UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DESARROLLO DE UNA HERRAMIENTA INTERACTIVA PARA LA AUTOMATIZACIÓN DEL PROCESO DE DECOMISIONADO DE SISTEMAS DATAWAREHOUSE

José María De Gregorio Domínguez
Tutor: Sinesio David Carvajal Tabasco
Ponente: Fernando Díez Rubio

ENERO 2017

DESARROLLO DE HERRAMIENTA INTERACTIVA PARA LA AUTOMATIZACIÓN DEL PROCESO DE DECOMISIONADO DE SISTEMAS DATAWAREHOUSE

AUTOR: José María De Gregorio Domínguez
TUTOR: Sinesio David Carvajal Tabasco

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero de 2017

Resumen (castellano)

En la actualidad son muchas las voces que afirman que nos encontramos en la era del “Big Data”. En el sector de las Tecnologías de la Información y la Comunicación (las ya más que famosas TICs) gran parte de las compañías se enfrentan a proyectos que tratan una cantidad de datos tan grande (y el volumen sigue creciendo constantemente) que supera la capacidad del software convencional para ser capturados, administrados y procesados en un tiempo razonable. La demanda de nuevos enfoques para tratar y resolver esta cuestión es lo que conocemos como Big Data. Las herramientas que engloban este ecosistema permiten el almacenamiento, proceso y tratamiento de los datos masivos de manera rápida y eficiente. Algunos ejemplos de este tipo de herramientas son HDFS (Hadoop Distributed File System), Hive (editor de bases de datos para HDFS), Sqoop (herramienta para la migración de datos) e incluso existen algunas otras herramientas de aprendizaje automático o “machine learning” que permiten realizar por ejemplo clasificaciones y predicciones sobre los datos que se están tratando que pueden resultar de gran utilidad para algunas entidades y compañías. Si bien es cierto que hay que tener cierta prudencia a la hora de implementar un sistema Big Data pues, si los volúmenes de datos que se van a tratar no son lo suficientemente grandes (aunque sea de manera potencial), es muy probable que no merezca la pena hacer uso de esta tecnología y sea más sensato no dejarse llevar por la “moda”.

Este Trabajo Fin de Grado surge por la necesidad que existe en el desarrollo de algunos proyectos relacionados con Big Data de ahorrar costes en tiempo al realizar la migración de los datos desde una base de datos tradicional a un sistema Big Data. Este proceso actualmente se realiza de manera “manual” realizando el transvase de la información tabla por tabla, lo que conlleva como hemos dicho un coste en tiempo a tener en cuenta, este proyecto pretende por tanto automatizar de alguna manera ese proceso de decomisionado para realizarlo con mayor rapidez.

Abstract (English)

Currently, there are many voices that claim that we are in the era of "Big Data". In the Information and Communication Technologies sector (also commonly known as ICT), many companies are facing projects that deal with such a large amount of data (and the volume keep growing constantly) that exceeds the capacity of the Software to be captured, managed and processed in a reasonable time. The demand for new approaches to address and resolve this issue is what we call Big Data. The tools that encompass this ecosystem allow the storage, processing and management of massive data fast and efficiently. Some examples of such tools are HDFS (Hadoop Distributed File System), Hive (a database editor for HDFS), Sqoop (a tool for data migration). There are many other machine learning tools, which allow to perform, for example, classifications and predictions about the data being processed that can be very useful for some entities and companies. Whilst some caution should be exercised when implementing a Big Data system, mainly if the volumes of data to be processed are not large enough (even potentially), it is likely that it

will not merit to make use of this technological ecosystem, being wiser do not get caught up by "trends".

This End-of-Grade dissertation arises because of the need to develop some projects related to Big Data to save costs in time when migrating the data from a traditional database to a Big Data system. This process is currently carried out by hand, carrying out the transfer of the information table by table. As mentioned above, this process entails a cost in time to be taken into account. This project aims to automate the decommissioning process to some extent, optimising said time costs.

Palabras clave (castellano)

Big Data, HDFS, Hive, Sqoop, decomisionado, almacén de datos

Keywords (inglés)

Big Data, HDFS, Hive, Sqoop, decommission, data warehouse

Agradecimientos

A la Universidad Autónoma de Madrid por la gran experiencia vivida y las oportunidades que me ha brindado.

A la Escuela Politécnica Superior y a su equipo docente por el trato humano y la excelente formación recibida, a los cuales profeso gran respeto, cariño y admiración.

A Everis por la oportunidad de trabajar con ellos y realizar este Trabajo de Fin de Grado.

A mi familia que siempre tuvo fe en mí.

A los amigos y compañeros con los que he compartido estos años y me ofrecieron su ayuda desinteresada cuando me hizo falta.

A todos los demás que en mayor o menor medida y de una manera u otra también hicieron su aportación durante este camino.

INDICE DE CONTENIDOS

1 Introducción ... 1
1.1 Motivación ... 3
1.2 Objetivos ... 5
2 Estado del arte ... 7
2.1 Introducción ... 7
2.1.1 Distribuciones que ofrecen Hadoop ... 7
2.2 Tecnologías utilizadas ... 7
2.2.1 HDFS ... 8
2.2.2 Sqoop ... 11
2.2.3 Hive ... 13
2.2.4 MapReduce ... 15
3 Diseño ... 17
3.1 Análisis de requisitos ... 17
3.1.1 Requisitos funcionales ... 17
3.1.2 Requisitos no funcionales ... 18
3.1.3 Selección de las herramientas a utilizar ... 18
3.2 Diseño de la aplicación ... 18
3.2.1 Diagrama de clases ... 19
3.2.2 Patrones de diseño ... 21
3.3 Metodología de trabajo ... 21
4 Desarrollo ... 23
4.1 Entorno de desarrollo ... 23
4.2 Desarrollo incremental del proyecto ... 23
4.2.1 Iteración I ... 23
4.2.2 Iteración II ... 25
4.2.3 Iteración III ... 26
4.3 Funcionalidad final de la aplicación ... 29
4.3.1 Flujo del programa ... 30
4.3.2 Parámetros y fichero de configuración ... 31
4.4 Otras consideraciones ... 31
4.4.1 Balanceo de Sqoop ... 31
4.4.2 Maven ... 32
5 Integración, pruebas y resultados ... 33
5.1 Pruebas unitarias ... 33
5.2 Pruebas de integración ... 33
5.3 Otros aspectos ... 37
6 Conclusiones y trabajo futuro ... 39
6.1 Conclusiones ... 39
6.2 Trabajo futuro ... 39
Referencias ... 41
Glosario ... 43

INDICE DE FIGURAS

FIGURA 1. TIPOS DE DATOS BIG DATA EN LA ACTUALIDAD 1

FIGURA 2. SISTEMA DISTRIBUIDO CON ALMACENAMIENTO DE DATOS ÚNICO 2

FIGURA 3. SISTEMA DISTRIBUIDO DE HADOOP 3

FIGURA 4. FASES DEL "CICLO" BIG DATA 4

FIGURA 6. EJEMPLO DE ALMACENAMIENTO EN HDFS (I) 8

FIGURA 7. EJEMPLO DE ALMACENAMIENTO EN HDFS (II) 9

FIGURA 8. EJEMPLO DE ALMACENAMIENTO EN HDFS (III) 9

FIGURA 9. LECTURA DE ALMACENAMIENTO EN HDFS 10

FIGURA 10. ESQUEMA BÁSICO DE SQOOP 11

FIGURA 11. DIAGRAMA DE FUNCIONAMIENTO DE SQOOP 11

FIGURA 12. DIAGRAMA DE FUNCIONAMIENTO DE SQOOP 2 13

FIGURA 13. ESQUEMA DATOS EN HIVE 14

FIGURA 14. EJEMPLO FLUJO DE DATOS EN MAPREDUCE. 15

FIGURA 15. DIAGRAMA DE CLASES DEL PROYECTO 19

FIGURA 16. DIAGRAMA METODOLOGÍA INCREMENTAL E ITERATIVA 22

FIGURA 17. PRIMERA VISTA DE LA APLICACIÓN 26

FIGURA 18. EJEMPLO CARGA TIPO APPEND CON DUPLICADOS 27

FIGURA 19. EJEMPLO CARGA TIPO INCREMENTAL SIN DUPLICADOS 27

FIGURA 20. PANEL SELECTOR DE CAMPO DE REFERENCIA PARA CARGA INCREMENTAL 28

FIGURA 21. VISTA FINAL DE LA APLICACIÓN 29

FIGURA 22. DIAGRAMA DEL FUNCIONAMIENTO DE LA APLICACIÓN 30

FIGURA 23. SELECCIÓN TABLA "ACCOUNTS" EN LA VISTA Y MÉTODO DE CARGA "FULL" 34

FIGURA 24. VISTA DEL HUE JOB BROWSER CON JOB DE TABLA "ACCOUNTS" PROCESANDO 34

FIGURA 25. VISTA DEL HUE JOB BROWSER CON JOB DE TABLA "ACCOUNTS" FINALIZADO CON ÉXITO 34

FIGURA 26. VISTA DEL DIRECTORIO "ACCOUNTS" EN HDFS CON SUS FICHEROS DE DATOS 35

FIGURA 27. VISTA DE DATOS EN BRUTO DEL PRIMER FICHERO DE "ACCOUNTS" 35

FIGURA 28. VISTA DE LA TABLA ACCOUNTS EN HIVE Y SUS METADATOS 36

FIGURA 29. DATOS DE MUESTRA DE LA TABLA ACCOUNTS EN HIVE 36

FIGURA 30. COMPARACIÓN NÚMERO DE REGISTROS EN TABLA FUENTE CON TABLA HIVE 36

1 Introducción

A día de hoy y aunque prácticamente todo el mundo desarrollado ha oído hablar del denominado "Big Data", son muchas las personas que no terminan de tener un concepto medianamente claro de en qué consiste exactamente. En las siguientes líneas vamos a tratar de aclararlo:

Si nos paramos un momento a pensar en las cantidades de datos e información que se manejan a nivel digital actualmente, nos damos cuenta de que el volumen que estos datos abarcan ha ido creciendo exponencialmente en los últimos años. En 2012 por ejemplo, la cantidad de datos generados anualmente era de 2.8 Zettabytes (10^{21} bytes), de los cuales además el 75% eran generados por los individuos en su uso de la red. Podemos encontrar evidencias de esto en el uso masivo que realizamos de la red ya sea por ejemplo, al descargar archivos, visualizar fotos y videos, conectar el GPS, enviar correos electrónicos o utilizar aplicaciones de mensajería instantánea y un largo etcétera. Se calcula que una persona media genera del orden de 5GB de datos al día. [6]

Tipos de datos a explorar

Vamos a tratar de clasificar todos los tipos de datos que actualmente se gestionan utilizando un enfoque Big Data:

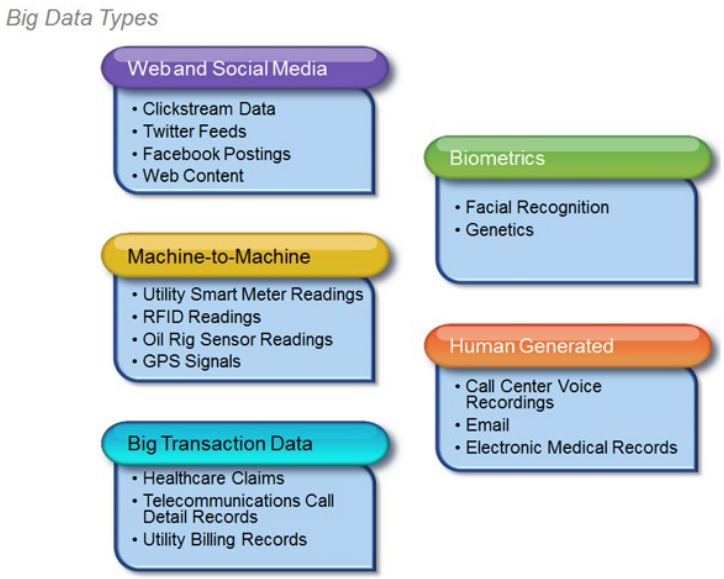

Figura 1. Tipos de datos Big Data en la actualidad

(Fuente: https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/)

- *Web and Social Media*: Información vía web proveniente de redes sociales, blogs, etc.

- *Machine-to-Machine (M2M)*: Comunicaciones entre máquinas y dispositivos como sensores o medidores que registran y transmiten datos a través de la red a otras aplicaciones que los traducen en información inteligible.

- *Big Transaction Data*: Registros de facturación, llamadas, operaciones bancarias, de bolsa, etc.

- *Biometrics*: Información biométrica de personas como huellas digitales, escáner de retina, reconocimiento facial, etc. Estos datos son información muy importante por ejemplo para las instituciones o agencias relacionadas con inteligencia, seguridad y defensa.

- *Human Generated*: Las que son directamente generadas por las personas, como correos y documentos electrónicos, registros de llamadas, notas de voz, etc. [7]

En determinadas aplicaciones el tiempo que puede llevar a un software convencional tratar y procesar esta cantidad de información tan gigantesca excede lo razonable. Por ello, han surgido otras tecnologías que permiten realizar esas tareas en de manera mucho más rápida y esto es lo que se conoce como Big Data, es decir, un ecosistema de nuevas tecnologías y herramientas que permiten almacenar, gestionar y procesar grandes cantidades de información de manera rápida y fluida. Una de estas tecnologías, tal vez la más conocida actualmente es Hadoop, y uno de los grandes pilares son los sistemas distribuidos.

Tradicionalmente, los datos son almacenados en una única localización central. Esta configuración es buena para cantidades limitadas de datos:

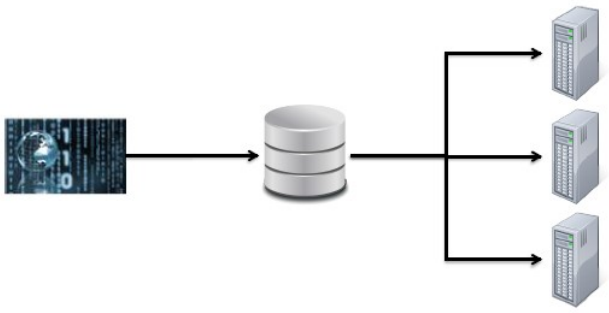

Figura 2. Sistema distribuido con almacenamiento de datos único

Sin embargo, para cantidades muy grandes de datos (muchos de los sistemas modernos y cada vez más) necesitamos otro enfoque puesto que el anterior supone un cuello de botella en el procesamiento y, por tanto, se pierde mucho tiempo.

