Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN

# SeQual-Stream: Herramienta de control de calidad de secuencias genéticas en un entorno Big Data mediante procesamiento en streaming


**Estudiante:** Óscar Castellanos Rodríguez
**Dirección:** Roberto Rey Expósito
Juan Touriño Domínguez


A Coruña, junio de 2021.

*A mis padres, a mis amigos y a todas las personas que han estado a mi lado*

**Agradecimientos**

A mis padres y familia, por haberme apoyado durante esta etapa académica y durante toda mi vida.

A mis amigos y compañeros, por todos los buenos momentos pasados a vuestro lado.

A mis tutores, Juan y Roberto, por haberme dado la oportunidad de llevar a cabo este proyecto y por su apoyo y guía durante el transcurso del trabajo.

A todos vosotros y a muchos otros, muchísimas gracias.

Óscar Castellanos Rodríguez

## Resumen

Este Trabajo de Fin de Grado (TFG) presenta la implementación de SeQual-Stream, una herramienta paralela derivada de SeQual que permite realizar controles de calidad sobre conjuntos de datos genómicos de forma escalable. Está orientada al procesamiento de conjuntos de datos masivos en entornos distribuidos, utilizando para ello el framework Big Data Apache Spark y HDFS como sistema de ficheros distribuido. SeQual-Stream adapta este procesamiento para realizarlo en modo streaming, a medida que se descargan los datos desde Internet y/o se copian a HDFS, acelerando los tiempos de ejecución al no necesitar esperar a tener completo el conjunto de datos de entrada para empezar su procesamiento.

La herramienta permite procesar conjuntos de datos genómicos aplicando operaciones que procesan cada secuencia de forma individual, incluyendo filtros individuales, recortadores y formateadores. Durante su desarrollo se siguió una metodología iterativa incremental, consistente en repartir el desarrollo de la aplicación en diferentes incrementos en los que se añaden nuevas funcionalidades o se mejoran las existentes.

Para analizar la mejora obtenida respecto a SeQual, se realizó una evaluación experimental en un entorno clúster de altas prestaciones, comparando sus tiempos de ejecución con diferentes operaciones, número de nodos y conjuntos de datos de tamaño significativo.

SeQual-Stream se encuentra disponible públicamente en el siguiente repositorio Git bajo una licencia GNU GPL: https://github.com/oscar-castellanos/SeQual-Stream.



## Abstract

This BSc Thesis presents the implementation of SeQual-Stream, a parallel tool derived from SeQual that allows performing quality controls on genomic datasets in a scalable way. It is oriented to process massive datasets in distributed environments, using the Big Data framework Apache Spark and HDFS as distributed file system. SeQual-Stream adapts this processing to perform it in streaming mode, as data is downloaded from the Internet and/or copied to HDFS, speeding up runtimes by not having to wait until the input dataset is complete to start processing.

The tool allows processing genomic datasets by applying operations that process each sequence individually, including individual filters, trimmers and formatters. During its development, an incremental iterative methodology was followed, consisting of distributing the development of the application in different increments in which new functionalities are added or existing ones are improved.

In order to analyze the improvement obtained with respect to SeQual, an experimental evaluation was carried out on a high-performance cluster environment, comparing their execution times with different operations, number of nodes and datasets of significant size.

SeQual-Stream is publicly available at the following Git repository under a GNU GPL license: https://github.com/oscar-castellanos/SeQual-Stream.

**Palabras clave:**

- Secuenciación de Nueva Generación (NGS)
- Big Data
- Apache Spark
- HDFS
- Streaming estructurado de Spark

**Keywords:**

- Next Generation Sequencing (NGS)
- Big Data
- Apache Spark
- HDFS
- Spark Structured Streaming

# Índice general

# Índice de figuras

# Índice de tablas

# Listados

Capítulo 1

# Introducción

En este capítulo introductorio se presenta el contexto y motivación de este Trabajo de Fin de Grado (TFG), describiendo además la estructura general del presente documento.

## 1.1 Motivación

La obtención de secuencias de ADN de seres vivos suele ser el primer paso en los estudios desarrollados por biólogos y bioinformáticos. El continuo desarrollo de las tecnologías Next Generation Sequencing (NGS) [1] ha dado lugar a un incremento vertiginoso en la cantidad de datos genómicos. En la actualidad se pueden generar cientos de millones de secuencias en una única ejecución disminuyendo drásticamente su coste. Sin embargo, la calidad de la secuenciación no es alta en todos los casos. El resultado de los análisis bioinformáticos puede verse perjudicado a causa de una baja calidad durante el proceso de secuenciación en algunos fragmentos de ADN. Por ello, los análisis genéticos actuales a menudo comienzan con una primera etapa donde se realiza un control de las secuencias de entrada, eliminando o modificando aquellas que no se consideran útiles.

SeQual [2, 3] es una herramienta paralela implementada en Java que permite realizar diversos controles de calidad sobre grandes conjuntos de datos genómicos de una forma escalable. Para ello hace uso de Apache Spark [4, 5], un framework de procesamiento de datos masivos o Big Data, y de HDFS [6], un sistema de ficheros distribuido integrado en Apache Hadoop [7]. SeQual hace uso de las capacidades de Spark para procesamiento por lotes (batch), con lo que los requisitos para poder empezar un control de calidad son: (1) el conjunto de datos de entrada debe estar completo antes de empezar su procesamiento; y (2) dicho conjunto debe encontrarse almacenado en HDFS. La descarga de los datos desde un repositorio remoto junto con su posterior copia a HDFS son operaciones costosas que retrasan considerablemente el inicio del control de calidad a realizar por SeQual. Este problema se ve especialmente agravado a medida que el tamaño de los conjuntos de datos genómicos a procesar aumenta.

## 1.2 Objetivos

El objetivo de este trabajo consiste en el desarrollo de **SeQual-Stream**, una herramienta para el control de calidad de secuencias genéticas que permita realizar un procesamiento de los datos genómicos en modo streaming mediante la adaptación de SeQual a este nuevo paradigma. Dicho soporte streaming se limitará a conjuntos de datos sin comprimir y para aquellas operaciones de SeQual que valoran cada secuencia de forma independiente a las demás (filtros individuales, recortadores y formateadores). De esta forma es posible aplicar dichos controles de calidad a los datos de entrada a medida que las secuencias genéticas se descargan desde Internet y/o se copian a HDFS. Este nuevo modo de funcionamiento reducirá los tiempos de ejecución al permitir el solapamiento de las operaciones de descarga y copia de los datos a HDFS con el procesamiento de los mismos por parte de SeQual-Stream.

Completado el desarrollo de la nueva herramienta, se realizará una evaluación comparativa del rendimiento con su contraparte SeQual en un entorno clúster usando diferentes conjuntos de datos de entrada y operaciones para analizar la mejora obtenida.

## 1.3 Estructura de la memoria

Este documento está dividido en 7 capítulos. A continuación se indica un breve resumen de cada uno:

1. **Introducción**: este mismo capítulo, donde se comenta la motivación que llevó al desarrollo del proyecto y se exponen los objetivos a alcanzar.

2. **Conceptos previos**: en este capítulo se exponen ciertos conceptos que rodean al proyecto y son necesarios para comprender el trabajo realizado.

3. **Herramientas y metodología**: se describen las principales tecnologías y herramientas utilizadas para el desarrollo del trabajo, además de la metodología seguida.

4. **Funcionalidades**: se describen las funcionalidades ofrecidas por la herramienta SeQual-Stream.

5. **Desarrollo**: este capítulo se centra en explicar las diferentes etapas seguidas en la implementación de la aplicación, así como su funcionamiento interno.

6. **Evaluación del rendimiento**: se exponen las pruebas de rendimiento de la aplicación streaming en comparación con la versión batch, utilizando conjuntos de datos de tamaño significativo.

7. **Conclusiones**: este último capítulo se dedica a exponer las principales conclusiones alcanzadas tras finalizar el trabajo, además de comentar posibles mejoras futuras para la aplicación.

Capítulo 2

# Conceptos previos

En este capítulo se introducen y explican los conceptos más importantes necesarios para entender el trabajo realizado en este TFG.

## 2.1 Bioinformática

La bioinformática [8] es una de las disciplinas científicas que más relevancia está teniendo en los últimos años. Su labor consiste en investigar, desarrollar y aplicar herramientas informáticas y computacionales para permitir y mejorar el manejo de datos biológicos, gracias al uso de herramientas que almacenan, organizan, analizan y permiten interpretar estos datos.

La bioinformática nació a comienzos de 1960 con la aplicación de métodos computacionales al análisis de la secuencia de proteínas. Su crecimiento fue ligado al desarrollo de la biología molecular, el descubrimiento del ADN y los avances en computación. El concepto que se tiene hoy en día de la bioinformática es algo distinto, ya que se considera una disciplina emergente que se ha hecho imprescindible para el manejo del enorme volumen de datos que generan las nuevas tecnologías 'ómicas' (genómica, proteómica, metabolómica...), haciendo del concepto de Big Data un activo fundamental en la biomedicina actual. Entre estas tecnologías, una de las que mayor relevancia ha tenido es la secuenciación de alto rendimiento o secuenciación masiva, que se popularizó a partir del 2004 con la secuenciación del genoma humano y que ha permitido obtener la secuencia genómica de muchos organismos conocidos y desconocidos.

El uso de la informática, de los lenguajes de programación y de las grandes infraestructuras computacionales son los pilares en los que se apoya la bioinformática para recopilar, manejar, almacenar y analizar los datos biológicos, desarrollando algoritmos o modelos matemáticos que extraen el máximo conocimiento de los datos y permiten aplicarlo directamente a la resolución de problemas biológicos o biomédicos.

Se puede resumir la bioinformática en 3 grandes objetivos:

1. **Organizar los datos en bases de datos.** La bioinformática organiza adecuadamente

los datos permitiendo a los investigadores acceder a la información existente y publicar nuevos datos a medida que estos se producen. Esta información es inservible hasta que se analiza, por lo que la curación de datos es también una tarea esencial realizada por la bioinformática.

2. **Desarrollar herramientas y recursos que ayudan en el análisis de datos.** Dentro de este ámbito es donde se pueden incluir herramientas como la propia SeQual.

3. **Utilizar estas herramientas para analizar los datos e interpretar los resultados.** La bioinformática ha revolucionado los estudios biológicos. Si estos estudios se concentraron tradicionalmente en los detalles de los sistemas, en la actualidad se pueden realizar análisis globales con todos los datos disponibles. Esto permite descubrir principios comunes y nuevas características de los sistemas biológicos.

### 2.1.1 NGS

La secuenciación de nueva generación (Next Generation Sequencing, NGS) [1] es un grupo de tecnologías diseñadas para secuenciar gran cantidad de segmentos de ADN de forma masiva y en paralelo, en menor cantidad de tiempo y a un menor coste por base. Su uso se dio inicialmente para detectar variantes de nucleótido único y cada vez se ha desarrollado más para otro tipo de variantes, como inserciones [9], deleciones [10] y grandes reordenamientos [11]. Gracias a los recientes desarrollos en las pruebas basadas en NGS, estas tecnologías se plantean como estrategias de gran utilidad para la prevención, el diagnóstico, el tratamiento y el seguimiento de un amplio espectro de enfermedades, incluidas condiciones genéticas, patologías crónicas y enfermedades infecciosas.

Como ya se mencionó en la Sección 1.1, pese a la gran capacidad de estas herramientas para generar un gran número de secuencias, no siempre tienen la calidad deseada; de ahí la importancia del desarrollo de herramientas como SeQual y SeQual-Stream para mejorar este proceso.

### 2.1.2 Formatos de representación de secuencias

En bioinformática hay diferentes formatos para representar las secuencias biológicas. Dos de ellos muy comunes y ampliamente usados son FASTQ [12] y FASTA [13]. Por este motivo, tanto SeQual como SeQual-Stream se han diseñado para trabajar con dichos formatos. Ambos están basados en texto plano y separan los componentes de cada secuencia biológica (típicamente secuencias de nucleótidos) en múltiples líneas.

En el formato FASTQ, además de la propia secuencia (sus bases) se añade información acerca de la calidad de la misma. Un ejemplo de secuencia en formato FASTQ se muestra en el Listado 2.1. Como se puede observar, está formada por cuatro líneas donde:

```
@ERR188245.4748
TGAGGCTTACTAGAAGTGTGAAAACGTAGGCTTGGATTAAGGCGACAGCGATTTCTAGGA
+
@@@FFFFDHFHFF@GHGIEGGIIFIJC@FGGEHEGGIGIGEHG@FGGGIIGCHIJJEHFE
```

Listado 2.1: Secuencia en formato FASTQ

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTN
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDF
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIV
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIA
IENY
```

Listado 2.2: Secuencia en formato FASTA

- Línea 1: Es un nombre o identificador de la secuencia, donde se permite añadir una descripción de forma opcional. Como primer carácter siempre se utiliza '@'.

- Línea 2: Bases de la secuencia, donde cada base se codifica con un único carácter ASCII.

- Línea 3: Un carácter separador, el '+'. Además, se puede incluir un comentario opcional.

- Línea 4: Puntuación de la calidad asignada a las bases. La calidad refleja el grado de confianza que el secuenciador tenía a la hora de determinar cada base de la secuencia. Cada puntuación se codifica con un único carácter ASCII, con lo cual, se tendrá el mismo número de caracteres en la secuencia (línea 2) que en la calidad. Habitualmente, esta calidad está calculada siguiendo la codificación de Illumina [14], empleando el sistema de puntuación Phred [15] +33. Es decir, a la puntuación de calidad Phred se le suma 33, obteniendo el carácter ASCII correspondiente.

En el formato FASTA cada secuencia se divide en nombre y bases, pero no se dispone de información de su calidad. Igual que en FASTQ, la primera línea indica el nombre, aunque en este caso el carácter inicial es el '>'. En la siguiente línea empiezan las bases de la secuencia, también codificada cada una con un único carácter ASCII. La gran diferencia es que las bases se pueden indicar en una única línea o en un número indeterminado de ellas. Un ejemplo de secuencia en formato FASTA se muestra en el Listado 2.2.

Finalmente, cabe mencionar que es posible y muy habitual almacenar múltiples secuencias FASTQ o FASTA en un mismo fichero. En la actualidad, estos ficheros suelen tener del orden de cientos de millones de secuencias genéticas.

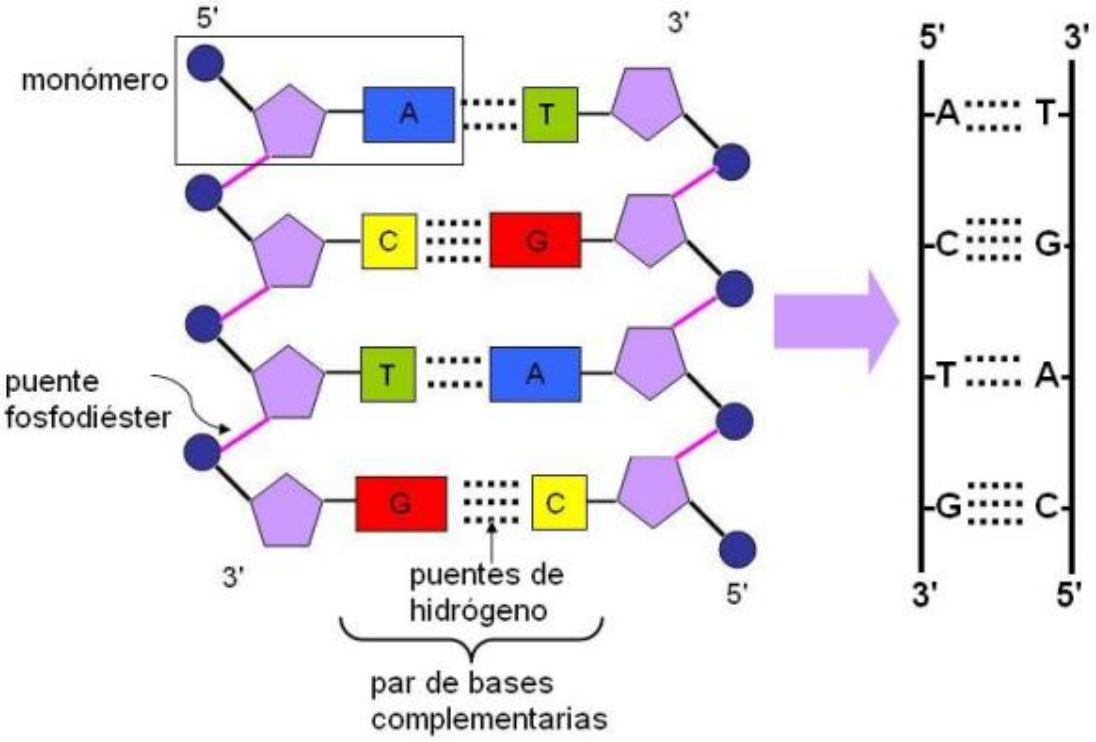


Figura 2.1: Extremos de una secuencia de ADN

### 2.1.3 Tipos de secuenciación

De la misma manera que existen diferentes formatos de representación de secuencias, también hay diferentes tipos de secuenciación para poder obtenerlas. Dos muy habituales en bioinformática son la secuenciación single-end y la paired-end [16, 17]. Por ello, tanto SeQual como SeQual-Stream se han diseñado para soportar ambos tipos de secuencias.

La diferencia clave reside en la forma de leer las bases a partir de una secuencia de ADN. Cuando se habla de extremos en las secuencias, se refiere a los extremos 5' y 3' que forman cada cadena de ADN. Como una secuencia de ADN está formada por dos cadenas de ADN complementarias (véase la Figura 2.1), habrá una segunda cadena con extremos 5' y 3' pero en sentido contrario.

La secuenciación single-end se refiere a secuenciar el ADN empezando desde un extremo de una cadena, una solución que permite generar un gran volumen de datos de forma rápida y económica. Por otro lado, la secuenciación paired-end utiliza dos lectores, donde cada uno lee una de las dos cadenas empezando en extremos opuestos y leyendo en dirección opuesta. Por ejemplo, en la Figura 2.1, un lector puede leer la cadena de la izquierda desde el extremo 5' al 3' (de arriba a abajo), mientras que otro lector leerá la cadena de la derecha desde el extremo 5' al 3' (de abajo a arriba).

La secuenciación paired-end permite obtener en general secuencias de mayor calidad y es capaz de detectar reordenamientos típicos del ADN como inserciones, deleciones e inversiones, pero también es más costosa, y a veces la precisión que ofrece single-end puede resultar suficiente para ciertos experimentos.

## 2.2 Tecnologías Big Data

A continuación, se ofrece una breve explicación sobre las principales tecnologías Big Data en las que se apoya el desarrollo de este proyecto.

### 2.2.1 MapReduce

MapReduce [18, 19] es un paradigma de procesamiento de datos caracterizado por dividirse en dos fases diferenciadas: Map y Reduce. Estos subprocesos asociados a la tarea se ejecutan de manera distribuida, en diferentes nodos de procesamiento o workers. Para controlar y gestionar su ejecución, existe un proceso Master o Job Tracker, que además es el encargado de aceptar los nuevos trabajos enviados al sistema por los clientes. MapReduce también es el nombre de la implementación propietaria de Google, quien propuso el modelo por primera vez en 2004, dada la necesidad de optimizar los resultados de las búsquedas de los usuarios en la web.

Para apoyar este tipo de procesamiento se utilizan tecnologías de almacenamiento de datos distribuidas, basadas en almacenar los datos en más de un nodo. Google utilizó el Google File System (GFS) [20], un sistema de ficheros distribuido de alto rendimiento que sigue una arquitectura master/worker.

A raíz del desarrollo de este modelo salieron a la luz otras tecnologías que lo implementaban. Es destacable la solución de código abierto Apache Hadoop [7]; o Apache Spark [4, 5], derivado del propio Hadoop. Como sistema de ficheros distribuido, Hadoop utiliza Hadoop Distributed File System (HDFS) [6], que se detallará más adelante. HDFS almacena los ficheros divididos en bloques de datos, con lo que ya proporciona la división previa de los datos que necesita Hadoop para su ejecución. Los resultados del procesamiento se pueden almacenar en el mismo sistema de almacenamiento HDFS, o bien en una base de datos o cualquier sistema externo soportado.

Por último, respecto a Spark, comentar que también proporciona integración con HDFS, aunque no es el único sistema de ficheros que soporta.

#### Fases en MapReduce

Las fases de un trabajo MapReduce son las siguientes:

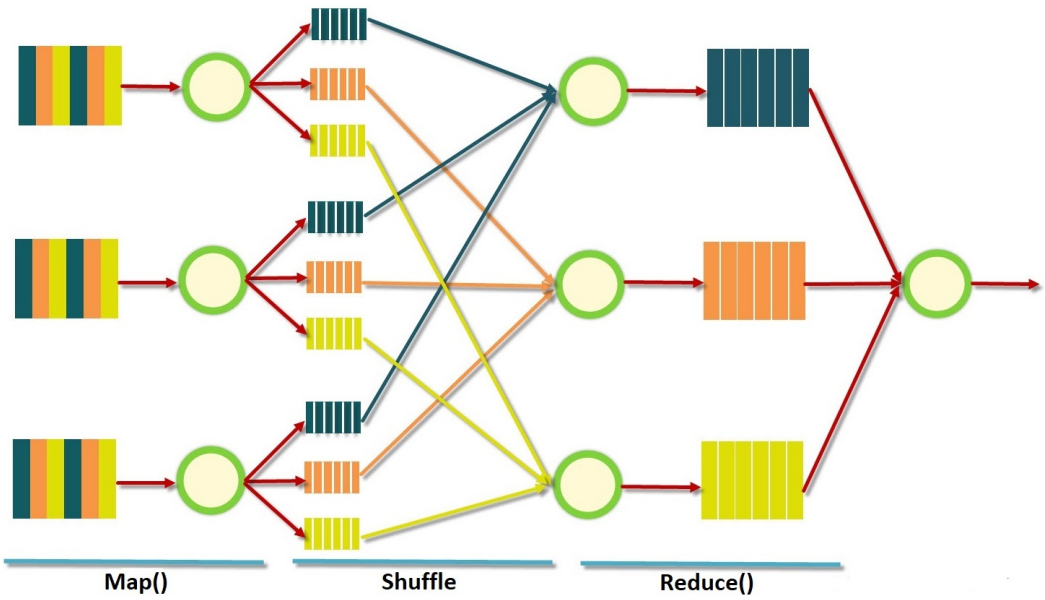


Figura 2.2: Flujo de MapReduce

- **Map**: toma como entrada un par clave-valor y devuelve una lista de pares clave-valor, pudiendo estar en un dominio diferente al dominio de entrada. Map se aplica en paralelo a cada ítem de la entrada de datos, generando listas de claves que distribuirá a los nodos encargados del procesamiento (workers) para que realicen el trabajo de forma paralela:

