la clase que se encargará de gestionar y ejecutar el algoritmo desarrollado en GATE, GateApp, por lo que una vez acabe el método se tendrá inicializado GATE, junto con el algoritmo NER y el corpus asociado. Esta clase se explicará con más detalle después, pero ahora conviene mencionar que para su creación es necesaria la clase DaemonGate, que también se explicará más adelante. A modo de resumen, el demonio de encarga de iniciar GATE, el algoritmo y el corpus asociado a dicho algoritmo, además de encargarse de la limpieza del corpus. Si no existe el demonio lo crea, le pasa la URI del algoritmo que se había añadido al contexto en la clase HadoopGateDriver y lo inicia. Si ya existe, significa que se había caído el *task* pero GATE sigue iniciado, al igual que el algoritmo y el corpus, así que simplemente se crearía un objeto GateApp que recupere el control del algoritmo y del corpus del demonio.

El método principal de la clase es `map(LongWritable offset, Text text, Context context)`. Se ejecuta después del anterior y se encarga de pasarle el algoritmo al *record* (`text`) para que se generen las anotaciones, y crear el modelo RDF a partir de ellas. Primero crea un documento GATE a partir del contenido del *record* y lo pasa como argumento a uno de los métodos de GateApp (`String annotation(Document document)`) para que ejecute el algoritmo sobre dicho documento. Si el documento está vacío o es una línea en blanco se evita su llamada. Este método devuelve un xml con las anotaciones que ha generado el algoritmo. Para su ejecución es necesario que el demonio no esté realizando la limpieza del corpus en ese momento, por eso se controla con semáforos que se han implementado en el demonio.

Cuando se termina la ejecución del algoritmo, se almacena en una variable el *path* del documento para poder referenciarlo al generar el RDF del documento, y el nombre del documento para escribirlo en el contexto como clave. También se obtiene el tamaño del documento para usarlo en la extracción de las anotaciones tipo *paragraph*.

Finalmente, en el caso de que el xml generado por el método de GateApp no sea null, se llama al método que genera el modelo RDF y se

escribe en el contexto el nombre del documento como clave y el modelo RDF del *record* como valor. Los pares <clave, valor> generados por los *mappers* serán los *intpus* de la siguiente etapa.

El método `generateRDF(Document document, long tamDoc)` genera el modelo RDF del documento pasado como argumento. Comienza obteniendo el conjunto de anotaciones del documento y anotando los párrafos usando una utilidad de GATE, porque en el caso de ANNIE no se etiquetaban (por eso se necesita el tamaño del documento). Cuando se anotan los párrafos se extrae el conjunto de anotaciones etiquetados como tales en el texto y se comienza a generar el modelo RDF a partir de esos dos conjuntos usando los métodos que se explicarán a continuación.

Finalmente, al crear el modelo se escribe en un *string*, se llama a `cleanModel()` para limpiar el modelo y se devuelve el *string* a la función principal del *mapper*, ya comentada anteriormente.

Los métodos que se usan para crear las tripletas RDF son los siguientes:

```
 * Obtenemos las anotaciones RDF para los párrafos
private void addAnnotationParagraph(AnnotationSet annotationsDoc, Annotation paragraph,

 * Saca las anotaciones (sentencias) contenidas en el párrafo, las transforma en resources y las añade
private void addAnnotationSentences(AnnotationSet annotationsDoc, Resource annotResource,

 * Saca las anotaciones de las palabras contenidas en la sentencia, las transforma en resources y las
private void addAnnotationTokens(AnnotationSet annotationsDoc, Resource annotSentenceResource,

 * Añade la anotación hija, que se le pasa como argumento, al recurso padre, como si fuera un property
private Resource addAnnotationChildren(Resource annotResource, Annotation annotationChildren,
```

El método `addAnnotationParagraph()` crea el recurso RDF para la anotación asociada al párrafo y le añade la propiedad del tipo de anotación junto al valor correspondiente (*paragraph*). Aparte crea el recurso para el documento junto con la propiedad asociada al nombre del documento. Al recurso del párrafo le añade otra propiedad para hacer referencia al recurso del documento. Para acabar, llama al método `addAnnotationSentences()` para agregar al recurso del párrafo las propiedades que hacen referencia a los recursos de las sentencias contenidas en dicho párrafo, además de crear los correspondientes recursos para cada una de ellas.

Los métodos `addAnnotationSentences()` y `void addAnnotationTokens()` siguen prácticamente el mismo patrón. Se filtran las anotaciones de un tipo determinado (sentence para las sentencias o *lookup*, *Person*, *Location*,… para los *tokens*) y por la anotación padre, es decir, se filtran las anotaciones que estén comprendidas en el rango de la anotación padre. Por ejemplo, si queremos obtener las sentencias de un párrafo se filtrarían las anotaciones del documento que tuvieran el tipo *sentence*, y el conjunto resultante se volvería a filtrar según el *offset* de inicio y fin de la anotación *paragraph* del párrafo al que pertenecen dichas sentencias.

El conjunto con las anotaciones resultantes se recorre para añadir cada anotación al modelo (addAnnotationChildren()) y, en el caso del método de las sentencias, se llama al método `addAnnotationTokens()` para que se extraigan y se añadan al modelo los tokens de la sentencia actual.

El último método sobre RDF, addAnnotationChildren(Resource annotResource, Annotation annotationChildren, Document document), se encarga de crear el recurso para esa anotación teniendo en cuenta si la anotación a agregar (`annotationChildren`) es una sentencia o un *token* y añadir al recurso las propiedades correspondientes con los valores obtenidos de la anotación (como el tipo de anotación y la clase a la que pertenece el recurso). Luego se crea un recurso para el documento y se añade como propiedad el nombre del documento. También se crea una propiedad en el recurso de la anotación hija para relacionarla con el recurso del documento.

En el caso de que la anotación hija haga referencia a un *token*, se genera un objeto con la entidad a la que hace referencia dicho *token*, almacenándose en la URI la instancia referenciada, si se obtiene de un recurso web o una base de datos, o el nombre de la entidad (si se obtiene de una lista estática). El recurso hijo tiene una propiedad que hace referencia al recurso de la entidad.

Por último, se añade el recurso creado al recurso padre a través de una propiedad de éste último, y se devuelve el recurso hijo.

En la Figura 8 se muestra un ejemplo sencillo de un modelo RDF generado por esta aplicación usando el algoritmo ANNIE para etiquetar palabras genéricas, en la que se puede observar las propiedades y los recursos que se usan. Las anotaciones de párrafos, sentencias y *tokens* tienen prácticamente las mismas propiedades:

- *TypeAnnotation*: Indica el tipo de la anotación (*Paragraph*, *Sentence* y para los *tokens*, *Lookup* u otros).
- *DocumentAssociated*: Documento del que provienen.
- *Subannotations*: Relaciona la anotación con sus anotaciones hijas. No existe en los *tokens*.