Hadoop es una de las posibles soluciones para esto:

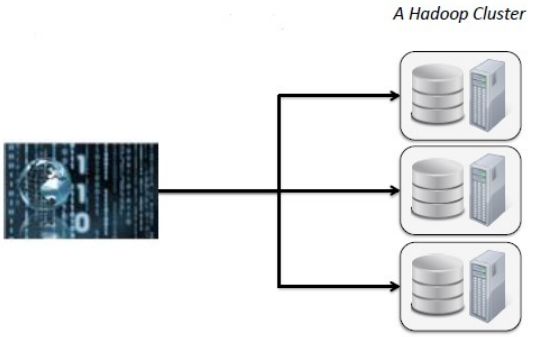

Figura 3. Sistema distribuido de Hadoop

Su enfoque consiste en llevar el programa a los datos más que los datos al programa. Sus dos conceptos clave son distribuir los datos cuando son almacenados y realizar el procesamiento donde están los datos. En el apartado de estado del arte veremos más en detalle cómo funciona esto.

Las tres "V" de Big Data

Como hemos visto, además del gran **volumen** de información, existe también una gran **variedad** de datos a tratar y que no siempre tendrán el formato específico que nos gustaría por ejemplo para almacenar en una base de datos relacional, sino que la variedad de formatos en los que llegará la información será también enorme. Sin embargo, las aplicaciones que realicen un análisis de todos estos datos requerirán que la **velocidad** de respuesta y procesamiento sea lo suficientemente alta para lograr obtener la información correcta en el momento adecuado. [7] [9]

Son precisamente estas tres necesidades o características (volumen, variedad y velocidad) las que nos sugieren la conveniencia de aplicar un paradigma Big Data, las cuales son sus principales ventajas.

1.1 Motivación

En la siguiente imagen vemos el proceso de un proyecto Big Data:

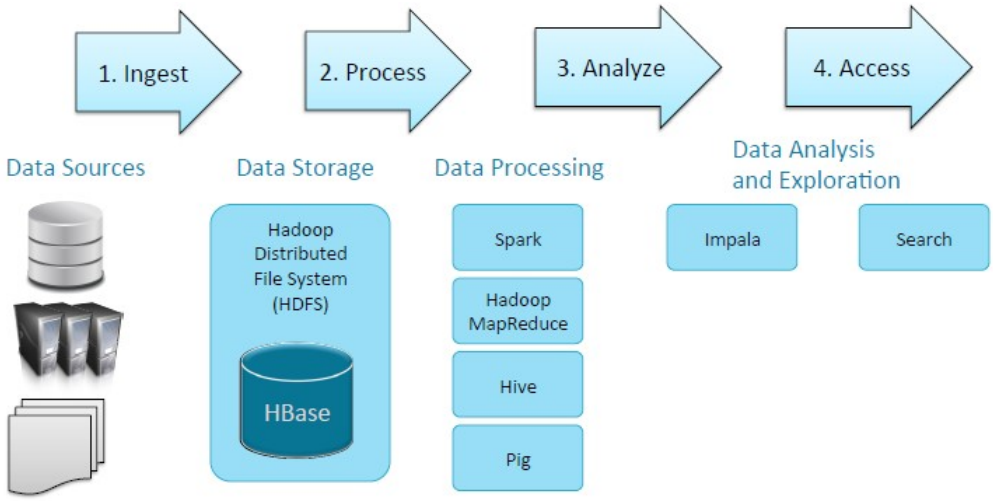

Figura 4. Fases del "ciclo" Big Data

Observamos sus distintas fases ordenadas y algunas de las herramientas que pueden utilizarse en cada una de ellas. La parte que compete a este TFG se fundamenta en la primera fase, es decir, la fase de ingesta. Hadoop, por su parte, puede extraer datos de distintas fuentes y en distintos formatos, por ejemplo, sistemas tradicionales de gestión de datos (como bases de datos), logs, ficheros importados, etc.

El tiempo que se tarda en realizar esta fase de ingesta de forma "manual" es elevado, por tanto, la motivación del TFG es precisamente esa, el desarrollo de una herramienta que permita automatizar este proceso para reducir los costes en tiempo de cualquier proyecto relacionado con esto. Concretamente en Everis (la empresa donde se ha realizado el TFG) ya existía una herramienta que funcionaba a base de scripts y que permitía automatizar parte de este proceso de ingesta. No obstante, el ámbito de aplicación de esta herramienta y su usabilidad eran bastante limitadas, ya que simplemente ayudaba al proceso de migración de los datos puro y duro, pero no creaba previamente un entorno de tablas con un formato adecuado en el sistema Big Data y además no disponía de una interfaz gráfica, lo que dificultaba el manejo de la herramienta por parte del usuario, especialmente si la cantidad de tablas presentes en el sistema a decomisionar era muy elevada. (Había que pasar cada tabla en un parámetro diferente)

Data Warehouse

Un data warehouse (o almacén de datos) es una colección de datos no volátil y variable en el tiempo que se orienta a un determinado ámbito, como una empresa, organización, institución, etc. y almacena todos los datos relacionados con la misma para ayudar en los procesos de toma de decisiones de la entidad y que además está diseñada y estructurada de tal manera que favorece el análisis y la divulgación eficiente de datos. Ralph Kimball, conocido autor en temas de data warehouse lo define como "una copia de las transacciones de datos específicamente estructurada para la consulta y el análisis". [8]

Estos almacenes de datos, son principalmente los que pretendemos decomisionar utilizando nuestra herramienta de automatización del proceso. Por otro lado, es importante entender que tanto los data warehouse como las bases de datos convencionales son una parte importante y relevante para una solución analítica y se vuelven mucho más útiles cuando se

utilizan conjuntamente con una plataforma Big Data. Es por ello, por lo que nuestra herramienta será de gran utilidad, especialmente cuando en lugar de realizar un sólo decomisionado, se utilicen en conjunto estas dos plataformas y el proceso de migración de datos entre ellas se realice con frecuencia.

1.2 Objetivos

El proyecto realizado en este TFG se propuso para realizar la ingesta concretamente de bases de datos tipo Oracle, pero se indicó que debía tenerse en cuenta a la hora de desarrollarlo, que en el futuro pudiera contemplar otros tipos de bases de datos y quizá también la ingesta de datos para otros tipos de fuentes que no fueran bases de datos. Todo esto se ha contemplado en el diseño, como veremos más adelante.

Por otro lado, una vez clara esta idea principal, para llevarla a cabo, la aplicación debía realizar los siguientes pasos:

En primer lugar, extraer los "metadatos" (nombres, tipos y longitud de tipo de cada columna) de todas las tablas presentes en la base de datos fuente que quisieran migrarse. Después de esto, "mapear" los tipos de datos de cada columna a los tipos de datos correspondientes en las futuras tablas que se crearan en el sistema Big Data (concretamente en Hive, también veremos ahora qué es y en qué consiste) y crearlas.

Por último, una vez estas tablas estuvieran creadas, debía realizarse la migración de los datos de las tablas de la base de datos fuente a estas tablas del sistema Big Data de manera que (teniendo en cuenta que el volumen de los datos a migrar será bastante elevado) este proceso se llevara a cabo utilizando paralelización o "multi-threading" para que los tiempos sean razonables.

Metodología ETL

Si nos fijamos en los pasos que hemos descrito, nos damos cuenta de que en el fondo estamos siguiendo una metodología ETL (*Extract, Transform and Load*) para el proceso de decomisionado. Veamos una definición para este proceso:

"Los procesos ETL implican las siguientes operaciones:

- **Extracción**: Acción de obtener la información deseada a partir de los datos almacenados en fuentes externas.
- **Transformación**: Cualquier operación realizada sobre los datos para que puedan ser cargados en el *data warehouse* o se puedan migrar de éste a otra base de datos.
- **Carga**: Consiste en almacenar los datos en la base de datos final, por ejemplo el almacén de datos objetivo normal." [8]

En nuestro caso, nosotros **extraemos** los metadatos que nos interesan de la fuente, los **transformamos** para obtener los tipos de datos correspondientes para crear las tablas equivalentes en Hive y finalmente **cargamos** la información desde la fuente en estas tablas destino.

2 Estado del arte

2.1 Introducción

El mercado del almacenamiento y procesamiento de grandes volúmenes de datos está directamente ligado a Hadoop. [1]
Hadoop es un framework de software libre que soporta aplicaciones distribuidas. Permite a las aplicaciones trabajar con miles de nodos y petabytes (10^{15} bytes) de datos. En sus inicios se inspiró en los Google Docs para MapReduce y Google File System. Actualmente es un proyecto de alto nivel que está siendo construido y usado por una comunidad global de contribuyentes (siendo Yahoo el mayor de ellos) utilizando el lenguaje de programación Java. [2]

El núcleo del framework de Hadoop está compuesto principalmente por los siguientes módulos:

- *Hadoop Common:* Contiene librerías y servicios que utilizan otros módulos de Hadoop.
- *Hadoop Distributed File System (HDFS):* Un sistema de archivos distribuidos que proporciona un elevado ancho de banda a través del clúster.
- *Hadoop YARN:* Una plataforma de gestión de recursos responsable de administrar los recursos de computación del clúster.
- *Hadoop MapReduce:* Una implementación del modelo de programación "MapReduce" para procesamiento de datos a gran escala. [3] [10]

2.1.1 Distribuciones que ofrecen Hadoop

Con el fin de facilitar la configuración e instalación de Hadoop, existen diversas distribuciones open-source, dos de las más conocidas hoy en día son Cloudera y Hortonworks.

En general, estas distribuciones no sólo incluyen el núcleo de Hadoop (HDFS, MapReduce, etc.) sino que además también integran otros proyectos de Apache que son un buen complemento (HBase, Mahout, Hive, etc.). Suelen contar además con otras características que facilitan mucho el trabajo, Cloudera por ejemplo, cuenta con una interfaz gráfica llamada Cloudera Manager que es útil para la administración y gestión de los nodos del clúster de Hadoop. [1]

2.2 Tecnologías utilizadas

En esta sección vamos a describir cómo funcionan y para qué se utilizan algunas de las tecnologías del ecosistema Big Data que hemos utilizado y están directamente relacionadas con este proyecto.

2.2.1 HDFS

Como hemos visto en las páginas anteriores, HDFS es el *Hadoop Distributed File System*, es decir, el sistema de ficheros distribuidos de Hadoop. Veamos ahora más en detalle cómo funciona:

En primer lugar, habría que mencionar, aparte de lo que ya hemos visto, algunos conceptos básicos. Como hemos dicho, HDFS está basado en el sistema de ficheros de google (GFS) y escrito en Java. Proporciona almacenamiento redundante para cantidades de datos masivos, en este punto utiliza conceptos tales como el de disponibilidad inmediata y estandarización.

Por otro lado, cabe decir que HDFS funciona mejor para un número alto, pero aun así sin llegar a ser extremo, de ficheros con muchos datos. Para hacernos una idea, cada fichero puede tener típicamente unos 100MB de datos o más. Y los tamaños de bloque idealmente tendrán entre 64 y 128MB de datos (tamaño de bloque por defecto, aunque es configurable) [2] [11]. Vamos a ver todo esto con un ejemplo:

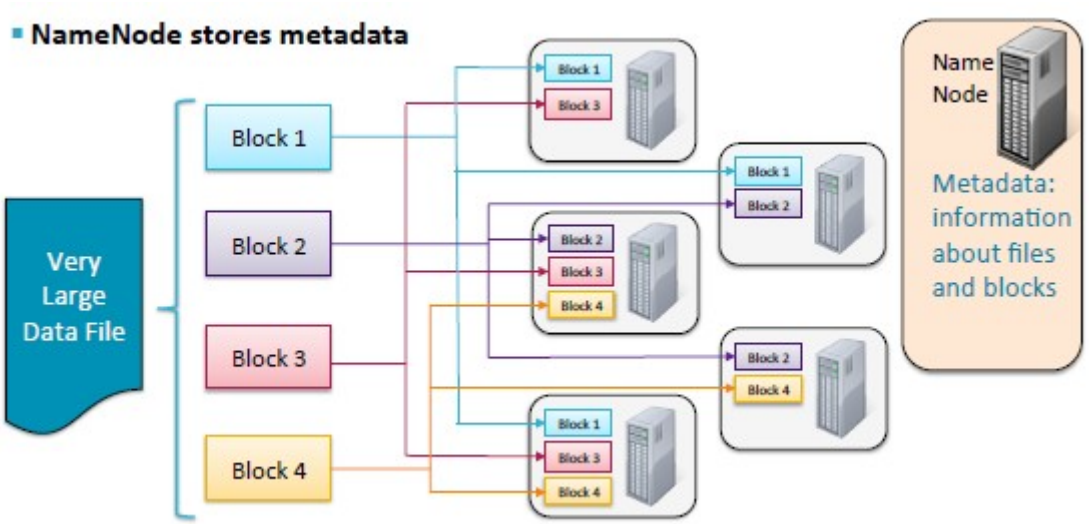

Figura 5. Ejemplo de almacenamiento en HDFS (I)

En esta imagen observamos cómo un fichero grande de datos es particionado en 4 bloques, cada uno de los cuales es almacenado en un nodo del clúster de HDFS y es replicado a su vez en otros dos (este parámetro es configurable). Se dispone además de otro nodo muy importante denominado **NameNode**, cuya función es almacenar "metadatos" acerca de los distintos bloques que componen a cada fichero almacenado y en que nodo se encuentra cada uno de estos bloques. Vamos a verlo más en detalle todavía:

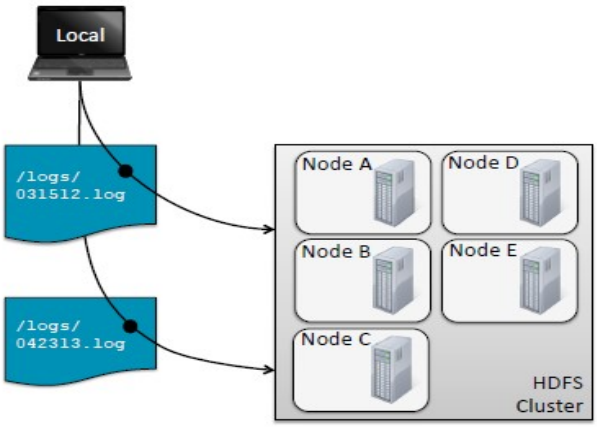

Figura 6. Ejemplo de almacenamiento en HDFS (II)