```
Map(k1,v1) -> list(k2,v2)
```

- **Shuffle**: es una fase intermedia entre Map y Reduce que se encarga de ordenar por clave los resultados generados por la fase Map y moverlos a cada nodo como entrada de la fase Reduce.

- **Reduce**: se aplica en paralelo para cada grupo generado por la fase Shuffle, es decir, se invoca una vez por cada clave única generada en la fase Map y genera un valor por cada invocación, obteniendo una nueva lista de pares clave-valor tras el final de esta fase:

```
Reduce(k2,list (v2)) -> list(k3,v3)
```

Para verlo de una forma gráfica, se muestra su funcionamiento en la Figura 2.2. La fase Map toma como entrada los datos repartidos en varios nodos. En la fase Shuffle los pares clave-valor con la misma clave se juntan y se envían al mismo nodo, donde se ejecutará la fase Reduce en cada uno de ellos, generando varias partes (ficheros) que se escribirán como salida del trabajo.

Como ya se comentó, Hadoop es la implementación de código abierto de MapReduce, y como su parte más interesante para este trabajo es HDFS, se detalla a continuación.

### 2.2.2 HDFS

Hadoop Distributed File System (HDFS) [6] es uno de los componentes principales del ecosistema Hadoop [7] que hace posible almacenar conjuntos de datos masivos con tipos de datos estructurados, semi-estructurados y no estructurados. HDFS está optimizado para almacenar grandes cantidades de datos y garantizar una alta disponibilidad y tolerancia a fallos, constituyendo una tecnología fundamental para el procesamiento Big Data.

HDFS es un sistema de ficheros distribuido implementado en Java que crea la abstracción de la existencia de un sistema de ficheros único, cuando físicamente los datos están repartidos en varios nodos. Esta distribución de los datos permite aumentar la velocidad de procesamiento y el paralelismo en las operaciones, y posibilita la replicación de los datos, la cual es clave para su alta disponibilidad.

La forma de almacenar cada fichero en HDFS es la siguiente: se divide en bloques de tamaño fijo (configurable por el usuario), por ejemplo de 128 MB, y estos se distribuyen entre los nodos que forman el clúster Hadoop. La replicación se aplica en función del llamado factor de replicación, que es un valor configurable (por defecto 3) que indica en cuántos nodos se debe guardar una copia de cada bloque que forma cada fichero. Naturalmente, esto también aumenta el almacenamiento necesario para los datos y el tiempo para su escritura, por lo que es necesario buscar un equilibrio entre una buena tolerancia a fallos y un buen rendimiento. Es posible desactivar la replicación indicando un factor de 1, con lo que cada bloque solo estará almacenado una vez.

HDFS sigue una arquitectura master/worker y, a continuación, se detallan sus componentes: el NameNode y los DataNodes.

- **NameNode**: el NameNode es un proceso Java que se ejecuta en el nodo maestro del clúster Hadoop. No se encarga de almacenar los datos en sí, sino de gestionar su acceso y almacenar sus metadatos. Se asemeja a una tabla de contenidos, en la que se asignan bloques de datos a los DataNodes. Debido a esto, necesita menos espacio de disco pero más recursos computacionales (memoria y CPU) que los DataNodes.

  Este componente es el único de HDFS que conoce la lista de ficheros y directorios de todo el clúster, así que el sistema de ficheros no se puede usar sin el NameNode. Para evitar que exista este punto único de fallo, en Hadoop 2 se permite la configuración de un NameNode secundario que toma el control si se detecta un fallo en el NameNode primario. A partir de Hadoop 3 se pueden configurar varios NameNodes secundarios.

- **DataNodes**: los DataNodes se corresponden con los procesos Java ejecutados en los nodos del clúster que almacenan los datos (los workers), encargándose de gestionar el almacenamiento local del nodo. Generalmente usan hardware básico con varios discos

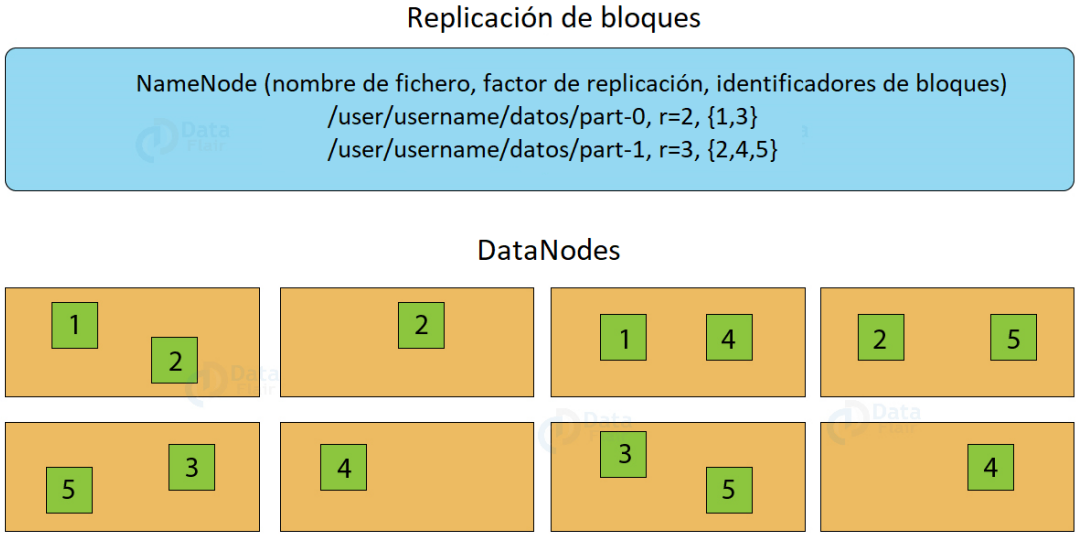


Figura 2.3: Replicación de datos en varios nodos en HDFS

locales de gran capacidad. A causa de su tipología, permiten aumentar la capacidad del sistema de una forma horizontal de forma efectiva y con un coste reducido.

Cabe mencionar que HDFS implementa un modelo write-once-read-many, lo que significa que no se pueden editar ficheros ya existentes, aunque sí se permite añadir datos nuevos. En las operaciones de escritura, el cliente debe comunicar la instrucción previamente al NameNode. Este comprueba los permisos y responde entonces al cliente con la dirección de los DataNodes en los que el cliente deberá empezar a escribir. Con un factor de replicación de 3, el primer DataNode copiará el bloque a otro DataNode, que entonces lo copiará a un tercero. Una vez que se han completado estas réplicas se enviará al cliente la confirmación de escritura. En las operaciones de lectura, el cliente pide al NameNode la localización de un fichero. Una vez que se han comprobado los permisos del cliente, el NameNode le envía la localización de los DataNodes que contienen los bloques que componen el fichero.

Para verlo de forma gráfica se incluye la Figura 2.3. En ella se observa que el NameNode tiene la información de dos ficheros, donde el primero tiene un factor de replicación de 2 y está formado por dos bloques (1 y 3). Por ello, se puede observar que los bloques 1 y 3 se encuentran replicados en dos DataNodes cada uno. La misma idea aplica al segundo fichero, que tiene en este caso un factor de replicación de 3 y está formado por tres bloques (2, 4 y 5).

### 2.2.3 Apache Spark

Apache Spark [4, 5] es un framework para el procesamiento de datos distribuidos diseñado para ser rápido y de propósito general. Como su propio nombre indica, ha sido desarrollado

en el marco del proyecto Apache, lo que garantiza su licencia de código abierto.

Como se comentó anteriormente, Spark surge como una evolución de Hadoop [7], cuya funcionalidad es muy rígida y limitada en el sentido de que no aprovecha al máximo las capacidades del procesamiento distribuido. Algunas de las mejoras que supone Spark frente a su predecesor son el procesamiento en memoria que disminuye las operaciones de lectura/escritura (puede llegar a ser hasta 100 veces más rápido que Hadoop al trabajar en memoria RAM, ya que Hadoop siempre almacena los resultados intermedios en disco), la posibilidad de análisis interactivo con SQL y la facilidad para interactuar con múltiples sistemas de almacenamiento persistente. Otra mejora clave para el desarrollo de este trabajo es que Spark permite procesamiento de flujos de datos o streams.

Cabe mencionar que Spark no almacena datos en sí mismo, centrándose en el procesamiento, a diferencia de Hadoop, que incluye tanto un sistema de almacenamiento persistente (HDFS) como un sistema de procesamiento de una manera muy integrada. Con lo cual, lo habitual es complementar Spark con un sistema de almacenamiento, como HDFS, aunque se permiten otras tecnologías.

Para entender esta herramienta de una forma más profunda, a continuación se exponen los conceptos clave de Spark:

- **Directed Acyclic Graph (DAG)**: un DAG es un grafo dirigido que no tiene ciclos, es decir, para cada nodo del grafo no hay un camino directo que comience y finalice en dicho nodo. Un vértice se conecta a otro, pero nunca a sí mismo. Aplicado a Spark, cada trabajo suyo crea un DAG de etapas que se ejecutan en el clúster. En comparación con MapReduce, que crea un DAG con dos estados predefinidos (Map y Reduce), los grafos DAG en Spark pueden tener cualquier número de etapas. Spark con DAG es más rápido que MapReduce, ya que tiene la capacidad de trabajar en memoria y solo escribirá a disco cuando lo necesite. Por el contrario, MapReduce siempre debe escribir en disco los resultados obtenidos en las etapas intermedias del grafo, es decir, entre las etapas Map y Reduce.

- **Resilient Distributed Datasets (RDDs)** [21]: un RDD se define como una colección de elementos inmutable, particionada a lo largo del clúster, tolerante a fallos y capaz de operar en paralelo. Constituyen la principal abstracción de datos en Spark y se encuentran distribuidos sobre los nodos del clúster. La inmutabilidad hace referencia a que no se puede cambiar un RDD ya existente, solo se pueden generar nuevos RDD a través de diferentes operaciones. Se pueden realizar dos tipos de operación sobre un RDD:

  - **Transformaciones**: Construyen un nuevo RDD a partir de uno existente. Es el caso de, por ejemplo, la operación map, que genera un nuevo RDD tras aplicar una función definida por el usuario sobre cada elemento del RDD de entrada; o

filter, que genera un nuevo RDD tras eliminar aquellos elementos del RDD que no cumplan una condición determinada.

- **Acciones**: Llevan a cabo una operación sobre un RDD y obtienen un valor resultado. Es el caso de, por ejemplo, count, que obtiene el número total de elementos de un RDD.

Gracias al DAG que crea Spark, al definir un nuevo RDD se guarda la secuencia de pasos necesaria para obtenerlo a partir del RDD original. Esto proporciona, por un lado, tolerancia a fallos, ya que ante cualquier fallo de un nodo del clúster se pueden recuperar los datos que pudiese tener simplemente volviendo a aplicar las operaciones del DAG. Por otro lado, también permite retrasar la ejecución efectiva de las transformaciones hasta que se ejecute una acción, utilizando un mecanismo de evaluación perezosa. Esto es interesante porque habilita que internamente se pueda optimizar la secuencia de transformaciones para obtener el mismo resultado de una forma más óptima evitando crear copias de los estados intermedios en cada paso.

#### Arquitectura

De forma similar a Hadoop, Spark sigue una arquitectura master/worker (véase la Figura 2.4), con lo que se tendrán un nodo maestro y uno o varios workers [22]. Por otro lado está el driver, que ejecuta la función main() de la aplicación Spark que genera el DAG, y crea el objeto SparkContext que coordina las aplicaciones y permite la conexión con el gestor de recursos del clúster. Este gestor puede ser el Standalone que ya viene integrado con el propio Spark u otros independientes como Hadoop YARN [23]. Es habitual que la aplicación driver se ejecute en el nodo maestro, pero no es estrictamente necesario. Es posible ejecutar el driver en un nodo worker o incluso desde una máquina externa que haga la petición del trabajo al nodo maestro correspondiente, aunque eso dependerá del gestor de recursos utilizado y su configuración. Una vez que se ha conectado con el gestor, el driver se encarga de obtener los ejecutores (Executors) en los nodos worker del clúster, que son los procesos que realizan los trabajos computacionales y almacenan los RDDs de la aplicación. Posteriormente, el driver envía el código de las tareas de procesamiento a los Executors para su ejecución.

#### Módulos adicionales

Las funcionalidades comentadas previamente son las consideradas como básicas que vienen en el módulo Spark Core, pero hay otros componentes que extienden sus funcionalidades:

- **Spark SQL**: pensado para el procesamiento de datos estructurados utilizando consultas relacionales. Con este módulo aparecen dos nuevas estructuras para almacenar los datos, los DataFrames y los Datasets.

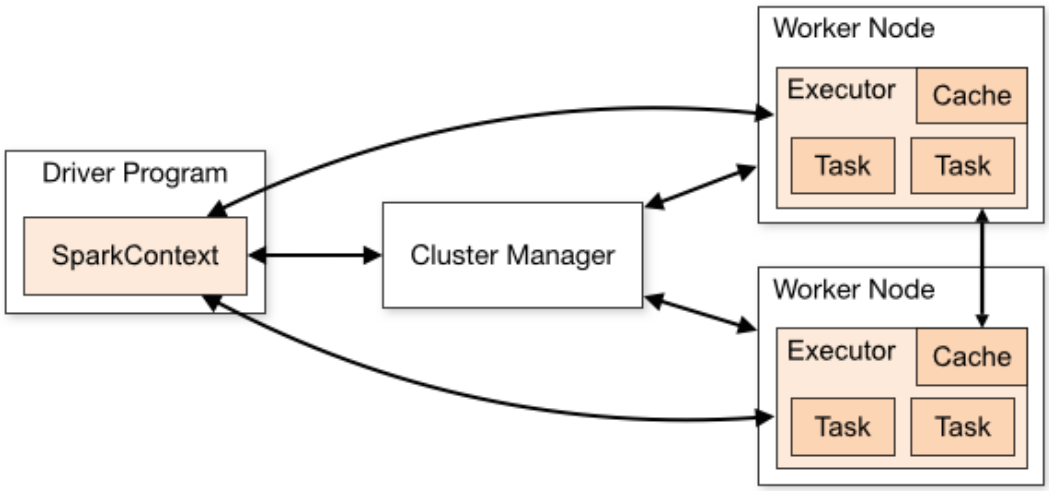


Figura 2.4: Arquitectura master/worker de Spark

- **Spark Streaming**: permite procesar flujos de datos en streaming, utilizando el concepto de los streams discretizados o DStreams, basados en dividir el flujo de datos entrante en pequeños lotes (batches) que se procesan rápidamente (micro-batch processing).
- **Spark Structured Streaming**: ofrece un modelo de streaming similar al anterior, basado en micro-batches, pero presenta una nueva API apoyada en los DataFrames y Datasets. Este módulo se detalla en la Sección 2.2.4 ya que es el utilizado en la implementación de SeQual-Stream.
- **MLlib**: es una biblioteca que incluye algoritmos de Machine Learning aprovechando las capacidades de Spark.
- **GraphX**: permite realizar computación paralela sobre grafos.

#### DataFrames y Datasets

Como se mencionó previamente, el módulo de Spark SQL proporciona dos alternativas a los RDDs para almacenar los datos, los DataFrames y los Datasets [24]:

- Los **DataFrames** organizan los datos en columnas, similar a una tabla en una base de datos relacional. A diferencia de los RDDs, permiten un mejor manejo de datos estructurados y se apoya en Spark Catalyst para optimizar las consultas realizadas.
- Los **Datasets** son una extensión de los DataFrames e intentan combinar las ventajas de los RDDs y los DataFrames en la misma API, es decir, la comodidad de uso de los RDDs y la optimización de rendimiento a través de Catalyst de los DataFrames. Además, añade

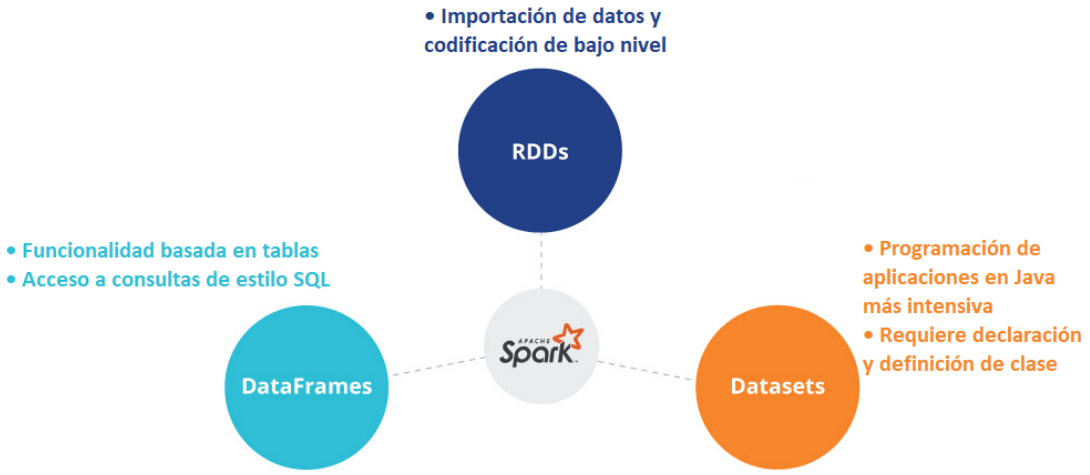


Figura 2.5: RDDs, DataFrames y Datasets en Spark

el concepto de los Encoders, que permiten representar los datos de la tabla como objetos Java, ofreciendo seguridad de tipado en compilación a diferencia de los DataFrames, donde un error de ese tipo se detectaría en ejecución. La construcción Dataset<Row> es una forma de indicar que el tipado del Dataset es genérico (no se conoce el tipo exacto), siendo equivalente a un DataFrame.

Tanto RDDs, como DataFrames y Datasets (véase la Figura 2.5) son formas válidas de representar los datos y, en general, funcionan de manera similar. Los DataFrames y Datasets se consideran más eficientes y ofrecen una vista de más alto nivel, aunque en algunos casos se puede preferir el uso de RDDs si se desea una funcionalidad y control a más bajo nivel.

### 2.2.4 Spark Structured Streaming

Mientras que SeQual utiliza la API para procesamiento por lotes (batch) de Spark basada en los RDDs, SeQual-Stream se desarrolló con el módulo de Spark Structured Streaming [25], utilizando los Datasets como estructura de datos. Por esto mismo, en este apartado se profundiza en dicho módulo.

Structured Streaming permite expresar los trabajos en streaming de la misma forma que se expresa un trabajo batch con datos estáticos. El motor de Spark SQL se encarga de ejecutarlo de forma incremental y continua, actualizando el resultado final a medida que llegan los datos en streaming.

Por defecto, las consultas de Structured Streaming se procesan internamente utilizando un modelo de micro-batches, procesando los streams como una serie de pequeños trabajos por lotes (batch). Este modo de procesamiento streaming consigue unas latencias de unos 100

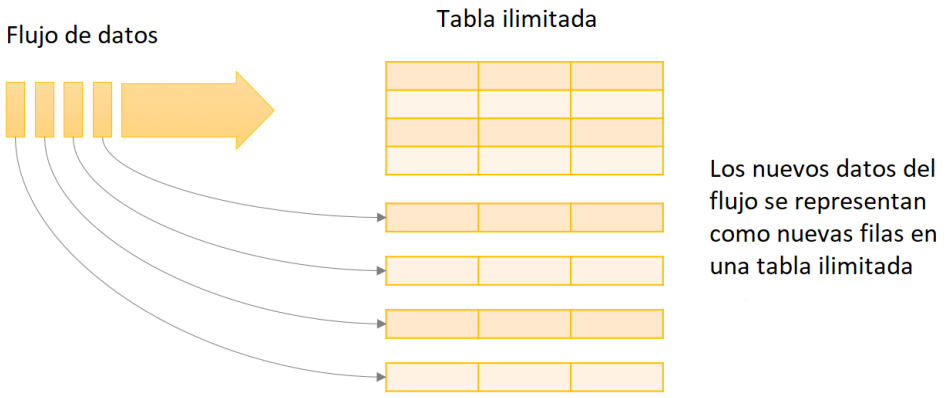


Figura 2.6: Modelo de Structured Streaming con micro-batches

ms y garantiza tolerancia a fallos. Desde Spark 2.3 también se permite un nuevo modo de procesamiento denominado Continuous Processing que puede conseguir latencias de 1 ms. Debido a que este nuevo modo todavía está en fase experimental, este TFG utilizó el modo por defecto de Structured Streaming.

Profundizando en el funcionamiento del modelo de micro-batches, la idea principal es tratar los datos como una tabla relacional a la que se le van añadiendo filas a medida que llegan nuevos datos, como se muestra gráficamente en la Figura 2.6. Cada vez que se actualice la tabla resultado, esta se escribe (persiste) en un sistema externo que puede ser un fichero o un topic de Kafka, entre otras opciones.

En función de los datos que se quieran escribir en la salida se diferencian 3 modos:

- **Complete mode**: en este modo se escriben todas las filas de la tabla en cada actualización.
- **Append mode**: en este modo se escriben solo las filas nuevas. Esto hace que solo sea aplicable cuando se espera que las filas existentes no cambien.
- **Update mode**: en este modo se escriben las filas que hayan cambiando desde la última actualización de la tabla.

Para proporcionar un ejemplo más concreto, se representa en la Figura 2.7 la ejecución de una consulta WordCount en la que los datos de entrada se escriben por consola con la herramienta nc, mientras que la aplicación Spark los captura para su procesamiento en streaming.

En el instante de tiempo $t = 1$, la aplicación ha leído las palabras “cat dog” y “dog dog”, con lo que ejecuta la consulta que realiza el conteo de palabras y se almacena en la tabla

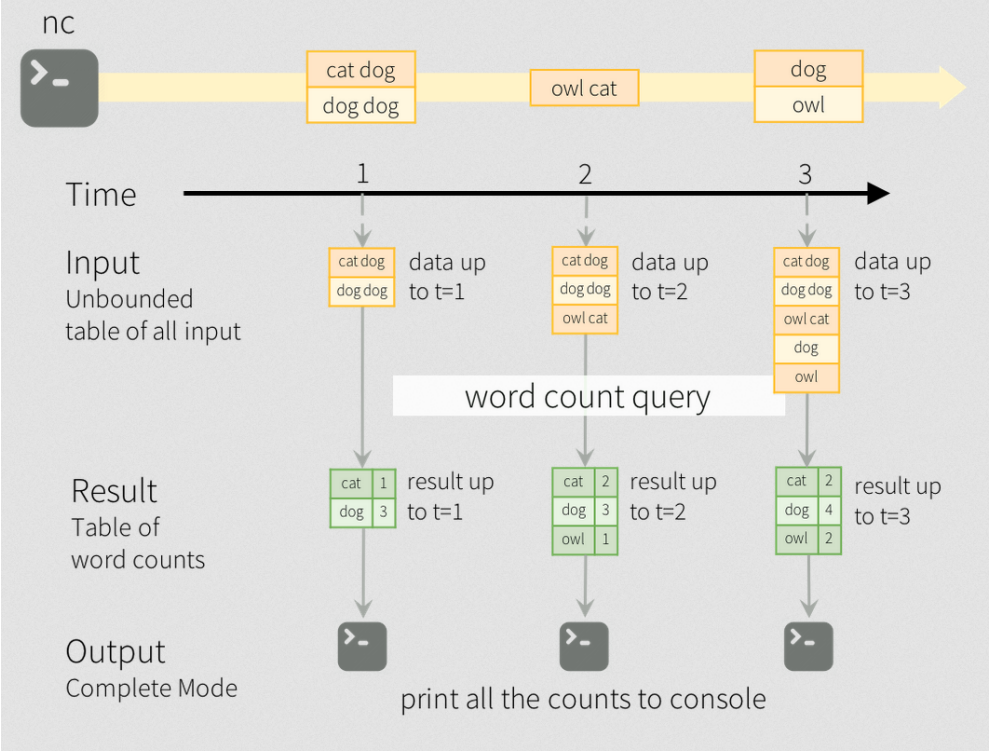


Figura 2.7: Ejemplo de WordCount en Spark Structured Streaming

resultado los valores “cat 1” y “dog 3”. A continuación, se escriben los resultados de salida por consola. En $t = 2$ recibe “owl cat”, actualizando así la fila que contenía “cat” y añadiendo una nueva fila para “owl”. Ya que el modo para la salida está establecido en Complete, se escribe por consola toda la tabla resultado. Para $t = 3$ el razonamiento a aplicar sería el mismo. Si el modo para la salida fuese Update, en $t = 1$ se escribiría lo mismo que antes, en $t = 2$ se escribiría “cat 2” y “owl 1”, pero no “dog 3” ya que no cambió desde la última vez, y en $t = 3$ se escribiría “dog 4” y “owl 2”. Esta consulta WordCount no sería aplicable en modo Append ya que las filas existentes podrían cambiar (y efectivamente, lo hacen).

Capítulo 3

# Herramientas y metodología

A continuación se comentan las diferentes herramientas que se utilizaron para el desarrollo de SeQual-Stream. También se exponen la metodología y plan de trabajo seguidos para el desarrollo del proyecto.

## 3.1 Tecnologías y herramientas

Dentro de las tecnologías y herramientas utilizadas, se pueden diferenciar aquellas que son claves para el funcionamiento de SeQual-Stream de las que se usaron como soporte al desarrollo.

### 3.1.1 Fundamentales

- **SeQual** [2, 3]: herramienta paralela de control de calidad de secuencias genéticas que se usó como base del desarrollo de la versión streaming.
- **Java** [26]: lenguaje de programación orientado a objetos con el que está escrito SeQual.
- **Spark** [4, 5]: framework de procesamiento Big Data.

### 3.1.2 Apoyo al desarrollo

- **Maven** [27]: herramienta para la gestión y construcción de proyectos Java, que permite la automatización de descarga e instalación de dependencias o el empaquetado de la aplicación para su uso.
