Autores
Sara Lanuza Orna
Arnau Gabriel Atienza

Dirigido por
Pedro García López

TRABAJO DE FIN DE GRADO

Alineación de genotipos en sistemas distribuidos

INGENIERÍA INFORMÁTICA

UNIVERSITAT ROVIRA I VIRGILI

Tarragona, 2023

Resumen

En los últimos años, se ha observado un crecimiento significativo en la demanda de migrar aplicaciones existentes hacia la nube, debido a las limitaciones inherentes de las ejecuciones locales y secuenciales. Este crecimiento ha sido especialmente notable en sectores relacionados con Big Data y análisis de datos. La nube ofrece una solución simplificada para gestionar la infraestructura necesaria en la ejecución de proyectos complejos, al hacer que los recursos computacionales sean más accesibles para los usuarios. En este proyecto en particular, se partió de una implementación antigua de herramientas de bioinformática destinadas al análisis de genomas en la nube, con el objetivo de mejorar su funcionamiento y rendimiento. Esta tarea implicó analizar más de 10.000 líneas de código, las cuales estaban escasamente documentadas, y posteriormente reimplementar el proyecto prácticamente desde cero. Además, fue necesario replantear el diseño arquitectónico del proyecto para aprovechar de manera más eficiente los recursos disponibles. Se implementó una arquitectura distribuida y paralela correcta, ya que la implementación anterior presentaba problemas que limitaban la escalabilidad del programa debido a un diseño subóptimo que no aprovechaba al máximo los recursos disponibles. Estas optimizaciones implicaron cambios en el manejo de los datos de entrada y los datos intermedios generados a lo largo del programa. El resultado de este trabajo ha sido una mejora sustancial en términos de coste, tiempo y escalabilidad del programa, permitiendo aprovechar al máximo los servicios que ofrece la computación en la nube. Además, se ha mejorado la calidad del código mismo, que ahora está mejor documentado y es más accesible para los usuarios interesados en utilizar el proyecto.

Resum

En els últims anys, s'ha observat un creixement significatiu en la demanda de migrar aplicacions existents cap al núvol, a causa de les limitacions inherents a les execucions locals i seqüencials. Això ha estat especialment important en els sectors relacionats amb BigData i anàlisis de dades. El núvol proporciona una solució simplificada per a la gestió de la infraestructura necessària en l'execució de projectes complexos, en fer que els recursos computacionals siguin més accessibles per als usuaris. En aquest projecte en particular, es va partir d'una antiga implementació d'eines de bioinformàtica destinades a l'anàlisi de genomes en el núvol, amb l'objectiu de millorar el seu funcionament i rendiment. Aquesta tasca va involucrar l'anàlisi de més de 10.000 línies de codi, les quals es trobaven escassament documentades, i la seva posterior reimplementació pràcticament des de zero. A més d'això, va ser necessari replantejar el disseny arquitectònic del projecte amb la finalitat d'aprofitar de manera més eficient els recursos disponibles. Es va implementar una arquitectura distribuïda i paral·lela correcta, atès que la implementació prèvia presentava problemes que limitaven l'escalabilitat del programa a causa d'un disseny subòptim que no aprofitava al màxim els recursos proporcionats. Aquestes optimitzacions han consistit en el canvi del maneig de les dades d'entrada i les dades intermèdies generades al llarg del programa. El resultat d'aquest treball ha resultat en una millora substancial de cost, temps i escalabilitat del programa, permetent aprofitar al màxim els serveis que ofereix la computació en el núvol. A més, s'ha millorat la qualitat generada del codi mateix, el qual ha quedat més ben documentat i més accessible per als usuaris que vulguin fer ús del projecte.

Abstract

In recent years, there has been a significant growth in the demand to migrate existing applications to the cloud, due to the inherent limitations of local and sequential execution. This has been especially important in industries related to Big Data and data analytics. The cloud provides a simplified solution for managing the infrastructure required in the execution of complex projects by making computational resources more accessible to users. In this particular project, we started from an old implementation of bioinformatics tools aimed at genome analysis in the cloud, with the goal of improving their operation and performance. This task involved the analysis of more than 10,000 lines of code, which were poorly documented, and their subsequent reimplementation practically from scratch. In addition, it was necessary to rethink the architectural design of the project in order to make more efficient use of available resources. A correct distributed and parallel architecture was implemented, since the previous implementation presented problems that limited the scalability of the program due to a suboptimal design that did not take full advantage of the resources provided. These optimizations have consisted of changing the handling of input data and intermediate data generated throughout the program. The result of this work has resulted in a substantial improvement in cost, time and scalability of the program, allowing to take full advantage of the services offered by cloud computing. In addition, the quality of the code itself has been improved, which has been better documented and more accessible to users who want to make use of the project.

Índice

Índice de figuras 6

Índice de cuadros 7

Índice de códigos 8

1 Introducción 9

1.1 Motivación del proyecto . . . 9
1.2 Problemas del proyecto . . . 10
1.3 Objetivos de nuestro trabajo . . . 10

2 Distribución del trabajo 12

2.1 Arnau Gabriel Atienza . . . 12
2.2 Sara Lanuza Orna . . . 12

3 Antecedentes 14

3.1 Tecnologías en la nube . . . 14
3.1.1 Proveedores de servicios en la nube . . . 15
3.1.2 Object Storage . . . 16
3.1.3 Funciones como Servicio y Serverless . . . 17
3.2 Modelo Map - Reduce . . . 18
3.2.1 Lithops Middleware . . . 20
3.3 Genómica y el alineamiento de secuencias . . . 21
3.4 Formato de ficheros de entrada . . . 22
3.4.1 FASTA . . . 22
3.4.2 FASTQ . . . 23

4 Análisis del sistema previo 24

4.1 Análisis de código . . . 24
4.2 Análisis de arquitectura . . . 24
4.2.1 Fases principales . . . 25
4.2.2 Preprocesamiento . . . 25
4.2.2.1 Particionado de ficheros FASTA . . . 25
4.2.2.2 Particionado de ficheros FASTQ . . . 26
4.2.2.3 Generación de la iterdata . . . 26
4.2.3 Map . . . 27
4.2.3.1 Mpileup . . . 28
4.2.4 Reduce . . . 29
4.3 Resumen . . . 31

5 Propuesta de la solución 32

5.1 Refactorización del código . . . 32
5.2 Particionado on the fly para fichero FASTA . . . 32
5.3 Eliminación de bloqueos . . . 33

5.4 Optimización de disco 36
5.5 Sistema de estadísticas y registros 36
5.6 Optimización de la generación y obtención del fichero GEM 37
5.7 Resumen 38