Aquí vemos como un usuario quiere almacenar dos ficheros (en este caso de tipo .log) en HDFS. Esto quedaría almacenado de la siguiente manera:

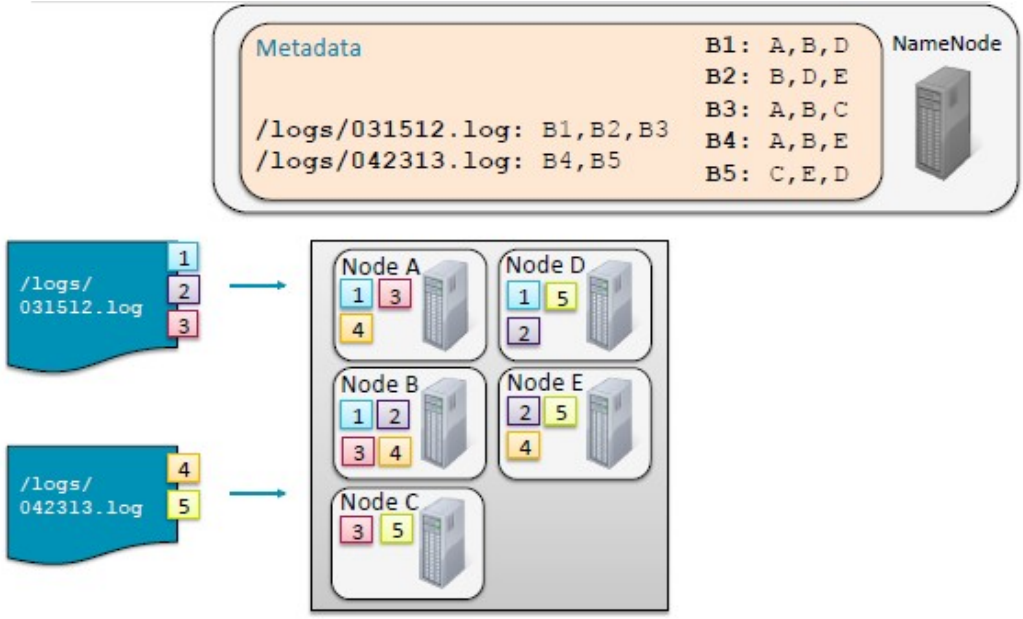

Figura 7. Ejemplo de almacenamiento en HDFS (III)

Observamos como el primer fichero se ha particionado en tres bloques y el segundo a su vez en dos bloques. Cada uno de estos bloques se ha almacenado en un nodo del clúster y se ha replicado en dos nodos más. Por otro lado, en el NameNode que mencionábamos, vemos como está almacenada la información correspondiente a los ficheros subidos, que bloques concretos pertenecen a cada fichero y en qué nodos pueden encontrarse esos bloques.

Ahora, si un cliente quiere consultar la información de uno de esos ficheros, realizaría una llamada al NameNode preguntándole por el fichero en cuestión, este le devolvería los bloques de información correspondientes a ese fichero y donde están alojados, el cliente

los recogería y los volvería a fusionar para obtener la información original completa como vemos en la siguiente imagen:

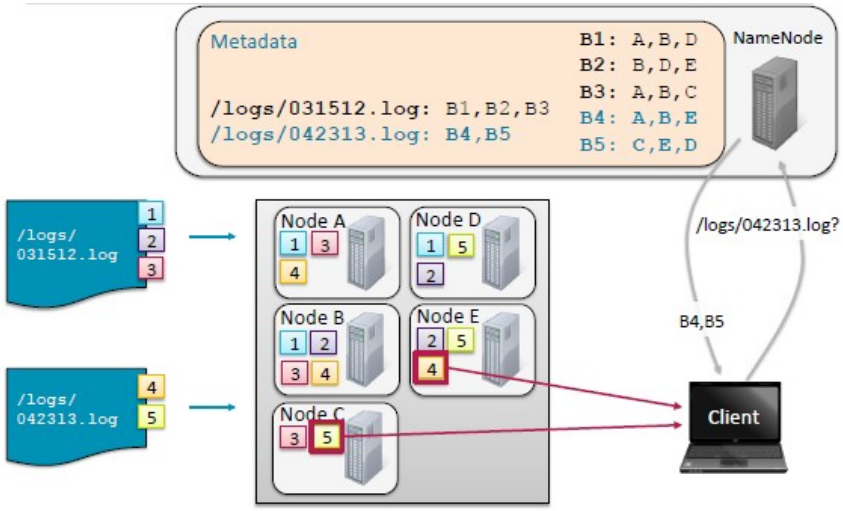

Figura 8. Lectura de almacenamiento en HDFS

Entendiendo todo este proceso y el funcionamiento de la división de los ficheros en bloques y su repartición y replicación en distintos nodos somos capaces de comprender uno de los grandes motivos por lo que la utilización de tecnologías Big Data no es conveniente cuando el volumen de datos a almacenar y procesar no es el adecuado (porque sea demasiado pequeño) ya que si, por ejemplo, almacenáramos en HDFS una cantidad muy grande de ficheros pequeños, los registros del NameNode crecerían exponencialmente y a la hora de buscar y encontrar cualquier información esto generaría una pérdida de eficiencia.

Con respecto al funcionamiento de HDFS es importante destacar también que la política de escritura de HDFS es lo que se conoce como "write once", esto es, en este sistema está permitido crear nuevos ficheros o borrarlos, pero no se permite modificarlos, de este modo cada fichero subido al sistema ha sido escrito una sola vez y no podrá volver a ser escrito a no ser que se modifique desde una fuente y se vuelva a subir a HDFS. Conocer este hecho es importante para entender qué sentido tienen los distintos tipos de carga que pueden realizarse con la herramienta que se ha desarrollado en este proyecto, como veremos más adelante. Tipos de carga de datos:

- Truncate: Cuando sólo me interesa la "última foto". Ej.: BD en la que se realizan modificaciones pero no queremos un histórico.

- Append con duplicados: Cuando en la BD se realizan modificaciones y queremos tener un histórico

- Append sin duplicados: Cuando en la BD no se van a realizar modificaciones en los datos previos y solo queremos traernos los que vayan llegando nuevos)

No es recomendable subir a HDFS muchos ficheros pequeños porque tendría muchas entradas en el Namenode y muchos duplicados de ficheros pequeños en los nodos del

clúster, lo que provocaría que se perdiera eficiencia (una de las grandes razones por la que no es conveniente utilizar Big Data cuando no se necesita).

2.2.2 Sqoop

Sqoop es un proyecto de software libre desarrollado originalmente por Cloudera. Su nombre es una contracción de "SQL-to-Hadoop". La función principal de esta herramienta es el intercambio de datos entre una base de datos (sólo funciona para bases de datos relacionales) y HDFS, es decir, permite la importación de datos desde una base de datos a HDFS y viceversa.

Por cada ejecución, Sqoop permite importar todas las tablas existentes en la base de datos, importar una sola tabla concreta o bien una parte de una tabla. Estos datos pueden ser importados en varios formatos (formatos de texto).

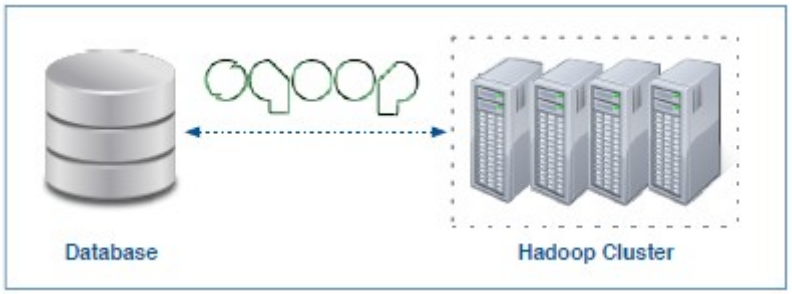

Figura 9. Esquema básico de Sqoop

Sqoop es una aplicación que funciona en el lado del cliente y se sirve de Hadoop MapReduce para realizar la importación. Veamos algo más en detalle que pasos sigue una importación típica de datos utilizando esta herramienta:

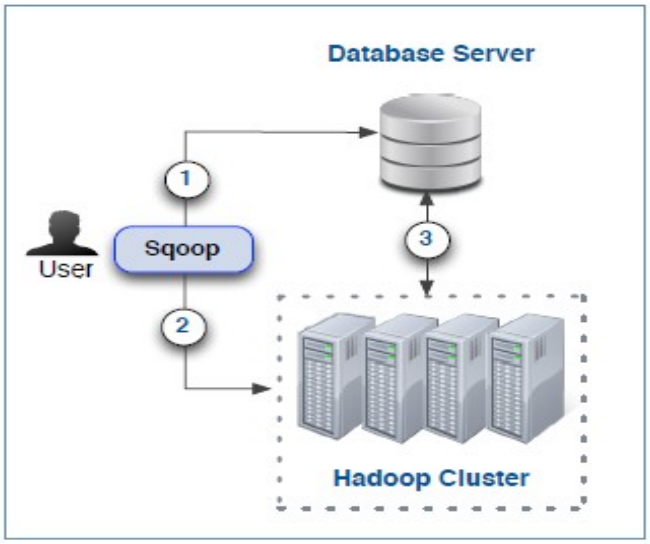

Figura 10. Diagrama de funcionamiento de Sqoop

Como vemos en la imagen, una importación con Sqoop consta principalmente de tres pasos:

1. Examina la tabla seleccionada (o todas las tablas) de la base de datos y sus "metadatos".

2. Crea y prepara un nuevo trabajo en el clúster de Hadoop.

3. Extrae los registros de la tabla y escribe estos datos en HDFS.

Estas migraciones de datos, puede realizarlas de manera paralelizada, y esto es otra de las grandes ventajas que tiene como herramienta Big Data, lo que reduce mucho los tiempos de migración de grandes volúmenes de datos. Por defecto, Sqoop realiza las importaciones utilizando 4 hilos, pero este parámetro puede modificarse (pudiendo utilizar incluso un solo hilo, aunque al hacer esto generalmente pierde gran parte de su sentido) al lanzar la ejecución utilizando el parámetro `–num-mappers` o `–m <número_hilos>` para abreviar. Aumentar el número de hilos puede mejorar la velocidad de la importación de datos, aunque se debe tener cuidado porque también aumentara la carga de procesamiento en el clúster.

Es importante resaltar que para poder utilizar más de un hilo, la tabla fuente cuyos datos van a importarse debe tener definida una `PRIMARY KEY`, si no es así, Sqoop no podrá dividir el trabajo entre los hilos (pues este campo clave es el que utiliza como referencia) y no podrá realizar el reduce de los datos procesados en los distintos hilos cuando termine la ejecución y por tanto fallará, aunque existe otra posibilidad que es indicarle por parámetro que campo queremos que utilice como referencia aunque no sea una clave primaria.

Para hacernos una idea más clara todavía vamos a ver un ejemplo de comando de ejecución de Sqoop:

```
sqoop import --table <nombre_tabla> --connect
jdbc:mysql://<bd_host>/<bd_nombre> --username <bd_usuario>
--password <contraseña> -m 5
```

Este es un ejemplo de ejecución para importar una sola tabla, si quisiéramos por ejemplo importar todas las tablas de la base de datos simplemente valdría con sustituir `import --table <nombre_tabla>` por `import-all-tables`.

Sqoop 2

Sqoop es estable y lleva utilizándose de manera satisfactoria durante años. No obstante, tiene algunas limitaciones, especialmente debido a la arquitectura de la parte del cliente.

Algunas de estas limitaciones son, entre otras, que al conectarse a la base de datos desde el cliente, este debe tener instalados los drivers JDBC necesarios y especificar su usuario y contraseña, además de requerir también conexión directa al clúster de HDFS desde el cliente. Por otro lado, y debido también a todo esto, es difícil de integrar en aplicaciones externas (veremos cómo se ha solucionado para este TFG) y como mencionamos anteriormente, sólo funciona con bases de datos relacionales.

Como solución de estos inconvenientes surge Sqoop 2 que sería la siguiente generación de Sqoop. En este entorno, existe un "Sqoop Server" que actúa de intermediario y el cliente, por su parte, sólo necesita tener conexión a este servidor. Este servidor ya se encarga de tener configuradas y realizar las conexiones tanto a la base de datos como al HDFS y proporcionar las credenciales necesarias, además realiza una mejor gestión de los recursos disponibles y al ser accesible desde otras aplicaciones, facilita mucho su integración en las mismas.

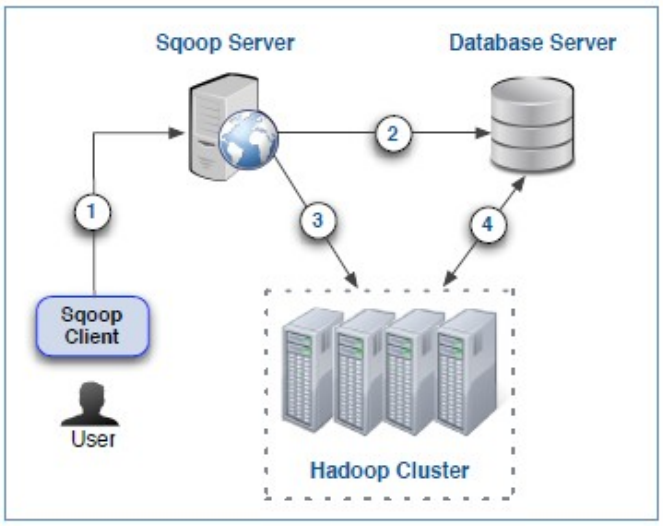

Figura 11. Diagrama de funcionamiento de Sqoop 2

En este proyecto, para tratar de aprender bien la base y al haber conseguido alcanzar una solución factible al problema de la integración, se ha utilizado el Sqoop original, pero no se descarta en el futuro, si la aplicación se utilizara de forma extensiva o frecuente, integrar Sqoop 2 en la misma.