Las anotaciones de los *tokens* tienen dos propiedades más:

- *EntityAssociated*: Entidad a la que hace referencia el *token*.
- *Class*: Clase a la que pertenece la entidad. En el ejemplo, si el *token* es "*company*", la clase se ha etiquetado como *Class* al ser una palabra genérica. Si se hubiera aplicado el algoritmo para etiquetar palabras que referencien a compañías, podría etiquetar cada compañía (IBM, Apple, Microsoft) con la clase *Company*.

Existen dos recursos más, el que hace referencia al documento y el que hace referencia a la entidad. El del documento hace referencia al que se encuentra en el HDFS (a través de la URI que identifica al recurso) y tiene una propiedad que almacena su nombre.

La URI de la entidad hace referencia a dicha entidad en Internet o crea una URI propia, al igual que los párrafos, sentencias y *tokens*, en el caso de que no haya una referencia a algún recurso de la Web. Este recurso tiene una propiedad en la que se almacena las distintas palabras que se han usado en el documento para referirse a dicha entidad.

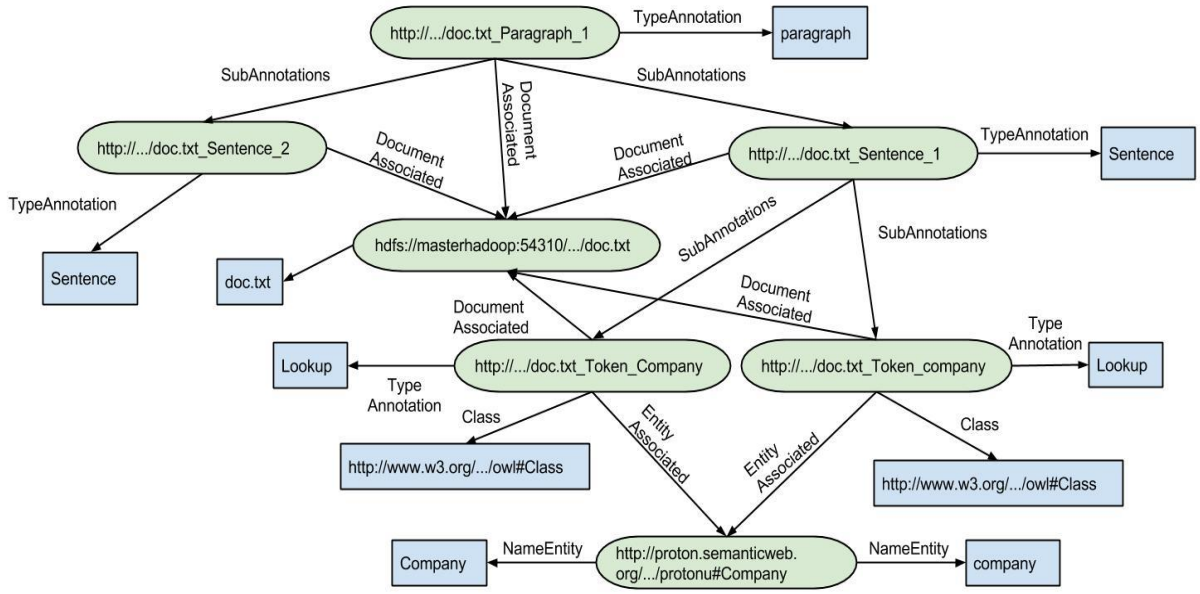


Figura 8. Modelo RDF de una ejecución del algoritmo

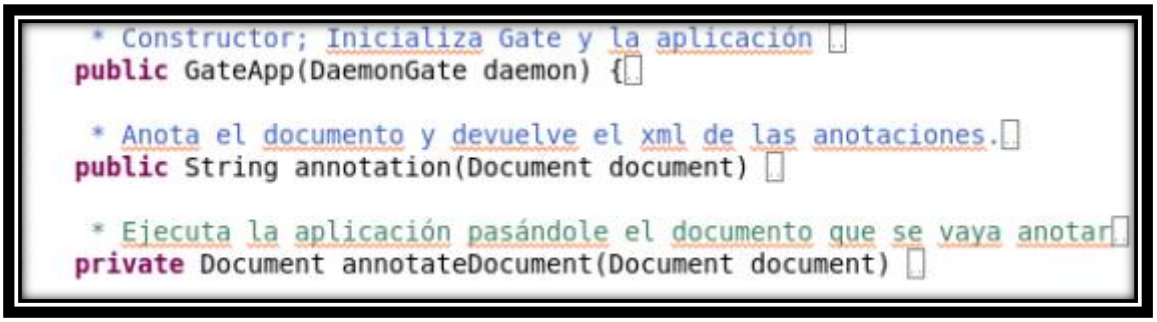
Para acabar, los últimos métodos del *mapper* son para limpiar las variables que almacenan el modelo y las propiedades y para obtener el daemonGate en el caso de que exista.

```
* Método para limpiar el modelo
   public void cleanModel(){

   //Método que busca si está el demonio activo y lo devuelve
   private Thread getDaemon() throws UnknownHostException{
```

En el método `cleanModel()` lo que se hace es cerrar el modelo actual y crear uno nuevo junto con las nuevas propiedades. Se usa en el método que genera el modelo RDF para evitar que en la siguiente ejecución del *map* se mantengan en el modelo. El método `getDaemon()` se encarga de buscar si existe un demonio tipo DaemonGate en el JVM del *mapper*. Este es el método que usa el *mapper* para saber si tiene que crear al demonio o no.

GateApp Es la clase que usa el *mapper* para controlar el algoritmo NER o cualquier algoritmo basado en GATE. Para ello, almacena en una variable el controlador de la aplicación (`application`) y en otra el corpus (`corpus`).



El *mapper* llama al constructor de la clase, pasándole el demonio como argumento. En él se realiza la petición al demonio de que inicie GATE (el demonio se encargará de ver si ya está iniciado o no) y se recupera de éste el controlador del algoritmo y el corpus.

El método `String annotation(Document document)` llama al método que anota un documento (`Document annotateDocument(Document document)`) y genera el xml del documento anotado, devolviéndolo como *string*. El *mapper* usa el xml para comprobar si se ha ejecutado el algoritmo o no.

Por último, el método `Document annotateDocument(Document document)` ejecuta el algoritmo, añadiendo antes al corpus de la aplicación de GATE el documento o fragmento de documento pasado como argumento. El resultado devuelto es el documento con las anotaciones implícitas, es decir, en el documento no se ven a simple vista, solo se pueden obtener al usar los métodos de las librerías de GATE sobre dicho documento.

DaemonGate Es una de las clases más importantes. Es un *thread* demonio con dos funciones:

- Iniciar Gate y preparar el algoritmo de minería de textos y el corpus la primera vez que se ejecuta y cada vez que GateApp lo requiera.
- Limpiar el corpus cuando alcance cierto tamaño (8.000 documentos), para evitar el bloqueo de GATE que se produce al tener corpus con 10.000 documentos.