6 Implementación **39**

6.1 Estructura principal 39
6.2 Preprocesamiento 42
6.2.1 Particionador FASTA: preprocess_fasta.py 43
6.2.1.1 Generación del fichero índice 43
6.2.1.2 Generación de las particiones 44
6.2.2 Particionador FASTQ: preprocess_fastq.py 44
6.3 Map 45
6.3.1 Coordinador: map_caller.py 45
6.3.2 Funciones de alineamiento: alignment_mapper.py 47
6.3.2.1 Generación del fichero GEM 47
6.3.2.2 Alineación inicial 48
6.3.2.3 Corrección de índices 49
6.3.2.4 Filtrado 50
6.3.3 Funciones Auxiliares de transferencia de datos: data_fetch.py 51
6.3.4 Finalización de la fase de Map 51
6.4 Reduce 52
6.4.1 Coordinador: reduce_caller.py 52
6.4.2 Funciones de reducción y distribución: *reduce_functions.py* 53
6.4.2.1 Distribución de índices 53
6.4.2.2 Reducción 54
6.4.2.3 Fusión final 55
6.5 Monitorización 55
6.6 Resumen 57

7 Validación **58**

7.1 Calidad del código 58
7.2 Evaluación del rendimiento 60
7.2.1 Particionador FASTA 60
7.2.2 Fase *map* 61
7.3 Prueba de escalabilidad 63
7.3.1 Parámetros 63
7.3.2 Fase de *Map* 63
7.3.3 Fase de Reduce 65
7.3.4 Resultados finales 67
7.3.5 Análisis de costes 69
7.4 Conclusiones 70

8 Futuro del proyecto **71**

9 Conclusión **72**

Referencias **73**

Anexo **74**

A Fichero pipeline.py **74**

A.1 Función Run_pipeline 74

B Fichero preprocess_fasta.py **74**

B.1 Función generate_faidx_from_s3 74

B.2 Función create_index_chunked 75

B.3 Función reduce_chunked_indexes 76

B.4 Función get_fasta_byte_ranges 77

B.5 Función get_fasta_byte_ranges 78

B.6 Funciones generate_fastqgz_index_from_s3 y generate_idx_from_gzip . . . 79

B.7 Función get_ranges_from_line_pairs 81

C Fichero stats.py **82**

C.1 Clase stats 82

D Fichero data_fetch.py **84**

D.1 Obtención del segmento FASTQ 84

E Fichero reduce_caller.py **85**

E.1 Coordinación de la fase de Reduce 85

Índice de figuras

2 Símbolo de AWS Lambda, la implementación de AWS de las Funciones como Servicio. 18
3 Ejemplo de modelo Map-Reduce. 19
4 Logo oficial de Lithops. 20
5 Transcripción del ADN [12]. 22
6 Flujo de datos. 25
7 Particionador de ficheros FASTA. 26
8 Esquema de la transferencia de datos en la anterior fase de Map. 27
9 Esquema de la transferencia de datos en la anterior fase de Reduce. 30
10 Particionador propuesto. 33
11 Pasos de la primera fase de alineamiento. 34
12 Pasos de la corrección del índice. 35
13 Pasos de la finalización de la fase de *Map*. 36
14 Pasos de la fase de generación del fichero GEM. 38
15 Estructura del proyecto. 39
16 Fases principales del proyecto. 40
17 Fase de preprocesamiento: particiondado del fichero FASTA. 60
18 Comparativa con la anterior y actual fase *map*. El tamaño de las particiones FASTQ es fijo. 61
19 Fase *map*: *Gem*. El tamaño de las particiones FASTQ es fijo. 61
20 Fase *map*: *Mapper* (subfase 1). El tamaño de las particiones FASTQ es fijo. 62
21 Fase *map*: *Filter_mpileup* (subfase 2). El tamaño de las particiones FASTQ es fijo. 62
22 Generación del fichero GEM 63
23 Alineamiento inicial 64
24 Corrección de los índices 64
25 Aplicación del índice corregido y generación del fichero Mpileup 65
26 Distribución de índices 66
27 Reducción final 66
28 Tiempos por fase 67
29 Transferencias de datos 68
30 Funciones lanzadas 68
31 Coste de las funciones Lambda según su memoria [8] 70

Índice de cuadros

1 Ejemplo de una secuencia en formato FASTA, que incluye una etiqueta de identificación, una descripción y la secuencia de nucleótidos de ADN correspondiente. 22
2 Ejemplo de un fichero FASTQ. 23
3 Ejemplo de 5 filas de un fichero mpileup. 29
4 Análisis del repositorio nuevo. 58
5 Análisis del repositorio antiguo. 58
6 Costes por fase 69
7 Tabla de costes de S3 Select. 70

Índice de códigos

1 Ejemplo de llamada y configuración del programa . . . 40
2 Preprocesado. . . . 42
3 Map. . . . 42
4 Reduce. . . . 42
5 Función *prepare_fasta_chunks*. . . . 43
6 Coordinación de las diferentes fases del Map . . . 45
7 Iterdata de la fase GEM. . . . 46
8 Iterdata de la fase Mapper. . . . 46
9 Iterdata de la fase Corrección de Índice. . . . 46
10 Iterdata de la fase Filtrado. . . . 47
11 Generación del fichero GEM en la función *gem_generator*. . . . 48
12 Alineación entre el fichero GEM y el fichero FASTQ en la función *aligner_indexer*. . . . 49
13 Corrección de los índices de un set de segmentos FASTQ en la función *index_correction* . . . 50
14 Filtrado con el índice corregido y generación del fichero Mpileup *filter_index_to_mpileup*. 50
15 Función auxiliar *fetch_fasta_chunk*. . . . 51
16 Configuración de S3 Select para la distribución de índices. . . . 53
17 Distribución de los índices. . . . 54
18 Configuración de S3 Select para la obtención de rangos de Mpileup. . . . 54
19 Fusión final de archivos. . . . 55
20 Función *timer_start* y *timer_stop*. . . . 55
21 Función *store_size_data*. . . . 56
22 Función *store_dictio*. . . . 56
23 Función *delete_stat*. . . . 56
24 Función *get_stats*. . . . 57
25 Función *load_stats_to_json*. . . . 57
26 Ejemplo de comentario en la implementación nueva . . . 59
27 Ejemplo de comentario en la implementación antigua . . . 59
28 Función *run_pipeline*. . . . 74
29 Función *generate_faidx_from_s3*. . . . 74
30 Función *create_index_chunked*. . . . 75
31 Función *reduce_chunked_indexes*. . . . 76
32 Función *get_fasta_byte_ranges*. . . . 77
33 Función *get_fasta_byte_ranges*. . . . 78
34 Función *generate_fastqgz_index_from_s3* y *generate_idx_from_gzip*. . . . 79
35 Función *get_ranges_from_line_pairs*. . . . 81
36 Clase *stats*. . . . 82
37 Función fetch_fastq_chunk() . . . 84
38 Función run_reducer() . . . 85

1 Introducción

La genómica, una disciplina científica fundamental en el mundo moderno, se dedica al estudio intensivo del genoma completo de un organismo. Esta disciplina abarca el análisis, la interpretación y la aplicación de datos genómicos en diversas áreas, como la medicina, la agricultura y la biotecnología. Su objetivo principal es comprender la estructura, la función, la evolución, el mapeo y la edición del genoma completo de un organismo.