2.2.3 Hive

Hive es un editor de consultas para los datos almacenados en HDFS. Es decir, permite trabajar con los datos almacenados en HDFS como si de una base de datos tipo SQL se tratara (aunque en realidad no lo sea).

Además de Hive existen otros tipos de editores de consultas para el sistema HDFS, uno de los más conocidos es Impala. A grandes rasgos, las principales diferencias entre ambos son que mientras Hive posee más características útiles (por ejemplo, soporta mayor número de tipos de datos y tipos de datos más complejos), Impala por su parte es mucho más rápido que Hive, esto se debe principalmente a que cuando Hive realiza cualquier consulta o proceso, la convierte en un job que se debe ejecutar en el motor de procesos de Hadoop utilizando Jobs de MapReduce, pero Impala realiza todo esto directamente en el clúster de Hadoop.

Las principales razones para utilizar estos editores es porque proporcionan análisis de datos a gran escala, y en los aspectos que les compete, son más productivos que escribir en

MapReduce o Spark (cinco líneas de HiveQL/ImpalaSQL pueden equivaler a doscientas de código Java) y además ofrecen interoperabilidad con otros sistemas, pueden extenderse mediante Java o scripts externos y muchas de las herramientas de Business Intelligence los soportan.

Es importante señalar que el clúster de Hadoop, aunque se trabaje con alguno de estos editores, no es un servidor de base de datos, tal y como lo entendemos de manera tradicional. En Hive podemos crear tablas, realizar consultas, etc. No obstante, hay que tener muy en cuenta que en Hive, aunque sí pueden añadirse nuevos registros, no es posible borrar o actualizar un registro o grupo de registros concreto, por lo que normalmente, siempre que se implementa un sistema Hadoop para dar soporte a algún tipo de trabajo con requerimientos de volúmenes de datos elevados, suele trabajarse a la par utilizándose tanto la base de datos fuente, como el sistema Big Data. De este modo si se quiere realizar por ejemplo alguna actualización de los datos existentes en HDFS en ese momento, habría que o bien borrar todos los datos asociados a una tabla concreta y volverlos a cargar de nuevo desde la base de datos fuente (con el proceso que ya hemos visto) o bien realizar un tipo de carga incremental que añada únicamente los datos nuevos que están presentes en la base de datos pero no en HDFS. Aquí vemos otro aspecto de las ventajas que supone la herramienta que se ha desarrollado en este proyecto, ya que permite automatizar en gran medida también este proceso.

Veamos rápidamente como trabajan estos editores:

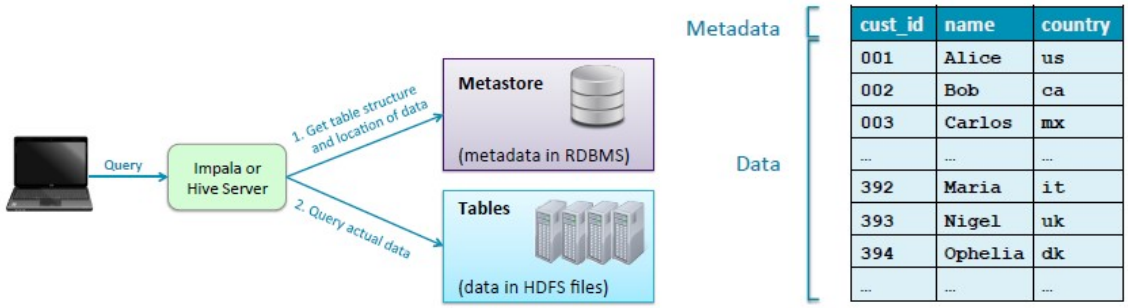

Figura 12. Esquema datos en Hive

Como vemos en estas imágenes, al lanzar una consulta, esta se envía al servidor de Hive o Impala y utiliza por un lado el denominado **Metastore** para obtener información sobre la estructura de la tabla a la que se está accediendo y la localización de sus datos y en segundo lugar realiza la consulta sobre los datos.

Por último, señalamos que existe una forma para que, a la hora de realizar la migración de los datos desde la base de datos fuente a HDFS utilizando Sqoop, este realice la creación de las tablas en Hive directamente, el principal problema de esto, sería que todos los tipos de datos los convertiría en Hive a tipo `STRING`, lo cual no es demasiado útil a la hora de realizar algunas consultas. Por esta razón en nuestro proyecto utilizamos Hive de manera independiente y en este punto lo preferimos a Impala porque nos ofrece como hemos visto mayor capacidad de acción. Más adelante explicaremos mejor todo esto.

2.2.4 MapReduce

MapReduce forma parte principal del núcleo de Hadoop. Este proceso tiene dos fases diferenciadas que ejecuta Hadoop, estas son: *Map*, que toma un conjunto de datos y lo convierte en otro separando los elementos en tuplas (pares clave-valor), y *Reduce*, que recoge las salidas del proceso *Map* como entrada y combina las tuplas en un conjunto más pequeño. Existe también una fase intermedia llamada *Shuffle* que es la encargada de obtener las tuplas generadas por el proceso *Map* y determinar qué nodo las procesará, dirigiendo estos datos de salida a una tarea *Reduce* específica. [7]

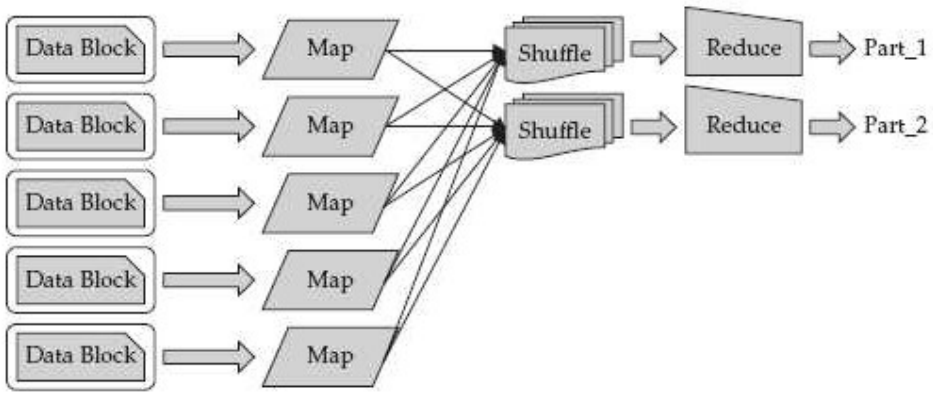

Figura 13. Ejemplo flujo de datos en MapReduce.

(Fuente: https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/)

Para más información sobre todas estas tecnologías puede consultarse el libro citado en la bibliografía: *Hadoop. The definitive guide.* (2012) de Tom White. [10]

3 Diseño

3.1 Análisis de requisitos

En esta sección vamos a tratar sobre el diseño que se llevó a cabo, como parte del proyecto, antes de llevar a cabo el desarrollo y la implementación de la aplicación. Debido a la metodología de trabajo que se ha seguido, como veremos al final de esta sección y a que no existía una propuesta bien definida por parte de la empresa en un principio sino tan sólo una idea general de lo que se quería obtener, el análisis de requisitos no es demasiado detallado. Partiendo de unas especificaciones iniciales imprescindibles, se ha ido añadiendo valor a la propuesta a medida que el proyecto se desarrollaba. Veamos entonces cuales eran los requerimientos obligatorios e imprescindibles desde un principio y en la siguiente sección veremos cómo se fue proponiendo y añadiendo nueva funcionalidad.

3.1.1 Requisitos funcionales

La aplicación debía realizar un decomisionado de un sistema Data Warehouse (entendámoslo como una base de datos) extrayendo los datos existentes en este y llevándolos a un sistema Big Data. Para ello, debía poseer, al menos, la siguiente funcionalidad:

Funcionalidad imprescindible:

- Extraer los metadatos de todas las tablas existentes en la base de datos fuente (entendemos por metadatos la estructura de las tablas, especialmente los tipos de datos de los campos y sus nombres).

- Realizar un "mapeo" de estos tipos de datos, analizándolos y estableciendo los tipos de datos que corresponderían a cada uno de ellos en el editor de consultas del sistema Big Data. (Ej.: un campo tipo `longtext` en una tabla BD se convierte a campo tipo `string` en una tabla en Hive)

- Crear las tablas vacías correspondientes en el editor de consultas del sistema Big Data con los tipos de datos correspondientes previamente mapeados.

- Realizar la migración de los datos de todas las tablas de la base de datos fuente a las tablas del editor de consultas del sistema Big Data y por consecuencia a HDFS. Esta migración de datos debía realizarse mediante multi-threading, es decir, lanzando varios hilos en paralelo.

Propuestas adicionales:

- Se planteaba la posibilidad de poder seleccionar las tablas cuyos datos se iban a migrar (poder elegir todas o elegir sólo algunas).
- Se planteaba la posibilidad de contar con una interfaz gráfica para facilitar esta selección y hacer la aplicación más interactiva y sencilla de utilizar.

3.1.2 Requisitos no funcionales

- La aplicación debía funcionar correctamente para realizar la migración de datos de una base de datos tipo Oracle.

- La aplicación debía ser lo suficientemente escalable y mantenible como para poder añadir con facilidad en el futuro la funcionalidad que permitiera incluir otros tipos de bases de datos fuente a decomisionar.

- La aplicación debía ser fácilmente portable para que pudiera ser utilizada sin problemas por las máquinas que se utilizan en la propia empresa o en la de los clientes.

- El rendimiento y velocidad del decomisionado de datos debía ser suficientemente bueno (tiempos razonables) teniendo en cuenta que iba a trabajarse con volúmenes de datos muy elevados, de ahí que se estableciera que el proceso que corresponde propiamente a la migración de los datos se hiciera utilizando paralelización o procesos multi-hilo.

3.1.3 Selección de las herramientas a utilizar

Teniendo en cuenta todos estos requisitos se establecieron las herramientas y tecnologías con las que se iba a trabajar para el desarrollo del proyecto. En primer lugar y como es evidente se utilizaría el framework de Hadoop y como editor de consultas se seleccionó Hive, principalmente prestando atención a las características expuestas en la sección del estado del arte.

Por otro lado, como lenguaje de programación se eligió Java, debido a que permite una sencilla integración de otras herramientas que serían necesarias, como por ejemplo conexiones JDBC, la portabilidad que ofrece es muy buena y era una de los lenguajes con los que más experiencia y mejor experiencia se contaba y mayor facilidad para el desarrollo y la implementación ofrecía en este caso.

Por último, se decidió utilizar Sqoop como herramienta para el proceso de migración de los datos, debido a que permite realizarlo con relativa facilidad pasándole una serie de parámetros y además también permite realizar este proceso utilizando paralelización e incluso puede configurarse por parámetro el número de hilos que se quieren utilizar para el proceso.

3.2 Diseño de la aplicación

3.2.1 Diagrama de clases

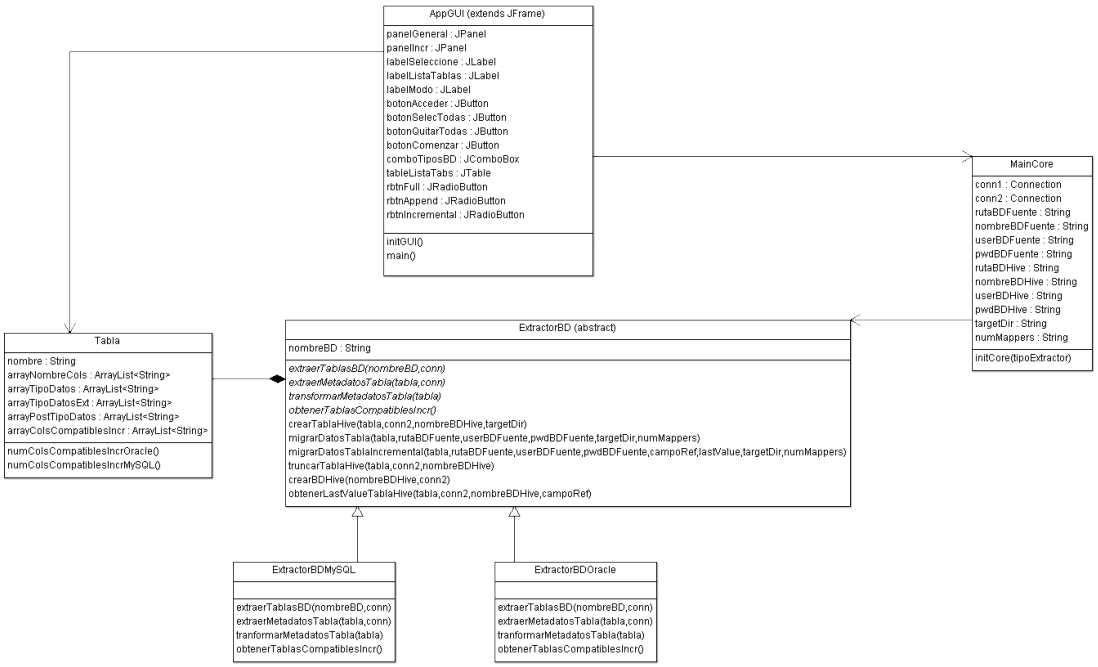

Figura 14. Diagrama de clases del proyecto (Fuente: Elaboración propia)

A la hora de realizarse este diagrama de clases, para hacer un diseño que fuera lo más sencillo y escalable posible, se tuvo muy en cuenta el requisito de que se pudiera añadir con facilidad en el futuro la funcionalidad que permitiera incluir otros tipos de bases de datos fuente a decomisionar. Para ello, se definió la clase abstracta `ExtractorBD`, la cual define como abstractos algunos de sus métodos. Siempre que se deseara implementar un nuevo tipo de base de datos concreto para que fuera compatible con la aplicación, debía implementarse una nueva clase concreta para ese tipo de base de datos fuente que heredara de la clase abstracta `ExtractorBD` (en nuestro caso tenemos implementadas las clases `ExtractorBDMySQL` y `ExtractorBDOracle`, cada una para su tipo de base de datos correspondiente). Los métodos no abstractos de la clase padre contienen funcionalidad que puede ser común para cualquier otro tipo de extractor que herede de ella y de esta manera no hay que implementarlos varias veces, pero los métodos abstractos sí deben escribirse obligatoriamente en todas las clases extractor que hereden de la clase padre `ExtractorBD`.