Esta clase tiene un constructor que crea el *thread* y lo convierte en demonio.

El resto de métodos son usados en el cuerpo (`run()`) del demonio. El método `run()` se divide en tres partes:

1. Inicia GATE, junto con el algoritmo y el corpus.
2. Bucle de ejecución. Aquí se realiza el trabajo más importante del demonio. Primero se comprueba si GateApp ha pedido que se inicie GATE y el algoritmo, en cuyo caso se procede a realizarlo (usando el método de la 1º parte). Después se realiza la limpieza del corpus si se ha llegado al tamaño máximo. Finalmente, comprueba si el *mapper* ha acabado para finalizar su ejecución, ya que solo tiene sentido en la fase del *mapping*.
3. Cierre de GATE, se limpia el corpus y se liberan los recursos.

```
public DaemonGate (String msg){

 * Cuerpo del demonio
public void run(){
```

La mayoría de los métodos son *getter* y *setter* de las variables de la clase, facilitando a las otras clases (GateApp y el mapper) que puedan interactuar con el demonio:

- Métodos para obtener y cambiar la URI del algoritmo de GATE: `getURI()` y `setURI (URI newUri)`.
- Métodos para obtener el controlador del algoritmo y el corpus: `getApplication()` y `getCorpus()`.
- Procedimientos para obtener y cambiar el *flag* para indicar si se debe iniciar GATE: `getFlagWantGate()` y `setFlagWantGate(boolean flagWantGate)`.
- Procedimiento para obtener (*acquire*) o dejar (*release*) el semáforo para controlar que no se manipule el corpus por dos o más objetos, lo usa el *mapper* para evitar que mientras se está ejecutando el

algoritmo sobre el corpus no se haga la limpieza, y viceversa: `getMutexGate(boolean b)`.

```
public void setURI (URI newUri){

public URI getURI (){

public CorpusController getApplication(){

public Corpus getCorpus (){

public boolean getFlagWantGate() {

public void setFlagWantGate(boolean flagWantGate) {

 * Método para utilizar el semáforo de limpieza. lo utiliza el map para
public void getMutexGate(boolean b){
```

A continuación se detallan el resto de métodos del demonio:

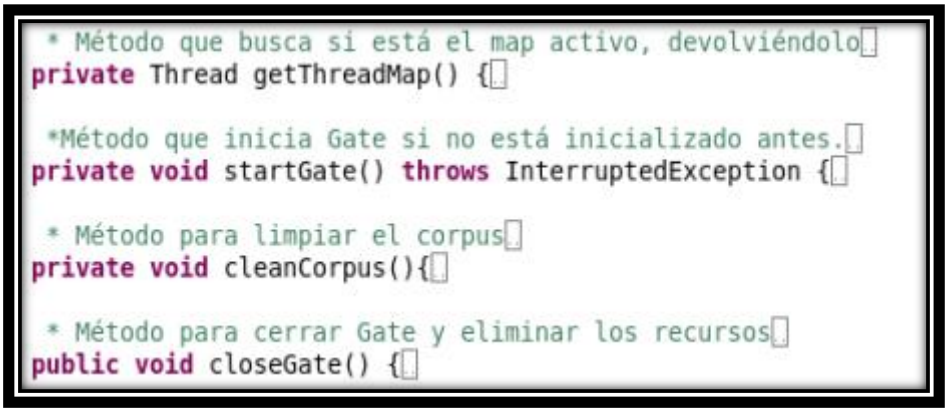
```
//Para detener al demonio
public void stopDaemon(){

 * Método que usa GateAnnie para indicar que hay que volver a
public void wantStartGate(){

public boolean isStartGate(){
```

El procedimiento `stopDeamon()` desactiva el booleano para detener el bucle del demonio y que termine de ejecutar su run. El método `wantStartGate()` lo usa GateApp para activar el *flag* en el caso de que no esté iniciado GATE, indicando que necesita su inicialización y la del algoritmo. Para evitar la posible concurrencia al tocar esa variable se usan semáforos. Cuando se termina el método se asegura que GATE está inicializado.

El procedimiento `isStartGate()` comprueba si GATE está inicializado y si las variables que almacenan el algoritmo y el corpus no son nulas, es decir, si ya están instanciadas.



```
 * Método que busca si está el map activo, devolviéndolo
private Thread getThreadMap() {

 *Método que inicia Gate si no está inicializado antes.
private void startGate() throws InterruptedException {

 * Método para limpiar el corpus
private void cleanCorpus(){

 * Método para cerrar Gate y eliminar los recursos
public void closeGate() {
```

El método para conseguir el *thread* del *map* si está activo es `getThreadMap()`, y lo usa el demonio para saber si tiene que parar su ejecución (no es una variable de clase).

El procedimiento `startGate()` inicia GATE si no está inicializado de antes, establece las rutas de los *plugins*, carga el algoritmo NER, crea un corpus y lo asigna al controlador del algoritmo. Usa semáforos para que no se modifique el *flag* que indica que hay que levantar GATE, y al acabar lo pone a falso.

El método `cleanCorpus()` recorre el corpus eliminando completamente cada documento, y luego elimina el corpus, destruyendo todos los recursos de cada documento y del propio corpus. Al final del método se crea un nuevo corpus y se asigna al controlador del algoritmo. El código de este método está controlado por semáforos para evitar la concurrencia en el corpus.

El último procedimiento, `closeGate()`, se encarga de limpiar el corpus con el método anterior y eliminarlo junto con el controlador de la aplicación.

Cuando acaba la fase del *mapping* comienza las etapas del *shuffling* y del *reducing*. El *shuffling* se hace automáticamente por Hadoop, y consiste en reunir los resultados de todos los *mappers* que tengan la misma clave, para enviárselo a un *reducer*. Como se ha visto, la clave que generan los *mappers* es el nombre del documento al que pertenece el modelo RDF generado por uno de sus fragmentos o del texto completo, así que el *shuffling* pasaría como argumentos al *reducer* la lista de todos los modelos generados en el *mapping*

para un único documento. Como ocurría con el *mapping*, los *node managers* asignados se encargan de crear tareas (*task*) para ejecutar el *reducer*.

HadoopGateReducer Clase que implementa el *reducer* de la aplicación. Al igual que el *mapper*, implementa el método de su interfaz (*Reducer*) que será llamado por los *tasks* asignados a la fase *reducing*.

```
public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
```

El método se encarga de unir los modelos RDF almacenados en la lista de valores (`values`) en un único modelo, obteniendo el modelo final que representa a las anotaciones del documento al que haga referencia la clave (`key`). Así pues, se recorre dicha lista y se va realizando la unión de conjuntos con el modelo final, que en la primera iteración se inicializará con el primer modelo RDF de la lista. Cuando acaba el bucle se escribe en el contexto del *reduce task* la misma clave que ha recibido y el modelo RDF resultante de la unión.