Dado que la genómica se basa en un volumen masivo de información, es necesario almacenar, procesar y analizar estos datos de manera eficiente. Cada genoma puede contener millones de secuencias de ADN, y el análisis de esta magnitud de datos requiere de un poder de procesamiento enorme, una capacidad que los sistemas tradicionales a menudo no pueden proporcionar de manera eficiente.

Aquí es donde adquiere importancia la utilización de sistemas distribuidos, que consisten en conjuntos de computadoras independientes, pero interconectadas, que trabajan en conjunto para lograr un objetivo común. Al dividir el procesamiento de los datos entre múltiples ordenadores o servidores, los sistemas distribuidos permiten manejar grandes volúmenes de información de forma más eficiente. En el ámbito de la genómica, estos sistemas pueden acelerar significativamente la secuenciación del genoma y el análisis de los datos, aumentando la eficiencia y la velocidad del análisis genómico.

En este contexto, el proyecto presentado se enfoca en llevar a cabo una refactorización y optimización de una implementación de herramientas de bioinformática diseñadas para el análisis de genomas en la nube. El objetivo es mejorar significativamente su eficiencia, escalabilidad y aprovechamiento de los recursos en la nube, aprovechando las ventajas de los sistemas distribuidos mencionados anteriormente. Al adaptar y optimizar estas herramientas, se espera acelerar y mejorar el análisis genómico, permitiendo un procesamiento más rápido y eficiente de los datos genómicos en entornos de nube.

1.1. Motivación del proyecto

Debido al interés en los sistemas distribuidos y su potencial, se decidió presentar este proyecto como Trabajo de Fin de Grado con el objetivo de explorar cómo estos sistemas pueden mejorar la eficiencia y escalabilidad de un programa.

Este trabajo se ha desarrollado bajo la supervisión del grupo de investigación CloudLab [7] de la Universidad Rovira y Virgili (URV). Este proyecto representa un caso de uso específico del proyecto europeo *CloudButton* [6]. El objetivo de *CloudButton* es proporcionar una solución escalable para el procesamiento de grandes volúmenes de datos, aprovechando las ventajas de las tecnologías en la nube. Este enfoque simplifica el modelo de programación y su ciclo de vida, permitiendo una mayor eficiencia y productividad en la manipulación de datos.

Otra razón que motivó la propuesta de este proyecto fue la necesidad de optimizar el procesamiento de datos genómicos. Aunque las soluciones existentes generaban los resultados deseados, no eran óptimas en términos de rendimiento y uso de memoria, lo que dificultaba su mantenimiento y escalabilidad. Por tal motivo, se propuso mejorar las deficiencias existentes en el procesamiento de datos, así como simplificar el modelo de programación y hacer el código más legible y fácil de mantener. En esencia, se llevó a cabo una reimplementación

completa del código para lograr estos objetivos.

A lo largo del desarrollo del proyecto, se contó con la colaboración de Paolo Ribeca (PhD) de la UK Health Security Agency (UKHSA) y Lucio Marcello (PhD) del James Hutton Institute (JHI). Su contribución fue esencial para el desarrollo del proyecto, ya que brindaron su amplia experiencia y conocimientos en el campo de la genómica.

1.2. Problemas del proyecto

Inicialmente, el proyecto fue implementado por un grupo de estudiantes. Sin embargo, debido a la falta de coordinación y comunicación entre las diversas partes involucradas, el estado del proyecto al momento de su ingreso era sumamente desorganizado. El repositorio original en el que se almacenaba el código presentaba un caos evidente, con archivos y estructuras dispersas y desordenadas. Además, la ausencia de documentación adecuada y completa complicaba aún más la asimilación del proyecto.

El desafío inicial consistió en comprender la arquitectura y el código existentes. El repositorio original contaba con más de 10.000 líneas de código. Debido a la falta de claridad y estructura en el repositorio, así como a la carencia de documentación detallada, se dedicó el primer mes de su trabajo exclusivamente a entender y ejecutar el código original. Fue necesario realizar un análisis exhaustivo de cada componente, identificar las dependencias y relaciones entre ellos, y establecer una comprensión sólida de la estructura general del sistema.

Aunque se había previsto la implementación de una arquitectura distribuida, se encontraron múltiples fallas de diseño que afectaban la escalabilidad del sistema debido a un uso ineficiente de los recursos ofrecidos por la nube. También se detectaron errores graves en el código del proyecto. Fue necesario identificar y corregir estos errores para garantizar un rendimiento óptimo y confiable.

1.3. Objetivos de nuestro trabajo

El objetivo principal del proyecto consistió en solucionar los problemas mencionados anteriormente para tener una base sobre la cual seguir trabajando en el futuro, dado el potencial del proyecto. Para lograr esto, se establecieron diferentes metas que se dividieron en tres áreas principales:

- **Refactorizado:** El objetivo principal de este proceso era crear un proyecto nuevo prácticamente desde cero, aprovechando de forma progresiva el código existente. Se optó por seguir un enfoque modular, dividiendo el proyecto en componentes más pequeños y cohesivos para facilitar su comprensión y mantenimiento.
- **Optimización del código:** Durante el proceso de refactorización del código, también se ha buscado mejorar tanto la eficiencia como la escalabilidad del sistema. Por un lado, se ha trabajado en la reducción del tiempo de respuesta de las ejecuciones, explorando formas más óptimas de generar la misma salida en menos tiempo. Por otro lado, se ha buscado disminuir el consumo de recursos para mejorar el rendimiento general del programa.

- **Rediseño de la arquitectura:** Por último, se han realizado cambios en el diseño del programa para optimizarlo en términos de una arquitectura distribuida. Estos cambios se han traducido en dos modificaciones arquitectónicas principales:
    - **Mejora en el tratamiento de archivos de entrada:** La implementación anterior generaba una cantidad considerable de datos duplicados, lo cual restringía la escalabilidad del sistema. Por consiguiente, se tomó la decisión de implementar un sistema de procesamiento de archivos que aprovechase de manera más efectiva las herramientas proporcionadas por la nube, posibilitando así una ejecución eficiente en un entorno distribuido.
    - **Eliminación de sincronismos:** Durante una fase del proyecto, se detectó la presencia de sincronismos entre numerosos procesos ejecutándose de manera distribuida, utilizando un servidor único como coordinador. Este enfoque resultaba poco escalable, ya que los distintos procesos del sistema podían quedar bloqueados esperando a que otros finalizaran, lo cual generaba un uso ineficiente de los recursos. El objetivo consistió en eliminar por completo estos sincronismos o bloqueos, permitiendo que todos los procesos se ejecuten de manera asíncrona.