- **GitHub** [28]: repositorio para almacenar proyectos software y mantener un control de versiones.
- **Eclipse** [29]: IDE de desarrollo con soporte para proyectos Java, entre otros.

- **BDEv** [30]: herramienta de benchmarking para la evaluación de frameworks de procesamiento Big Data. Utilizado para conseguir un despliegue rápido de un clúster Spark con HDFS en el que poder ejecutar aplicaciones Spark.
- **SSH**: herramienta que permite el acceso remoto a otros equipos, utilizado para la conexión con el clúster donde se realizó la evaluación del rendimiento.
- **Microsoft Project** [31]: herramienta para la administración de proyectos.

## 3.2 Metodología

La metodología seguida en este TFG ha sido la iterativa e incremental. Consiste en planificar el proyecto dividiéndolo en diversos bloques temporales llamados iteraciones. En cada iteración se repite un proceso de trabajo similar (por ejemplo, análisis, diseño, implementación y pruebas) para proporcionar un resultado completo sobre el producto final. Un esquema de este modelo se puede ver en la Figura 3.1. Así, las funcionalidades del proyecto software se van ampliando y/o mejorando con cada iteración, y al final de cada una se obtiene un producto utilizable por el cliente.

Son múltiples los beneficios que aporta esta metodología. Por destacar los más interesantes, se puede mencionar la flexibilidad que ofrece, ya que permite reunirse con el cliente (que en este caso serían los tutores del TFG) de forma periódica con la finalización de cada iteración para revisar el estado del producto y tomar decisiones respecto a cómo continuar, especialmente útil en proyectos donde el propio cliente no tiene del todo claro los requisitos

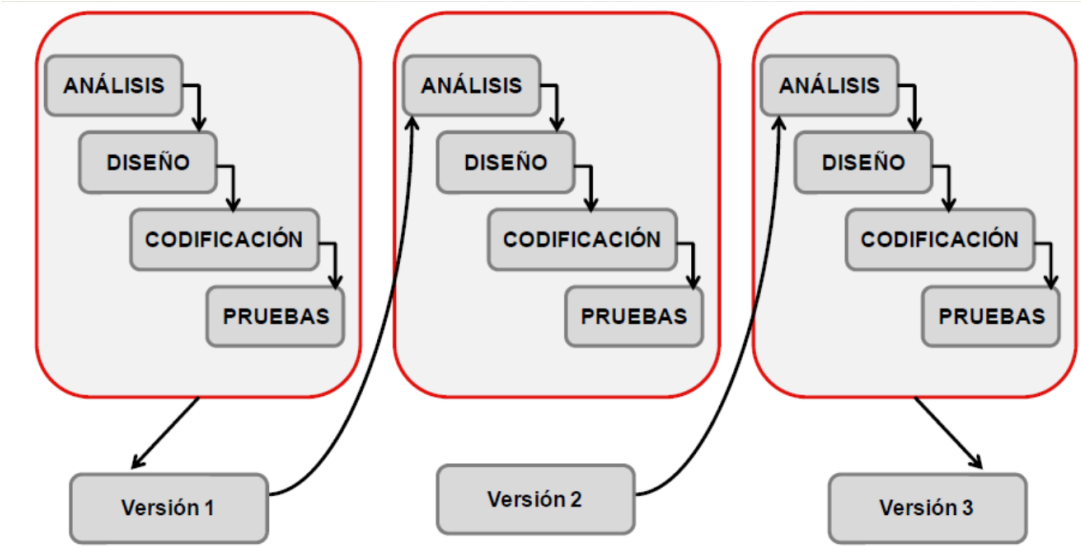


Figura 3.1: Esquema de la metodología iterativa incremental

que necesita. Otra gran ventaja es que permite gestionar mejor la complejidad del proyecto, al estar dividido en partes más pequeñas y fácilmente gestionables, en contraposición a considerar la totalidad del proyecto desde el inicio.

### 3.2.1 Aplicación de la metodología

Aplicando esta metodología al presente TFG, se definieron los siguientes incrementos:

1. **Flujo básico**: como primer paso, se define un flujo simple de ejecución desde la lectura de un fichero, su procesamiento a través de una única operación y su escritura final, utilizando la API de Spark Structured Streaming, aunque en este punto no es necesario que el procesamiento soporte el funcionamiento para ficheros en descarga. Se empieza con ficheros FASTQ, para simplificar más adelante la adaptación para FASTA.

2. **Procesamiento en streaming**: esta iteración se centra en poder procesar un fichero mientras se está descargando desde un servidor remoto o desde Internet.

3. **Procesamiento de secuencias paired-end**: en este incremento se añade la funcionalidad de procesar secuencias pareadas, basada en procesar dos ficheros de secuencias a la vez.

4. **Implementación del resto de operaciones**: se adaptan el resto de operaciones a soportar por SeQual-Stream y el procesamiento de ficheros en formato FASTA.

5. **Mejoras y optimizaciones**: se realiza un análisis general para identificar partes mejorables o alguna posible optimización.

6. **Resolución de cuellos de botella**: por último, una vez llevada la ejecución de la aplicación a un entorno clúster real, se analizan los posibles cuellos de botella que es previsible que puedan surgir, y así, optimizar los tiempos de ejecución finales de la herramienta.

En la Figura 3.2 se muestra el diagrama de Gantt correspondiente al desarrollo a lo largo del tiempo.

### 3.2.2 Planificación general

Las fases en las que se reparte el trabajo en general son las siguientes:

1. **Preparación**: se requiere una primera fase de formación en las herramientas a utilizar y en la propia herramienta SeQual que sirve de base del trabajo, así como una introducción a los conceptos relacionados con el dominio que rodea a la misma.

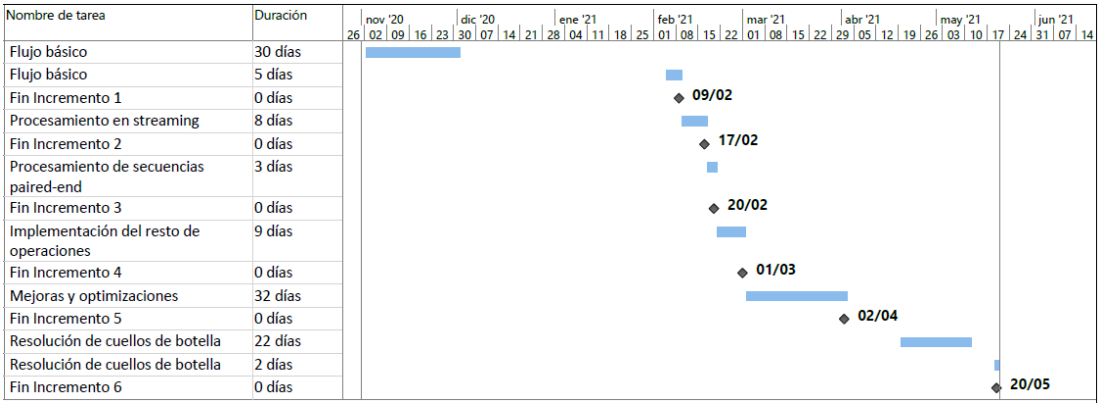


Figura 3.2: Diagrama de Gantt del desarrollo de los incrementos

2. **Desarrollo de la herramienta**: esta es la fase de implementación de la aplicación, realizada de forma iterativa incremental. Cada incremento tiene las siguientes fases genéricas:

   - Análisis: unas primeras consideraciones sobre el estado actual del progreso y los siguientes pasos.
   - Diseño: un diseño más concreto de las decisiones tomadas en la fase anterior.
   - Codificación: la implementación que llevará a cabo las funcionalidades buscadas en el incremento.
   - Pruebas: una fase de pruebas para verificar el correcto desempeño del código.

3. **Evaluación en un entorno clúster y detección de posibles cuellos de botella**: en esta fase se lleva la aplicación SeQual-Stream a un entorno clúster y se evalúa su rendimiento en comparación con SeQual para analizar la mejora obtenida. En el caso de detectar cuellos de botella que ralenticen su ejecución, se lleva a cabo un incremento para solventarlos.

4. **Documentación**: por último, una fase de redacción de la memoria que recoge el trabajo realizado.

La supervisión del proyecto con los tutores se realizó al final de cada iteración del desarrollo de la herramienta, comentando su estado en ese momento de forma telemática.

Capítulo 4

# Funcionalidades

En este capítulo se exponen todas las funcionalidades ofrecidas por la herramienta desarrollada en este TFG. Las funcionalidades principales son un subconjunto de las de SeQual, que están divididas en Filtrado individual y grupal, Recortado, Formateado y Estadísticas. Como se comentó en la Sección 1.2, las operaciones consideradas en este trabajo son las que operan sobre cada secuencia de forma individual, es decir, las funcionalidades de Filtrado individual, Recortado y Formateado. También existen otras funcionalidades transversales, necesarias para el funcionamiento de la herramienta.

## 4.1 Filtrado individual

Con el objetivo de establecer ciertos límites de calidad, se permite realizar filtrados sobre las secuencias en función de ciertos parámetros establecidos por el usuario. Concretamente, con los filtros individuales se comprueba cada secuencia para valorar si debe conservarse o eliminarse del resultado.

Los filtros individuales disponibles actualmente son:

- **LENGTH**: Filtra las secuencias de acuerdo a una longitud máxima y/o mínima indicada.
- **QUALITY**: Filtra las secuencias de acuerdo a una calidad media máxima y/o mínima indicada. La puntuación de calidad de cada base se calcula siguiendo el encoding de Illumina [14].
- **QUALITYSCORE**: Filtra las secuencias de acuerdo a una calidad por base máxima y/o mínima indicada, eliminándolas si cualquiera de sus bases no se encuentra entre los límites. La puntuación de calidad de cada base se calcula de igual forma que en el filtro anterior.
- **NONIUPAC**: Filtra las secuencias si contienen bases Non-IUPAC (es decir, cualquier otra base distinta de A, T, G, C o N).

- **GCBASES**: Filtra las secuencias de acuerdo a su cantidad de bases G(uanina) y C(itosina) según una cantidad máxima y/o mínima indicada.
- **GCCONTENT**: Filtra las secuencias de acuerdo a su contenido (porcentaje) de bases G(uanina) y C(itosina) según un contenido máximo y/o mínimo indicado.
- **NAMB**: Filtra las secuencias de acuerdo a su cantidad de bases N ambiguas según una cantidad máxima y/o mínima indicada.
- **NAMBP**: Filtra las secuencias de acuerdo a su porcentaje de bases N ambiguas según un porcentaje máximo y/o mínimo indicado.
- **BASEN**: Filtra las secuencias en función de si contienen un número mínimo y/o máximo de uno o varios tipos de base (o incluso grupos de bases).
- **BASEP**: Filtra las secuencias en función de si contienen un porcentaje mínimo y/o máximo de uno o varios tipo de base (o incluso grupos de bases).
- **PATTERN**: Filtra las secuencias en función de la ausencia de un patrón especificado (es decir, si no contiene el patrón, la secuencia se elimina) junto con sus repeticiones (por ejemplo, el patrón ATC con dos repeticiones sería ATCATC).
- **NOPATTERN**: La contraparte del anterior, filtra las secuencias en función de la existencia de un patrón especificado (es decir, si contiene el patrón, la secuencia se elimina) junto con sus repeticiones.

## 4.2 Recortado

Otra forma para mejorar la calidad de los conjuntos de datos se basa en el recortado (o trimming). Consiste en permitir eliminar ciertas bases que se consideren sobrantes de cada secuencia, de distintas formas y en función de los parámetros indicados por el usuario.

Los recortadores disponibles actualmente son:

- **TRIMLEFT**: Recorta las secuencias de acuerdo a una cantidad de posiciones indicada a partir del extremo 5' (véase la Figura 2.1), correspondiente con la izquierda de la cadena de bases.
- **TRIMRIGHT**: Recorta las secuencias de acuerdo a una cantidad de posiciones indicada a partir del extremo 3', correspondiente con la derecha de la cadena de bases.
- **TRIMLEFTP**: Recorta las secuencias de acuerdo a un porcentaje del total de bases indicado a partir del extremo 5' (izquierda).

- **TRIMRIGHTP**: Recorta las secuencias de acuerdo a un porcentaje del total de bases indicado a partir del extremo 3' (derecha).
- **TRIMQUALLEFT**: Recorta las secuencias hasta alcanzar una calidad media de secuencia especificada a partir del extremo 5' (izquierda).
- **TRIMQUALRIGHT**: Recorta las secuencias hasta alcanzar una calidad media de secuencia especificada a partir del extremo 3' (derecha).
- **TRIMNLEFT**: Recorta los N-terminal tails con una longitud mínima especificada en el extremo 5' (izquierda). Un N-terminal tail es un conjunto de bases N encontrado al inicio o fin de una secuencia. Por ejemplo, las tres Ns de la secuencia NNNATCGAT forman un N-terminal tail al inicio.
- **TRIMNRIGHT**: Recorta los N-terminal tails con una longitud mínima especificada en el extremo 3' (derecha).
- **TRIMLEFTTOLENGTH**: Recorta las secuencias hasta alcanzar una longitud máxima especificada a partir del extremo 5' (izquierda).
- **TRIMRIGHTTOLENGTH**: Recorta las secuencias hasta alcanzar una longitud máxima especificada a partir del extremo 3' (derecha).

## 4.3 Formateado

Esta funcionalidad es la que permite realizar cambios de formato en los conjuntos de datos. Los formateadores disponibles actualmente son:

- **DNATORNA**: Transforma las secuencias ADN a secuencias ARN.
- **RNATODNA**: Transforma las secuencias ARN a secuencias ADN.
- **FASTQTOFASTA**: Transforma las secuencias en formato FASTQ a secuencias en formato FASTA, perdiendo la información de la calidad de las bases.

## 4.4 Transversales

Por último, en este apartado se agrupan las funcionalidades necesarias para el uso y configuración de la herramienta por parte del usuario. Son las siguientes:

- **Lectura de datasets en formato FASTA**: Permite leer datasets de secuencias en formato FASTA, tanto si están completos o en proceso de escritura/descarga. Pueden estar almacenados en HDFS o en otro sistema de ficheros.

- **Lectura de datasets en formato FASTQ**: Permite leer datasets de secuencias en formato FASTQ, tanto si están completos o en proceso de escritura/descarga. Pueden estar almacenados en HDFS o en otro sistema de ficheros.
- **Lectura de datasets paired-end en formato FASTA**: Permite leer datasets de secuencias paired-end en formato FASTA, tanto si están completos o en proceso de escritura/descarga. Pueden estar almacenados en HDFS o en otro sistema de ficheros. Las secuencias paired-end deben estar separadas en dos ficheros diferentes.
- **Lectura de datasets paired-end en formato FASTQ**: Permite leer datasets de secuencias paired-end en formato FASTQ, tanto si están completos o en proceso de escritura/descarga. Pueden estar almacenados en HDFS o en otro sistema de ficheros. Las secuencias paired-end deben estar separadas en dos ficheros diferentes.
- **Escritura de secuencias resultantes**: Permite escribir las secuencias resultantes tras las operaciones en la ruta indicada, generando dos carpetas distintas en caso de ser paired-end. Este tipo de escritura se realiza por defecto, escribiendo el resultado en múltiples ficheros de texto dentro de varias subcarpetas.
- **Escritura de secuencias resultantes a fichero individual**: Permite escribir las secuencias resultantes tras las operaciones en un fichero individual en la ruta indicada (con el mismo formato que el de entrada o en FASTA si se aplicó el formateador FASTQTOFASTA a un fichero FASTQ), o en dos ficheros en caso de ser paired-end.
- **Configuración de modo de ejecución de Apache Spark**: Permite configurar el modo de ejecución de Spark, siendo "local[*]" por defecto (lo que implica emplear todos los núcleos de la máquina en la que se lanza el trabajo).
- **Configuración de nivel de log mostrado al usuario**: Permite configurar el nivel de log que se le mostrará al usuario por parte de Spark, así como de otras bibliotecas. El nivel por defecto es ERROR.
- **Generación y lectura de fichero de configuración/especificación de parámetros**: Permite generar un fichero plantilla donde se pueden especificar las diferentes operaciones a realizar, así como los parámetros necesarios para las mismas.
- **Selección de modo Batch o modo Streaming**: Permite seleccionar el uso de SeQual (versión batch) o el uso de SeQual-Stream.

# Capítulo 5

# Desarrollo

En este apartado se expone el desarrollo completo de SeQual-Stream, dividido en los incrementos explicados en la Sección 3.2.

## 5.1 Preparación

Esta primera fase sirvió como preparación y formación sobre las diferentes herramientas y tecnologías claves para todo el trabajo a realizar posteriormente.

En primer lugar, se realizaron pruebas con la herramienta SeQual con el objetivo de familiarizarse y adquirir un primer nivel de entendimiento de sus funcionalidades generales. Además, se aprendieron conceptos acerca de la bioinformática relacionados directamente con el uso de la herramienta, tales como los formatos de secuencias FASTQ y FASTA o la secuenciación single-end y paired-end.

El siguiente paso consistió en adquirir formación acerca del framework de procesamiento Big Data Spark, que constituye la base del proyecto. Este aprendizaje se realizó mediante la lectura de su propia documentación y de otros manuales técnicos, además de probar su funcionamiento con algún ejemplo básico. Una vez adquiridos los conocimientos básicos sobre Spark, se pasó a profundizar sobre el módulo de Structured Streaming, también leyendo la documentación oficial y realizando pruebas con ejemplos integrados en la descarga del propio framework.

Por último, se estudió en detalle el código fuente de SeQual, para ver cómo utiliza internamente Spark y analizar aquellas partes claves a tener en cuenta en su adaptación a un paradigma de procesamiento en streaming.

## 5.2 Incremento 1 - Flujo básico

En esta fase se inició la implementación de la nueva herramienta SeQual-Stream, empezando por el primer incremento que establece un flujo básico de ejecución. Esto incluye la lectura de un fichero de entrada con las secuencias, su procesamiento por parte de una única operación de SeQual y su escritura final en disco. La lectura inicial se realizó con un único fichero (modo single-end) y en formato FASTQ. Una vez que se tiene la implementación con FASTQ, el desarrollo posterior para el soporte FASTA resulta más sencillo.

### 5.2.1 Lectura

Empezando con la parte de la lectura del fichero de entrada, la API de Structured Streaming funciona monitorizando un directorio indicado por el usuario y procesando los ficheros que se escriban en el mismo. Un parámetro importante a indicar es el tipo de fuente de entrada, que en este caso será simplemente de tipo fichero. Habiendo escogido esta clase, se debe seleccionar también el formato que tienen los ficheros a procesar, pudiendo elegir entre texto, CSV, JSON, ORC o Parquet. Como ninguno se ajusta específicamente al caso de uso de la herramienta, se escogió el formato de texto que tiene un carácter genérico.

Cuando la API de Spark Structured Streaming lee un fichero, carga sus datos en varias filas de la tabla de resultados y, por defecto, en cada fila de la tabla carga una línea del fichero. Esto no es adecuado para trabajar con el formato FASTQ, ya que lo interesante es poder cargar cada secuencia (4 líneas, ver Listado 2.1) en una única fila de la tabla para posteriormente realizar el procesamiento sobre cada fila. Sin embargo, la API permite especificar un carácter o cadena separador de línea diferente al utilizado por defecto (que es el salto de línea: \n). La elección del carácter/cadena separador no es algo trivial, ya que se podría pensar en utilizar el carácter '@' puesto que todas las secuencias FASTQ empiezan con él, pero hay que considerar que la línea de calidad también puede contener ese carácter, invalidando, por tanto, su uso. En esta primera versión de la herramienta se utilizó como separador una cadena contenida como inicio en todos los nombres de secuencias del fichero de pruebas usado ("@ERR"), lo que permite realizar el mapeo de las 4 líneas de cada secuencia a una fila de la tabla de forma correcta. Este método solo funcionaría para este fichero de pruebas (o similares). Por ello, en un incremento posterior se resuelve este aspecto para soportar cualquier fichero de entrada.

Con todo lo expuesto anteriormente, la herramienta ya sería capaz de cargar 4 cadenas de texto en una fila de la tabla, pero lo interesante es tener un objeto secuencia por fila. Por tanto, se añadió una función map que procesa cada fila y genera una secuencia a partir de sus 4 líneas, generando un Dataset tipado con la clase que representa a una secuencia, obteniendo así un Dataset<Sequence> . Esto garantiza que en dicho Dataset solo se almacenan secuencias correctamente formadas. En el Listado 5.1 se muestra un fragmento del código encargado de