Por otro lado, esta clase padre está compuesta a su vez por objetos de la clase `Tabla`. Esta clase tabla es suficientemente genérica para englobar la funcionalidad necesaria para recoger los datos de las tablas de los tipos de bases de datos fuente incluidos hasta ahora en la funcionalidad y con alta probabilidad seguiría funcionando sin problemas para cualquier otro tipo de base de datos relacional que pueda añadirse e incluso también para las de tipo No SQL. Esta clase `Tabla` contiene los atributos necesarios para almacenar como hemos dicho la información que necesitamos de las tablas cuyos datos se van a migrar, para ello define un nombre de tabla y varios arrays de `Strings` que recogen los metadatos necesarios (nombre de los campos, tipo de dato de estos y su extensión). Además existen algunos otros que serán útiles para almacenar los metadatos mapeados para la creación de

las tablas en Hive y otro para registrar los campos de la tabla que son compatibles con el tipo de carga incremental (más adelante veremos en qué consiste). A excepción de este último atributo y del que almacena el nombre de la tabla, la ventaja que encontramos en utilizar estos arrays de `Strings` es que se su información se va almacenando siempre ordenadamente, es decir, prestando atención a los índices el primer nombre de campo `arrayNombreColumna(0)` se corresponderá con el primer tipo de dato de campo `arrayTipoDatos(0)`, y del mismo modo con la extensión `arrayTipoDatosExt(0)` y con su tipo de dato mapeado `arrayPostTipoDato(0)` y así sucesivamente. De este modo se establece una correspondencia entre la información contenida en los distintos arrays en base a sus índices, de alguna manera es como si definiéramos una matriz de datos pero por partes. El único inconveniente de esta clase es que debe definirse un método `numColsCompatiblesIncr<tipoBD>()` por cada tipo de base de datos fuente compatible con la aplicación, pero cómo esto fue una de las últimas modificaciones que se llevaron a cabo y además es un solo método y por ahora sólo existen dos tipos de BD compatibles, se ha preferido hacerlo así en lugar de definir una clase `Tabla` específica para cada tipo de base de datos. (Este método obtiene el número de columnas compatibles con el tipo de carga incremental en esa tabla, y es útil para saber si existe al menos una columna compatible, de esta manera, si no es así, al seleccionar la opción de ese tipo de carga en la interfaz, esta sólo mostrará las tablas que sean compatibles con el mismo. El método debe implementarse una vez para cada tipo de BD porque cada una define sus tipos de datos de una manera diferente).

La clase `MainCore` nos es útil para realizar la lectura por fichero de los parámetros de configuración de la aplicación y guardarlos en sus atributos correspondientes, así como inicializar las conexiones por JDBC tanto a la base de datos fuente como a Hive. Además contiene una instancia de la clase `ExtractorBD`. De esta manera se consigue que al realizar llamadas a los métodos de la clase `ExtractorBD` no haya que realizar una nueva conexión por cada ejecución de los distintos métodos, si no que las conexiones necesarias se realizan previamente con el método `initCore()` de la clase `MainCore` y posteriormente, cuando se llama a los métodos de la clase `ExtractorBD` que lo necesitan, se les pasan estas conexiones por argumento.

Por último, la clase `AppGUI` es la encargada de implementar la interfaz gráfica y su funcionalidad. Contiene todos los componentes de la GUI que le son necesarios, así como una instancia de la clase `MainCore` (que contenía a su vez una instancia de la clase `ExtractorBD`). De esta manera se pueden incluir las llamadas a los métodos de estas clases en el código correspondiente a la funcionalidad de la GUI.

Es cierto que hubiera sido mejor práctica haber extraído esa parte de implementación de la funcionalidad de la GUI y haberla incluido en otra clase aparte (a modo de controlador, dejando en `AppGUI` tan sólo la vista) pero en el siguiente apartado veremos por qué se ha dejado así por el momento.

NOTA: Las clases `ExtractorBD`, `Tabla` y `MainCore` poseen los métodos getters y setters correspondientes para sus atributos, pero no se han incluido en el diagrama por simplicidad.

Gracias a este diseño, cualquiera que posea un mínimo conocimiento razonable sobre Java y programación orientada a objetos será capaz de añadir con relativa facilidad la

funcionalidad correspondiente a los tipos de bases de datos fuente nuevos que deseen incluirse. Para ello, únicamente tendrá que implementar una nueva clase que herede de `ExtractorBD` y escribir al menos sus métodos abstractos, añadir el método correspondiente en la clase `Tabla` y añadir un par de líneas en el método `initCore`() de la clase `MainCore` y la opción correspondiente para la selección de ese tipo de BD concreto en el combo Box de la GUI.

3.2.2 Patrones de diseño

Al tratarse de una aplicación no excesivamente compleja ni extensa a nivel de código, no se ha seguido específicamente ningún patrón de diseño como tal, pues dada la relativa sencillez del código se ha considerado que no merecía la pena. No obstante, si se han seguido algunas pautas o directrices útiles que guardan relación con algunos patrones aunque no se hayan implementado en su completitud. Lo más destacable sería el patrón de diseño MVC (Modelo-Vista-Controlador) que se ha tratado de seguir en términos generales separando en clases bien definidas la parte del Core o funcionalidad lógica (modelo) de la aplicación de la interfaz gráfica de usuario (vista).

Por otro lado, cómo la interfaz gráfica era también bastante simple, la parte correspondiente al controlador (lo que mediaría entre la vista y el modelo) se ha integrado directamente en la vista, salvo alguna excepción.

Aun así se considera que sería una buena práctica, si la aplicación sigue creciendo en un futuro, implementar este MVC de manera completa para facilitar la escalabilidad de la aplicación y la integración de nuevas funcionalidades o vistas que se quieran añadir.

3.3 Metodología de trabajo

Durante el desarrollo de este proyecto se ha seguido una metodología iterativa e incremental cuya descripción podemos ver en la siguiente imagen:

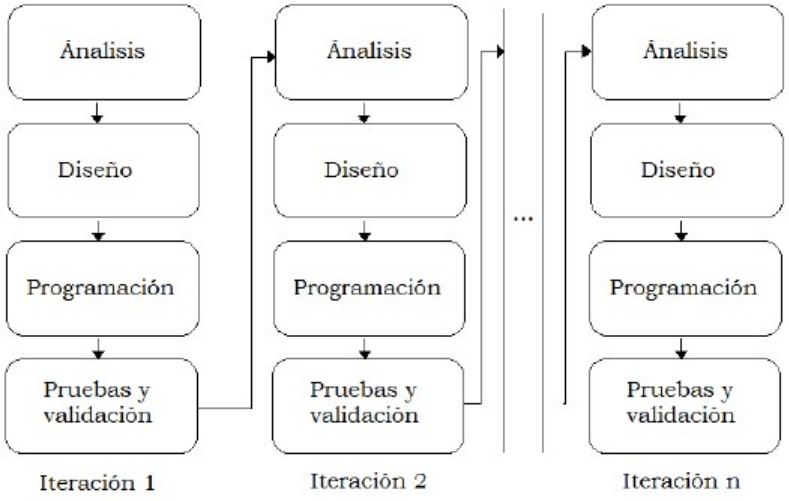

Figura 15. Diagrama metodología incremental e iterativa (Fuente: Google Images)

Esta metodología de trabajo consiste en realizar una o varias iteraciones, cada una de las cuales consta de cuatro fases, estas son:

- Análisis: Se analiza el problema que se quiere resolver y su coste y viabilidad.
- Diseño: Se plantean soluciones para la resolución del problema y se da un enfoque lo más detallado posible de cómo se van a desarrollar esas soluciones.
- Programación: Se implementan las soluciones planteadas en el diseño.
- Pruebas y validación: Se realizan pruebas sobre la nueva funcionalidad implementada y se comprueba si su funcionamiento es el correcto.

En nuestro proyecto, como hemos mencionado, se ha seguido esta metodología para ir añadiendo valor a la propuesta inicial a medida que se iban cumpliendo los objetivos requerimientos mínimos con éxito. Era al final de la fase de pruebas y validación donde si se comprobaba que el trabajo y la funcionalidad añadida en esa iteración era correcta, se realizaban nuevas propuestas de mejora de la aplicación y se pasaba a la siguiente iteración para abordar esas nuevas propuestas.

4 Desarrollo

Como comentamos en la sección anterior, se ha seguido una metodología de trabajo iterativa e incremental, por tanto a lo largo de esta sección comentaremos a grandes rasgos en qué consistieron las iteraciones que aportaron nueva funcionalidad a la aplicación más allá de los requisitos mínimos de la propuesta inicial hasta el producto final y comentaremos también algunos otros aspectos destacables del desarrollo del proyecto.

4.1 Entorno de desarrollo

Como entorno de desarrollo se ha utilizado una máquina virtual distribuida por Cloudera, más concretamente "Cloudera-QuickStart-VM-5.8". Esta máquina dispone del framework de Hadoop instalado (versión 2.6.0-cdh5.8.0) y añade muchas herramientas útiles del ecosistema Big Data, como Hive, Impala, Sqoop, etc. Además ofrece la aplicación web Hue (Hadoop User Experience) para poder visualizar y gestionar gráficamente todo el entorno de Hadoop instalado, lo cual facilita bastante la tarea y también añade el denominado Cloudera Manager que permite gestionar también de manera gráfica todas estas herramientas.

Por nuestra parte, hemos añadido como complemento en la VM los gestores de bases de datos de MySQL y Oracle (además de sus JDBC correspondientes) para trabajar con ellos durante el desarrollo del proyecto y realizar pruebas. También añadimos el entorno de desarrollo Spring Tool Suite (STS) que resultó bastante útil a la hora de crear y desarrollar la aplicación como proyecto de Maven y también hubo que actualizar la versión de Java a la 1.8, ya que la que viene con la VM es la 1.7 y STS utiliza la 1.8.

4.2 Desarrollo incremental del proyecto

Veamos ahora un poco más en detalle en qué consistieron las distintas iteraciones incrementales que se realizaron durante el desarrollo de la aplicación, viendo los problemas o nuevas propuestas de funcionalidad que se querían abordar y cómo se les dio solución.

4.2.1 Iteración I

En esta primera versión de la aplicación, simplemente se pretendía dar solución a los requerimientos mínimos que se plantearon previos a la fase de diseño. Haciendo un resumen rápido, recordamos que la aplicación debía extraer los metadatos de todas las tablas presentes en la base de datos fuente, realizar una transformación de estos metadatos a los correspondientes en Hive, crear las tablas (vacías) equivalentes en Hive utilizando estos metadatos transformados y por último migrar los datos desde la base de datos fuente a HDFS y las tablas creadas en Hive utilizando Sqoop.

En este punto es importante mencionar que Sqoop, al realizar una migración es capaz de crear las tablas en Hive automáticamente añadiendo la opción "-hive-import". No obstante, al hacerlo, Sqoop le asigna a todos los tipos de datos de los campos existentes en las tablas

creadas el tipo `STRING`, lo cual limitará bastante algunos aspectos, como consultas o procedimientos que se quieran ejecutar en el futuro, es por ello por lo que es importante realizar esa fase previa de creación de las tablas en Hive con sus correspondientes metadatos que realiza nuestra aplicación.

Para obtener los nombres de las tablas existentes en la base de datos fuente, así como los metadatos de las mismas que necesitamos, realizamos consultas tipo SQL, mediante la creación previa de `Statements` y su método `executeQuery`(). En esta iteración la aplicación sólo funcionaba para bases de datos Oracle, por tanto, las consultas que se ejecutaban eran las siguientes:

`SELECT TABLE_NAME FROM USER_TABLES`: Esta consulta devuelve los nombres de todas las tablas de usuario existentes.

`SELECT COLUMN_NAME FROM all_tab_columns WHERE table_name = '<nombreTabla>'`: Esta consulta permite obtener los nombres de todas las columnas o campos de una tabla concreta.

`SELECT DATA_TYPE FROM all_tab_columns WHERE table_name = '<nombreTabla>'`: Esta consulta permite obtener los tipos de datos de todos los campos de una tabla concreta. Ej.: `Number`, `Date`, `Varchar`, etc.

`SELECT DATA_LENGTH FROM all_tab_columns WHERE table_name = '<nombreTabla>`: Esta consulta permite obtener la longitud de los tipos de datos de todos los campos de una tabla concreta. Ej.: `Varchar(20)`

Para realizar la migración, debido a la dificultad (como veíamos en la sección del estado del arte) de integrar Sqoop dentro de la propia aplicación, se optó por realizar una ejecución por comando desde la aplicación al Sqoop instalado en la máquina donde se está ejecutando la aplicación utilizando la función de java `Runtime.getRuntime().exec()`. El comando de Sqoop que se utiliza en esta ejecución es el siguiente:

```
sqoop import --connect <rutaBDFuente> --username <usernameBDFuente>
--password <pwdBDFuente> --table <nombreTabla>  --target-dir  <targetDir>
+ <nombreTabla> --append -m <numMappers>
```

En este comando distinguimos los siguientes parámetros:

- `connect`: Ruta JDBC donde se encuentra nuestra base de datos fuente.
  Ej.: `jdbc:mysql://localhost/nombrebd`.

- `username` y `password`: Nombre de usuario y contraseña necesarios para realizar la conexión con la base de datos fuente.

- `Table`: Nombre de la tabla de la base de datos fuente cuyos datos se van a migrar.

- `Target-dir`: Ruta HDFS donde se cargaran los datos. En nuestro caso, tendremos una misma ruta general y luego para cada tabla crearemos un directorio distinto dentro de esa ruta. Ej.: `/user/pruebas/nombreTabla`.

- `Append`: Es necesario incluirlo debido a que estamos cargando datos en tablas previamente creadas en Hive.

- `M`: Numero de mappers (hilos) que utilizará Sqoop en su proceso de migración de datos.

Al estar trabajando con Oracle, es importante tener en cuenta que si se realizan modificaciones en las tablas fuente, antes de ejecutar Sqoop, deberá ejecutarse el comando `ANALYZE TABLE <nombreTabla> COMPUTE STATISTICS` para que los cambios y actualizaciones que se hayan realizado queden completamente registrados y Sqoop los tenga en cuenta.