La ejecución de la aplicación finaliza cuando se escribe el resultado de todos los *reducers* en ficheros llamados part-r-xxxxx, donde las x referencian al identificador de cada *task*. Dichos ficheros se crean en el directorio del HDFS que el usuario ha pasado como argumento en el *output* y contendrán pares <nombre del documento, modelo RDF>. Por ello, para poder tener un modelo global de todos los documentos procesados hay que usar la otra aplicación del sistema.

En la Figura 9 se muestra el diagrama de clases del programa TFG_ParallelizationApp.

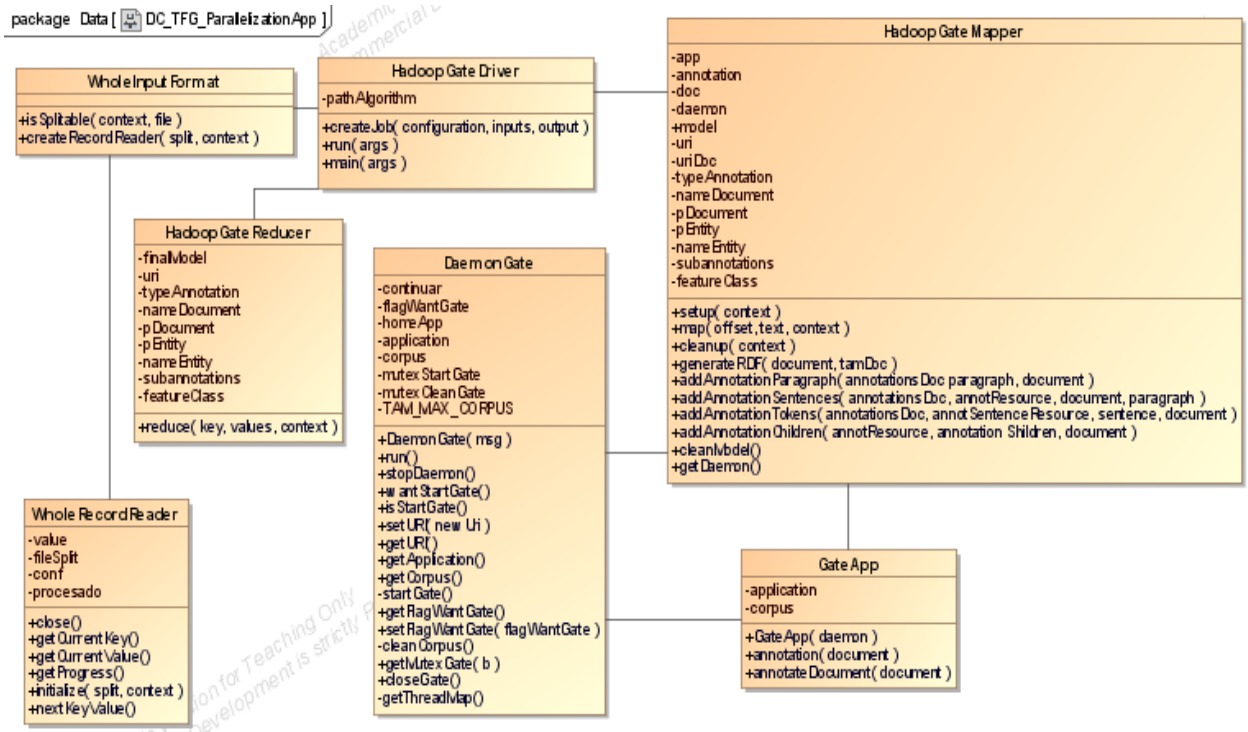


Figura 9. Diagrama de clase de TFG_ParallelizationApp

La segunda aplicación, TFG_UnionModels, se encarga de unir en un único modelo global todos los modelos RDF generados por la primera aplicación, lanzada en Hadoop. Este proyecto es de menor envergadura que el anterior y está compuesto por tres clases: Main, Filtro y UnionModels.

Main Es la clase ejecutable y se encarga de crear un objeto UnionModels y ejecutar su método para la unión de los modelos, pasándole como parámetro el nombre del directorio que haya introducido el usuario como argumento.

```java
public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        UnionModels unionModels = new UnionModels();
        try {
            unionModels.unionModels(args[0]);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

Filtro Esta clase es usada por la clase UnionModels para filtrar los ficheros que hay en la carpeta de salida generada por Hadoop, indicando si son ficheros con resultados de la ejecución de la aplicación o no. Estos ficheros tienen la siguiente estructura: part-r-xxxxx, donde las 'x' hacen referencia al número del *task* que lo ha generado. Comienzan por el 0, es decir, el primer fichero es part-r-00000.

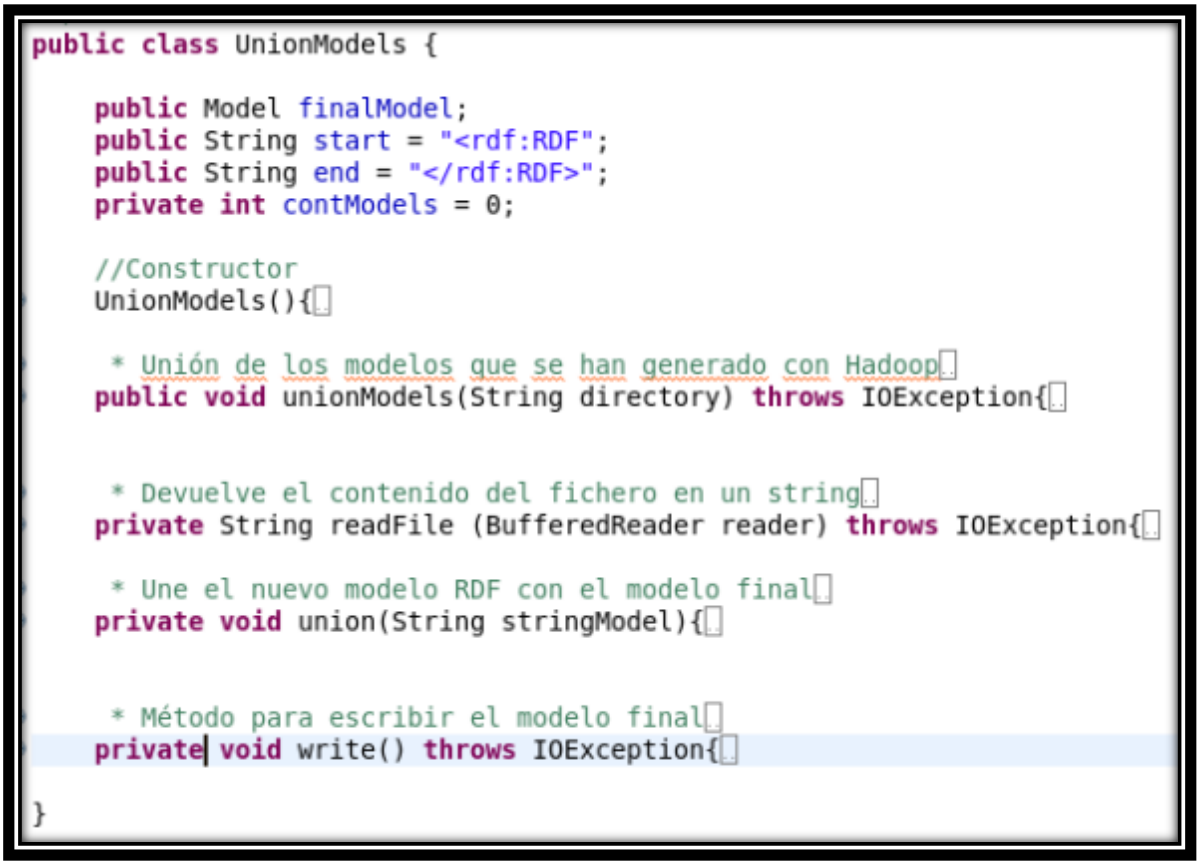
```java
public class Filtro implements PathFilter{