```
Dataset<Row> lines = sparkSession
    .readStream()
    .option("lineSep", "@ERR")
    .text(inputDir);

Dataset<Sequence> seqs = lines
    .filter("value != ''")
    .map( tuple -> {
        String[] sequence = tuple.toString().split("\\n");

        return new Sequence(sequence[0].substring(1), sequence[1],
    sequence[2], sequence[3]);

    }, Encoders.bean(Sequence.class));
```

Listado 5.1: Fragmento de código de lectura de secuencias single-end en formato FASTQ

realizar el proceso de lectura descrito.

### 5.2.2 Escritura

El siguiente paso fue la escritura, obteniendo así un flujo de funcionamiento muy sencillo en el que simplemente se escribe en la salida el fichero leído en la entrada. La API Structured Streaming necesita una indicación de la forma de realizar la escritura. Si se indica una ruta de destino, se interpreta que la escritura se hará en un fichero almacenado en disco, pero hay otras opciones como el envío a un topic de Kafka. De la misma forma que en la lectura, al seleccionar la escritura en fichero hay que indicar un formato soportando las mismas opciones, así que se optó por el formato de texto genérico. Previamente a la escritura, se aplica una función map para transformar las secuencias a cadenas de texto y obtener así el formato requerido (Dataset<String>).

Otra opción importante es el modo de salida, pudiendo escoger entre Append (por defecto), Complete y Update (véase la Sección 2.2.4). Se optó por el modo Append, ya que solo escribe las nuevas filas y en nuestro caso de uso nunca se van a modificar filas (secuencias) ya procesadas. Este es el funcionamiento que interesa para las operaciones de filtrado individual, recortado y formateado que procesan cada secuencia de forma individual.

Por último, mencionar el parámetro que permite indicar un directorio donde Spark almacenará información del estado del trabajo para utilizarlo a modo de checkpoint y permitir la recuperación ante un fallo. La API devuelve un objeto de la clase StreamingQuery, encargado de llevar el estado de la petición. El flujo de ejecución Java continúa tras haber definido la petición de escritura, pudiendo acabar prematuramente, por lo que se añadió un bucle encargado de revisar el objeto StreamingQuery y analizar si el procesamiento terminó. Si esto es así se termina la petición, en caso contrario se esperan unos milisegundos y se vuelve a iterar

```
Dataset<String> seqs = sequences.map(s -> {
    return s.toString();
},Encoders.STRING());

StreamingQuery query = seqs.writeStream()
    .outputMode("append")
    .format("text")
    .option("path", output)
    .option("checkpointLocation", checkpointDir)
    .start();

while (query.isActive()) {
    if (!query.status().isDataAvailable() &&
        !query.status().isTriggerActive() &&
        !query.status().message().equals("Initializing sources")) {

        query.stop();
  }
  query.awaitTermination(1000);
}
```

Listado 5.2: Fragmento de código de escritura de secuencias single-end

sobre el bucle. Un ejemplo del código encargado de la escritura se encuentra en el Listado 5.2.

### 5.2.3 Primera operación

A continuación se implementó una operación para realizar algún tipo de procesamiento entre lectura y escritura, escogiendo el filtro de LENGTH (ver Sección 4.1) por su sencillez. La adaptación de este filtro al modo streaming fue un proceso relativamente rápido, ya que una de las ventajas comentadas de la API Structured Streaming es poder expresar un trabajo como si fuese batch. Fue necesario hacer el cambio de RDDs a Datasets y otros detalles que no funcionan al estar en un entorno streaming. Por ejemplo, en batch se hace una comprobación de si el RDD está vacío para decidir si realizar la operación; esto en streaming fallaría ya que al ser un proceso continuo no se permite comprobar si en un determinado momento el Dataset está vacío.

### 5.2.4 Selección del modo de procesamiento

Finalmente, se adaptó la herramienta para poder seleccionar el modo de procesamiento: batch o streaming. El módulo de SeQual que proporciona la interfaz de línea de comandos es SeQual-CMD, encargado de interpretar las opciones indicadas por el usuario en el fichero de configuración y en el propio comando usado para su ejecución, delegando en una clase llamada AppService que gestiona a nivel genérico toda la operativa interna. Se creó una nueva clase interfaz implementada de forma concreta por un AppService de batch y otro para streaming.

SeQual-CMD pasa así a depender de esa clase interfaz y se escoge la implementación concreta en función de un nuevo parámetro en el fichero de configuración.

## 5.3 Incremento 2 - Procesamiento en streaming

En el incremento anterior se obtuvo una versión funcional capaz de leer un fichero completo de entrada a través de la API Structured Streaming, pero sin poder procesarlo correctamente cuando el fichero está en proceso de escritura a HDFS y/o en descarga. El motivo es la forma de funcionar que tiene la monitorización de Spark sobre un directorio concreto. Aunque es posible detectar cuándo se escribe un nuevo fichero y procesarlo, no se tiene la capacidad de detectar actualizaciones en un fichero previamente leído; una vez que se procesa un fichero, este no se vuelve a tener en cuenta.

Para solventar este problema, el método que se diseñó fue crear un proceso que se encargase de leer el fichero de entrada y copiarlo al directorio de HDFS que es monitorizado por la aplicación Spark. El fichero se copiará dividido en uno o más ficheros, dependiendo del proceso de descarga. Esto también implica que el fichero de entrada no necesita estar almacenado en HDFS de forma obligatoria, ya que ese proceso se encargará de copiarlo en HDFS en el momento de la ejecución. Este nuevo método se lleva a cabo mediante un thread de forma paralela a la aplicación Spark, al que denominaremos como "thread de lectura".

El funcionamiento en detalle del thread de lectura es el siguiente: se copian todos los datos que haya disponibles del fichero de entrada en un momento dado a un nuevo fichero en HDFS dentro del directorio monitorizado por Spark. Cuando no haya más datos disponibles, y como realmente no se puede saber si es porque el fichero ya está completo o porque está todavía en proceso de descarga y no hay nuevos datos, se esperan unos segundos y se vuelve a iterar, copiando los nuevos datos a otro fichero diferente en HDFS. El proceso continúa iterando hasta que se supere un cierto tiempo de timeout en el que no se detecten nuevos datos, considerando que el fichero está completo. En este contexto, hay varias cuestiones que se deben considerar a continuación.

### 5.3.1 Secuencias incompletas

Cabe mencionar que la descarga de un fichero no se realiza teniendo en cuenta las secuencias que tiene, sino que se hace a nivel de byte. Por tanto, cuando se realice la lectura y se llegue al final de los datos disponibles en un momento dado, pero la descarga sigue en curso, la última secuencia leída estará incompleta la mayoría de las veces (bien porque le falta alguna línea o porque a la última línea le faltan caracteres).

Esto supone un problema cuando Spark procese el fichero, ya que se encontrará con secuencias incompletas. Para evitarlo, el thread de lectura siempre escribe secuencias completas,

```
static String readLine(BufferedReader reader) {
    String readString;

    try {
        int c = reader.read();
        while(c != -1 && (char) c != '\n') {
          readString += c;
          c = reader.read();
        }

        if (c != -1) readString += (char) c;

    } catch (Exception e) {
      throw new RuntimeException(e);
    }

    if (readString.equals("")) return null;
    else return readString;
}
```

Listado 5.3: Método readLine propio

por lo que necesitará leer varias líneas (4 líneas para el caso de FASTQ), considerar si estas forman o no una secuencia completa, y escribirlas en caso de que así sea. En caso contrario, se queda a la espera de que se descarguen nuevos datos que completen la secuencia.

Detectar si una secuencia está incompleta porque le faltan líneas es trivial, ya que al intentar realizar la lectura se obtendrá un resultado nulo. Sin embargo, detectar que a la última línea de una secuencia le faltan bytes requiere conocer el último carácter de dicha línea (se considerará completa en caso de ser un salto de línea). Para ello, se implementó una versión propia del método readLine utilizado para leer cada línea (véase el Listado 5.3), ya que el método proporcionado por la clase BufferedReader de Java elimina el carácter de salto de línea (\n).

### 5.3.2 Atomicidad

La segunda cuestión importante es la atomicidad. Este concepto hace referencia a realizar una operación en un único paso o, si está formada por varios pasos, asegurarse de que se hagan todos o ninguno, y que desde un sistema externo no se puedan apreciar pasos intermedios. Este concepto es un requisito para la API Structured Streaming a la hora de copiar ficheros en el directorio que monitoriza y leerlos de forma correcta. Si se piensa en realizar la copia de forma no atómica, esta tardará un determinado tiempo durante el que es posible observar cómo se van agregando los datos, un aspecto que se hace más evidente para conjuntos de datos de gran tamaño que tarden mucho tiempo en copiarse. Esto representa un problema para Spark ya que no tiene el conocimiento de si se está en proceso de copia o no, así que en

cuanto detecte el fichero (sus primeros bytes) lo leerá y procesará, aunque esté incompleto.

Por este motivo, se necesita garantizar que la copia desde el thread de lectura se hace de forma atómica, y para ello se utiliza la operación a nivel de fichero de move o rename. Dentro de un mismo sistema de ficheros, mover un archivo de una localización a otra no es más que renombrar la ruta del sistema en la que se encuentra, lo cual es una operación inmediata y de un único paso (atómica). Aprovechando esta idea, se crea un directorio en HDFS que sirve de almacenamiento temporal para cada fichero mientras se realiza el proceso de copia. Cuando se termine de copiar, se mueve al directorio en HDFS monitorizado por Spark, desde el cual ya será leído y procesado.

### 5.3.3 Lectura de HDFS

En este punto, la herramienta puede copiar un fichero que está en proceso de descarga dentro del sistema de ficheros local, pero no funcionaría si se estuviese descargando directamente en HDFS. El motivo es que se utiliza un objeto InputStream para ir leyendo los datos del fichero, cuyo funcionamiento en local es correcto ya que es capaz de actualizar los datos disponibles según se actualiza el contenido del fichero en disco. Pero si se desea leer desde HDFS, el InputStream obtiene el conjunto de datos disponibles en ese momento, pero no se actualiza según se escribe en HDFS.

La solución pasa por crear un nuevo objeto InputStream en cada iteración del proceso de copia, lo que asegura que se están obteniendo los datos más recientes. Hay que tener en cuenta que al crear un nuevo InputStream se reinicia la posición de lectura del fichero. Por tanto, es necesario llevar la contabilidad de los bytes copiados hasta el momento para poder saltar dicha cantidad cada vez que se reinicie la posición. Es importante el detalle de contar los bytes "copiados", es decir, los que se leyeron y se escribieron en fichero, porque esto excluye a las secuencias incompletas que se leen pero no se escriben. Así, al entrar en una nueva iteración, se inicia con la lectura de la secuencia que no se pudo completar en la iteración anterior.

Esta solución también proporciona mayor flexibilidad en el almacenamiento de los ficheros de entrada, ya que el problema de la no actualización del InputStream presente en HDFS se podría dar en otros sistemas de ficheros distribuidos similares.

### 5.3.4 Terminación de la lectura

La última consideración a tener en cuenta es determinar la terminación del proceso de lectura, que ya se mencionó previamente que se hacía mediante un timeout. El problema es que el thread de lectura no puede saber al 100% cuándo un fichero se ha terminado de descargar completamente, así que cuando se lleve un tiempo sin poder copiar nuevos datos, se considera que la descarga ha terminado. De nuevo, resaltar el matiz de "copiar" datos y no "leerlos", ya

que esto excluye a las secuencias incompletas. Si el timeout se reiniciase al leer datos, también se reiniciaría al volver a leer las líneas de la secuencia incompleta de la iteración anterior. Esto no representaría un problema a no ser que la última secuencia del fichero de entrada estuviese incompleta de forma permanente, con lo que el proceso entraría en un bucle infinito leyendo esas últimas líneas. En todo caso, esto significaría que el fichero de entrada está mal formado.

Una vez definido cómo detener el thread de lectura, falta por controlar la terminación del proceso de escritura de Spark. Como ahora el thread de lectura dictamina cuándo se acaba este proceso, se utiliza una variable booleana compartida entre threads que indica el estado del mismo. Por su parte, el código de Spark se mantiene similar, pero para terminar o no la query de escritura, además de la condición de que no haya ninguna operación en progreso, se considera también el valor de la variable booleana previamente mencionada.

### 5.3.5 Funcionalidad para juntar las partes

Finalmente, en este incremento se añadió la funcionalidad para juntar las partes generadas en la salida en un único fichero, ya que facilita la realización de pruebas y la verificación de los resultados finales.

El método básicamente consiste en ir leyendo cada parte que forma la salida de Spark y copiarla al mismo fichero resultado, aunque hay que considerar el orden correcto para leer las partes. Cuando Spark funciona en modo batch y se procesa un fichero, las partes generadas como resultado están ordenadas alfabéticamente (part-0000, part-0001...). Sin embargo, utilizando la API Structured Streaming solo aparecen ordenadas por nombre las partes generadas a partir del mismo fichero. Cuando se procesan varios, los nombres de los ficheros resultado no son indicativos del orden entre los ficheros de entrada. Una primera aproximación tomada en este incremento fue considerar los tiempos de escritura de las partes como primer criterio de ordenación y, de haber coincidencia, usar como segundo criterio el orden alfabético. En el incremento 5 (Sección 5.6) se volverá a reconsiderar esta problemática del orden para obtener una solución más precisa y robusta.

## 5.4 Incremento 3 - Procesamiento de secuencias paired-end

La siguiente funcionalidad a implementar fue la del procesamiento de secuencias paired-end, basada en operar sobre dos ficheros de entrada simultáneamente.

El primer paso fue crear un nuevo proceso de lectura en un thread capaz de copiar el contenido de los dos ficheros de entrada para su posterior procesamiento con Spark. La idea por la que se optó fue la de combinar cada secuencia del primer fichero con su correspondiente pareja en el segundo, y escribir ambas en un único fichero en HDFS que leerá Spark. La parte de la copia es similar a la versión single-end explicada previamente, con la diferencia de que

en lugar de leer las líneas en grupos de cuatro para formar una secuencia, ahora se lee un grupo de cuatro líneas para un fichero y otro grupo de cuatro para el segundo, manejando así ocho líneas (o dos secuencias) al mismo tiempo. A continuación se escriben en el fichero resultado y se obtienen las siguientes ocho líneas. En este punto surge un conflicto con la versión anterior, relacionado con la cadena que hace de separador de líneas en la lectura con Spark. Lo que interesa es cargar 8 líneas en una fila y obtener así una secuencia paired por fila, pero el separador utilizado era una cadena contenida en los nombres de todas las secuencias del fichero de pruebas utilizado. Esta aproximación no es válida al desear juntar dos secuencias (cada una con su nombre) en una misma fila.

La solución consiste en añadir una cadena específica al inicio de cada pareja de secuencias y realizar la separación de líneas con dicha cadena. De esta forma, no solo se solventa este problema, sino también el de que la cadena utilizada anteriormente solo era válida para el fichero de pruebas. La nueva solución funciona en todos los casos, por lo que se adopta también para la versión single-end, añadiendo una cadena al inicio de cada secuencia. Respecto a la cadena a utilizar, aunque podría ser cualquiera, se requiere cierto criterio para su elección ya que todos los caracteres utilizados en ella pueden aparecer tanto en el nombre de la secuencia como en la cadena de calidad. Si la cadena escogida está contenida en alguna parte de una secuencia se realizaría una división no deseada. Por este motivo se debe evitar el uso de una cadena muy corta con altas probabilidades de coincidir con el contenido de una secuencia. Tampoco es conveniente que sea muy larga ya que aumentaría en exceso el tamaño del fichero a escribir en HDFS (hay que tener en cuenta que se agrega esta cadena a cada secuencia). Un ejemplo de cadena válida sería: “@STREAM_GEN_SEQ”.

Habiendo considerado esta cuestión, desde Spark ya se estaría creando un Dataset genérico (Dataset<Row>) con las 8 líneas de cada pareja de secuencias en cada fila de la tabla. Se necesita añadir una función map que lo transforme a un Dataset de secuencias, considerando las nuevas líneas para la conversión de cada secuencia paired. Se puede ver un código ejemplificando esto en el Listado 5.4.

Para adaptar el filtro LENGTH, el filtrado en sí es sencillo de modificar ya que en el modo paired-end simplemente se aplica la función tanto a la secuencia como a su pareja (véase el Listado 5.5). Sin embargo, resulta más problemático determinar la selección entre la ejecución single o paired del filtro. En modo batch, simplemente se obtiene la primera secuencia del RDD y se observa si es paired o no; si lo es, se considera todo el conjunto de datos como paired y se procesa de ese modo. Por lo ya comentado en la Sección 5.2.3, este tipo de operación no está permitido en un modelo streaming, pues no se puede comprobar si en un determinado momento el Dataset tiene una primera fila. La solución implementada para solventar este problema consiste en crear y pasar como parámetro una variable booleana que indica si las secuencias son paired o no, lo cual se conoce desde el inicio de la ejecución cuando se indican

```
Dataset<Row> lines = sparkSession
    .readStream()
    .option("lineSep", reader.SEQUENCE_PAIR_NAME_PREFIX)
    .text(inputDir);

Dataset<Sequence> seqs = lines
    .filter("value != ''")
    .map(tuple -> {
        String[] sequence = tuple.toString().split("\\n");

        Sequence s = new Sequence(sequence[0].substring(1),
    sequence[2], sequence[4], sequence[6]);
        s.setPairSequence(sequence[1], sequence[3], sequence[5],
    sequence[7]);
        return s;

        }, Encoders.bean(Sequence.class));
```

Listado 5.4: Fragmento de código de lectura de secuencias paired-end en formato FASTQ

```
if (isPaired) {
  return sequences.filter(s -> this.filter(s, limMin, limMinUse,
    limMax, limMaxUse)
          && this.filterPair(s, limMin, limMinUse, limMax,
    limMaxUse));
}
// single end
return sequences.filter(s -> this.filter(s, limMin, limMinUse,
    limMax, limMaxUse));
```

Listado 5.5: Fragmento de código de filtro LENGTH adaptado para single- y paired-end

dos ficheros como entrada.

Por último, para la parte de la escritura se divide el Dataset de secuencias en otros dos Datasets a través de dos funciones map, uno con las secuencias correspondientes al primer fichero y otro con las del segundo, creando una petición de escritura para cada uno en directorios diferentes. Un ejemplo del código del proceso descrito se muestra en el Listado 5.6.

## 5.5 Incremento 4 - Implementación del resto de operaciones

Este incremento se dedicó a la implementación del resto de operaciones pendientes, incluyendo el resto de filtros individuales, los formateadores, los recortadores y la lectura y procesamiento de secuencias en formato FASTA.

Respecto a la adaptación de los filtros, se actuó de forma similar a la adaptación del filtro LENGTH, requiriendo el mismo ajuste para usar Datasets en vez de RDDs. Para los formateadores y recortadores la idea es similar, pero como en estas operaciones se utiliza la función

```
firstSequences = sequences.map(sequence ->
    SequenceUtils.getFirstSequenceFromPair(sequence),
    Encoders.bean(Sequence.class));

secondSequences = sequences.map(sequence ->
    SequenceUtils.getSecondSequenceFromPair(sequence),
    Encoders.bean(Sequence.class));

Dataset<String> seqs1 = firstSequences.map(s -> {
    return s.toString();
},Encoders.STRING());

Dataset<String> seqs2 = secondSequences.map(s -> {
    return s.toString();
},Encoders.STRING());

StreamingQuery query1 = seqs1.writeStream()
    .outputMode("append")
    .format("text")
     .option("path", output + OUTPUT_FOLDER_NAME_FIRST_PAIR)
    .option("checkpointLocation", checkpointDir1)
    .start();

StreamingQuery query2 = seqs2.writeStream()
    .outputMode("append")
    .format("text")
    .option("path", output + OUTPUT_FOLDER_NAME_SECOND_PAIR)
    .option("checkpointLocation", checkpointDir2)
  .start();

```

Listado 5.6: Fragmento de código de escritura de secuencias paired-end

map en lugar de filter, se requiere añadir un Encoder de la clase representante de una secuencia para tipar correctamente el Dataset resultado, ya que podría ser diferente al de entrada (aunque no es el caso). Otra diferencia respecto a la versión batch es la gestión de las operaciones específicas de FASTQ, que actúan en función de la calidad de las secuencias como el filtro de QUALITY (ver Sección 4.1)„ que filtra en función de su calidad media. En SeQual, se obtiene la primera fila del RDD y se comprueba si dispone de cadena de calidad para decidir si se debe realizar una operación de este tipo. Como ya se comentó que esto en streaming no es posible, se añade una variable con el mismo objetivo cuyo valor se obtiene en función del formato del fichero de entrada.

Otro detalle introducido con la implementación de los formateadores es la posibilidad de cambiar de formato FASTQ a FASTA, lo cual genera un problema en la funcionalidad de juntar las partes en un único fichero resultado. Este fichero necesita tener la extensión correcta (.fastq/.fq o .fasta/.fa), pero el formato de la entrada ya no es siempre un indicativo del formato de la salida. Para solventar este problema, además del formato del fichero de entrada, se

```
List<String> basesLines = new ArrayList<>();

line1 = ReaderUtils.readLine(reader);
lineAux = ReaderUtils.readLine(reader);

while (lineAux != null && !isLineName(lineAux)) {
    basesLines.add(lineAux);
    lineAux = ReaderUtils.readLine(reader);
}

//La función isLineName es la siguiente:

private boolean isLineName(String line) {
    return line.startsWith(">");
}
```

Listado 5.7: Fragmento de código del thread de lectura de secuencias single-end en formato FASTA

considera una variable booleana en función de si se ejecutó este formateador, y así conocer la extensión que debe tener la salida.

Por otro lado, el soporte para secuencias en formato FASTA presenta la problemática de que sus bases pueden estar almacenadas en un número indeterminado de líneas, a diferencia de FASTQ que siempre tiene una única línea para las bases (y cuatro en total para cada secuencia). Para ello, se requiere un nuevo proceso de lectura en un thread capaz de leer una primera línea (el nombre de la secuencia) y $N$ líneas correspondientes a las bases. Esta última parte se gestiona introduciendo cada línea de bases en una lista hasta que no haya más bases o se encuentre el nombre de la siguiente secuencia (que debe empezar siempre con el carácter '>'). En el Listado 5.7 se muestra un fragmento de código ejemplificando este concepto. Respecto al procesamiento paired-end, la idea es la misma pero aplicada a dos ficheros simultáneamente. En este proceso de copia de ficheros FASTA, se vuelve a aplicar la agregación de una cadena a modo de prefijo o separador de cada secuencia para la posterior división en líneas con Spark, donde se necesitan dos nuevas clases capaces de interpretar las nuevas líneas en single y paired, como se aprecia en los Listados 5.8 y 5.9. Por último, no se requieren nuevos cambios en los filtros, recortadores, formateadores ni en la escritura en disco para funcionar con secuencias en formato FASTA.

## 5.6 Incremento 5 - Mejoras y optimizaciones

En esta sección se describe un incremento que analiza el estado actual de la herramienta para detectar posibles mejoras y añadir optimizaciones. Los principales cambios realizados se detallan en los siguientes apartados.