Por otro lado, para crear las tablas en Hive se utiliza una sentencia `CREATE TABLE`, veamos un ejemplo:

```
CREATE TABLE <nombreTabla> (id INTEGER, nombre STRING, apellido STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘,’ LOCATION ‘path’
```

En esta sentencia observamos que hay que especificar un delimitador para los campos que se utiliza para distinguir los distintos campos de la información “en bruto” en HDFS, en nuestro caso, actualmente utilizamos comas (,). Además también se especifica una localización, esta es el path HDFS donde se almacenan los datos correspondientes a esa tabla, que hacemos coincidir con el valor del parámetro `targetDir` que utilizábamos en Sqoop, de esta manera queda enlazada la información contenida en HDFS con sus tablas Hive correspondientes. Gracias a esto, con una sola carga de datos, la información aparece automáticamente tanto en HDFS (en bruto) como en las tablas Hive.

Como complemento, en esta iteración también se añadió la funcionalidad correspondiente para leer desde un fichero XML todos los parámetros necesarios para la ejecución de la aplicación y además, para permitir la selección de tablas por interacción mediante consola.

4.2.2 Iteración II

Al ver que se estaban obteniendo buenos resultados, a partir de esta iteración se empezó a tener más en cuenta el posible uso de la aplicación como herramienta frecuente de trabajo a la hora de actualizar datos en HDFS en lugar de una sola migración y decomisionado inicial. En base a esto se propuso como mejora añadirle a la aplicación una interfaz gráfica (GUI, Graphic User Interface) para facilitar el trabajo de selección de tablas (especialmente para bases de datos con un numero de tablas elevado) y para hacerlo más intuitivo.

Para ello se utilizó la librería Swing de Java y se construyó una interfaz gráfica sencilla (un solo panel) en el que se incluían: Un combo box para poder seleccionar el tipo de base de datos con el que se iba a trabajar y mediante la selección realizada en este combo box, la aplicación instanciaba un objeto de la clase correspondiente al tipo de extractor de base de

datos. (Por cada tipo de base de datos con el que se pudiera trabajar debe existir una clase con los métodos correspondientes implementados y una opción seleccionable en el combo box, aunque en este momento sólo funcionaba para Oracle). Un panel (con scroll) donde aparecieran todas las tablas existentes en la BD fuente indicando su nombre y un checkbox de selección y un par de botones para dar la opción de seleccionar o excluir todas las tablas automáticamente. Aquí vemos una primera versión de la interfaz gráfica de usuario:

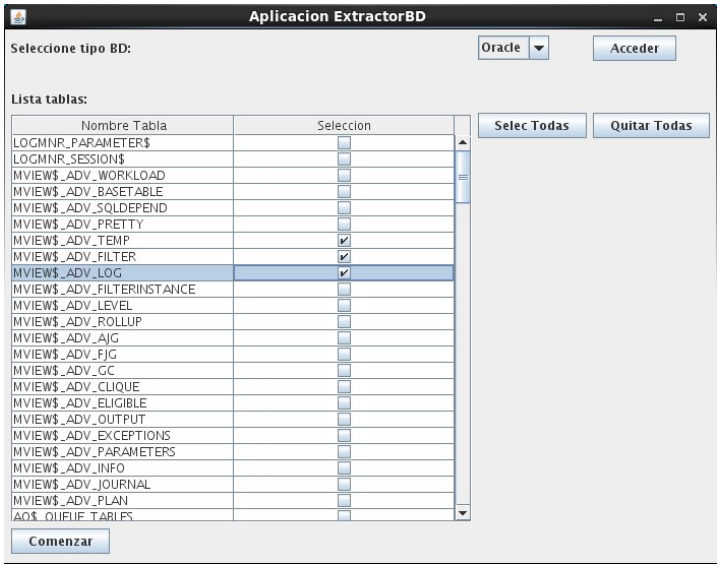

Figura 16. Primera vista de la aplicación

En base a este nuevo enfoque, también se modificó ligeramente la sentencia de la creación de tablas en Hive, añadiendo al `CREATE TABLE` la frase `IF NOT EXISTS`, de modo que si en lugar de realizarse una primera carga, se estaba realizando una carga posterior en el que las tablas ya estuvieran creadas previamente, no hubiera problema.

4.2.3 Iteración III

Hasta este momento, la base de datos destino en Hive debía estar previamente creada y el tipo de carga que se hacía es el denominado como "Full", es decir, se crean las tablas correspondientes en Hive si no existen previamente y se cargan todos los datos de las tablas de la BD fuente, y si ya existen se truncan (se borra su contenido) y se le vuelven a insertar todos los datos completos. Como mejora para esta iteración se propuso en primer lugar crear la base de datos destino en Hive de manera automática si no existía previamente (nombre configurable por parámetro) y, por otro lado, contemplar tres tipos distintos de carga de datos, estos son:

- Full: Como hemos visto, se crean las tablas destino si no existen previamente, o si ya existen, se borra su contenido y se vuelven a añadir de nuevo todos sus datos.

- Append: Se cargan todos los datos existentes de las tablas fuente en las tablas destino, aunque en estas ya existieran datos previos, de esta manera se generan registros duplicados que en algunos casos pueden ser de utilidad. Además no habrá

conflictos puesto que en Hive no existen restricciones de este tipo como claves primarias, etc.

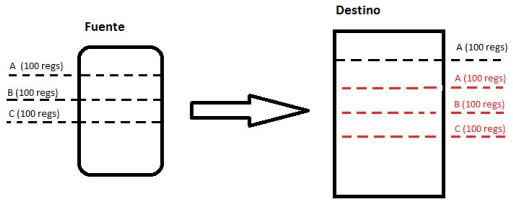

Figura 17. Ejemplo carga tipo Append con duplicados (Fuente: Elaboración propia)

- Incremental: Se cargan únicamente los datos que se han añadido recientemente en las tablas de la base de datos fuente y que no están incluidos en el destino (que contiene los datos de otras cargas anteriores pero no los nuevos).

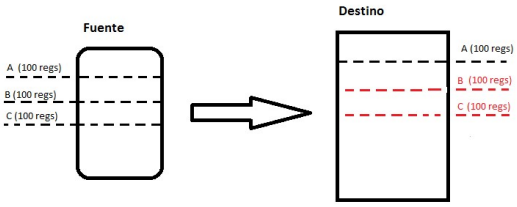

Figura 18. Ejemplo carga tipo Incremental sin duplicados (Fuente: Elaboración propia)

Para añadir esta funcionalidad fue necesario modificar ligeramente la interfaz gráfica, añadiendo tres radiobuttons para permitir la selección de los diferentes tipos de carga planteados.

Al incluir el parámetro `--append` en el comando Sqoop que utilizábamos previamente, para realizar el tipo de carga de datos Append, utilizamos exactamente el mismo comando (incluso misma función para simplificar código), la única diferencia está en que si se elige el modo de carga Full, este previamente realizará un `TRUNCATE` de todas las tablas seleccionadas. No obstante, este truncate no da lugar a errores porque según la estructura secuencial del código, está implementado de tal forma que las sentencias truncate que se puedan realizar, siempre serán a posteriori de la creación previa de las tablas (si no existen ya), por tanto, de esta manera se controla y se evita este error.

Para el tipo de carga incremental, la funcionalidad extra a añadir era bastante más complicada, esto se debe principalmente a que para realizar este tipo de migración de datos, Sqoop debe emplear como referencia un campo (generalmente de tipo `INTEGER` o `DATETIME` o equivalentes) y además especificar el valor más alto de ese campo ya existente en destino. Por ejemplo, si tenemos una tabla con registros de datos de clientes y

a cada cliente le hemos asociado un identificador único e incremental y hemos realizado una carga previa de los 100 primeros clientes en destino, si ahora añadimos 50 clientes nuevos en la tabla fuente y queremos migrar esta información al destino haciendo una carga incremental, deberemos especificar el campo de referencia (en este caso `id_cliente`) y el último valor cargado en destino (en este caso el valor 100), de esta manera se cargaran únicamente los registros de los 50 últimos clientes añadidos.

Para enfrentar esto y darle una funcionalidad lo más automática posible, se emplearon las siguientes soluciones: En primer lugar, al seleccionar el radiobutton correspondiente al tipo de carga incremental en la GUI, el panel nos mostrará sólo aquellas tablas que sean compatibles con este tipo de carga (que tengan algún campo de tipo entero o fecha o equivalente). Después, una vez decidamos que tabla queremos cargar, deberemos hacer doble clic sobre su celda de selección y se abrirá un dialogo que nos mostrará el nombre y el tipo de dato de los campos compatibles con este tipo de carga y nos permitirá seleccionar el campo que queremos utilizar como referencia.

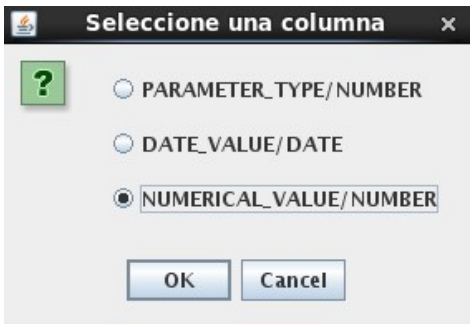

Figura 19. Panel selector de campo de referencia para carga incremental

Una vez seleccionemos el campo clicaremos en Aceptar y la aplicación internamente realizará una consulta a la tabla Hive correspondiente y obtendrá el valor más alto existente en esa tabla para el campo seleccionado, entonces recogerá y almacenara ese valor para pasárselo al comando Sqoop que vemos a continuación para realizar la carga de manera automática:

```
sqoop import --connect <rutaBDFuente> --username <usernameBDFuente> --
password <pwdBDFuente> --table <nombreTabla>  --target-dir  <targetDir> +
<nombreTabla> --incremental append –check-column <campoRef> --last-value
<ultimoValor_campoRef> -m <numMappers>
```

A diferencia del comando Sqoop que veíamos anteriormente para los tipos de carga de datos Full y Append, en este hemos añadido los siguientes parámetros:

`--incremental append`: Para indicarle que queremos realizar un tipo de carga incremental.
`--check-column`: Para indicarle el campo que queremos utilizar como referencia.
`--last-value`: Para indicarle el valor máximo actual del campo que utilizamos como referencia.

Es importante aclarar que para realizar este tipo de carga, debe hacerse tabla por tabla y el técnico que la esté utilizando tendrá la labor de conocer la coherencia de los valores del campo que va a utilizar como referencia con los datos que pretende cargar, es decir, si el

campo es de tipo id incremental como en el ejemplo que veíamos antes, en principio no habrá ningún problema, sin embargo, si por ejemplo estuviera migrando datos de una tabla que contuviera registros de películas y en lugar de realizar la carga por supongamos un id de película la realizará utilizando como referencia el campo fecha de estreno, no tendría mucho sentido, pues si el valor más alto de ese campo en destino fuera por ejemplo "2016", solo cargaría películas con fecha de estreno de 2016 en adelante pero podría haber nuevos registros en fuente que no estuvieran en destino con películas que tuvieran una fecha de estreno anterior a ese año y por tanto, esos datos no los estaría cargando en destino y los estaría "perdiendo" por el camino.

Por último, en esta última iteración se añadió también la funcionalidad necesaria para que la aplicación pudiera trabajar también con bases de datos fuente de tipo MySQL. De esta manera se extendió el ámbito en el que puede ser útil como herramienta y se afianzó la idea de que el diseño que se había llevado a cabo era bueno, ya que esta adición de código se pudo llevar a cabo de manera modular y relativamente sencilla.

4.3 Funcionalidad final de la aplicación

Veamos el aspecto final de la interfaz gráfica de usuario:

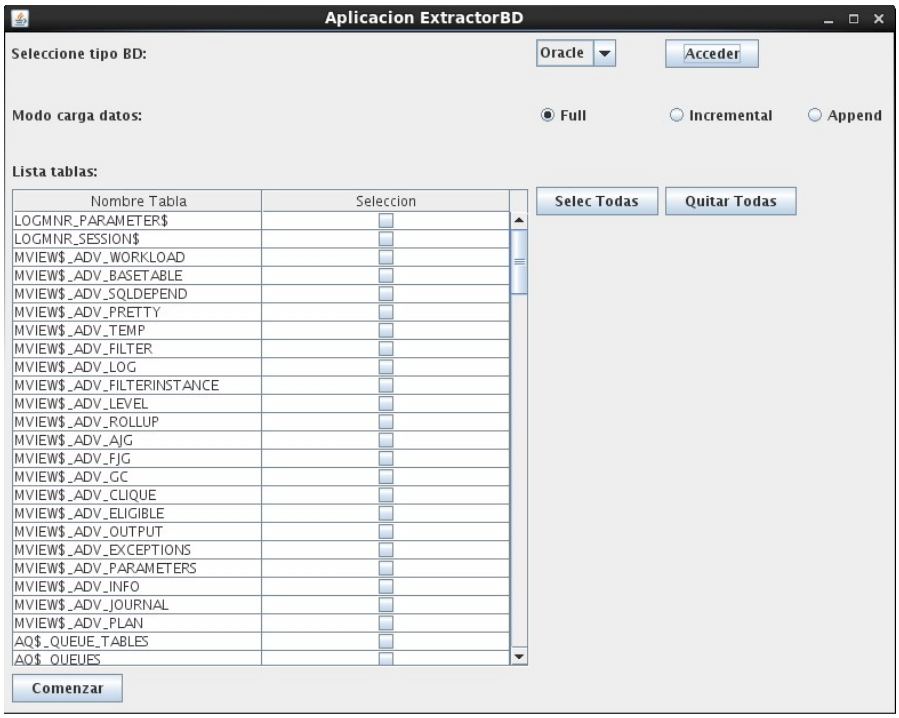

Figura 20. Vista final de la aplicación

Observamos la inclusión del selector de modo de carga de datos mediante radiobuttons. Además cabe destacar que la interfaz cuenta con la funcionalidad necesaria para habilitar o deshabilitar ciertos botones según el estado en el que se encuentre el sistema o las opciones que haya seleccionadas. Por ejemplo: Como el modo de carga incremental se lanza después de elegir un campo de referencia de una tabla mediante el uso de un `JDialog`, se deshabilita el botón "Comenzar" que es el que correspondería a los demás tipos de carga.

De esta manera conseguimos mejorar la usabilidad de la aplicación y prevenir posibles errores en tiempo de ejecución.