    //Devuelve true si el fichero empieza por part-r-*
        //(fichero de salida de Hadoop)
    @Override
    public boolean accept(Path path) {
        return path.getName().startsWith("part-r-");
    }

}
```

UnionModels Es la clase que implementa la lógica de esta aplicación. Tiene un constructor que crea un modelo RDF vacío y lo asigna a la variable `finalModel`.

Su método principal es `unionModels()` y usa como apoyo los otros métodos privados de la clase.



```
public class UnionModels {

    public Model finalModel;
    public String start = "<rdf:RDF";
    public String end = "</rdf:RDF>";
    private int contModels = 0;

    //Constructor
    UnionModels(){

     * Unión de los modelos que se han generado con Hadoop
    public void unionModels(String directory) throws IOException{


     * Devuelve el contenido del fichero en un string
    private String readFile (BufferedReader reader) throws IOException{

     * Une el nuevo modelo RDF con el modelo final
    private void union(String stringModel){


     * Método para escribir el modelo final
    private void write() throws IOException{

}
```

Los métodos de soporte son los siguientes:

- El método `readFile (BufferedReader reader)` se encarga de devolver el contenido del fichero que se le pasa como argumento en un *string*.
- `write()` crea un fichero (Modelo-Final.txt) en el que escribe el modelo RDF global generado por la unión de todos los modelos.
- El método `union(String stringModel)` crea un modelo a partir del *string* que se le pasa como argumento y lo une al modelo final. La unión de los modelos es como la unión matemática, es decir, no duplica recursos ya existentes, como los de las entidades.

El método principal es `unionModels(String directory)` y su función es la de unir todos los modelos generados en la ejecución de Hadoop. Para ello, lo primero que hace es crear la configuración necesaria para conectarse al HDFS, añadiendo como recursos los ficheros de configuración hdfs-site.xml y core-site.xml, además de especificar los parámetros del *FileSystem* local y el

del HDFS. Con esta configuración se puede acceder al HDFS (deben estar corriendo el *namenode* y los *datanodes*) y, por tanto, al directorio *output* generado por la primera parte del sistema. Usando la clase Filtro comentada antes, filtramos los archivos del directorio para obtener aquellos ficheros que contengan la salida de la ejecución, ya que en ese directorio también se almacenan más ficheros que tienen que ver con el éxito del programa, entre otros. Se ha controlado que en el caso de no encontrar los ficheros el programa acabará sin generar excepciones o errores.

En el caso de encontrar los ficheros, extrae el contenido usando el primer método de soporte, luego recorre dicho contenido para extraer cada modelo RDF que aparece y se lo pasa al método que añade el modelo extraído con el modelo final. Una vez que se ha realizado la unión de todos los modelos, se usa el método `write()` para escribir el modelo global en un fichero.

La parte más difícil ha sido la extracción de los modelos. El uso de StringTokenizer y la función Split() de los String ha facilitado la tarea al poder tokenizar el contenido según las variables `start` y `end`, lo que permitía descartar los tokens que tuvieran las claves (los nombres de los documentos). Los detalles a tener en cuenta son:

- Al tokenizar desaparecen las etiquetas al principio y al final de los modelos, por lo que antes de crear el modelo a partir de ese contenido hay que agregarle dichas etiquetas.
- En el caso de que un modelo se quede partido, es decir, una parte del modelo esté en un fichero y el resto en otro, lo primero que se hace es mirar si se da dicha característica comprobando si sólo queda un token con un modelo en el fichero actual y si el índice de aparición de una etiqueta de fin de modelo es más pequeño que el de inicio de modelo en el siguiente fichero. En ese caso, se juntarían el string con el primer fragmento del modelo con el del segundo, y se procedería como siempre.

En la Figura 10 se muestra el diagrama de clases de la aplicación TFG_UnionModels.

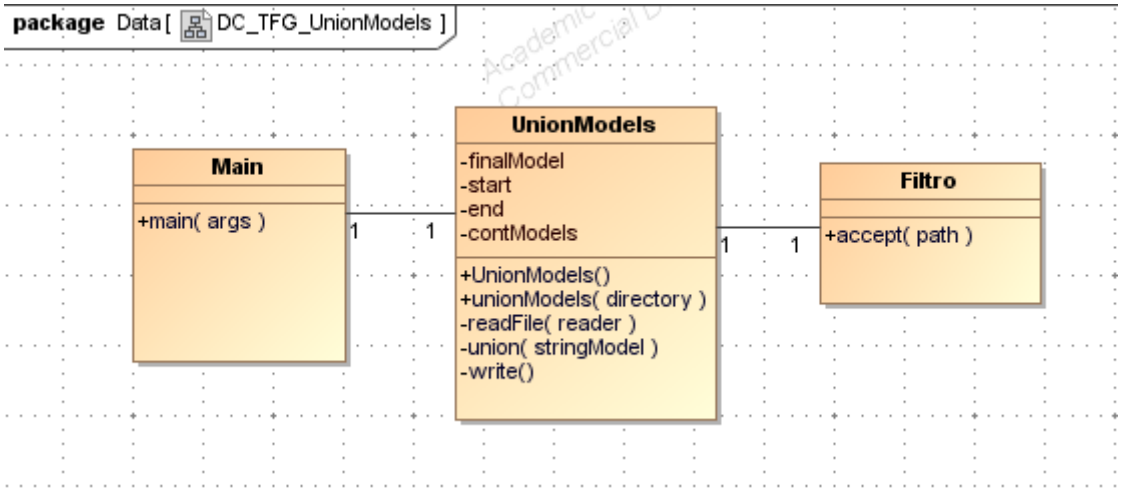


Figura 10. Diagrama de clase de TFG_UnionModels

En el apartado 7.2 se muestra el resultado de la ejecución del sistema completo usando ANNIE como algoritmo a paralelizar.

4.3 Pruebas del sistema y análisis de los resultados

Esta última fase consistió en la limpieza y refactorización del código, y en la realización de las últimas pruebas al sistema. Además, se incluye también el análisis de los resultados obtenidos al ejecutar las aplicaciones.

i. Pruebas del sistema

Se realizó una batería de pruebas finales que consistieron en ejecutar la aplicación TFG_ParallelizationApp (con y sin el *splitting*) con una serie de documentos en inglés, y la aplicación TFG_UnionModels con los resultados de la anterior. Las ejecuciones se realizaban con documentos de diverso tamaño y con caracteres extraños y/o líneas en blanco.

Las otras pruebas realizadas se centraron en el demonio, de tal forma que se probó el correcto funcionamiento de la limpieza (al llegar al límite, que se puso en 3 documentos durante las pruebas) sin alterar demasiado el funcionamiento del *mapper*, evitando el acceso y modificación simultáneo del corpus. También se comprobó que, en el caso de que se cayera un *task* junto con GATE, el demonio volviera a iniciar GATE y a preparar el algoritmo con el corpus.

ii. Análisis de los resultados

Los análisis a realizar consistían en dos:

- Análisis y comparación del tiempo total de procesamiento de los algoritmos de minería de textos al ejecutarlos con y sin paralelización.
- Análisis de la posible pérdida de información al trocear los documentos.

Para la realización del primer análisis se eligieron 4 documentos pequeños para el procesamiento. Por un lado, se calcularon los tiempos en la ejecución del algoritmo de minería de textos sin paralelizar usando un programa que lo ejecutara a través de Eclipse y que generase los modelos RDF de los documentos. Por otro lado, se usó la aplicación TFG_ParallelizationApp para paralelizar dicho algoritmo, usando los mismos textos.

Los tiempos obtenidos se muestran a continuación:

	Tiempo Ejecución
Con paralelización (TFG_ParallelizationApp)	41 segundos

Sin paralelización	16.581 segundos

La diferencia es de 24.5 segundos, aproximadamente. Sin paralelización es más rápido que paralelizándolo ya que al tratar textos pequeños Hadoop es bastante ineficiente. El tamaño mínimo de un bloque de Hadoop es de 64 MB, por lo que no se recomienda ejecutarlo con ficheros de menor tamaño que un bloque, porque no compensa el coste de inicializar y poner en marcha la estructura frente a la ejecución secuencial. Este hecho se ha contrastado con la realización de esta prueba.

Para documentos de mayor tamaño sí compensa paralelizarlo, siempre y cuando se tengan suficientes máquinas. Usando dos nodos, la ejecución se ralentiza hasta el punto de llegar a congelarse, por la falta de recursos. Usando unas 20 máquinas, el tiempo de ejecución se reduce a más de la mitad.