```
Dataset<Sequence> seqs = lines
    .filter("value != ''")
    .map( tuple -> {
        String[] sequence = tuple.toString().split("\\n");
        sequence = (String[]) ArrayUtils.remove(sequence,
    sequence.length-1);

        String basesLines = "";
        for (int i = 1; i < sequence.length; i++) {
            basesLines += sequence[i];
            if (i < sequence.length - 1) basesLines += "\n";
        }
        return new Sequence(sequence[0].substring(1), basesLines);

}, Encoders.bean(Sequence.class));
```

Listado 5.8: Fragmento de código de lectura de secuencias single-end en formato FASTA

```
Dataset<Sequence> seqs = lines
    .filter("value != ''")
    .map( tuple -> {
        String[] sequence = tuple.toString().split("\\n");
        sequence = (String[]) ArrayUtils.remove(sequence,
    sequence.length-1);

        String basesLines = "";
        int i = 1;
        while (i < sequence.length && !sequence[i].startsWith(">"))
    {
            basesLines += sequence[i];
            if (!sequence[i+1].startsWith(">")) basesLines += "\n";
            i = i + 1;
        }

        // now i is in position of sequence pair name
        String pairBasesLines = "";
        int j = i+1;

        while (j < sequence.length) {
            pairBasesLines += sequence[j];
            if (j < sequence.length - 1) pairBasesLines += "\n";
            j = j + 1;
        }

        Sequence s = new Sequence(sequence[0].substring(1),
    basesLines);
        s.setPairSequence(sequence[i], pairBasesLines);
        return s;

}, Encoders.bean(Sequence.class));
```

Listado 5.9: Fragmento de código de lectura de secuencias paired-end en formato FASTA

### 5.6.1 Secuencias con timestamp

Como se mencionó en la Sección 5.3.5, la capacidad de obtener las partes resultado ordenadas alfabéticamente no es un proceso directo. Cuando se usa procesamiento batch, los ficheros generados por Spark como salida se nombran en orden utilizando un código numérico indicativo del número de partición del RDD o DataFrame/Dataset (part-0000, part-0001...). En streaming también aparecen ordenadas las partes del mismo fichero, pero no se asegura el orden cuando se procesan varios. Por ejemplo, si un fichero de entrada se copia en dos ficheros en HDFS, el primero de ellos podría dividirse en dos particiones procesadas por dos tareas en Spark que generarían dos ficheros resultado (por ejemplo, part-0000-87bd23db y part-0001-f2843e01), mientras que del segundo fichero se podrían generar otros dos como resultado (part-0000-2e4f18b9 y part-0001-57fb935b). Al finalizar, no es posible interpretar el orden entre los cuatro ficheros resultado simplemente a partir de sus nombres.

La primera aproximación mencionada en la Sección 5.3.5 se basa en utilizar el tiempo de escritura de cada fichero resultado como criterio de ordenación, creando un proceso que itere desde el fichero más antiguo hasta el más reciente, renombrando cada parte para garantizar así un orden alfabético. De haber coincidencia en el tiempo, el siguiente criterio de ordenación a seguir sería el alfabético. En el ejemplo anterior, resultaría en 4 partes llamadas part-0000, part-0001, part-0002 y part-0003. La funcionalidad de juntar las partes en un único fichero resultado simplemente necesitaría leerlas en orden alfabético. Sin embargo, esta aproximación no es perfecta ya que realmente no hay garantías de que Spark procese y escriba los resultados de forma ordenada al ser alimentado a la entrada con diferentes ficheros. Es decir, se podría dar el caso de que se escriba un fichero, un poco más tarde se escriba uno nuevo y justo en ese momento Spark detecte ambos, no garantizando que se procesen en orden de llegada. Además, es frecuente que en el reparto de las tareas de procesamiento de un fichero justamente la última tarea tenga menor coste computacional porque se haya generado a partir de los últimos bytes del fichero, procesándose y escribiéndose en la salida más rápido que las anteriores. Lo más conveniente en ese caso sería que todas las partes se escriban aproximadamente al mismo tiempo para aplicar el segundo criterio de ordenación (el alfabético).

La solución definitiva consiste en embeber un timestamp en las secuencias que sea representativo del orden entre los diferentes ficheros de entrada a procesar. Se habla de timestamp, pero no es necesario utilizar un tiempo absoluto, se puede simplemente aplicar un número para establecer un orden relativo entre las secuencias. Como el proceso que conoce perfectamente el orden de los ficheros copiados es el thread de lectura, este se encargará de añadir un entero representativo de la iteración o fichero a cada secuencia leída, mientras que el proceso de Spark debe leerlo e interpretarlo. Para esta última parte se creó una nueva clase que encapsula cada secuencia con su timestamp (SequenceWithTimestamp), así que ahora se manejan Datasets tipados con esta nueva clase, lo que implica un ligero reajuste en las operaciones de

```
if (isPaired) {
    return sequences.map(sequence -> this.doTrimPair(sequence,
    limit), Encoders.bean(SequenceWithTimestamp.class));
}

return sequences.map(sequence -> this.doTrim(sequence,
    limit),Encoders.bean(SequenceWithTimestamp.class));

// la función doTrim es la siguiente y doTrimPair es similar a ella:
private SequenceWithTimestamp doTrim(SequenceWithTimestamp
    sequenceWithTimestamp, Integer limit) {

    Sequence sequence = sequenceWithTimestamp.getSequence();
    int length = sequence.getLength();
    if (length > limit) {
        sequence.setSequenceString(
            sequence.getSequenceString().substring(limit)
        );
        if (sequence.getHasQuality()) {
            sequence.setQualityString(
                sequence.getQualityString().substring(limit)
            );
        }
    }
    return sequenceWithTimestamp;
}
```

Listado 5.10: Fragmento de código del recortador TRIMLEFT adaptado a la clase SequenceWithTimestamp

filtrado, recortado y formateado, pues deben operar con la secuencia almacenada dentro de esta nueva clase (véase el Listado 5.10 como ejemplo para el recortador TRIMLEFT).

Por otro lado, es necesario que la fase de escritura tenga en cuenta el timestamp para establecer un orden, pero no se desea que el timestamp se vea reflejado en el contenido del fichero final. Existe una función que se adapta expresamente a este propósito: partitionBy. Esta transformación particiona el conjunto de datos en diferentes particiones o grupos según el criterio marcado por una columna, y posteriormente se elimina dicha columna del resultado. Cuando se procede a escribir el resultado en disco, se generará una carpeta por cada partición en la ruta de destino donde se almacenarán las partes correspondientes a esa partición. Para ver un ejemplo más concreto, si se dispone de un conjunto de datos referente a diferentes ciudades de Estados Unidos y en cada fila se indica el estado al que pertenece cada ciudad, se podría particionar por estado para agrupar así todas las ciudades del mismo estado. En la fase de escritura se crearía una carpeta por cada estado como se observa en la Figura 5.1.

Aplicado al caso de uso de esta aplicación, dado un Dataset con secuencias con timestamp se requiere una separación de sus datos en una columna con las secuencias convertidas en cadenas de texto, listas para ser escritas, y otra columna con los timestamps para realizar

```
$ ls -lrt zipcodes-state
total 24
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AZ'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=FL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=NC'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=PR'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=TX'/
-rw-r--r-- 1 prabha 197121 0 Mar  4 21:18  _SUCCESS
```

Figura 5.1: Ejemplo de particionado

posteriormente un particionado siguiendo esta columna. Esto resulta en la agrupación de las secuencias generadas a partir del mismo fichero (mismo timestamp) dentro de la misma partición. En el Listado 5.11 se muestra el código referente al particionado. Este proceso acaba con varias subcarpetas dentro de la carpeta de destino, del estilo timestamp=0, timestamp=1, ..., timestamp=n, cada una con sus partes ordenadas alfabéticamente, asegurando así un orden para todos los ficheros resultado.

Por último, se modificó el método que renombraba todas las partes para seguir el nuevo criterio de ordenación, consiguiendo que todas ellas se encuentren en un mismo directorio en lugar de estar repartidas en varios subdirectorios. Este último aspecto se discute de nuevo en la Sección 5.7, ya que no es un proceso estrictamente necesario para tener un orden y se comprobó que no resulta muy eficiente en términos de rendimiento. La funcionalidad de juntar las partes en un único fichero resultado se mantiene sin cambios, dado que simplemente debe seguir leyendo las partes en orden alfabético.

### 5.6.2 Limitación de datos copiados en cada iteración

Hasta este punto, la forma de realizar la operación de copia por parte del thread de lectura consiste en leer todos los datos disponibles en un momento dado y copiarlos en el mismo fichero. Para ficheros de entrada de pequeño tamaño esta aproximación no supone ningún inconveniente, pero cuando se empiezan a considerar ficheros con tamaños significativos surge el problema de que no se aprovecha de forma adecuada la capacidad de cómputo de forma paralela al proceso de copia. Supongamos que tenemos un fichero de gran tamaño ya disponible de forma completa en el sistema de ficheros local. En ese caso, SeQual-Stream lo copiaría entero a HDFS y Spark solo podría empezar a procesarlo una vez que acabe la operación de copia (y al ser un fichero de gran tamaño la copia tardaría un tiempo considerable). Este comportamiento no sería muy diferente a subir directamente el fichero de entrada a HDFS y procesarlo con SeQual en modo batch. Para ficheros en proceso de descarga en el sistema de

```
Dataset<Row> seqs = sequences.flatMap(
    getOutputFlatMapFunction(),
    RowEncoder.apply(getOutputStructType()));

StreamingQuery query = seqs.writeStream()
    .outputMode("append")
    .format("text")
    .option("path", output)
    .option("checkpointLocation", checkpointDir)
    .partitionBy("timestamp")
    .start();

// siendo las funciones getOutputStructType y
    getOutputFlatMapFunction las siguientes:

private static StructType getOutputStructType () {
    StructType structType = new StructType();
    structType = structType.add("timestamp",DataTypes.StringType,
    false);
    structType = structType.add("sequence",DataTypes.StringType,
    false);
    return structType;
}

private static FlatMapFunction<SequenceWithTimestamp, Row>
    getOutputFlatMapFunction () {
    return new FlatMapFunction<SequenceWithTimestamp, Row>() {

        private static final long serialVersionUID = 1L;

        @Override
        public Iterator<Row> call (SequenceWithTimestamp seq)
    throws Exception {
            String timestamp = seq.getTimestamp();
            String sequence = seq.getSequence().toString();

            List<String> data = new ArrayList<>();
            data.add(timestamp);
            data.add(sequence);
            List<Row> list = new ArrayList<>();
            list.add(RowFactory.create(data.toArray()));

            return list.iterator();
        }
    };
}
```

Listado 5.11: Fragmento de código del proceso de particionado

ficheros local podría ocurrir exactamente lo mismo si la velocidad de descarga es superior a la de copia.

Para evitar esto, la solución adoptada consiste en establecer un límite en el número de

bytes que se pueden copiar en una misma iteración (o fichero) del thread de lectura, aprovechando así para los casos mencionados anteriormente el solapamiento entre el procesamiento de SeQual-Stream y la carga de entrada/salida en disco, que se realizarían en paralelo. Este límite no es estricto, se pueden sobrepasar los bytes que sean necesarios para no dejar incompleta una secuencia al final. En este punto del proyecto, el límite se estableció a un valor de forma arbitraria, pero en un incremento posterior se refinó para que se calcule en función de las características del clúster donde se ejecute. Esta idea se detalla en la Sección 5.7.2.

### 5.6.3 Adaptación de la GUI

Finalmente, en este incremento también se realizó la adaptación de la interfaz gráfica integrada en SeQual (módulo SeQual-GUI) para admitir el uso de SeQual-Stream.

Por un lado, se añadió un nuevo botón en la GUI para seleccionar el uso del modo streaming (véase la Figura 5.2). Por otro lado, en el código que controla la lógica de la interfaz gráfica se añadió la gestión del valor del botón mencionado y, de forma similar a la interfaz de consola de comandos, el controlador pasa ahora a depender de una clase interfaz implementada de forma concreta por la versión batch y por la versión streaming, utilizando una u otra en función del valor del botón.

## 5.7 Incremento 6 - Resolución de cuellos de botella

Una vez obtenida una versión completa y funcional de la herramienta, se evaluó su rendimiento en un entorno clúster utilizando conjuntos de datos con un tamaño significativo, del orden de decenas de GB. Durante estas primeras pruebas se detectó que la velocidad de ejecución de la herramienta era demasiado lenta, así que se llevó a cabo este incremento para detectar y depurar los cuellos de botella, estando la mayoría de ellos relacionados con el proceso de copia inicial desde el thread de lectura.

### 5.7.1 Optimización de las funciones usadas en el proceso de copia en HDFS

El primer paso fue la optimización o reemplazo de ciertos métodos que ofrecían un bajo rendimiento, de los cuales podemos destacar los siguientes:

- El método skip del BufferedReader utilizado para "saltar" los bytes ya copiados en iteraciones previas es una función poco eficiente, ya que lee cada carácter antes de llegar a la posición deseada, pudiendo tardar del orden de varios minutos en hacerlo. Por ello, se reemplazó por el método seek del InputStream, que realiza un salto "inmediato".
- La implementación propia del método readLine creado anteriormente (véase el Listado 5.3) resultó ser poco eficiente debido a la forma de concatenar la cadena a devolver con

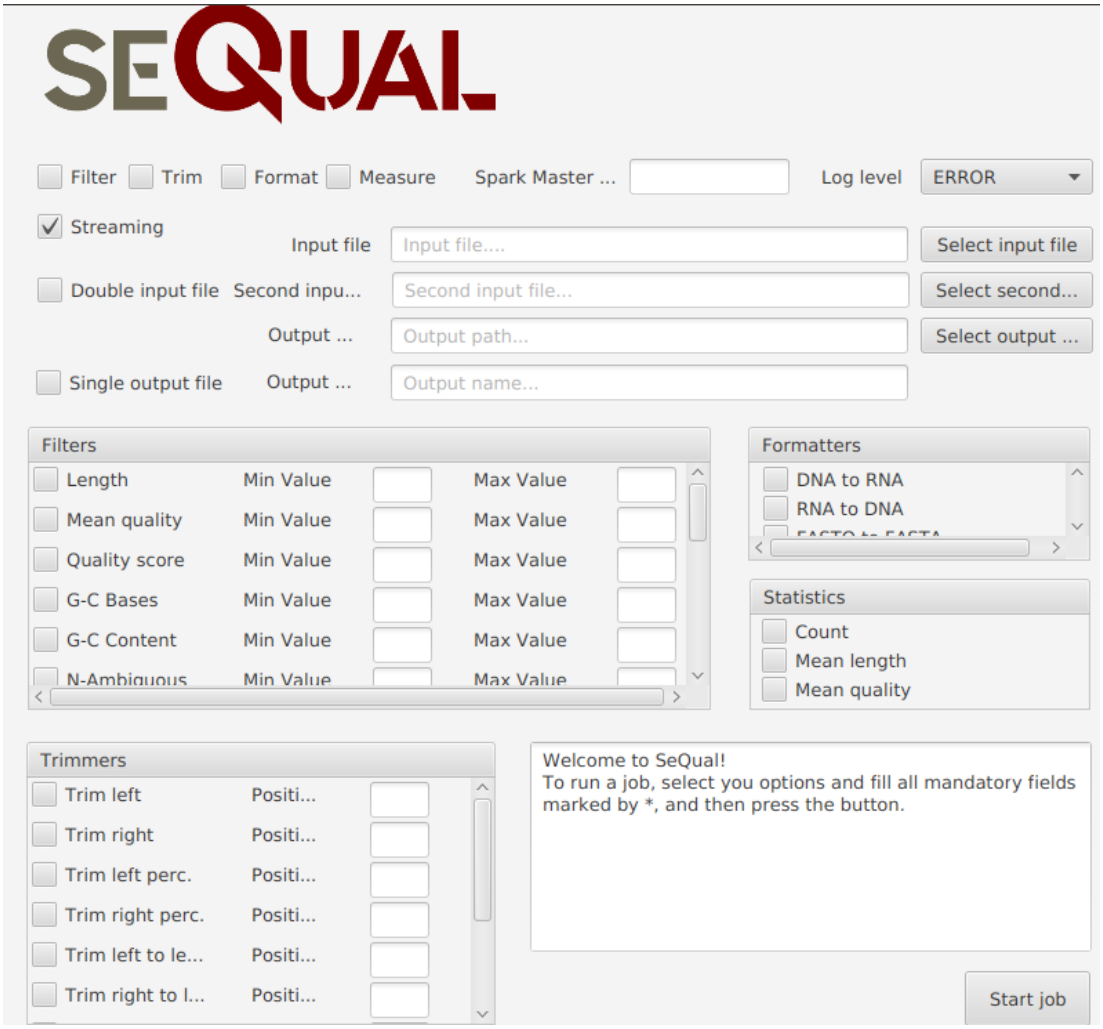


Figura 5.2: Interfaz gráfica adaptada para usar SeQual-Stream

cada nuevo carácter. Esta concatenación se hacía de manera directa (string1 = string1 + string2). Dado que los objetos String de Java son inmutables, internamente en lugar de concatenar se está creando un nuevo objeto String con el nuevo contenido, siendo muy ineficiente a la hora de hacer muchas concatenaciones seguidas. Por ello, se cambió el código para realizar la concatenación con un objeto StringBuilder, el cual sí es mutable y mucho más eficiente. En el Listado 5.12 se muestra el método modificado.

Tras realizar los cambios descritos en este apartado, la ejecución sufrió una aceleración considerable, pero aún no era lo suficientemente rápida. Además, presentaba un problema de escalabilidad cuando se añadían más nodos para el procesamiento, ya que el principal cuello de botella seguía siendo la operación de copia inicial, así que se optó por la paralelización de este proceso.