4.3.1 Flujo del programa

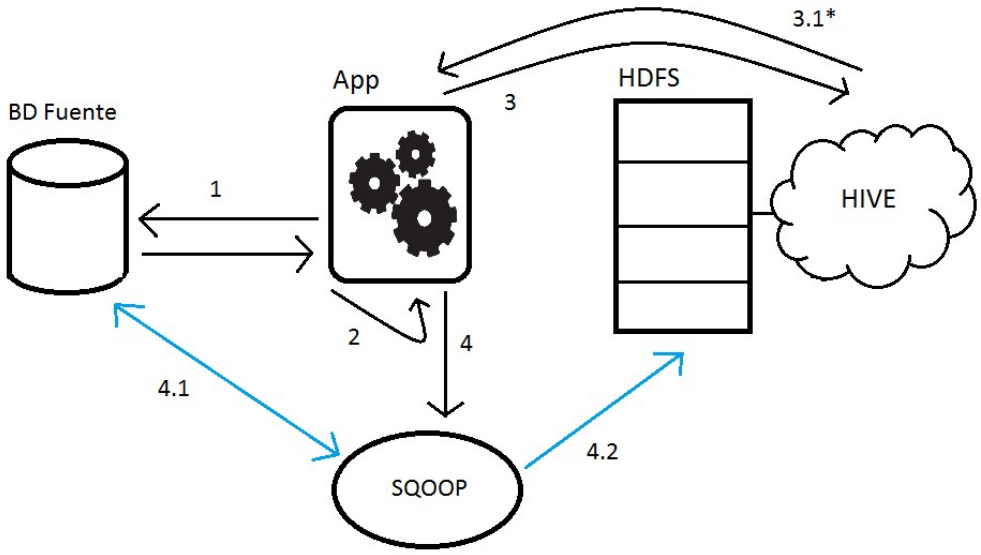

Figura 21. Diagrama del funcionamiento de la aplicación (Fuente: Elaboración propia)

En esta imagen vemos un diagrama de la funcionalidad general de la aplicación y el flujo que sigue ordenadamente en su ejecución. Veámoslo:

1. La aplicación se conecta a la base de datos fuente y realiza consultas para obtener los nombres de todas las tablas existentes en esta y los metadatos de cada una de ellas.

2. La aplicación procesa toda esta información, la almacena creando un objeto tabla por cada tabla existente y muestra al usuario la lista de tablas disponibles. Una vez hecho esto, el usuario elige las tablas cuyos datos desea migrar y el sistema procesa la información necesaria para crear los metadatos correspondientes a esas tablas en sus futuras tablas equivalentes en Hive.

3. La aplicación se conecta a Hive, crea una base de datos concreta en Hive (a no ser que ya exista) y crea las tablas elegidas en Hive con sus metadatos equivalentes (si no existen previamente).

   3.1*. Si las tablas ya existían y se ha elegido el modo de carga incremental, el sistema realiza una consulta en Hive para obtener el valor más alto del campo de referencia elegido para utilizarlo en el comando Sqoop correspondiente a la migración incremental.

4. La aplicación realiza una llamada a la herramienta Sqoop (externa a la aplicación) y le pasa los parámetros necesarios para realizar la migración de datos que se desea.

   4.1. y 4.2. Sqoop conecta con la base de datos fuente y extrae los datos necesarios y después los almacena en HDFS.

4.3.2 Parámetros y fichero de configuración

Para el correcto funcionamiento de la aplicación es imprescindible incluir un fichero de configuración que contiene los parámetros necesarios para su ejecución. En este caso utilizamos un fichero XML que contiene los siguientes elementos (veamos un ejemplo):

```
<datos>
  <parametros nombre="BDFuente">
        <ruta_jdbc>jdbc:oracle:thin:@localhost:1521:XE</ruta_jdbc>
        <nombre_bd>XE</nombre_bd>
        <username>userOracle</username>
        <password>userOraclepwd</password>

  </parametros>

  <parametros nombre="BDHive">
        <ruta_jdbc>jdbc:hive2://localhost:10000</ruta_jdbc>
        <nombre_bd>pruebas</nombre_bd>
        <username>userHive</username>
        <password>userHivepwd</password>
        <targetDir>/user/cloudera/pruebasHiveTFG/</targetDir>
        <numMappers>4</numMappers>
   </parametros>
</datos>
```

Observamos dos elementos principales:

- BD Fuente: Contiene los parámetros necesarios para conectar con la base de datos fuente. Estos son su ruta de conexión para JDBC, nombre de la base de datos, usuario y contraseña.
- BD Hive: Contiene los mismos parámetros que el elemento anterior pero para la base de datos en Hive y además incluye los parámetros para especificar la ruta en HDFS donde se almacenaran los datos en bruto (`targetDir`) y para especificar el número de hilos con los que se desea que Sqoop realice la migración de los mismos (`numMappers`).

4.4 Otras consideraciones

4.4.1 Balanceo de Sqoop

A la hora de ejecutar Sqoop para realizar la migración de los datos de las tablas utilizando varios mappers (hilos), existe una restricción que consiste en que debe existir una `PRIMARY KEY` en la tabla a migrar o al menos especificar un campo de referencia. Este

campo, ya sea la PK o el especificado como referencia mediante la opción `--split-by` es el que utiliza Sqoop para realizar la partición de los datos en base a la cual se lanzarán y se les dará una carga de trabajo a los distintos hilos de ejecución. Además este campo es imprescindible, pues sin él, Sqoop no tendría forma de agrupar de nuevo todos los registros procesados al finalizar la migración de los datos. Veamos un ejemplo, si utilizamos la PK como referencia (que es lo que hace Sqoop por defecto) y tenemos 100 registros y especificamos 4 hilos de ejecución, Sqoop dividirá la carga de trabajo en 25 registros para cada hilo debido a que cada campo de referencia (al ser una PK) contiene un valor diferente. Este ejemplo como vemos está bien balanceado, sin embargo, si utilizáramos como referencia otro campo distinto que tuviera 80 registros con el mismo valor para ese campo y 20 registros con otro valor diferente del primero pero iguales entre ellos y lanzáramos 2 hilos, la carga se dividiría en 80 registros para el primer hilo y 20 para el segundo, lo cual estaría descompensado. Por este motivo es importante procurar permitirle a Sqoop que utilice las PKs que utiliza por defecto y que cada tabla a migrar posea la suya o si no, al menos especificar un campo de referencia que esté lo más balanceado posible para no perder eficiencia.

4.4.2 Maven

Maven es una herramienta para la gestión y construcción de proyectos Java. Tiene un modelo de construcción simple basado en XML. [5] En nuestro proyecto, el empleo de Maven ha sido de gran utilidad durante el desarrollo, debido a que a la hora de añadir alguna librería externa o driver que se necesitarán (como los necesarios para las conexiones mediante JDBC para MySQL, Oracle y Hive) se han incluido a través de esta herramienta en lugar de tener que descargarlos manualmente e incluirlo en el proyecto como fichero .jar.