Por tanto, el objetivo principal de este trabajo ha sido solucionar problemas existentes y sentar bases sólidas para futuros desarrollos. Se ha realizado una reimplementación del código, optimizado su rendimiento y rediseñando la arquitectura para adaptarla a un entorno distribuido.

2 Distribución del trabajo

En esta sección se detallará de forma cronológica las tareas llevadas a cabo por ambas partes desde el inicio del curso académico hasta la presentación del proyecto. A lo largo de esta sección, se observarán las actividades que se realizaron de manera independiente y en conjunto.

2.1. Arnau Gabriel Atienza

1. **Comprensión del código:** El primer mes de trabajo consistió en comprender la arquitectura y el código de la implementación antigua del proyecto. Aunque ambos nos ayudamos a asimilar el código, Arnau se centró en la fase de *Map* (explicada en la sección 4.2.3). Esta tarea tuvo una duración aproximada de un mes.

2. **Reimplementación de la fase de *Map*:** Una vez comprendido el funcionamiento previo se reimplementó esta fase del proyecto de manera que funcionase con el mismo comportamiento que la implementación previa. Esta fase tuvo una duración de un mes.

3. **Optimizaciones de la fase de *Map*:** Esta tarea consistió en solventar los problemas de arquitectura de la implementación previa del *Map*. Esta fase tuvo una duración aproximada de 2 meses.

4. **Reimplementación de la fase de *Reduce*:** Esta tarea consistió en volver a implementar la fase de *Reduce* siguiendo los patrones de calidad de código usados en las anteriores fases. Esto tuvo una duración aproximada de un mes.

5. **Implementación de métricas:** Se implementaron diferentes mecanismos de monitorización usando las herramientas desarrolladas por Sara. Esto tuvo una duración de un par de semanas.

6. **Ejecución de experimentos:** Se ejecutaron diferentes experimentos para probar el rendimiento de la nueva implementación. Estos experimentos y sus análisis tuvieron una duración aproximada de un mes y medio.

7. **Memoria del proyecto:** Se elaboró la memoria de este proyecto. El trabajo fue distribuido entre los dos, realizando cada uno los apartados relacionados con los puntos mencionados anteriormente. Se realizaron diversas reuniones para poner en común los avances de ambos y asegurar la calidad de la memoria.

2.2. Sara Lanuza Orna

A continuación se mostrarán las tareas realizadas por la estudiante Sara durante la realización del trabajo final de grado:

1. **Comprensión del código:** Como se comentó en la sección anterior (ver sección 2.1), el primer mes estuvo dedicado a entender la arquitectura del proyecto y el funcionamiento del código. Una vez comprendido el funcionamiento, Sara se centró en la fase de preprocesado, la cual se explicará posteriormente en la sección 4.2.2.

2. **Reimplementación de un particionador de ficheros:** En la fase de preprocesamiento, se llevó a cabo la reimplementación completa de uno de los particionadores de ficheros, de la cual se hablará más adelante en las secciones 5.2 y 6.2.1. Una vez reimplementado el particionador, se adaptaron las fases posteriores al nuevo tipo de datos de salida y se realizaron experimentos con el fin de verificar que la nueva implementación era más eficiente que la anterior y que era completamente funcional para cualquier tamaño de archivos de entrada. La tarea tuvo una duración aproximada de tres meses.

3. **Mejora del código de la fase de preprocesamiento:** Se realizó una refactorización y optimización de la fase de preprocesamiento. Este proceso tuvo una duración aproximada de uno a dos meses.

4. **Implementación de un registro:** Se implementó una estructura para registrar y analizar el código. Esta tarea fue completada en menos de una semana.

5. **Implementación de métricas:** Junto con Arnau, se implementaron las métricas utilizando el sistema de registros implementado previamente en todas las fases, las cuales para ese momento ya habían sido optimizadas y refactorizadas. La tarea fue realizada en un día.

6. **Experimentos:** Se realizaron varias ejecuciones de todo el proyecto con el propósito de poder comprobar si se había producido una mejora en cuanto al rendimiento general comparado con el proyecto antes de ser modificado y localizar las fases que podrían requerir de una optimización. La tarea fue realizada en tres semanas.

7. **Elaboración de la memoria del proyecto:** Se elaboró la memoria del proyecto. El trabajo fue distribuido equitativamente entre las diferentes secciones principales de la memoria. A medida que cada sección era escrita, se leía conjuntamente y se discutían posibles cambios y mejoras que se podrían realizar. Esta tarea se hizo a lo largo de todo el proyecto, en especial los dos últimos meses.

3 Antecedentes

En este apartado se explicarán las diferentes tecnologías, servicios y herramientas necesarios para el desarrollo de este proyecto.

En primer lugar, se explicará en que consiste la ejecución en la nube y los diferentes servicios que ofrece. Entre dichos servicios se encuentran las funciones como servicio y el almacenamiento de objetos, las dos piezas más fundamentales para la ejecución del programa. También se explicará en que consiste el modelo *Map - Reduce*, un modelo de programación distribuida en el que se basa la implementación.

Por último, se explicará en qué consiste la genómica, una disciplina científica que se dedica a analizar y comprender la estructura, función, evolución y regulación de los genomas. Posteriormente, se procederá a explicar en qué consiste el alineamiento de secuencias, una técnica fundamental en la genómica que permite comparar y encontrar similitudes entre las secuencias de ADN o ARN.

3.1. Tecnologías en la nube

Cuando se habla de tecnologías en la nube o *cloud* se habla del acceso a diferentes servicios relacionados con las Tecnologías de la Información a través de internet. Además, en lugar de hacer uso de servidores e infraestructura general propia, se hace a través de proveedores externos.

Los diferentes tipos de servicios que se pueden llegar a ofrecer se suelen clasificar en las siguientes categorías:

- **Software como servicio** (*Software as a Service o SaaS*). En este modelo la empresa ofrece la posibilidad de alojar las aplicaciones en sus propios servidores. Algunos ejemplos son el correo electrónico, software de gestión, herramientas de procesado, etc. Este servicio está orientado a usuarios.
- **Plataforma como servicio** (*Platform as a Service o PaaS*). Este modelo permite a los desarrolladores desarrollar, ejecutar y gestionar sus aplicaciones en la nube sin tener que preocuparse de gestionar la propia infraestructura subyacente.
- **Infraestructura como servicio** (*Infrastructure as a Service o IaaS*). Este modelo busca dar más control a las empresas o desarrolladores a la hora de gestionar su infraestructura. Este modelo se puede considerar un "alquiler" de recursos informáticos, como pueden ser servidores, almacenamiento, redes, etc. Se relega el mantenimiento de infraestructura en sí a la empresa proveedora.
- **Función como servicio** (*Function as a Service o FaaS*). Este modelo es bastante reciente y consiste en ejecutar una porción de código en una máquina externa gestionando mínimamente los recursos. Lo que busca este servicio es la posibilidad de ejecutar código con una alta carga computacional sin necesidad de preocuparse por la estructura donde se está ejecutando. Este servicio resulta vital para el desarrollo de este proyecto.