```
static String readLine(BufferedInputStream reader) {
    String readString;
    StringBuilder readStr = new StringBuilder();

    try {
        int c = reader.read();
        while(c != -1 && (char) c != '\n') {
            readStr.append((char) c);
            c = reader.read();
        }

        if (c != -1) readStr.append((char) c);

      } catch (Exception e) {
        throw new RuntimeException(e);
      }

    readString = readStr.toString();
    if (readString.equals("")) return null;
    else return readString;
}
```

Listado 5.12: Método readLine propio optimizado

### 5.7.2 Paralelización del proceso de copia en HDFS

El siguiente paso consistió en la paralelización del proceso de copia en HDFS mediante múltiples threads. La idea general es conseguir que cada thread lea un fragmento del fichero de entrada y genere un fichero con esos datos en HDFS para que sea procesado por Spark.

En primer lugar se implementó de forma que cada thread lee una cantidad de datos igual al límite de bytes establecido anteriormente. Es decir, siendo $X$ dicho límite, el primer thread copiará los primeros $X$ bytes, mientras que el siguiente copia los $X$ siguientes, y así sucesivamente. En este punto se observó que establecer un límite de forma arbitraria no funcionaría bien en todos los escenarios, ya que un límite bajo podría funcionar bien en un determinado clúster, pero no aprovechar la capacidad de un clúster más grande. Esto se debe a la forma en que Spark genera las tareas de procesamiento en cada nodo. Cada fichero en HDFS está dividido en bloques del mismo tamaño, y al realizar un procesamiento sobre él se creará una partición por cada bloque (comportamiento por defecto) y se asignará una tarea de procesamiento de cada partición. Cada tarea será procesada por un núcleo de cómputo disponible en el clúster Spark. Por ejemplo, si se tiene un fichero de 1 GB y un tamaño de bloque en HDFS de 128 MB, el fichero se dividirá en 8 bloques, 8 particiones y 8 tareas, de forma que se podría explotar el rendimiento de como mucho 8 núcleos de cómputo. Teniendo en cuenta esta última idea, se puede calcular de forma dinámica en cada ejecución la cantidad total mínima de bytes a copiar en cada thread para aprovechar al máximo los recursos disponibles del clúster, multiplicando el tamaño de bloque de HDFS por la cantidad de núcleos disponibles. En el

ejemplo anterior, con un tamaño de bloque de 128 MB y 8 núcleos en el clúster, la cantidad mínima a leer sería 1 GB, y siempre se debería intentar llegar a ese valor (aunque habrá casos en los que no se pueda porque no hay tantos datos disponibles en el fichero de entrada).

Optimizado ese aspecto, quedaría pendiente el reparto entre threads. La idea es repartir la funcionalidad de copia del thread de lectura principal entre uno o varios threads de lectura auxiliares (workers), mientras que el thread principal se encargará únicamente de la gestión de los threads auxiliares (véase el Listado 5.13). Respecto a la cantidad de threads a utilizar, se debe tener en consideración los recursos disponibles en la máquina que ejecuta el código principal de la aplicación (el driver), que será la encargada del proceso de copia, teniendo en cuenta que ya se están utilizando al menos dos threads (la aplicación principal y el thread de lectura principal). Por ejemplo, si se dispone de 32 núcleos en esa máquina, se usarán como máximo 30 threads auxiliares. Solo resta dividir la cantidad mínima de bytes a copiar entre el número máximo de threads para obtener el límite de bytes para cada uno. Para evitar crear un thread que lea muy pocos bytes se considera como límite inferior el tamaño de bloque de HDFS. Si la división da un valor menor, se recalcula el número máximo de threads en función del límite inferior y la cantidad total mínima a copiar.

Un detalle a considerar es que los threads auxiliares no pueden compartir el mismo objeto InputStream, ya que se desea que cada uno lea desde posiciones diferentes del fichero de entrada. Cada thread podría crear su propio InputStream y saltar los bytes que necesite, pero una forma más óptima es hacer que el thread de lectura principal se encargue de esta gestión para cada uno de ellos, ya que si se detecta que el salto falla es debido a que no hay más datos disponibles. De esta forma puede evitarse la creación de threads innecesarios.

Otra consideración importante es reiterar la idea de que el límite de bytes no es un límite estricto, se podrá sobrepasar para evitar dejar incompleta la última secuencia copiada. Ligado a esto, también hay que tener en cuenta que cuando un thread empieza su lectura es improbable que le coincida exactamente en el inicio de una nueva secuencia, sino en medio de ella. Es decir, una secuencia que está posicionada en medio de la lectura de dos threads diferentes. El problema se evita simplemente saltando esa primera secuencia incompleta, dado que el thread anterior deberá encargarse de copiarla. Detectar que no se está leyendo el inicio de una secuencia es algo directo de hacer en el formato FASTA, ya que solo los nombres contienen (y empiezan) por el carácter '>'. Sin embargo, en FASTQ se requiere leer las líneas de dos en dos para poder tener más contexto y determinar si alguna de ellas corresponde a un nombre de una secuencia. Por ejemplo, si las dos líneas empiezan por '@', significa que la primera es una cadena de calidad y la siguiente es el nombre de la secuencia siguiente. Tras realizar el proceso de copia, cada thread auxiliar deberá reportar al thread principal la cantidad real de bytes que ha copiado. De esta forma, en la siguiente iteración se conoce la cantidad de bytes iniciales que deben saltarse, y el primer thread (el único que no tiene un thread anterior) empezará

```
List<FQReaderWorker> workerList = new ArrayList<>();

for (int i = 0; i < num_readers; i++) {

    FQReaderWorker worker = new FQReaderWorker();
    FSDataInputStream fsInputThread = fsInput.open(new
    Path(inputFile));

    try {
        fsInputThread.seek(skipChars + partition_skip*i);

        worker.startRead(iteration, i, partition_skip,
        new BufferedInputStream(fsInputThread), fsOutput,
    this.tmpDir, this.outputDir, SEQUENCE_NAME_PREFIX);

        workerList.add(worker);