Con respecto al segundo análisis, se ejecutaron las dos aplicaciones desarrolladas, TFG_ParallelizationApp y TFG_UnionModels con dos documentos pequeños. Primero se ejecutó la paralelización sin trocear los documentos usando las clases WholeInputFormat y WholeRecordReader para evitar el *splitting* del texto. Después se realizó la ejecución de las aplicaciones, pero sin el uso de las clases anteriores, es decir, permitiendo que se trocearan los documentos en varios *records*. Finalmente, se compararon los modelos RDF generados en ambas ejecuciones. El resultado no fue el que esperaba, ya que los modelos eran iguales, en lo que se refiere a la clasificación de los *tokens* (las entidades). En teoría, al procesar el documento troceado debería haber pérdida de información para clasificarlas, pero en este caso no fue así. Las entidades estaban clasificadas en las mismas categorías que en el modelo de los documentos sin trocear.

La explicación de este resultado reside en el diccionario usado y en el tipo de NER del algoritmo usado. ANNIE es un algoritmo tipo NER sencillo, por lo que se podría decir que realmente no usa el contexto del documento sino que se basa en listas estáticas. Con la introducción del LKB Gazetteer tampoco hay mucha diferencia, ya que usa un diccionario limitado debido a la capacidad de las máquinas. Por lo tanto, el reconocimiento de entidades se limita al

diccionario y a las reglas del *tagger*, independientemente del contexto, obteniendo como resultado la misma clasificación.

A pesar de que las entidades se clasificaban en la categoría correcta, la estructura, en lo que se refiere a párrafos y sentencias, era en ciertos casos diferente. En la tabla siguiente se muestran las diferencias encontradas en la estructura:

Texto	Modelo Sin trocear	Modelo Troceado
Oracle's Larry Ellison has long threatened to acquire IBM set to take over Canadian software maker Cognos. SAP no longer plans to buy Business Objects.	Paragraph 172	Paragraph 72 Paragraph 21 Paragraph 22

En el modelo generado por los documentos enteros se diferencian los párrafos según si hay una línea en blanco diferenciándolos, es decir, dos saltos de línea. Sin embargo, en el modelo de los documentos troceados, se diferencian según si existe el retorno de carro. Esto se debe a la implementación del *RecordReader* del *TextInputFormat* que, tal y como se explicó en este capítulo, genera un *record* por cada salto de línea. Por eso se generan tres párrafos en vez de uno.

Como resultado de este análisis, se concluyó que para algoritmos que no usen el contexto o que lo usen de forma mínima es recomendable trocear los documentos, porque no se pierde información a la hora de clasificar las entidades y se obtiene mayor rendimiento. A pesar de que podría crearse más anotaciones de párrafos que los que existen, para ciertas actividades ésto no supone un problema al no usarlos, como por ejemplo para la clasificación de textos según las palabras claves.

En el capítulo 7 se encuentran los ficheros utilizados para hacer los análisis. El resto de los materiales se encuentran en el CD adjuntado junto con la memoria.

4.4 Consideraciones

En este apartado se detallan las consideraciones que se han tenido en cuenta durante el desarrollo de las aplicaciones y las restricciones para el buen funcionamiento de las mismas:

- Los documentos para procesar, así como el algoritmo NER usado deben estar en el HDFS y debe estar disponible durante la ejecución de ambas aplicaciones.
- No debe existir una carpeta para el *output* en el HDFS (con el mismo nombre elegido por el usuario), ya que al ejecutar el *job* se crea automáticamente.
- Es recomendable que los documentos estén en inglés y en formato txt.
- El directorio que se le pasa como argumento a la aplicación TFG_UnionModels debe contener los ficheros generados por Hadoop (con nombre part-r-xxxxxx) y éstos deben tener como contenido pares <nombreDocumento, modelo RDF>. Se ha considerado el caso en el que el directorio no tenga dichos ficheros, cuyo resultado sería la terminación del programa sin generar ninguna salida.
- La aplicación TFG_ParallelizationApp funciona para la versión YARN (2.2.0), pero no se asegura su correcto funcionamiento en versiones anteriores.
- Hay rutas necesarias que deben existir para el buen funcionamiento del sistema completo:

Local:

-/usr/local/hadoop/hadoop-2.2.0/etc/hadoop/core-site.xml

-/usr/local/hadoop/hadoop-2.2.0/etc/hadoop/hdfs-site.xml

HDFS:

-/user/hadoop/application.zip

Para cumplir dicho requisito se recomienda crear un usuario en la máquina local y en el HDFS llamado hadoop, que gestione todo lo relacionado con dicho *framework*. La instalación de Hadoop se

recomienda hacerlo directamente en el directorio /usr/local/hadoop, para no tener que mover los ficheros.

En el apartado Manual de usuario del capítulo 7 se especifican los comandos para usar las aplicaciones y algunos detalles a tener en cuenta para su funcionamiento.

5. Conclusiones y trabajos futuros

En esta sección se expondrán las conclusiones alcanzadas durante el desarrollo de este trabajo y se listarán una serie de posibles trabajos de fin de grado o extensiones del mismo.

5.1 Conclusiones

Este trabajo se engloba en la línea *Gestión y Análisis de Datos*. Dicha línea tiene una gran importancia actualmente gracias a los grandes avances del *Big Data* y a la creciente necesidad de procesar los datos de textos no estructurados. Por ello, su campo de estudio es muy amplio, abarcando una considerable cantidad de técnicas diferentes.

El objetivo principal de este TFG es la paralelización de algoritmos de minería de textos usando la filosofía *MapReduce* de Hadoop para solucionar dos problemas: permitir usuarios concurrentes y mejorar los tiempos de ejecución.

Para alcanzar dicho objetivo, el desarrollo del sistema se estructuró en varias etapas. Se destaca la primera de ellas, la elección y configuración del sistema, ya que en ella se realizó la mayor parte del estudio del arte y la primera toma de contacto con el framework Hadoop. La filosofía *MapReduce* resultó ser muy intuitiva y fácil de implementar, incluso en varios lenguajes, usándose en múltiples aplicaciones de diversos sectores. Esta etapa fue la que más tiempo requirió, seguida del análisis y construcción del sistema.

Como resultado final han sido obtenidas dos aplicaciones que permiten la paralelización de algoritmos NER, extrayendo las anotaciones sobre entidades que generan estos algoritmos a partir de un corpus y creando un modelo RDF global con la información de todas esas anotaciones. Dicho modelo podrá ser usado por otras herramientas o aplicaciones, especialmente por aquellas centradas en la Web semántica.

Por lo tanto, el sistema final ha sido satisfactorio, puesto que se han implementado y cumplido todos los requisitos. Sin embargo, durante la realización de dicho trabajo se ha llegado a la conclusión de que no es recomendable usar la paralelización con Hadoop cuando los documentos de entrada son muy pequeños (aproximadamente menores a un bloque, es decir, 64MB) o cuando hay pocas máquinas (dos, por ejemplo), ya que prácticamente no hay beneficios en lo que a tiempo de ejecución se refiere.