En los recientes años se ha hecho aparente la necesidad de este tipo de servicios para el análisis de datos. El rápido incremento de datos a procesar ha provocado la búsqueda de alternativas más eficientes para procesar, guardar y analizar dichos datos. Las tecnologías en la nube ofrecen un gran abanico de herramientas según las necesidades de cada proyecto.

La nube da acceso a una infraestructura masiva con una necesidad mínima de mantenimiento por parte del usuario, por lo que se elimina la necesidad de sistemas complejos y costosos de hardware. De esta manera, no es necesario aumentar la infraestructura propia a medida que el volumen de datos incrementa, de manera que se ofrece una alta escalabilidad, siempre y cuando la implementación sea correcta.

Por tanto, las principales ventajas de la computación en la nube pueden resumirse en:

- **Escalabilidad:** Al no tener que gestionar nosotros mismos los recursos, resulta más sencillo escalar tanto vertical (mejores máquinas) como horizontalmente (más máquinas).
- **Flexibilidad:** No importa la potencia de la maquinaria de los desarrolladores, ya que siempre que dispongan de acceso a internet podrán acceder a estos recursos en línea.
- **Seguridad:** La mayoría de grandes proveedores de servicios en la nube tienen capas de seguridad avanzadas.
- **Reducción de costes:** En muchos casos, el uso de estos proveedores reduce los costes relacionados con las tecnologías de la información, ya que a menos que se trate de una empresa muy grande, el manejo de una infraestructura de gran volumen no suele ser rentable para desarrolladores pequeños o medianos.
- **Actualizaciones y mejoras automáticas:** Los proveedores del servicio suelen encargarse de mantener su infraestructura actualizada y tanto a nivel de hardware como software.

El principal problema del procesado en la nube es que muchos de los procesos actuales no están adaptados para este tipo de entornos, por lo que es necesario realizar un proceso extra de adaptación. La calidad de esta adaptación es de vital importancia, ya que si implementación no es correcta, los recursos no se pueden usar de la manera más eficiente posible y la escalabilidad puede verse comprometida. Por tanto, es importante comprender correctamente los recursos que estamos utilizando, los cuales están explicados a continuación.

3.1.1. Proveedores de servicios en la nube

A la hora de escoger un proveedor de servicios en la nube es importante escoger uno confiable y adecuado a las necesidades del proyecto. Otros factores importantes a tener en cuenta incluyen:

- **Experiencia y reputación.** Un proveedor con buena reputación garantiza que el servicio sea de calidad y que no puedan aparecer problemas imprevistos a largo plazo.

- **Medidas de seguridad.** Esto es especialmente importante si los datos que se van a tratar son sensibles.
- **Escalabilidad.** Es importante saber como responderá el sistema ante una carga elevada de trabajo.
- **Costes.** Dependiendo del tipo de proyecto a desarrollar, puede que un proveedor ofrezca mejores cuotas que otros. Puede que para algunos casos de uso pequeños la nube no sea la opción más económica.

(a) Amazon Web Services.

(b) Google Cloud.

Algunos de los principales proveedores de servicios en la nube incluyen [25]:

- Amazon Web Services
- Microsoft Azure
- Google Cloud Platform
- Alibaba Cloud
- IBM Cloud

Para este proyecto se ha usado Amazon Web Services debido a la familiaridad con su ecosistema y a sus altas prestaciones.

3.1.2. *Object Storage*

Object Storage es una arquitectura de almacenaje de datos no estructurados, la cual divide estos datos en unidades conocidas como objetos. El equivalente de estos objetos en un sistema de ficheros tradicional sería un simple fichero. Al contrario que los sistemas de ficheros, los objetos no siguen un sistema jerárquico, sino que cada uno es completamente único. Aun así, por motivos de organización se pueden asignar unos prefijos a los identificadores de dichos objetos para simular un sistema de carpetas. Los diferentes repositorios de datos son llamados *buckets*.

La principal ventaja de *Object Storage* es la escalabilidad prácticamente sin límites y el bajo coste para almacenar altos volúmenes de datos. Además, según el proveedor de este servicio, se suele ofrecer un alto nivel de seguridad contra fallos, por lo que no es necesario preocuparse de errores de hardware que causen posibles perdidas de datos.

Este servicio de tipo Infraestructura como Servicio se ha vuelto muy popular en los últimos años debido a las ventajas en cuanto a escalabilidad, flexibilidad y seguridad. Su escalabilidad permite manejar grandes volúmenes de datos y aumentar el espacio de almacenamiento según sea necesario. Su flexibilidad permite almacenar cualquier tipo de objeto y que este sea accesible desde cualquier lugar. Como los objetos se pueden cifrar y proteger con autenticación y autorización, también es muy seguro.

En cuanto a arquitectura, el almacenamiento de objetos suele hacer uso de arquitecturas distribuidas y escalables que permiten el almacenamiento de los datos en una infraestructura de hardware y software muy escalable. Cuando un objeto se carga en el almacenamiento de objetos, este se replica automáticamente en diversos nodos de almacenamiento para así garantizar su disponibilidad en caso de fallada de hardware. Para equilibrar el tráfico se hace uso de diversas técnicas de balanceo de carga entre dichos nodos para que la lectura y escritura de los datos sea lo más rápida y eficiente posible.

Debido a estos motivos, algunos de los principales usos de *Object Storage* son los siguientes:

- Análisis de datos
- Data Lakes
- Aplicaciones nativas en la nube
- Archivado de datos
- Copias de seguridad
- Machine Learning

Para este proyecto, *Object Storage* será vital para guardar los datos de entrada, los datos intermedios que tengan que ser enviados entre diferentes procesos y para guardar los resultados finales.

3.1.3. Funciones como Servicio y Serverless

A pesar de los muchos beneficios que ofrece la ejecución en la nube, la tarea de configurar un servidor o *cluster* y los diferentes recursos asociados puede resultar complicada para los analistas de datos, quienes simplemente buscan ejecutar sus algoritmos de la manera más rápida y eficiente posible. Es a partir de este problema que nace el concepto de *Serverless*, también conocido como *Function as a Service*.

Function as a Service se trata de un paradigma de computación en la nube que resigna la gestión de los servidores y la infraestructura al proveedor del servicio, de manera que el usuario solo necesita centrarse en el desarrollo del código y no en la gestión de recursos. El proveedor se encargará de escalar los recursos de manera automática según la demanda, lo cual nos permite escalar de manera el procesado de nuestros datos a través del modelo *map-reduce*, el cual se explicará más adelante. Los usuarios solo pagan el coste de computación concreto que ha hecho falta para ejecutar su código.

Este modelo se basa en que una función (un segmento de código) se trata de una unidad de ejecución autónoma e independiente. Cada función se ejecutará en un entorno completamente aislado de todas las otras funciones, de manera que siempre se obtendrá el mismo resultado.