    } catch (IOException e) {
        // seek failed
      break;
    }
}

// wait for all threads
for (FQReaderWorker worker: workerList) {
    worker.join();
}
```

Listado 5.13: Fragmento de código del thread de lectura principal para FASTQ single-end respecto a la creación y finalización de threads de lectura auxiliares

en el inicio de una nueva secuencia. También se aprovecha para saber si se ha copiado algún byte o no, y detener la lectura en caso de considerar que la descarga del fichero ha finalizado o realizar una nueva iteración.

Finalmente, respecto al timestamp utilizado para cada fichero, se sigue la misma idea previamente explicada en la Sección 5.6.1, solo que en vez de indicar simplemente el número de iteración se añade también el número de thread. Por ejemplo, en la iteración 3, el thread 5 indicará su timestamp como: $3 - 5$.

### 5.7.3 Factor de replicación de HDFS

Un tema importante a considerar es la limitación en el rendimiento de entrada/salida que impone el propio disco. Esto ya era un factor notorio en SeQual debido a que, al existir múltiples procesos subiendo ficheros a HDFS, los bloques que se deban escribir a la vez en el mismo disco se verán limitados por la velocidad máxima que proporcione el disco, que se convierte en el cuello de botella. Sin embargo, en SeQual-Stream este factor se agrava todavía más al considerar la copia a HDFS de los distintos fragmentos del fichero de entrada, generando un

solapamiento con la subida de partes y ralentizando ambos procesos.

Aunque no es un problema que se pueda evitar por completo, sí es posible mitigarlo hasta cierto punto reduciendo el factor de replicación de HDFS, el cual es especialmente problemático cuando su valor es mayor, igual o cercano al número de DataNodes del clúster. Por ejemplo, considerando un factor de replicación de 3 y 3 DataNodes, cada fichero que se escriba conlleva la escritura de varios bloques en todos los DataNodes, congestionando en gran medida la entrada/salida en disco. Si se disponen de más nodos y el mismo factor de replicación, la carga se balancearía mejor y se aliviaría la congestión en cada nodo, aunque evidentemente un factor igual a uno es el que siempre obtiene el mejor rendimiento.

Teniendo en cuenta todo lo anterior, para obtener un buen rendimiento en general se redujo el factor de replicación a 1 únicamente en los ficheros copiados a HDFS, mejorando considerablemente el rendimiento cuando se utilizan pocos nodos, mientras que no se modifica el factor de replicación por defecto del clúster. Esto podría generar la pérdida de algún bloque de los ficheros generados en caso de que se caiga un nodo del clúster, pero no es tan problemático como si fuese un proceso batch ya que la copia del fichero de entrada es un proceso continuo y, si se cae un DataNode, no se le vuelven a enviar bloques para escribir.

### 5.7.4 Renombrado de partes

Como se comentó en la Sección 5.6.1, tras la escritura de los ficheros resultado particionados en subcapertas según un timestamp, había una fase posterior para renombrarlas siguiendo el orden y juntarlas bajo una misma carpeta.

En este punto se reconsideró esta fase y se eliminó para mejorar los tiempos de ejecución. En realidad no es necesaria porque las partes ya están ordenadas de forma alfabética en cada subcarpeta, y las subcarpetas están ordenadas entre sí según el timestamp. No solo empeoraba el rendimiento lo suficiente como para considerar su descarte, especialmente notorio en procesamiento paired al tener muchas más partes, sino que también empeoraba más a medida que se aumentaba el número de nodos del clúster. Esto es debido a que en ese caso se dividen más las tareas, creándose por tanto más partes y reduciendo el rendimiento global.

Habiendo quitado esta fase, la funcionalidad de juntar las partes en un único fichero se modificó para tener en cuenta la ordenación mediante subcarpetas.

Capítulo 6

# Evaluación del rendimiento

En este capítulo se exponen los resultados de la evaluación experimental realizada en un entorno clúster, con el objetivo de analizar el rendimiento de SeQual-Stream y realizar una comparación de la mejora obtenida respecto a SeQual.

## 6.1 Entorno de pruebas

El entorno de pruebas utilizado para la evaluación experimental fue el clúster de altas prestaciones Plutón [32]. Su configuración hardware se divide en un único nodo frontal o cabecera y los nodos de cómputo. El nodo cabecera sirve como único punto de acceso desde el exterior para que los usuarios puedan loguearse e interactuar con el clúster (véase la Figura 6.1). En este nodo los usuarios pueden editar/compilar sus códigos y enviar trabajos al planificador o sistema de colas para su posterior ejecución en los nodos de cómputo. También hace la función de servidor Network Attached Storage (NAS) en el clúster, donde se almacenan de forma persistente los ficheros de los usuarios, que luego pueden ser accedidos desde todos los nodos de cómputo a través de NFS.

Los nodos de cómputo son los encargados de proporcionar los recursos computacionales del clúster (como CPU, memoria y aceleradoras) y se encuentran agrupados de forma lógica en cabinas de cómputo, donde cada una dispone de un número variable de nodos con características hardware muy similares. Actualmente existen 3 cabinas, teniendo cada una de ellas 17, 2 y 6 nodos de cómputo, respectivamente.

Para la evaluación del rendimiento se escogió la primera cabina (cabina 0) por su mayor número de nodos (17), y así poder analizar la escalabilidad de la herramienta al aumentar en mayor medida la cantidad de nodos utilizados en los experimentos. Las características hardware específicas de cada nodo de la cabina 0 se indican en la Tabla 6.1.

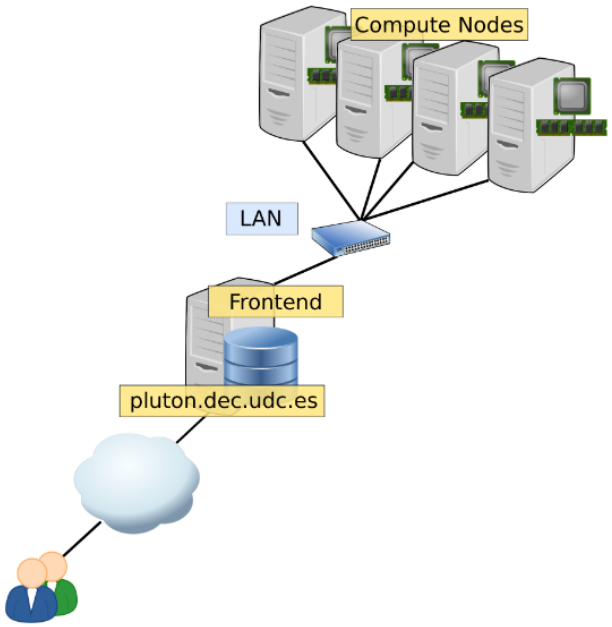


Figura 6.1: Estructura del clúster Plutón

| | 17 nodos (compute-0-{0-16}) |
|---|---|
| **CPU: Modelo** | 2 × Intel Xeon E5-2660 Sandy Bridge-EP |
| **CPU: Velocidad/Turbo** | 2.20 GHz/3.0 GHz |
| **#Núcleos por CPU** | 8 |
| **#Threads por núcleo** | 2 |
| **#Núcleos/Threads por nodo** | 16/32 |
| **Caché L1/L2/L3** | 32 KB/256 KB/20 MB |
| **Memoria RAM** | 64 GB DDR3 1600 Mhz |
| **Discos** | 1 × HDD 1 TB SATA3 7.2K rpm |
| **Redes** | InfiniBand FDR & Gigabit Ethernet |

Tabla 6.1: Descripción de los nodos de cómputo de la cabina 0 del clúster Plutón

## 6.2 Pruebas a realizar

Las pruebas de rendimiento se realizaron sobre conjuntos de datos de tamaños y características diferentes usando funcionalidades representativas de cada grupo de operaciones. Las operaciones escogidas fueron las siguientes (ver el Capítulo 4 para más detalle):

- Filtros individuales
  - QUALITY: Filtrar por calidad media mínima.
  - NONIUPAC: Filtrar si contiene caracteres Non-IUPAC.

- Formateadores
  - DNATORNA: Transformar las secuencias ADN a secuencias ARN.
- Recortadores
  - TRIMRIGHTP: Recortar las secuencias un porcentaje a partir del extremo 3'.

Los conjuntos de datos utilizados son ficheros FASTQ paired-end obtenidos del Sequence Read Archive (SRA) [33], un repositorio público de datos genómicos que pertenece al National Center for Biotechnology Information (NCBI) [34]. Los ficheros contienen secuencias de ADN reales obtenidas de células del ser humano, y sus características en cuanto a tamaño son las siguientes:

- **SRR567455_1.fastq** y **SRR567455_2.fastq**: cada fichero ocupa unos 45 GB y contiene unos 251 millones de secuencias con 76 bases cada una.
- **SRR11442499_1.fastq** y **SRR11442499_2.fastq**: cada fichero ocupa unos 62 GB y contiene unos 251 millones de secuencias con 99 bases cada una.

Estos ficheros se utilizaron para realizar cuatro mediciones diferentes sobre cada una de las operaciones evaluadas, de la siguiente forma:

1. **Medición 1**: Medición con entrada single-end sobre el fichero SRR567455_1.fastq.
2. **Medición 2**: Medición con entrada paired-end sobre los ficheros SRR567455_1.fastq y SRR567455_2.fastq.
3. **Medición 3**: Medición con entrada single-end sobre el fichero SRR11442499_1.fastq.
4. **Medición 4**: Medición con entrada paired-end sobre los ficheros SRR11442499_1.fastq y SRR11442499_2.fastq.

Las pruebas se realizaron desplegando un clúster Spark con HDFS usando la herramienta BDEv [30], como se mencionó en la Sección 3.1. Para HDFS se ha utilizado un factor de replicación de 3 y un tamaño de bloque de 128 MB. Los ficheros de entrada se encontraban almacenados de forma completa en el sistema de ficheros local del servidor NAS del clúster. En cada experimento SeQual-Stream realiza la lectura de los ficheros de entrada directamente desde la NAS a través de NFS y los copia en HDFS. En SeQual es necesario copiar previamente la entrada en HDFS, debiendo sumar el tiempo requerido para ello en la evaluación comparativa. Para que la comparación sea lo más justa posible, dado que SeQual-Stream copia los ficheros en HDFS con un factor de replicación de 1, para SeQual también se hizo la copia con el mismo valor.

## 6.3 Comparación de rendimiento

A continuación se muestran los resultados de la evaluación del rendimiento para ambas versiones utilizando las operaciones y mediciones mencionadas anteriormente. Cada gráfica mostrada representa la ejecución de una operación sobre una medición, ejecutada por SeQual y por SeQual-Stream, para diferente número de nodos. En el eje X se muestra la cantidad de nodos workers de Spark usados para cada resultado o, dicho de otra forma, la cantidad de nodos de cómputo utilizados en el clúster excluyendo al nodo maestro, ya que no aporta capacidad computacional durante el procesamiento. En el eje Y se muestra el tiempo de ejecución en segundos para cada resultado. También cabe mencionar que en todas las ejecuciones se utilizaron los 16 núcleos proporcionados por cada nodo de cómputo.

### 6.3.1 Filtro QUALITY

Los resultados utilizando el filtro QUALITY se pueden observar en las Figuras 6.2, 6.3, 6.4 y 6.5, representando las mediciones 1, 2, 3 y 4, respectivamente. Se puede apreciar una tendencia muy similar en todas ellas, y es que el uso de SeQual-Stream disminuye los tiempos respecto a SeQual en buena medida cuando se usan pocos nodos y, al ir aumentándolos, esta diferencia se va reduciendo hasta que los tiempos son muy similares. En las mediciones 2 y 4, que procesan en modo paired-end, la mejora obtenida es mucho más clara.

En la mediciones 1 y 3, SeQual-Stream es hasta 1.5 veces más rápido que la versión batch, mientras que obtiene unas aceleraciones de 2.7 y 2.4 en las mediciones 2 y 4, respectivamente.

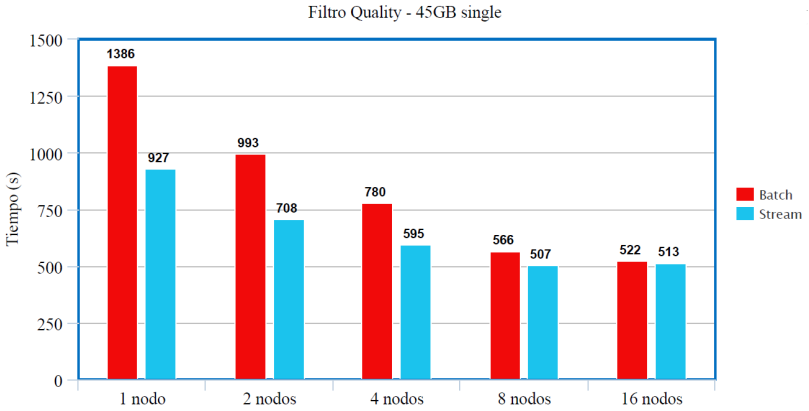


Figura 6.2: Medición 1 - Filtro QUALITY por calidad mínima de 25

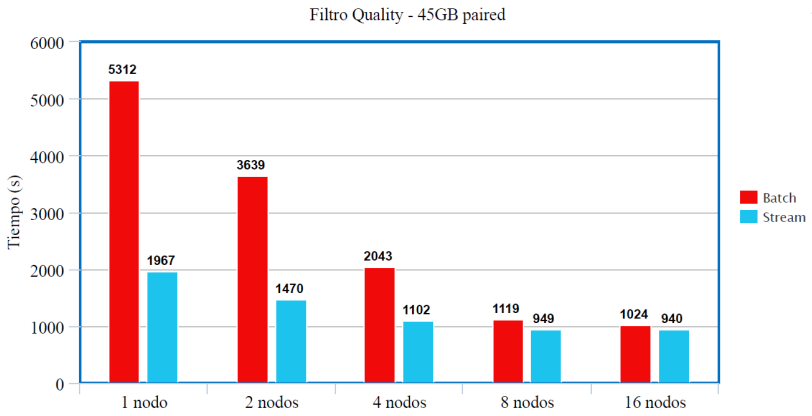


Figura 6.3: Medición 2 - Filtro QUALITY por calidad mínima de 25

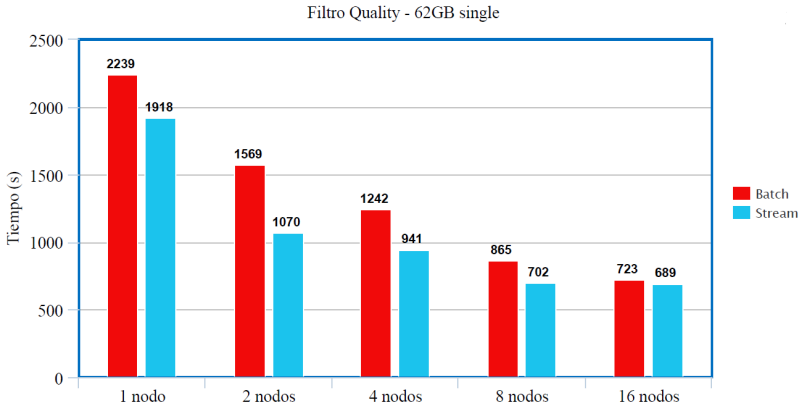


Figura 6.4: Medición 3 - Filtro QUALITY por calidad mínima de 25

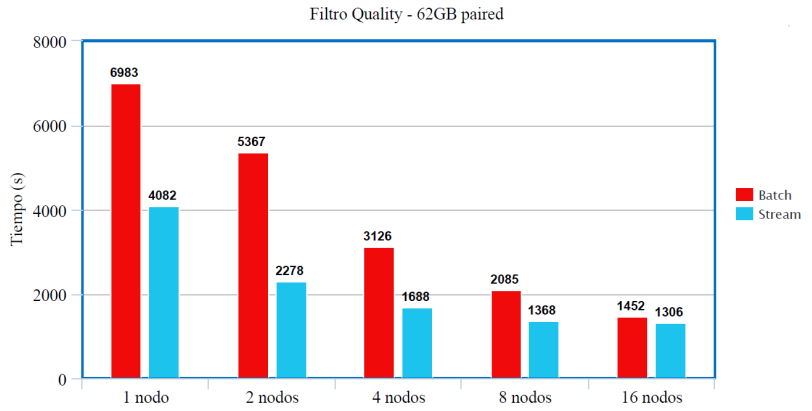


Figura 6.5: Medición 4 - Filtro QUALITY por calidad mínima de 25

### 6.3.2 Filtro NONIUPAC

Los resultados utilizando un filtro NONIUPAC se pueden observar en las gráficas de las Figuras 6.6, 6.7, 6.8 y 6.9, representando las mediciones 1, 2, 3 y 4, respectivamente. Se sigue una tendencia muy similar al filtro anterior, siendo las aceleraciones máximas obtenidas en este caso de 1.4, 2.2, 1.2 y 2, para cada una de cuatro las mediciones realizadas.

### 6.3.3 Formateador DNATORNA

Los resultados utilizando un formateador DNATORNA se muestran en las Figuras 6.10, 6.11, 6.12 y 6.13, representando las mediciones 1, 2, 3 y 4, respectivamente. Las aceleraciones obtenidas por SeQual-Stream en esta operación son de 1.4, 2.1, 1.8 y 2, para cada una de las mediciones.

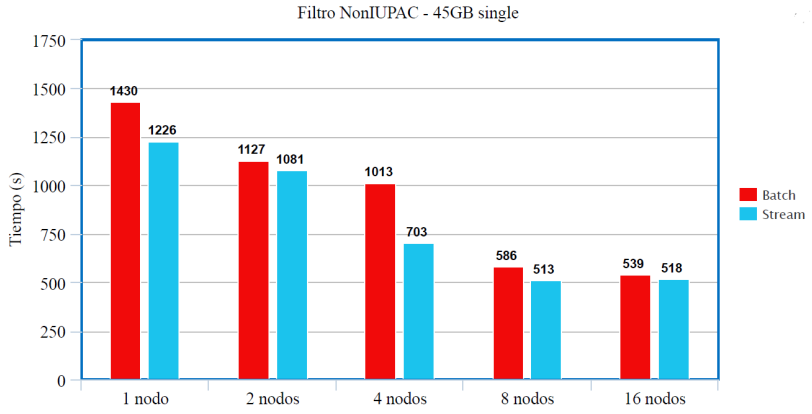


Figura 6.6: Medición 1 - Filtro NONIUPAC

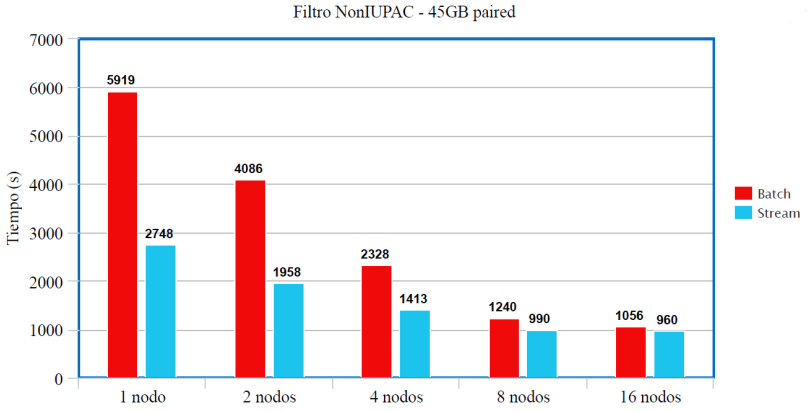


Figura 6.7: Medición 2 - Filtro NONIUPAC

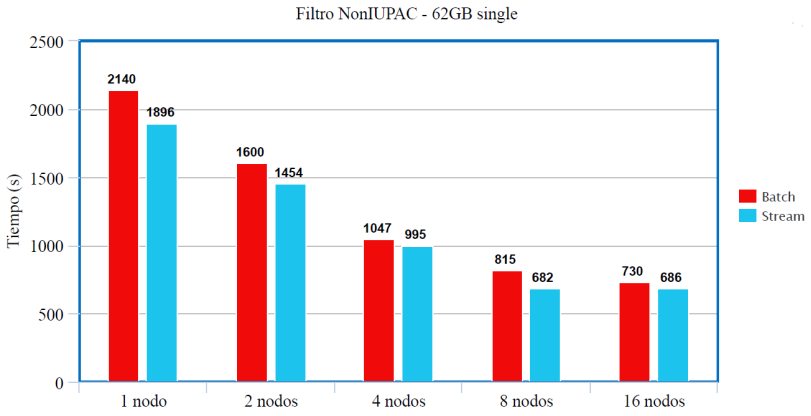


Figura 6.8: Medición 3 - Filtro NONIUPAC

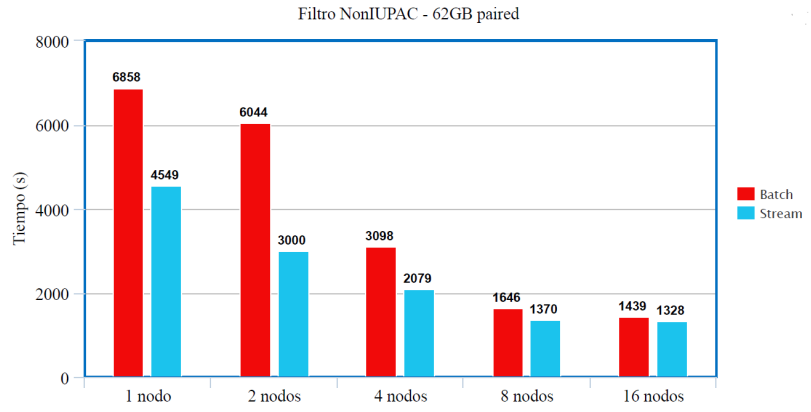


Figura 6.9: Medición 4 - Filtro NONIUPAC

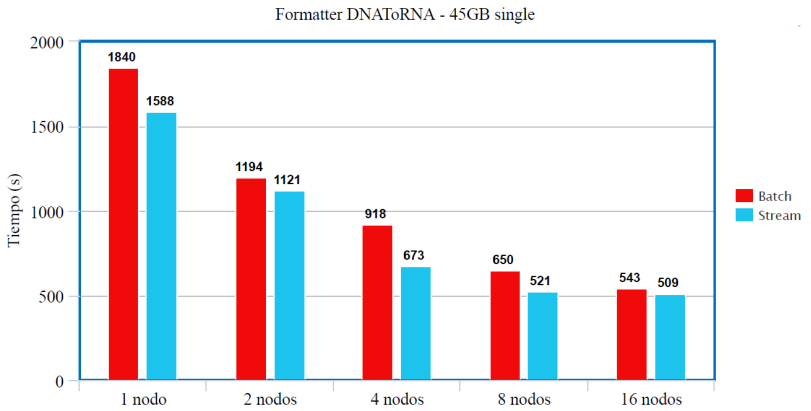


Figura 6.10: Medición 1 - Formateador DNATORNA

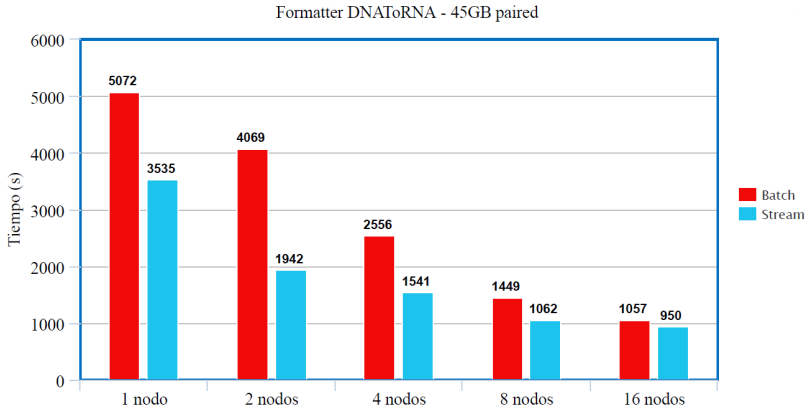


Figura 6.11: Medición 2 - Formateador DNATORNA

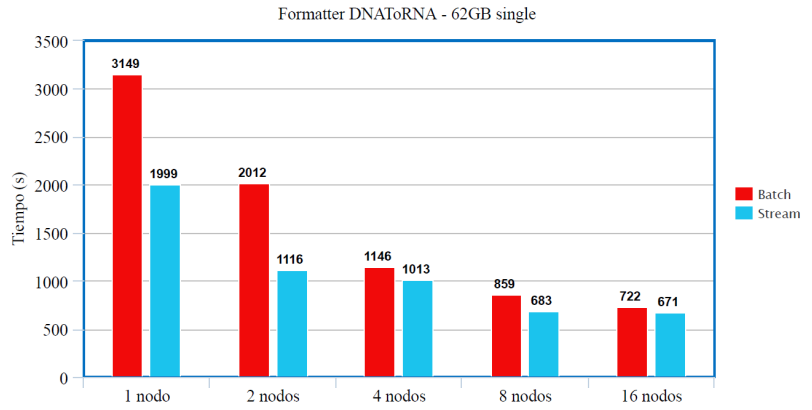


Figura 6.12: Medición 3 - Formateador DNATORNA

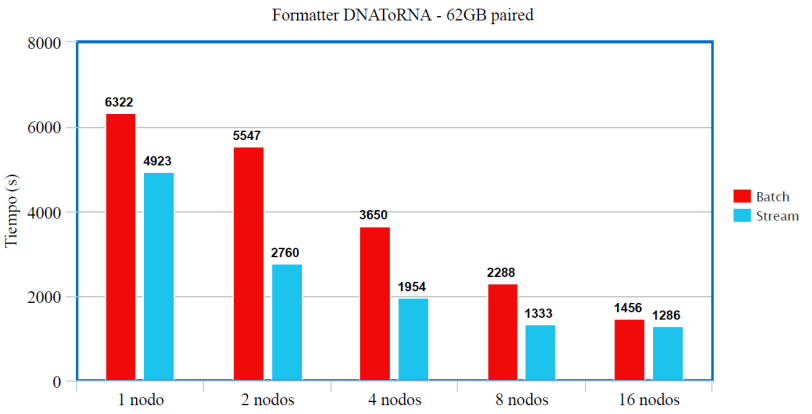


Figura 6.13: Medición 4 - Formateador DNATORNA

### 6.3.4 Recortador TRIMRIGHTP

Por último, se muestran los resultados utilizando un recortador TRIMRIGHTP en las gráficas de las Figuras 6.14, 6.15, 6.16 y 6.17, representando las mediciones 1, 2, 3 y 4, respectivamente. Para esta operación se obtienen las siguientes aceleraciones máximas para cada medición: 1.6, 2.6, 1.7 y 2.3.

### 6.3.5 Conclusiones

De los resultados obtenidos se pueden extraer una serie de conclusiones valorando diferentes parámetros. Respecto al tipo de operación, las que obtuvieron una aceleración ligeramente superior fueron el filtro QUALITY y el recortador TRIMRIGHTP, frente a las otras dos operaciones (filtro NONIUPAC y formateador DNATORNA). Además, estas dos últimas

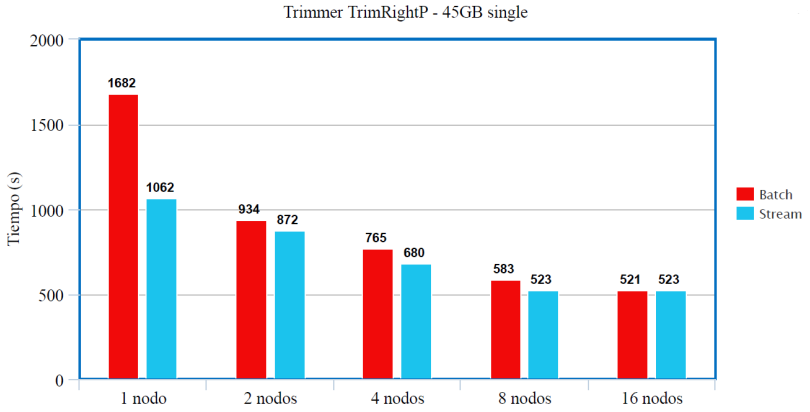


Figura 6.14: Medición 1 - Recortador TRIMRIGHTP recortando el 10%

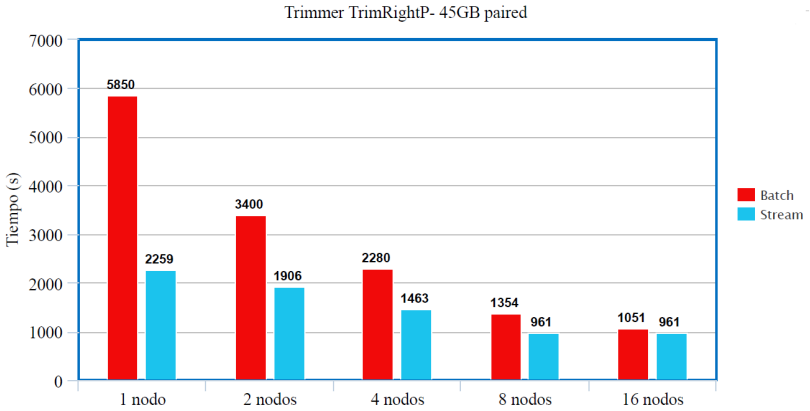


Figura 6.15: Medición 2 - Recortador TRIMRIGHTP recortando el 10%

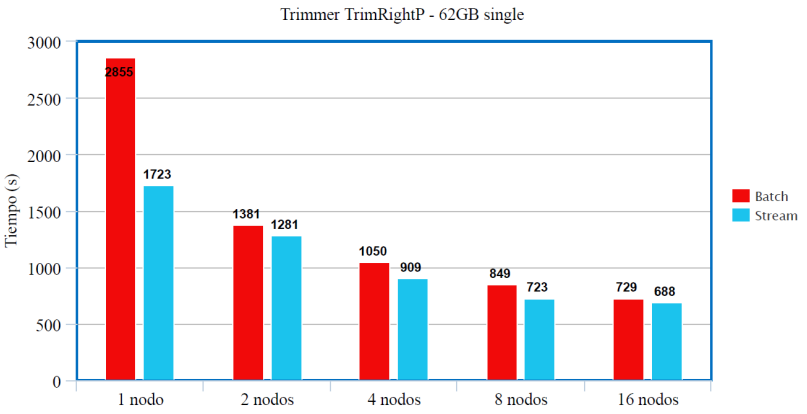


Figura 6.16: Medición 3 - Recortador TRIMRIGHTP recortando el 10%

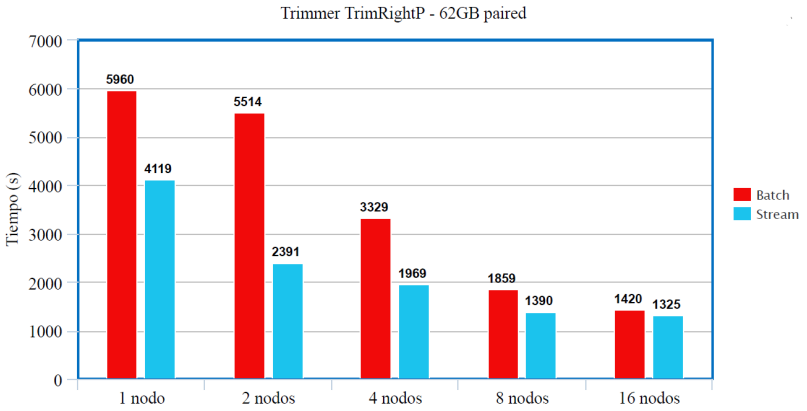


Figura 6.17: Medición 4 - Recortador TRIMRIGHTP recortando el 10%

fueron las operaciones que, en general, obtuvieron los tiempos de procesamiento más altos para ambas herramientas. El principal motivo tiene que ver con la cantidad de datos escritos en HDFS como salida final. Mientras que el filtro QUALITY quita secuencias del conjunto de datos de entrada y TRIMRIGHTP las hace más pequeñas (recorta sus bases), DNATORNA siempre escribe la misma cantidad de secuencias que lee como entrada (simplemente hace un cambio de formato), y el filtro NONIUPAC no elimina ninguna de ellas en este caso ya que las bases de los conjuntos de datos utilizados ya están con nomenclatura IUPAC. Esto implica que las operaciones que más datos escriben a la salida incurren en una mayor sobrecarga en los discos de los nodos worker, que actúan de DataNodes. Esto en SeQual-Stream es más limitante porque, de forma paralela al procesamiento y escritura de las diferentes partes de la salida, también se está realizando la copia del fichero (o ficheros) de entrada en HDFS.

El siguiente punto a analizar es la escalabilidad de las herramientas. Se puede apreciar que cuando se aumenta en gran medida el número de nodos (8 y 16), la mejora obtenida por

SeQual-Stream se reduce y los tiempos acaban convergiendo prácticamente al mismo valor. El motivo es que llega un punto en el que se dispone de tanta capacidad computacional y de tantos discos en los que poder repartir la escritura de las partes, que el procesamiento y la escritura de los resultados son lo suficientemente rápidos como para que el factor limitante del rendimiento sea la velocidad de copia del fichero de entrada. A la hora de copiar el fichero (o ficheros) de entrada en HDFS, la fase de escritura sí puede mejorar al disponer de más nodos, pero no ocurre lo mismo con la fase de lectura ya que se depende del rendimiento del servidor NAS del clúster y del protocolo NFS. Este es un factor que afecta tanto a la versión batch como a la streaming, y termina convirtiéndose en el principal cuello de botella.

Otro tema a considerar es cómo evoluciona la aceleración obtenida por SeQual-Stream al aumentar el tamaño de los datos de entrada. Cuando se procesa en modo single-end, no se aprecia mucha diferencia entre el fichero de 45 GB y el de 62 GB, pues ambos rondan aproximadamente los mismos valores de mejora obtenida. Lo mismo ocurre entre las mediciones paired-end, aunque en la medición 2 se aprecia que tiende a ser algo mayor respecto de la 4. Donde sí existe una clara diferencia es en las mediciones de procesamiento paired-end respecto de las mediciones single-end. Se puede observar que al duplicar la cantidad de secuencias procesadas en relación a su versión single, las mejoras obtenidas por SeQual-Stream son muy superiores. Por ejemplo, para el filtro QUALITY y las mediciones 1 y 2 (Figuras 6.2 y 6.3), las aceleraciones máximas eran de 1.5 y 2.7, respectivamente; viéndolo porcentualmente, en la medición 1 se redujo el tiempo un 33% respecto a SeQual, mientras que en la medición 2 se redujo hasta un 63%. El motivo es que al aumentar significativamente la cantidad de datos de entrada también aumenta proporcionalmente el tiempo necesario para copiar el/los fichero/s en HDFS, procesarlos y escribir sus partes, con lo que paralelizar este proceso mediante un modelo streaming es muy beneficioso y es lo que se buscaba precisamente con el desarrollo de esta herramienta.

Por último, cabe mencionar un aspecto relacionado con las características del clúster utilizado, y es que sus recursos son compartidos con otros usuarios que ejecutan sus aplicaciones (que es lo habitual en estos entornos). Para la carga computacional de las herramientas esto no genera ningún problema, ya que se garantiza en todo momento el aislamiento entre los nodos de cómputo utilizados en los experimentos de forma que ningún otro usuario pueda interferir en un trabajo en progreso. Sin embargo, el servidor NAS debe dar servicio de almacenamiento a todos los usuarios del clúster. Por tanto, es posible que varios trabajos de diferentes usuarios estén accediendo al servidor NAS de forma simultánea, ralentizándose entre sí (el servidor NFS se convierte en un cuello de botella). Esto es un inconveniente para SeQual, porque puede interferir con la operación de copia en HDFS, pero ahí acaba el problema ya que su procesamiento posterior no se ve afectado. En SeQual-Stream, sin embargo, es un factor que influye durante toda la ejecución y, por ello, en función de los trabajos exis-

tentes en el clúster en cada momento los resultados obtenidos podrían variar. Aunque no es un factor muy limitante, además de ser inevitable en este clúster por sus características, sí es algo a tener en cuenta.

## 6.4 Pruebas realizadas en descarga

Para explotar otra de las ventajas principales del desarrollo de esta herramienta, también se realizaron una serie de experimentos realizando la descarga de los ficheros de entrada desde un servidor remoto.

Las condiciones son las mismas que las comentadas en la Sección 6.2, con la diferencia de que solo se realizaron las pruebas para los ficheros más grandes de 62 GB (mediciones 3 y 4), y que los conjuntos de datos ahora no se encuentran almacenados previamente en el servidor NAS, sino que están alojados en el servidor Web del Grupo de Arquitectura de Computadores (GAC) [35] y se requiere su descarga. Se han descargado desde un servidor interno a la universidad para garantizar una velocidad de descarga más o menos constante que permitiese obtener resultados de rendimiento con poca variabilidad, evitando las fluctuaciones típicas si se realizara la descarga desde servidores en Internet. Para SeQual-Stream, los ficheros de entrada se descargan desde el servidor Web al servidor NAS del clúster y la herramienta los procesa según se escriben en la NAS, leyéndolos a través de NFS y copiándolos en HDFS. En el caso de SeQual, se realizaron unas pruebas que miden el tiempo que se tarda en descargar los ficheros y copiarlos directamente en HDFS. Luego se obtuvo la diferencia de tiempo entre este proceso y lo que se tardaba en copiarlos en HDFS cuando ya se tenían descargados, sumando ese tiempo adicional a los resultados anteriores de SeQual.