5.2 Trabajos futuros

Para finalizar se proponen una lista de propuestas para extensiones de este TFG o futuras líneas de TFGs relacionadas:

- Realización de una interfaz gráfica para la aplicación, permitiendo el uso intuitivo de ésta y la posibilidad de ejecutar el programa de la unión de modelos, mostrando el resultado final por pantalla. También sería recomendable mostrar la información relevante de la ejecución de Hadoop y facilitar al usuario decidir algunos parámetros de configuración.

- Ampliar las aplicaciones para permitir la paralelización y la generación de modelos RDF de algoritmos NPL no desarrollados en GATE.

- Basarse en ontologías y BD web o propias para la generación de modelos RDF, independientemente del algoritmo y el *gazetteer* introducido.

- Desarrollar un algoritmo completo de minería de textos usando este desarrollo como parte del subsistema para su paralelización.

- Implementar la paralelización de algoritmos usando otro framework de desarrollo, y comparar los resultados generados. Se podría extender a la paralelización en la nube.

- Paralelizar la aplicación encargada de la unión de todos los modelos RDF (TFG_UnionModels).

6. Bibliografía

[1] «Bioledge,» [En línea]. Available: http://www.bioledge.eu/top.html.

[2] D. A. Cockburn, «Using Both Incremental and Iterative Development,» 2008. [En línea]. Available: http://www.crosstalkonline.org/storage/issue-archives/2008/200805/200805-Cockburn.pdf.

[3] «Text Mining,» [En línea]. Available: http://en.wikipedia.org/wiki/Text_mining.

[4] «Unstructured Data and the 80 Percent Rule,» [En línea]. Available: http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/.

[5] McKinsey Global Institute, «Big data: The next frontier for innovation, competition, and productivity,» May 2011. [En línea]. Available: http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation.

[6] «Autonomy,» [En línea]. Available: http://www.texttechnologies.com/category/vendors/autonomy/.

[7] «SAS Text Analytics,» [En línea]. Available: http://www.sas.com/en_us/software/analytics.html#text-analytics.

[8] «AeroText,» [En línea]. Available: http://www.rocketsoftware.com/products/rocket-aerotext.

[9] B. Liu, de *Sentiment Analysis and Opinion Mining*, Morgan & Claypool Publishers, 2012, pp. 5-30.

[10] M. R. Mehl, «Quantitative Text Analysis,» [En línea]. Available: http://dingo.sbs.arizona.edu/~mehl/eReprints/Text%20analysis%20Handbook.pdf.

[11] J. Carbonell, «El procesamiento del lenguaje natural, tecnología en transición,» [En línea]. Available: http://cvc.cervantes.es/obref/congresos/sevilla/tecnologias/ponenc_carbonell.htm.

[12] A. Suárez y M. Palomar, «Desambiguación del sentido y del dominio de las palabras con modelos de probabilidad de Máxima Entropía.,» 5 Mayo 2002. [En línea]. Available: http://journal.sepln.org/sepln/ojs/ojs/index.php/pln/article/view/3303/1792.

[13] X. N. T. L. M. W. W. W. Tanabe L, «GENETAG: a tagged corpus for gene/protein named entity recognition.,» 2005. [En línea]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15960837.

[14] E. J.-R. V. L. S. G. R. B. D. R.-S. Antonio Jimeno, «Assessment of disease named entity recognition on a corpus of annotated sentences,» [En línea]. Available: http://www.biomedcentral.com/1471-2105/9/S3/S3.

[15] «Gate,» [En línea]. Available: https://gate.ac.uk/.

[16] D. M. K. B. I. R. Hamish Cunnigham, «ANNIE: a Nearly-New Information Extraction System,» de *Developing Language Processing Components with GATE Version 8*, pp. 117-137.

[17] A. R. M. L. Bo-Christer Björk, «Scientific journal publishing: yearly volume and open access availability,» 2009. [En línea]. Available: http://www.informationr.net/ir/14-1/paper391.html.

[18] Oracle, «Information Management and Big Data,» [En línea]. Available: http://www.oracle.com/technetwork/topics/entarch/articles/info-mgmt-big-data-ref-arch-1902853.pdf.

[19] «Apache Hadoop,» [En línea]. Available: http://hadoop.apache.org/.

[20] A. Chauhan, «Master Slave architecture in Hadoop,» 2012. [En línea]. Available: http://blogs.msdn.com/b/avkashchauhan/archive/2012/02/24/master-slave-architecture-in-hadoop.aspx.

[21] T. White, «The Hadoop Distributed Filesystem,» de *Hadoop: The Definitive Guide*, O'Reilly Media / Yahoo Press, 2012, pp. 45-83.

[22] R. Merriman, «Massive Performance Gains and Cost Savings using Hadoop,» 2011. [En línea]. Available: http://blogs.avalonconsult.com/blog/enterprise-web/cloud-computing/massive-performance-gains-and-cost-savings-using-hadoop/.

[23] Salesforce, «¿Qué es Cloud Computing?,» [En línea]. Available: http://web.archive.org/web/20121130221321/http://www.itnews.ec/marco/000035.aspx.

[24] R. N. C. S. B. M. A. S. N. R. B. Marcos D. Assuncao, «Big Data Computing and Clouds: Trends and Future Directions,» 2013. [En línea]. Available: http://arxiv.org/pdf/1312.4722v2.pdf.

[25] «IBM Cloud,» [En línea]. Available: http://www.ibm.com/cloud-computing/es/es/.

[26] «Amazon Web Services,» [En línea]. Available: http://aws.amazon.com/es/.

[27] «Resource Description Framework (RDF),» [En línea]. Available: http://www.w3.org/RDF/.

[28] «BDpedia,» [En línea]. Available: http://dbpedia.org/About.

[29] E. M. M. Rodríguez, «RDF: Un modelo de metadatos flexible para las bibliotecas digitales del próximo milenio,» [En línea]. Available: http://www.cobdc.org/jornades/7JCD/1.pdf.

[30] «W3C - Web Semántica,» [En línea]. Available: http://www.w3c.es/Divulgacion/GuiasBreves/WebSemantica.

[31] «Jena,» [En línea]. Available: http://jena.apache.org/.

[32] «Maven,» [En línea]. Available: http://maven.apache.org/.

[33] «Wiki Hadoop Windows,» [En línea]. Available: https://wiki.apache.org/hadoop/Hadoop2OnWindows.

[34] T. White, «YARN (MapReduce2),» de *Hadoop: The Definitive Guide*, O'Reilly Media/Yahoo Press, 2012, pp. 194-200.

[35] «Cloudera,» [En línea]. Available: http://www.cloudera.com/content/cloudera/en/home.html.

[36] «Hortonworks,» [En línea]. Available: http://hortonworks.com/.

[37] T. White, Hadoop: The Definitive Guide, O'Reilly Media/Yahoo Press, 2012.

[38] «JAPE,» [En línea]. Available: https://gate.ac.uk/sale/tao/splitch8.html.

[39] «LKB Gazetteer,» [En línea]. Available: https://confluence.ontotext.com/display/KimDocs37EN/Large+Knowledge+Base+(LKB)+gazetteer.

[40] «SPARQL,» [En línea]. Available: http://www.w3.org/TR/rdf-sparql-query/.

[41] «Eclipse Juno,» [En línea]. Available: http://www.eclipse.org/juno/.

7. Anexos Técnicos

Los documentos mencionados en este capítulo se encuentran en la carpeta *Documentacion/Anexos* del CD aportado con la memoria.

7.1 Manual de usuario

Para usar el sistema Hadoop debe estar instalado en las máquinas en las que se vaya a ejecutar y los servicios deben estar activos. En uno de los nodos masters se lanzará la aplicación TFG_ParallelizationApp, utilizando el comando **yarn jar**:

```
[hadoop@masterhadoop ~]$ yarn jar /home/hadoop/TFG_ParallelizationApp-1.0.jar ga
te.hadoop.HadoopGateDriver documents output
```

Los argumentos de dicho comando son:

- El jar de la aplicación. La ruta se debe ajustar a la localización de este jar.
- Clase principal que ejecuta el *job*. No es variable.
- Documentos o directorio de entrada. Es variable y se deben encontrar en el HDFS.
- Directorio de salida. No debe existir en el HDFS.

Los ficheros resultantes de la ejecución se almacenan en el directorio que se haya especificado, en el HDFS. El zip con el algoritmo de minería de textos que se vaya a paralelizar debe subirse al HDFS con el nombre “application.zip” en la ruta “/user/hadoop”.

La segunda aplicación, TFG_UnionModels, es un poco más restrictiva. Se puede ejecutar el proyecto desde un IDE como Eclipse o el jar desde una consola de comandos.

```
java -jar TFG_UnionModels.jar hdfs://masterhadoop.local:54310/user/hadoop/salida3DocPeq
```

El argumento que se le introduce es la ruta del directorio resultante en la ejecución anterior. La restricción que tiene es que dos ficheros de configuración deben estar en una ruta determinada, concretamente:

-/usr/local/hadoop/hadoop-2.2.0/etc/hadoop/core-site.xml
-/usr/local/hadoop/hadoop-2.2.0/etc/hadoop/hdfs-site.xml

Este programa devuelve un fichero llamado “Modelo-Final.txt” con el modelo RDF generado a partir de todos los modelos del directorio de salida de la anterior aplicación.

La ruta del directorio debe escribirse con la siguiente estructura (sin las comillas):

```
"hdfs://masterhadoop.local:54310/user/hadoop/output"
```

- La máquina y el puerto donde se accede al HDFS.
- La ruta del directorio en el HDFS

7.2 Resultados del sistema usando ANNIE

Un ejemplo de los resultados de la ejecución del sistema completo se puede encontrar en la ruta *Analisis/Segundo/Troceado*. También se pueden encontrar en el directorio *Codigo*.

- **TFG_ParallelizationApp**

En el documento *salidaAnalisis2Troceado/part-r-00000*. Es el fichero generado por la ejecución de Hadoop.

- **TFG_UnionModels**

En el documento *Modelo-Final.txt.* Este es el fichero que reúne todos los modelos generados por los diferentes ficheros de salida de Hadoop en un único modelo.

7.3 Ficheros utilizados para el análisis

Los ficheros mencionados se encuentran en el CD aportado junto con la memoria.

Análisis 1

En la ruta Analisis/Primero.

- Ejecución Standalone. En el documento *ExecutionResult_Standalone.txt*
- Ejecución Paralelizada. En el documento *Analisis1DocsPeqs.txt*

Análisis 2

En la ruta *Analisis/Segundo*.

- Modelo Sin trocear. En el documento *Sin trocear/Modelo-Final.txt.*
- Modelo Final Troceado: En el documento *Troceado/Modelo-Final.txt.*