Figura 2: Símbolo de AWS Lambda, la implementación de AWS de las Funciones como Servicio.

La principal ventaja de este servicio es el desacople la necesidad de mantenimiento de la infraestructura de la ejecución del código en sí. Se puede escalar la computación de manera horizontal con facilidad debido a que las funciones se ejecutan de manera independiente, en su propio sistema aislado, el cual suele ser borrado al acabar la ejecución.

Debido a que el sistema donde se ha ejecutado la función es borrado al finalizar (o poco después de acabar), los datos para la tarea deben ser adquiridos de un repositorio externo, el cual suele ser *Object Storage*. Los datos de entrada se descargan de *Object Storage* y en caso de que los resultados tengan un tamaño considerable, son también subidos ahí. Esto puede llegar a suponer un problema, ya el proceso de descarga y subida de datos puede impactar el coste de la función cuando el volumen de datos es alto, ya que la transferencia no es inmediata y, aunque la función no está realizando una computación como tal, ese tiempo de transferencia cuenta igual a la hora de determinar el coste.

Para este proyecto se hará uso de AWS Lambda, la implementación de AWS de *Function as a Service*. A menudo se hablará de "Funciones Lambda", haciendo referencia a una o varias funciones que se están ejecutando en este servicio.

3.2. Modelo Map - Reduce

El modelo Map - Reduce permite procesar grandes entradas de datos de manera paralela, aprovechando las ventajas de escalabilidad de las funciones *serverless*. Cuando se haga mención a *mapear* funciones a lo largo del proyecto, se referirá al acto de lanzar múltiples funciones de manera paralela.

El mapeo consiste en llamar a un cierto número de funciones y asignar una porción de los datos de entrada a esta. La principal consideración a tener en esta fase es el balance de carga entre todas las funciones, es decir, intentar que todas procesen la misma cantidad de datos para intentar que todas tengan una duración similar. A medida que las funciones ejecutan el proceso pertinente sobre los datos, van subiendo los resultados a *Object Storage* y finalizando.

Cuando los datos de entrada consisten en múltiples ficheros distintos, resulta sencillo distribuir un fichero a cada función, pero no siempre es así. En la mayoría de casos, solo hay un fichero de gran tamaño, por lo que será necesario hacer un paso previo de preprocesamiento para preparar dicho fichero para su distribución.

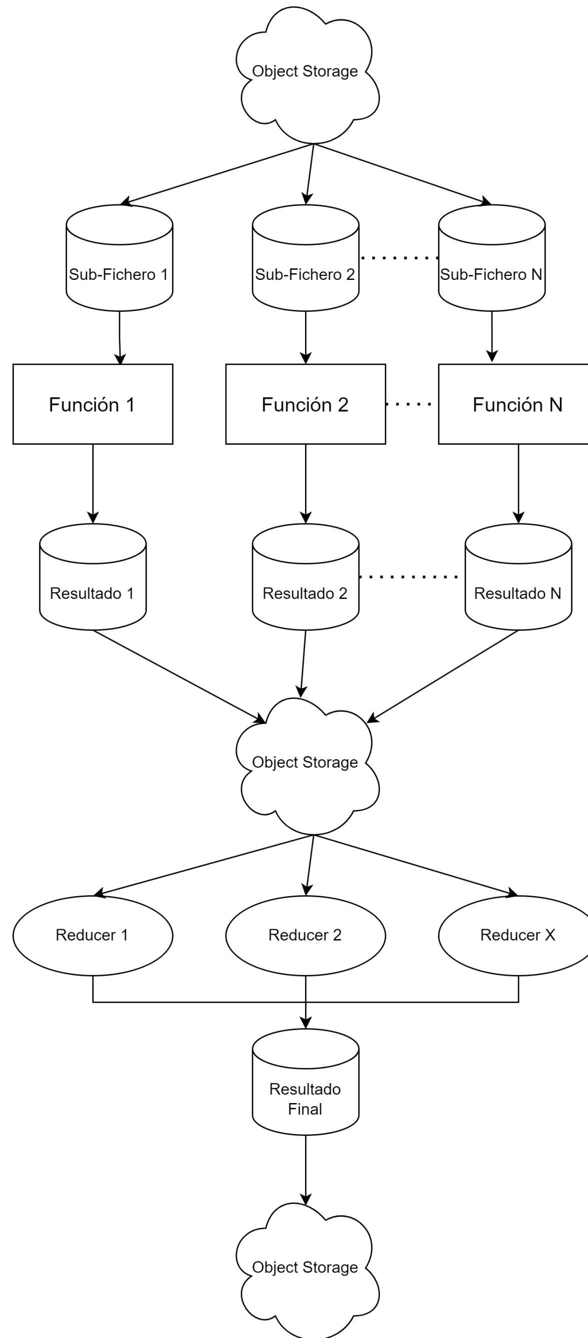

Figura 3: Ejemplo de modelo Map-Reduce.

Una vez todas las funciones de la fase de Map han terminado, se ejecuta la fase de *reduce*, en la cual uno o varios procesos llamados Reductores o *Reducers* obtienen los resultados generados en la fase de Map y los juntan en un fichero (o varios en algunos casos). En algunos casos esta fase puede ser simplemente concatenar dichos resultados o ejecutar una operación simple, pero como es en el caso de este proyecto, puede ser necesario un procesado complejo, llegando esta fase a tener una duración similar a la de Map.

En el ejemplo de la figura 3, tenemos N sub-ficheros que actuarán a modo de datos de entrada, y que se mapearán a N funciones, es decir, un fichero por función. Cada función generará 1 fichero de resultado, el cual se subirá a *Object Storage*. A continuación se lanzarán X *Reducers*,

por ejemplo se podrían asignar 5 ficheros a cada *reducer*, de manera que tendríamos N/5 *Reducers*.

Si solo se usase un único *Reducer*, la función sola generaría el fichero final y lo subiría a *Object Storage*, pero en casos como este donde hay varios *Reducers* es necesario aplicar algún otro método como *Multipart-Upload*, un mecanismo que permite que cada función suba una parte del fichero final y el propio proveedor de *Object Storage* se encarga de concatenar dichas partes en un solo fichero.

Como se puede observar, el modelo *Map-Reduce* permite escalar de manera masiva el procesamiento de los datos a través del paralelismo de funciones, pero es necesario añadir algunas fases adicionales, como el Preprocesado de los datos de entrada y la Reducción. Estas dos fases son tan importantes como el Map en sí, ya que, de lo contrario, el rendimiento general puede verse altamente comprometido.

3.2.1. Lithops Middleware

Lithops [21] se trata de una herramienta que actúa a modo de Middleware para el uso de Funciones como Servicio y el modelo *Map - Reduce*. Lithops proporciona una interfaz unificada que permite desarrollar aplicaciones sin tener que preocuparse por las diferencias entre los distintos proveedores.