Para trabajar y describir el proyecto software a construir, sus dependencias de otros módulos y componentes externos, orden en que se construyen estos elementos, etc. Maven utiliza el fichero pom.xml (Proyect Object Model). [5] Para añadir por ejemplo el JDBC necesario para MySQL utilizamos una dependencia:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.40</version>
</dependency>
```

Al actualizar o salvar el proyecto, el motor de Maven descargará automática y dinámicamente el plugin del repositorio correspondiente (pueden incluirse varios repositorios a consultar en el fichero pom.xml) y siempre que queramos añadir algún otro modulo o componente, simplemente deberemos buscar en el repositorio de Maven (https://mvnrepository.com) cual es la dependencia correspondiente que queremos añadir (probablemente existirán varias versiones), incluirla en el fichero y repetir el proceso.

Para más información pueden consultarse las referencias o el siguiente enlace: https://maven.apache.org/

5 Integración, pruebas y resultados

5.1 Pruebas unitarias

Se han realizado pruebas unitarias para comprobar el correcto funcionamiento de todas las clases existentes en el proyecto y especialmente de los métodos que implementan. Hay que tener en cuenta que el correcto funcionamiento de algunos métodos depende de la correcta ejecución previa de otros. En el programa se ha controlado prestando gran atención a la secuencialidad del código, sin embargo, durante las pruebas o bien se han forzado para poderlos testear o bien se ha probado su funcionamiento durante las pruebas de integración.

Durante esta fase de pruebas se ha prestado especial atención a:

- Pruebas de conexión, mediante los JDBC integrados en la aplicación, tanto a las bases de datos fuente compatibles, como a Hive.
- Obtención de los datos esperados con las consultas realizadas automáticamente a las bases de datos fuente o a Hive haciendo uso de `Statements` y `ResultSets`.
- Verificación del almacenamiento en la aplicación de los datos obtenidos en las consultas y correcta transformación de los metadatos a los equivalentes necesarios para la creación de las tablas en Hive.
- Correcta creación de las tablas correspondientes en Hive (vacías).
- Llamada a Sqoop y el proceso de migración de datos que ejecuta.
- Verificación de la visualización y adecuación de la GUI (sin funcionalidad integrada).

5.2 Pruebas de integración

En esta fase de pruebas comprobamos el correcto funcionamiento de la aplicación en su conjunto, es decir, verificamos que todos los componentes de la misma (que en principio ya se ha comprobado que funcionan correctamente por separado) funcionan correctamente al ensamblarlos y la aplicación tiene la funcionalidad esperada y cumple con los requisitos que se marcaron.

Veamos un ejemplo rápido de ejecución utilizando los parámetros de conexión a las bases de datos fuente y Hive (ruta JDBC, nombreBD, usuario y contraseña) y además la ruta HDFS `/user/training/pruebasDemo4/` y 4 mappers para Sqoop:

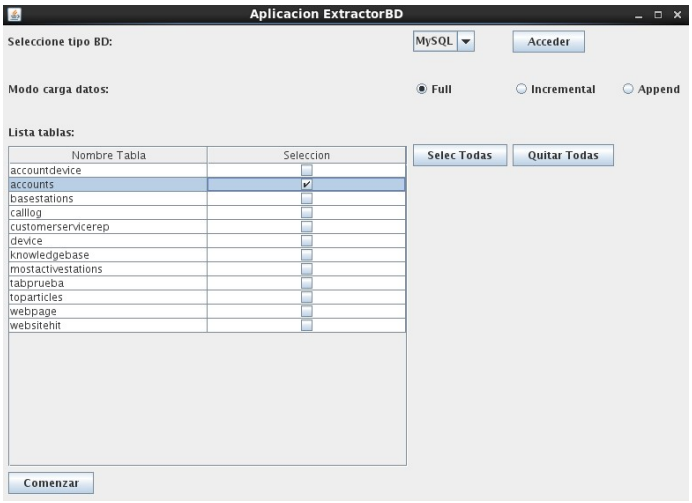

Figura 22. Selección tabla "accounts" en la vista y método de carga "Full"

Accedemos mediante la aplicación a una base de datos fuente tipo MySQL que contiene esas tablas y seleccionamos la tabla `accounts` y el método de carga "Full" y le damos a "Comenzar".

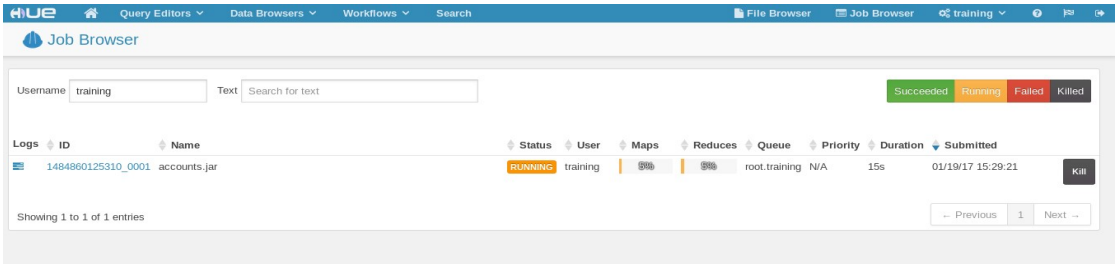

Figura 23. Vista del HUE job browser con job de tabla "accounts" procesando

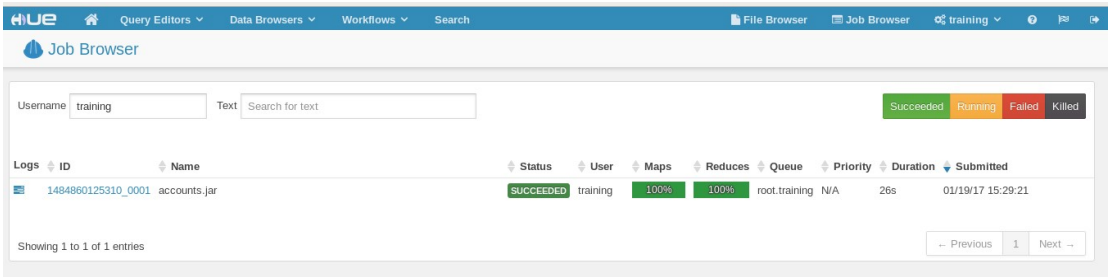

Figura 24. Vista del HUE job browser con job de tabla "accounts" finalizado con éxito

La consola nos va mostrando información acerca de cómo se está ejecutando el proceso y si además accedemos a HUE (Hadoop User Experience) podemos seguirlo de una manera más gráfica, vemos cómo se genera un job que se va ejecutando hasta que finalmente se termina con éxito. Accedamos ahora a HDFS y Hive para comprobar que los datos se han cargado correctamente:

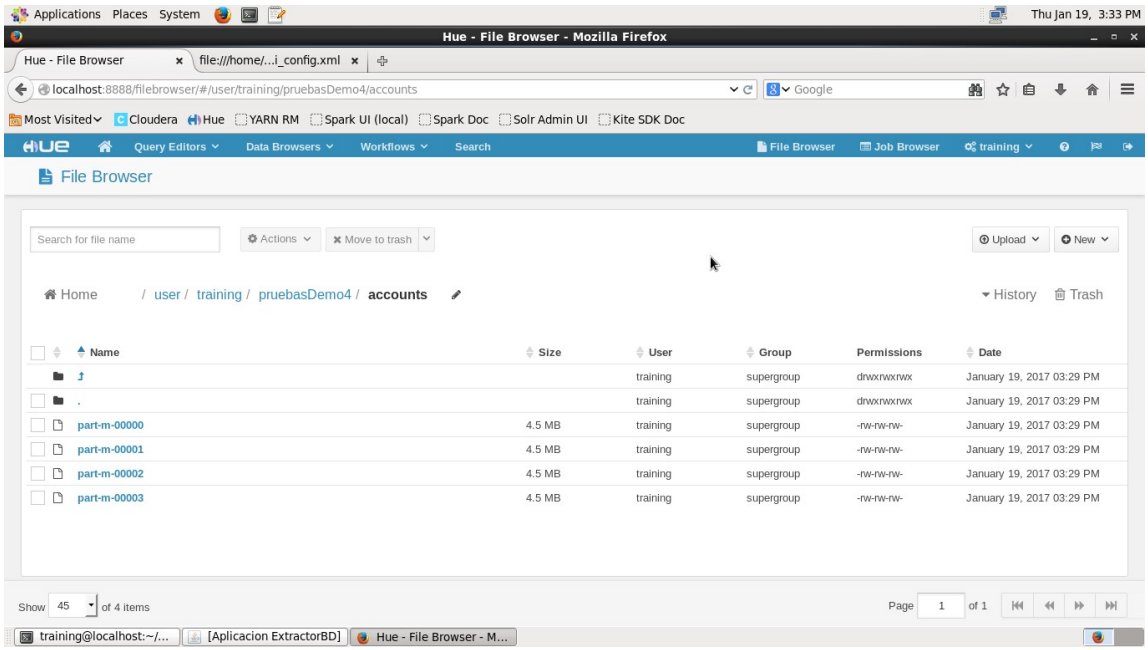

Figura 25. Vista del directorio "accounts" en HDFS con sus ficheros de datos

Efectivamente vemos cómo se ha generado un directorio con el nombre de la tabla seleccionada en la ruta HDFS que hemos pasado por parámetro y en él se han cargado 4 ficheros de datos diferentes (uno por cada mapper de Sqoop) con un tamaño de 4.5MB cada uno (cada fichero se corresponde con un bloque de datos en HDFS correspondientes a los datos que había contenidos en la tabla fuente). Si accedemos ahora a uno de ellos podemos ver la información en bruto:

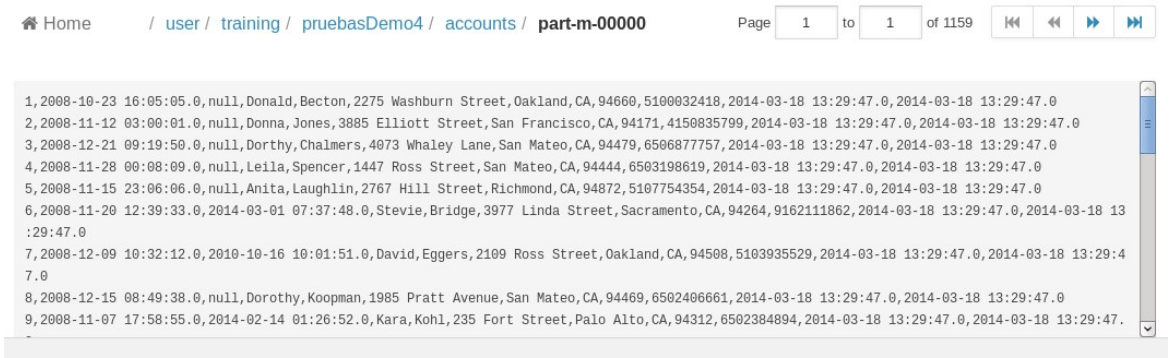

Figura 26. Vista de datos en bruto del primer fichero de "accounts"

Accedamos ahora a Hive para comprobar que también se ha creado la tabla correspondiente y se han añadido los datos a la misma de manera estructurada:

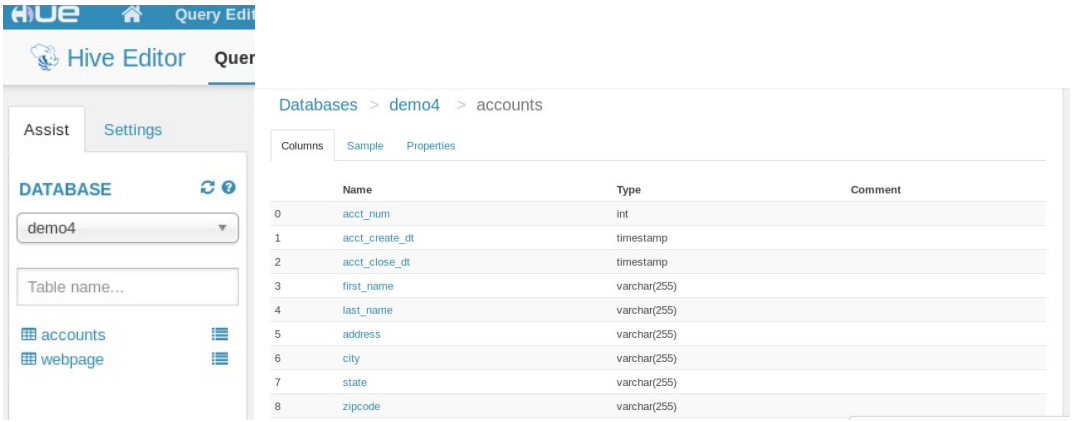

Figura 27. Vista de la tabla accounts en Hive y sus metadatos

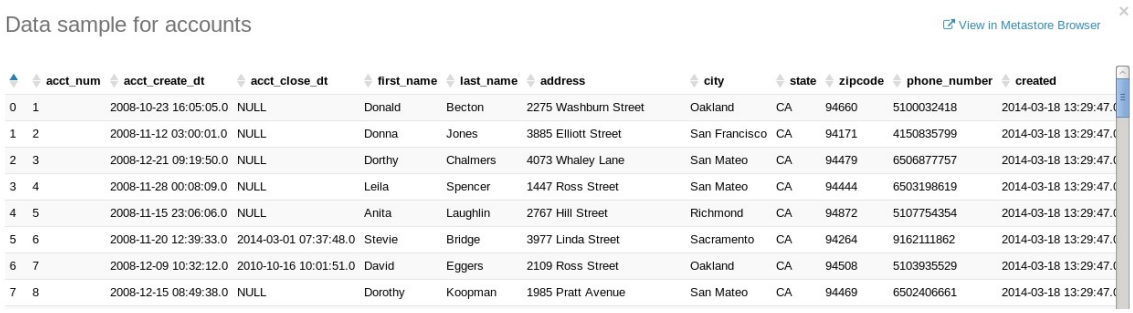

Figura 28. Datos de muestra de la tabla accounts en Hive

Observamos cómo se ha creado en la base de datos Hive indicada por parámetro la tabla `accounts` y que sus metadatos también son los adecuados. Comprobamos también que en la tabla se han insertado los datos correspondientes visualizando una muestra de los mismos. Si para más seguridad, queremos verificar el número de registros existentes en la nueva tabla podemos ejecutar la consulta `SELECT count(*) FROM accounts` y comprobamos que efectivamente coinciden con el número de registros existentes en la tabla `accounts` de la base de datos fuente:

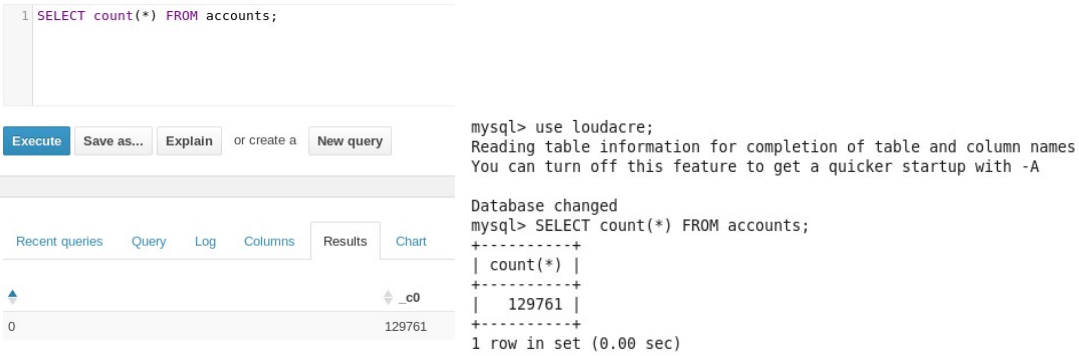

Figura 29. Comparación número de registros en tabla fuente con tabla Hive

5.3 Otros aspectos

Se ha procurado controlar todos los casos de error posibles, mostrando su descripción por pantalla al usuario y permitiendo a este solventarlos si era posible en lugar de terminar directamente la ejecución. No obstante, al no haber tenido acceso a bases de datos reales y al haber trabajado en local, es posible, que a la hora de utilizar la aplicación en un entorno de producción, puedan descubrirse nuevos casos de error que no se han controlado.

Hubiera sido interesante también haber realizado unas pruebas de rendimiento modificando el parámetro correspondiente al número de mappers de Sqoop, pero al estar trabajando en un entorno pseudo-distribuido (el que proporciona la VM), es decir, simulado y con una sola máquina en el clúster, no tiene mucho sentido llevarlas a cabo hasta que no se tenga acceso a un entorno de pruebas más realista.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

De acuerdo con la demanda actual de proyectos basados en Big Data, podemos afirmar que la herramienta que se ha desarrollado podría llegar a ser bastante útil gracias a la automatización que realiza del proceso de decomisionado y migración de datos al entorno de Hadoop, lo cual permite ahorrar tiempo cada vez que sea necesario realizar este proceso ya que no habrá necesidad de realizarlo de manera manual y esto a su vez permitirá que un mayor número de profesionales que dispongan de unos conocimientos técnicos mínimos acerca de las tecnologías implicadas puedan utilizarla y realizar el proceso con facilidad, lo cual también es otro aspecto positivo.

Como conclusión personal, el desarrollo del presente Trabajo Fin de Grado, me ha permitido desarrollar numerosas competencias y destrezas adquiridas durante la carrera y aplicar parte de los conocimientos obtenidos en la misma, los cuales han sido realmente útiles durante la elaboración de este proyecto. En particular, las competencias adquiridas en asignaturas como análisis y proyecto de análisis y diseño de software han sido de gran utilidad tanto para plantear y elaborar un buen diseño de la aplicación como para implementarlo en lenguaje Java. También disponer de una buena base previa de programación (Programación I y II) y de conocimientos sobre bases de datos relacionales tipo SQL que se han aplicado ampliamente a la hora de buscar, seleccionar e implementar las consultas a realizar. Por último cabe mencionar también asignaturas como informática y sociedad, útil para ampliar la mente y tener mayor facilidad para buscar, entender y plasmar la información referente a tecnologías tan novedosas como Big Data, y por último pero no por ello menos importante, conceptos vistos en asignaturas como sistemas informáticos II que han sido útiles a la hora de entender el sistema distribuido de Hadoop y de implementar consultas automáticas en el código haciendo uso de Connections y Statements, y la asignatura ingeniería del software prestando atención a conceptos importantes como el de la usabilidad, la escalabilidad y la mantenibilidad, que se han tenido muy en cuenta a la hora de diseñar e implementar la aplicación.

6.2 Trabajo futuro

Como trabajo futuro para esta herramienta se han propuesto los siguientes puntos o ideas:

- Realizar pruebas de rendimiento y de control de errores en un entorno distribuido real y con bases de datos reales. (Muy recomendable)

- Pasar por el fichero de parámetros de configuración varios esquemas o bases de datos fuente y añadir un selector (además del ya existente para el tipo) que permita seleccionar la que se desee.

- Añadir nuevos tipos de bases de datos fuente compatibles con la herramienta.

- Añadir la funcionalidad necesaria para poder realizar varias cargas de datos de tipo incremental con una sola ejecución si es posible, en lugar de una por una.
- Parametrizar de alguna manera un campo de referencia para que Sqoop lo utilice cuando realiza el proceso de migración con más de un hilo para cuando no se desea utilizar la `PRIMARY KEY` por defecto o esta no existe en la tabla fuente.

Referencias

[1] “Big Data y Hadoop. Cloudera vs Hortonworks” http://mukom.mondragon.edu/ict/big-data-y-hadoop-cloudera-vs-hortonworks/ (Último acceso: 22/01/2017)

[2] “Hadoop” Wikipedia https://es.wikipedia.org/wiki/Hadoop#cite_note-poweredby-2 (Último acceso: 22/01/2017)

[3] “Apache Hadoop” Wikipedia https://en.wikipedia.org/wiki/Apache_Hadoop (Último acceso: 22/01/2017)

[5] “Maven” Wikipedia https://es.wikipedia.org/wiki/Maven (Último acceso: 22/01/2017)

[6] “Big Data” Wikipedia https://es.wikipedia.org/wiki/Big_data (Último acceso: 22/01/2017)

[7] “¿Qué es Big Data? IBM https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/ (Último acceso: 22/01/2017)

[8] “Almacén de datos” Wikipedia https://es.wikipedia.org/wiki/Almac%C3%A9n_de_datos#Funciones_ETL_.28extracci.C3.B3n.2C_transformaci.C3.B3n_y_carga.29 (Último acceso: 22/01/2017)

[9] “Big Data. The management revolution” Andrew McAfee & Erik Brynjolfsson http://www.rosebt.com/uploads/8/1/8/1/8181762/big_data_the_management_revolution.pdf (Último acceso: 22/01/2017)

[10] White,T (2012) *Hadoop. The definitive guide (Third edition)* USA: Editorial O’Reilly

[11] “Introduction to Hadoop” Cloudera http://people.apache.org/~larsgeorge/SAP-Summit/Slides.pdf (Último acceso 22/01/2017)

NOTA: La mayor parte de las imágenes utilizadas (excepto en las que se cita la fuente específicamente) se han obtenido de las transparencias del curso de introducción a Big Data de Cloudera [12].

Glosario

API	Application Programming Interface
HDFS	Hadoop Distributed File System
JDBC	Java DataBase Connectivity
GUI	Graphic User Interface
SQL	Structured Query Language
XML	eXtensible Markup Language
STS	Spring Tool Suite (Entorno de desarrollo)
MVC	Modelo Vista Controlador (Patrón de diseño)
HUE	Hadoop User Experience
PK	Primary Key
BD	Base de datos (DB, DataBase en inglés)
VM	Virtual Machine (MV, Máquina Virtual en castellano)