### 6.4.1 Filtro QUALITY

Los resultados utilizando un filtro QUALITY se pueden observar en las Figuras 6.18 y 6.19 para las mediciones 3 y 4, respectivamente.

Se puede observar que cuando se utiliza un número alto de nodos ahora sí existe una diferencia apreciable en los tiempos de ejecución entre las versiones batch y streaming, a diferencia de los experimentos anteriores donde estos tiempos eran muy similares. Las aceleraciones máximas obtenidas por SeQual-Stream son 1.9 y 2.2 para las mediciones 3 y 4, respectivamente. Además, en contraposición al escenario sin descarga, las mejoras obtenidas por SeQual-Stream en los peores casos siguen siendo significativas: 1.3 y 1.4 veces más rápido que SeQual.

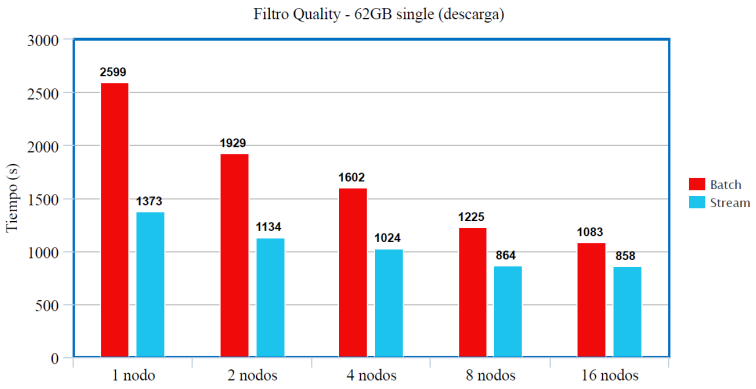


Figura 6.18: Medición 3 - Filtro QUALITY por calidad mínima de 25 descargando el fichero de entrada

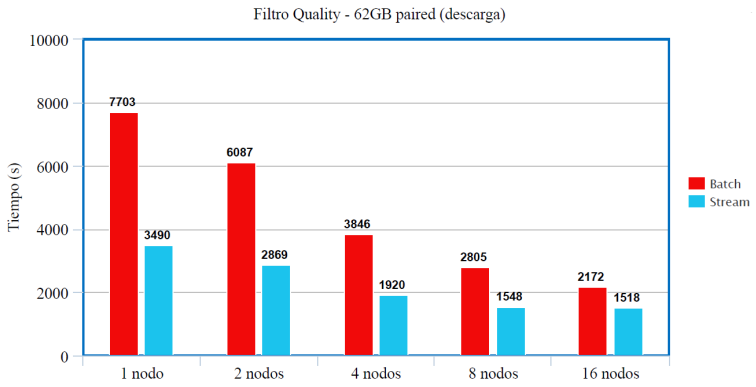


Figura 6.19: Medición 4 - Filtro QUALITY por calidad mínima de 25 descargando los ficheros de entrada

### 6.4.2 Filtro NONIUPAC

Los resultados utilizando un filtro NONIUPAC se muestran en las Figuras 6.20 y 6.21 para las mediciones 3 y 4, respectivamente. En este caso se obtiene una aceleración máxima de 1.5 para la medición 3 y de 2.3 para la medición 4. Las aceleraciones mínimas son de 1.3 y 1.4, respectivamente.

### 6.4.3 Formateador DNATORNA

Los resultados utilizando un formateador DNATORNA se pueden observar en las Figuras 6.22 y 6.23 para las mediciones 3 y 4, respectivamente.

De forma similar a los casos anteriores, las aceleraciones máximas obtenidas por la versión streaming son de 1.8 para la medición 3 y 2.4 para la medición 4, mientras que las mínimas son de 1.3 y 1.5, respectivamente.

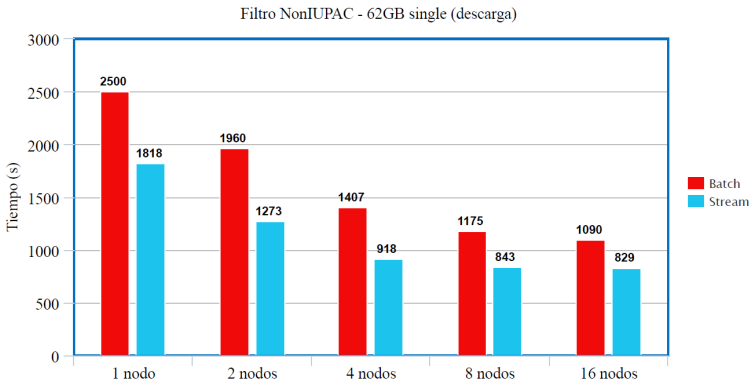


Figura 6.20: Medición 3 - Filtro NONIUPAC descargando el fichero de entrada

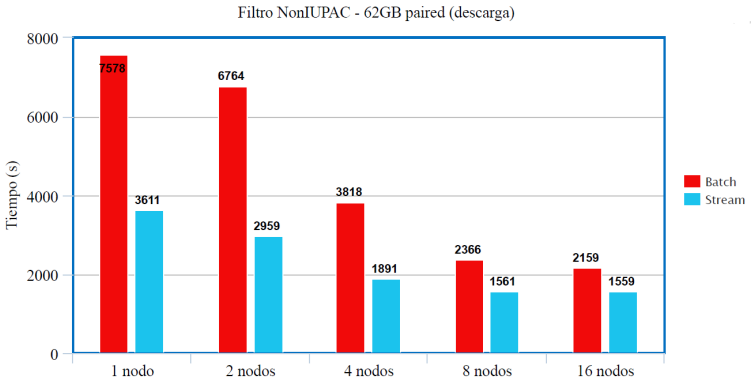


Figura 6.21: Medición 4 - Filtro NONIUPAC descargando los ficheros de entrada

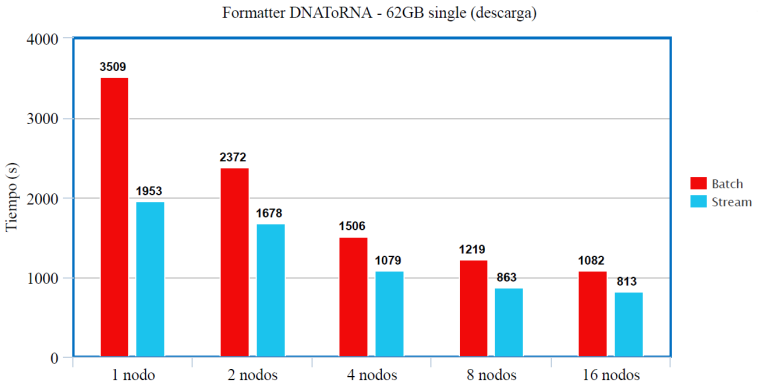


Figura 6.22: Medición 3 - Formateador DNATORNA descargando el fichero de entrada

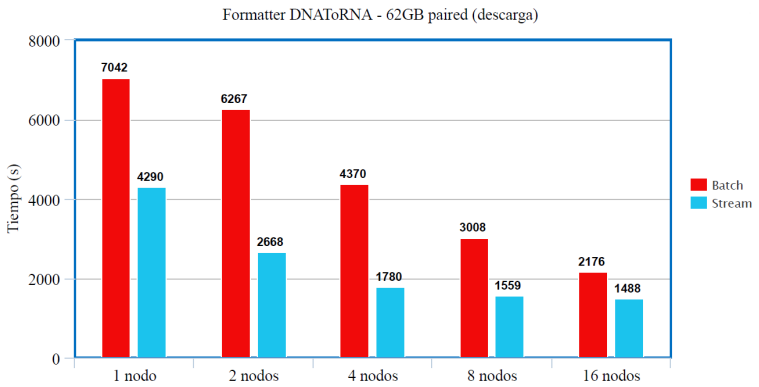


Figura 6.23: Medición 4 - Formateador DNATORNA descargando los ficheros de entrada

### 6.4.4 Recortador TRIMRIGHTP

Los resultados utilizando un recortador TRIMRIGHTP se muestran en las Figuras 6.24 y 6.25 para las mediciones 3 y 4, respectivamente.

En esta operación las aceleraciones máximas obtenidas son muy similares: 2.1 y 2.2 para las mediciones 3 y 4, respectivamente, mientras que las mínimas son de 1.3 y 1.4.

### 6.4.5 Conclusiones

De esta batería de pruebas que realizan la descarga de los conjuntos de datos desde una localización remota podemos extraer las siguientes conclusiones en comparación con los experimentos anteriores. Las mejoras obtenidas por SeQual-Stream en descarga son, en general, ligeramente superiores que las de sus contrapartes sin descarga. Aunque lo más notorio es que incluso en los casos más desfavorables, con 8 y 16 nodos, ahora sí se obtiene cierta mejora,

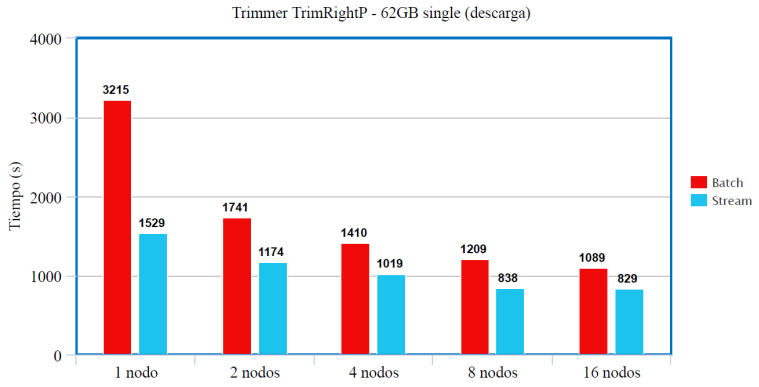


Figura 6.24: Medición 3 - Recortador TRIMRIGHTP recortando el 10% y descargando el fichero de entrada

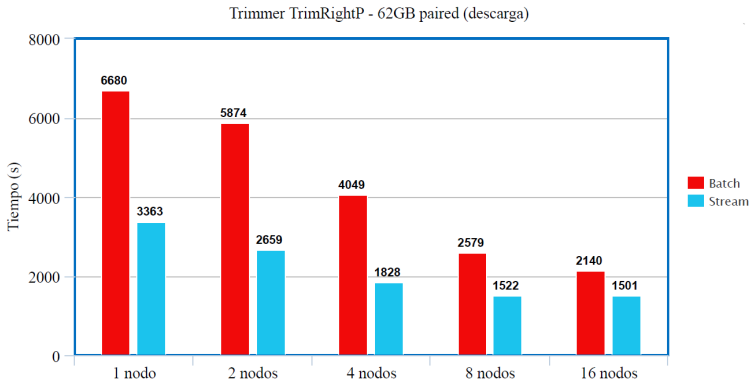


Figura 6.25: Medición 4 - Recortador TRIMRIGHTP recortando el 10% y descargando los ficheros de entrada

mientras que antes esta era mínima o inexistente. Esto ocurre porque al tardar más tiempo el proceso de copia de los ficheros de entrada en HDFS debido a la descarga, la versión streaming saca mayor provecho de no tener que esperar a que termine la descarga para poder iniciar el procesamiento.

Otro detalle destacable es que las mejoras con el fichero single de la medición 3 aumentan en mayor medida que las mejoras con los ficheros paired de la medición 4, que se mantienen bastante similares. El motivo es que la lectura sobre dos ficheros requiere ir copiando ambos de forma simultánea para juntar cada secuencia con su pareja. Al realizar la descarga es altamente probable que ambos ficheros no se descarguen de forma sincronizada, es decir, que uno lo haga de forma más rápida que el otro, y los datos adicionales de los que se dispone del fichero que se descarga más rápido no son aprovechables por SeQual-Stream, ya que es necesario esperar a que lleguen las secuencias correspondientes del segundo fichero. Por tanto, esto termina siendo más ineficiente que copiar un único fichero, como ocurre en el modo de procesamiento single-end.

# Capítulo 7

# Conclusiones

Como último capítulo de la memoria, se exponen una serie de conclusiones a raíz del trabajo realizado en este TFG, además de unas consideraciones sobre posibles funcionalidades a añadir a la herramienta en un futuro.

## 7.1 Reflexiones principales

Se puede considerar el trabajo realizado como altamente satisfactorio, al conseguir una herramienta paralela capaz de realizar controles de calidad genética mediante un modelo de procesamiento en streaming. La evaluación comparativa del rendimiento respecto a la versión batch demuestra que SeQual-Stream puede llegar a ser el doble de rápido o incluso más. Esto conlleva que pueda resultar una herramienta más útil en aquellos casos reales donde se necesite llevar a cabo múltiples experimentos muy pesados, ya que una aceleración así sobre cada experimento resultaría en una mejora global muy significativa.

Centrándome un poco más en lo que significó este trabajo para mí, a nivel técnico y formativo, me ha permitido conocer y profundizar sobre tecnologías Big Data de código abierto, como Spark, HDFS y, en general, el entorno Hadoop, herramientas de gran interés en el ámbito empresarial y que no se incluyen en el actual plan de estudios de la titulación. De la misma forma, también adquirí formación en conceptos de bioinformática, un campo muy extenso e interesante y que cada vez tiene más relevancia hoy en día. Además, he tenido la oportunidad de trabajar en un entorno real HPC, con un clúster de altas prestaciones como Plutón [32] y aprender a interactuar con un planificador de trabajos como Slurm [36] para la ejecución de los experimentos. También es destacable lo aprendido sobre el paradigma streaming y los retos que ello conlleva, como la problemática de que los datos aparezcan incompletos al final de una lectura. A nivel personal, me ha resultado una experiencia muy enriquecedora, al permitirme afrontar un nuevo tipo de trabajo y de este calibre, aprendiendo de primera mano la importancia de seguir metodologías como la iterativa incremental para un desarrollo más

sencillo y fluido.

Cabe mencionar que la herramienta desarrollada en este TFG se encuentra disponible en el siguiente repositorio Git: https://github.com/oscar-castellanos/SeQual-Stream. SeQual-Stream se distribuye bajo una licencia GNU General Public License [37], con lo que se permite que cualquier usuario pueda descargar, usar, modificar y redistribuir la aplicación, aportando en su crecimiento y expansión de funcionalidades.

## 7.2 Relación con la titulación

El desarrollo de esta herramienta ha conseguido que aprenda sobre determinados conceptos completamente desconocidos para mí hasta el momento, pero también he podido aprovechar y complementar la formación recibida en otras asignaturas de la titulación. Dentro de ellas, se pueden destacar asignaturas troncales como **Programación I** y **Programación II**, donde aprendí conceptos básicos y fundamentales sobre programación en general, o **Diseño Software**, donde me formé sobre el lenguaje Java y su paradigma orientado a objetos, además de consejos y patrones para obtener un buen diseño de la aplicación. También son destacables las asignaturas de **Internet y Sistemas Distribuidos**, donde aprendí a utilizar la herramienta Maven como gestor de proyectos Java, y **Concurrencia y Paralelismo**, donde se profundiza en conceptos importantes como el threading y el paralelismo entre núcleos de procesamiento de un sistema computacional.

Por otro lado, también cabe mencionar lo aprovechado de materias específicas de la mención en **Tecnologías de la Información**. Por ejemplo, **Integración de Aplicaciones**, donde se vuelve a trabajar con Maven, y **Administración de Infraestructuras y Sistemas Informáticos**, que aborda conceptos relacionados con la computación de altas prestaciones y los sistemas de ficheros distribuidos (tópico donde se incluye a HDFS), además de aprovechar la oportunidad para realizar un trabajo tutelado opcional sobre el despliegue de un clúster virtual con Spark y HDFS.

Finalmente, no está de más hacer mención a la aplicación de competencias generales del título [38], pudiendo destacar:

- **A20**: "Conocimiento y aplicación de los principios fundamentales y técnicas básicas de la programación paralela, concurrente, distribuida y de tiempo real."

- **A22**: "Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería de software."

## 7.3 Trabajo futuro

Por último, se realiza una consideración final sobre algunas funcionalidades o mejoras que sería interesante implementar en un futuro para aumentar la potencia y flexibilidad de la herramienta.

Lo primero sería adaptar las operaciones de SeQual que realizan el procesamiento teniendo en cuenta todo el conjunto de secuencias: las operaciones de filtrado grupal y las estadísticas. Para aprovechar el rendimiento en un modelo streaming se requeriría realizar un procesamiento de forma incremental a medida que se descargan datos nuevos. Por poner un ejemplo, para la estadística de COUNT (cuenta de secuencias), lo ideal sería ir contabilizando el número de secuencias según se leen y se procesan en un contador que se vaya incrementando Aunque la idea general es simple, la implementación puede no ser tan directa y habría que reflexionar sobre ella.

Otra funcionalidad de interés sería la capacidad de procesar datos comprimidos y así evitar el tiempo necesario para su descompresión. Aunque esta funcionalidad tampoco existe actualmente en SeQual, en el caso de SeQual-Stream sería una característica importante, dado que al realizar una descarga desde Internet de un conjunto de datos, la mayoría de las veces se pueden descargar en fomato comprimido y así depender en menor medida de la velocidad de descarga y el tráfico de red. Esta funcionalidad supone un reto debido a cómo los diferentes formatos de compresión almacenan la información. Por ejemplo, formatos como gzip guardan cierta información del fichero al final del comprimido, haciendo que sea necesario esperar a que la descarga se complete para poder iniciar la descompresión. Sería más adecuado utilizar formatos de compresión de bloque, que comprimen el fichero de entrada dividido en bloques que se pueden descomprimir por separado. Alternativamente, se podría utilizar alguna técnica adicional para solventar la problemática con los formatos como gzip y permitir su descompresión en streaming.

Otro detalle tiene que ver con el proceso de copia de ficheros en HDFS. Actualmente, se utiliza un factor de replicación de 1 para conseguir un buen rendimiento, pero sería interesante que ese valor se calcule de forma dinámica en función de los nodos del clúster, aumentándolo si se considera que hay suficientes nodos como para mantener un alto rendimiento. Relacionado con esto, también podría ser útil que la cantidad de bytes mínima total a copiar en cada iteración del thread de lectura se calcule de manera continua, en función del estado actual del clúster. Por ejemplo, si durante la ejecución se pierde un nodo, lo ideal sería actualizar y reducir ese valor para adaptarse a la nueva capacidad de cómputo.

# Lista de acrónimos

**BDEv** Big Data Evaluator.

**DAG** Directed Acyclic Graph.

**GPL** General Public License.

**GUI** Graphical User Interface.

**HDFS** Hadoop Distributed File System.

**HPC** High Performance Computing.

**IDE** Integrated Development Environment.

**NAS** Network Attached Storage.

**NFS** Network File System.

**NGS** Next Generation Sequencing.

**RDD** Resilient Distributed Dataset.

**YARN** Yet Another Resource Negotiator.

# Glosario

**Apache Hadoop** Framework de código abierto que permite el procesamiento distribuido de grandes conjuntos de datos.

**Apache Spark** Framework de código abierto de programación para procesamiento de datos distribuidos diseñado para ser rápido y de propósito general.

**DataNode** Proceso Java que se ejecuta en un nodo worker de un clúster Hadoop, encargado de almacenar los datos de su sistema de ficheros.

**Extremo 3'** Representa el extremo de una hebra de ADN o ARN que coincide con el grupo hidroxilo del tercer carbono de la respectiva ribosa o desoxirribosa terminal.

**Extremo 5'** Representa el extremo de una hebra de ADN o ARN que coincide con el grupo fosfato del quinto carbono de la respectiva ribosa o desoxirribosa terminal.

**Formato FASTA** Formato de representación de secuencias basado en texto, donde cada base de una secuencia se codifica con un único carácter ASCII y en el que cada secuencia se identifica con un nombre.

**Formato FASTQ** Formato de representación de secuencias basado en texto, donde cada base de una secuencia tiene una puntuación de calidad, y tanto las bases como las puntuaciones se codifican con un único carácter ASCII. También cada secuencia se identifica con un nombre.

**NameNode** Proceso Java que se ejecuta en el nodo maestro del clúster Hadoop, encargado de gestionar su sistema de ficheros.

**Next Generation Sequencing (NGS)** Conjunto de tecnologías diseñadas para analizar gran cantidad de ADN de forma masiva.

**Puntuación Phred** Medida de calidad en la identificación de las nucleobases generadas por la secuenciación automatizada de ADN.

**Secuenciación paired-end** Tipo de secuenciación que utiliza dos lectores, donde cada uno lee una cadena diferente de la secuencia empezando en extremos opuestos.

**Secuenciación single-end** Tipo de secuenciación que lee una única cadena de una secuencia.

**SeQual** Herramienta paralela diseñada para el control de calidad de secuencias genéticas de forma masiva.

**Spark Structured Streaming** Módulo de Apache Spark diseñado para el procesamiento de datos en streaming.

# Bibliografía


[1] K. A. Phillips, "Assessing the value of next-generation sequencing technologies: An introduction," *Value in Health*, vol. 21, no. 9, pp. 1031–1032, 2018.

[2] R. R. Expósito, R. Galego-Torreiro, and J. González-Domínguez, "SeQual: Big Data tool to perform quality control and data preprocessing of large NGS datasets," *IEEE Access*, vol. 8, pp. 146 075–146 084, 2020.

[3] ——, "SeQual," 2019. [En línea]. Disponible en: https://github.com/UDC-GAC/SeQual

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: a unified engine for Big Data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[5] "Apache Spark." [En línea]. Disponible en: https://spark.apache.org

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*. Incline Village, NV, USA, 2010, pp. 1–10.

[7] "Apache Hadoop." [En línea]. Disponible en: https://hadoop.apache.org

[8] N. M. Luscombe, D. Greenbaum, M. Gerstein *et al.*, "What is bioinformatics? An introduction and overview," *Yearbook of Medical Informatics*, vol. 1, no. 83–100, 2001.

[9] Genome.gov, "Mutación Genética - Inserción." [En línea]. Disponible en: https://www.genome.gov/es/genetics-glossary/Insercion

[10] ——, "Mutación Genética - Deleción." [En línea]. Disponible en: https://www.genome.gov/es/genetics-glossary/Delecion

[11] A. Griffiths, W. Gelbart, J. Miller, and R. Lewontin, *Modern Genetic Analysis*. New York, USA: WH Freeman Publishers, 1999, ch. Chromosomal rearrangements. [En línea]. Disponible en: https://www.ncbi.nlm.nih.gov/books/NBK21367/

[12] "FASTQ Format." [En línea]. Disponible en: https://learn.gencore.bio.nyu.edu/ngs-file-formats/fastq-format/

[13] Zhang Lab, "FASTA Format." [En línea]. Disponible en: https://zhanglab.dcmb.med.umich.edu/FASTA/

[14] Illumina, Inc., "Quality Score Encoding." [En línea]. Disponible en: https://support.illumina.com/help/BaseSpace_OLH_009008/Content/Source/Informatics/BS/QualityScoreEncoding_swBS.htm

[15] Dr. Phil Green's group, "Phred - Quality Base Calling." [En línea]. Disponible en: https://www.phrap.com/phred/

[16] Illumina, Inc., "Advantages of paired-end and single-read sequencing." [En línea]. Disponible en: https://emea.illumina.com/science/technology/next-generation-sequencing/plan-experiments/paired-end-vs-single-read.html

[17] The Sequencing Center, "What are paired-end reads?" [En línea]. Disponible en: https://thesequencingcenter.com/knowledge-base/what-are-paired-end-reads/

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*. San Francisco, CA, USA, 2004, pp. 137–150.

[19] ——, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. Bolton Landing, NY, USA, 2003, p. 29–43. [En línea]. Disponible en: https://doi.org/10.1145/945445.945450

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA, USA, 2012, pp. 15–28.

[22] "Cluster Mode Overview." [En línea]. Disponible en: https://spark.apache.org/docs/latest/cluster-overview.html

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. Santa Clara, CA, USA, 2013.

[24] "Spark SQL, DataFrames and Datasets Guide." [En línea]. Disponible en: https://spark.apache.org/docs/latest/sql-programming-guide.html

[25] "Structured Streaming Programming Guide." [En línea]. Disponible en: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[26] "Java." [En línea]. Disponible en: https://www.java.com

[27] "Maven." [En línea]. Disponible en: https://maven.apache.org/

[28] "Github." [En línea]. Disponible en: https://github.com/

[29] "Eclipse IDE." [En línea]. Disponible en: https://www.eclipse.org/ide/

[30] J. Veiga, J. Enes, R. R. Expósito, and J. Touriño, "BDEv 3.0: Energy efficiency and microarchitectural characterization of Big Data processing frameworks," *Future Generation Computer Systems*, vol. 86, pp. 565–581, 2018.

[31] "Microsoft Project." [En línea]. Disponible en: https://www.microsoft.com/es-es/microsoft-365/project/project-management-software

[32] "Pluton Cluster." [En línea]. Disponible en: http://pluton.dec.udc.es/

[33] National Center for Biotechnology Information, "Sequence Read Archive." [En línea]. Disponible en: https://www.ncbi.nlm.nih.gov/sra

[34] ——, "National Center for Biotechnology Information." [En línea]. Disponible en: https://www.ncbi.nlm.nih.gov/

[35] "Grupo de Arquitectura de Computadores (GAC)." [En línea]. Disponible en: https://gac.udc.es

[36] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple Linux utility for resource management," in *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*. Seattle, WA, USA, 2003, pp. 44–60.

[37] The Free Software Foundation (FSF), "GNU General Public License," 2007. [En línea]. Disponible en: https://www.gnu.org/licenses/gpl-3.0.html

[38] Facultade de Informática da Coruña, "Guía docente. Competencias del título," 2021. [En línea]. Disponible en: https://guiadocente.udc.es/guia_docent/index.php?centre=614&ensenyament=614G01&consulta=competencies&idioma=cast