Algunas de las características principales son:

1. *Cross-platform*: Múltiples proveedores de servicios en la nube soportados, por lo que el mismo código se puede ejecutar a través de diferentes proveedores (AWS Lambda, Google Cloud Functions, IBM Cloud Functions ...).
2. Fácil de usar y aprender.
3. Habilidad de detectar e informar de errores de manera local.
4. Escalado dinámico de los datos.

Figura 4: Logo oficial de Lithops.

Esta herramienta se compone de dos partes principales: el entorno de ejecución y el sistema de almacenamiento. El entorno de ejecución debe ser definido con una imagen *Docker*[1], para así poder inicializar los contenedores donde se ejecutarán las funciones. El sistema de almacenamiento será el *Object Storage* donde se almacenarán y recuperarán los datos de las funciones, lo que permite acceder a los datos de manera eficiente.

[1]Herramienta de tipo Plataforma como Servicio que permite el desarrollo de imágenes para la virtualización de Sistemas Operativos conocidos como contenedores.

Esta herramienta será utilizada en forma de librería para ejecutar todas las funciones que deban ser ejecutadas en la nube. El lenguaje utilizado para esta librería y, por ende, para este proyecto, es *Python*. Este lenguaje actuará a modo de coordinador entre las diferentes fases que componen el proyecto, haciendo uso de estructuras de datos óptimas o relegando las tareas computacionales intensas a otros lenguajes de scripting.

3.3. Genómica y el alineamiento de secuencias

La genómica, un campo especializado de la biología, se encarga de investigar la totalidad del material genético, también conocido como ADN, presente en un organismo. Este conjunto completo de genes y demás elementos genéticos conforman lo que denominamos como el genoma de un organismo. La tarea principal de la genómica incluye identificar y caracterizar todos los genes y componentes funcionales del genoma, comprendiendo la funcionalidad y estructura integral de estos.

El estudio de la genómica no sólo se limita a la identificación y caracterización de los genes, sino que también busca entender cómo estos genes interactúan entre sí y con el entorno. Cada interacción tiene un papel crucial, pues contribuyen a la manifestación de diferentes rasgos y funciones en un organismo. En otras palabras, la genómica se encarga de descifrar el código genético para entender cómo las interacciones genéticas y ambientales dan lugar a la diversidad de vida que observamos.

Este campo de estudio no sólo proporciona una comprensión más profunda de la biología fundamental de un organismo, sino que también tiene aplicaciones significativas en áreas como la medicina personalizada, la biotecnología y la conservación de especies. A través de la genómica, los científicos están abriendo nuevos caminos para tratar enfermedades, mejorar los cultivos y proteger la biodiversidad.

En este contexto, un método muy utilizado en genómica es el alineamiento de secuencias, que consiste en comparar una secuencia (un fragmento de ADN, una lectura) con un genoma de referencia, con el objetivo de encontrar todas las regiones en el genoma de referencia que tengan una secuencia similar. Esta información es valiosa, ya que permite identificar las partes del genoma que están activas bajo diferentes condiciones biológicas.

El alineamiento de secuencias se realiza en cinco etapas principales:

1. Inicialmente, se realiza un alineamiento continuo de la secuencia de interés, referida como la lectura, con el genoma de referencia. Si existe información anotada, la lectura se alinea también con los transcritos anotados.

2. En la segunda etapa, la lectura se alinea de manera discontinua con el genoma de referencia. Esto se debe al proceso biológico conocido como empalme, en el cual una secuencia continua en el transcripto[2] puede derivarse de bloques no contiguos en el genoma, llamados exones. Después de eliminar las secuencias intercaladas, conocidas como intrones, estos exones son transcritos conjuntamente.

[2]Un transcripto consiste en una copia de una parte de ADN, hecha en una molécula de ARN.

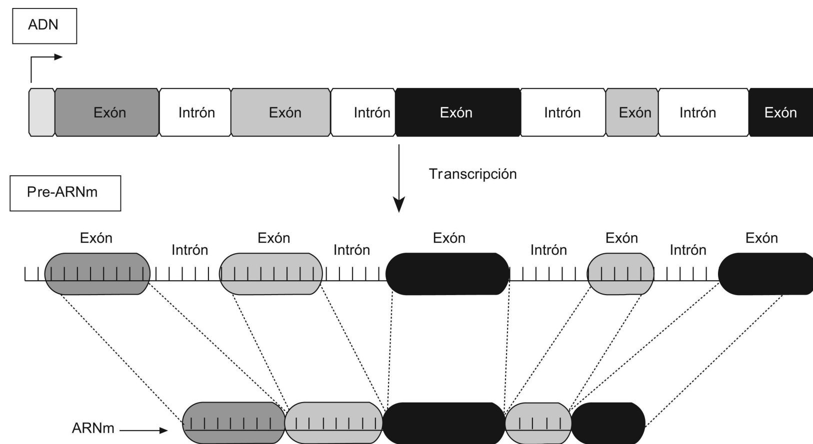

Figura 5: Transcripción del ADN [12].

3. La tercera etapa implica recopilar todos los intrones detectados en la etapa anterior y los presentes en la anotación. A partir de esta recopilación se genera una anotación ampliada con la que se alinea la lectura.
4. En la cuarta etapa, todas las alineaciones generadas en las etapas anteriores se recopilan y fusionan para eliminar redundancias.
5. En la etapa final, las alineaciones se clasifican en función de su calidad. Esta clasificación se basa en la cantidad de diferencias en comparación con la secuencia de referencia y en la similitud con otras secuencias. Cuantas más diferencias haya y menos única sea la secuencia, menor será la puntuación asignada.

3.4. Formato de ficheros de entrada

En este proyecto se emplean dos formatos de ficheros de entrada: FASTA[13] y FASTQ[14]. Estos formatos son extensivamente usados en el campo de la bioinformática para el almacenamiento y análisis de datos de secuenciación de ADN[1], ARN[3] y proteínas. A continuación, se proporcionará una explicación de ambos tipos de ficheros de entrada utilizados en el proyecto.

3.4.1. FASTA

Iniciando con el formato FASTA, este se caracteriza por almacenar secuencias de nucleótidos o aminoácidos. Se trata de un formato simple y fácil de interpretar.

```
>sp|A4GXA9|EME2_HUMAN EME2 OS=Homo sapiens OX=9606 GN=EME2 PE=1 SV=3
MARVGPGRAGVSCQGRGRGRGGSGQRRPPTWEISDSDAEDSAGS
EAAARARDPAGERRAAAEALRLLRPEQVLKRLAVCVDTAILEDA
GADVLMEALEALGCECRIEPQRPARSLRWTRASPDPCPRSLPPE
VWAAGEQELLLLLEPEEFLQGVATLTQISGPTHWVPWISPETTA
...
```

Cuadro 1: Ejemplo de una secuencia en formato FASTA, que incluye una etiqueta de identificación, una descripción y la secuencia de nucleótidos de ADN correspondiente.

Cada registro comienza con una línea de encabezado que empieza con el símbolo “>” seguido de una identificación única para la secuencia y una descripción opcional separada por un espacio en blanco. La secuencia de nucleótidos o aminoácidos se escribe en una o varias líneas debajo de la línea de encabezado.

3.4.2. FASTQ

Los ficheros con formato FASTQ almacenan secuencias de nucleótidos. Sin embargo, a diferencia del formato FASTA, incorporan información de calidad para cada base en la secuencia.

```
@SIM:1:FCX:1:15:6329:1045 1:N:0:2
TCGCACTCAACGCCCTGCATATGACAAGACAGAATC
+
<>;##=><9=AAAAAAAAAA9#:<#<;«<????#=
...
```

Cuadro 2: Ejemplo de un fichero FASTQ.

Cada registro consta de cuatro líneas: la primera comienza con el símbolo “@”, seguida de una identificación única para la secuencia; la segunda línea contiene la secuencia de nucleótidos; la tercera línea comienza con el símbolo “+”, seguida de información adicional opcional; y la cuarta línea contiene una cadena de calidad para cada base en la secuencia.

4 Análisis del sistema previo

En esta sección se llevará a cabo un análisis exhaustivo de las secciones y elementos más significativos del programa. En este análisis, se busca revisar el código fuente original, así como los archivos de entrada y salida utilizados. El objetivo es comprender el funcionamiento del programa e identificar posibles problemas de rendimiento que puedan corregirse durante la reimplementación, mejorando así su funcionalidad.

4.1. Análisis de código

A lo largo del análisis de las 10.000 líneas de código existentes, se han identificado diversos problemas estructurales. Debido a estos problemas, resulta imposible explicar de manera clara cómo estaba estructurado el repositorio original, lo que evidencia la necesidad de realizar un cambio total en dicha estructura.

Uno de ellos es la existencia de archivos de código de longitud excesiva, algunos llegando a tener casi 1.000 líneas de código. Esto dificulta la legibilidad y comprensión del contenido, lo que puede ocasionar problemas al realizar cambios en ellos.

Otro problema detectado es la presencia de archivos duplicados motivados por el versionado, como se muestra a continuación:

1. varcall_lithops_demo_v5.py
2. varcall_lithops_demo_v6.py
3. varcall_lithops_demo_v7.py
4. varcall_args.tsv
5. varcall_args2.tsv
6. varcall_args3.tsv

Este tipo problema se encuentra bastante repetido a lo largo del repositorio original. Este tipo de versionado dificulta mucho el mantenimiento y actualización del código.

También existen problemas de estandarización del código, lo cual dificulta el trabajo si participan múltiples programadores en un mismo proyecto. Esto lleva a que hayan “estilos” de programación variantes a lo largo de todo el programa.

Por último, es importante destacar que la mayoría de los archivos importantes se encuentran dispersos en una misma carpeta junto con archivos que ni siquiera se utilizan, lo que resulta en una falta de modularidad que agrava los problemas mencionados anteriormente.

4.2. Análisis de arquitectura

En este apartado se van a analizar las diferentes fases por las que pasa el código y se van a desglosar dichas fases en diferentes partes para así encontrar puntos de mejora en la ejecución.

4.2.1. *Fases principales*

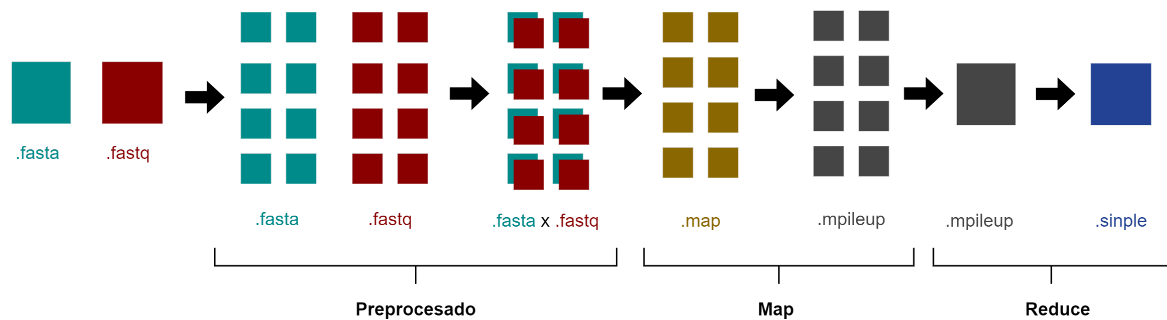

Figura 6: Flujo de datos.

El proyecto previo se puede dividir en 3 fases importantes tal y como se muestra en la figura 6. Estas fases siguen el modelo *map - reduce*:

1. **Preprocesado:** Se particionan los dos tipos de datos de entrada para crear diferentes segmentos. Estos segmentos serán emparejados con cada segmento del otro tipo. Cada pareja de segmentos será asignada a los procesos de la siguiente fase.

2. **Map:** Se procesan las parejas de segmentos para generar los diferentes alineamientos. Cada pareja de segmentos de entrada genera un fichero de salida.

3. **Reduce:** Se procesan los datos generados en la anterior fase para así fusionarlos en un único fichero de salida.

4.2.2. *Preprocesamiento*

Durante el procedimiento de preprocesamiento, se emplean una serie de procesos con el objetivo de preparar los ficheros de entrada para su uso en un sistema distribuido. Esta fase consta de dos secciones principales:

1. **Particionado de ficheros de entrada:** Este proceso se encarga de adaptar los ficheros de entrada para facilitar su procesamiento posterior.

2. **Generación de iterdata:** En esta etapa, se combinan los fragmentos de los ficheros de entrada generados en el proceso anterior. El objetivo es crear conjuntos de datos que puedan ser utilizados para el procesamiento y análisis posterior, y que permitan la alineación de secuencias entre ambos ficheros.

4.2.2.1. *Particionado de ficheros FASTA*

El proceso de particionado consiste en fragmentar un fichero FASTA en secciones más pequeñas que se almacenan en la nube para su posterior uso en el proyecto. Para ello, el particionador divide el fichero original en fragmentos de un tamaño determinado por el usuario. Como el fichero FASTA puede contener múltiples secuencias, cada una con

una cabecera y una base de secuencia, es necesario ajustar los fragmentos resultantes para garantizar la integridad de las secuencias.

El ajuste implica la búsqueda simultánea de las diferentes secuencias en cada fragmento para detectar si la primera y/o última secuencia se encuentra dividida entre la cabecera y la base. Dependiendo del caso y si se trata de la primera o la última secuencia, se traslada al fragmento anterior o siguiente o se deja en el fragmento actual. De esta forma, se asegura que cada fragmento contenga secuencias completas y consecutivas y que no se pierda información esencial.

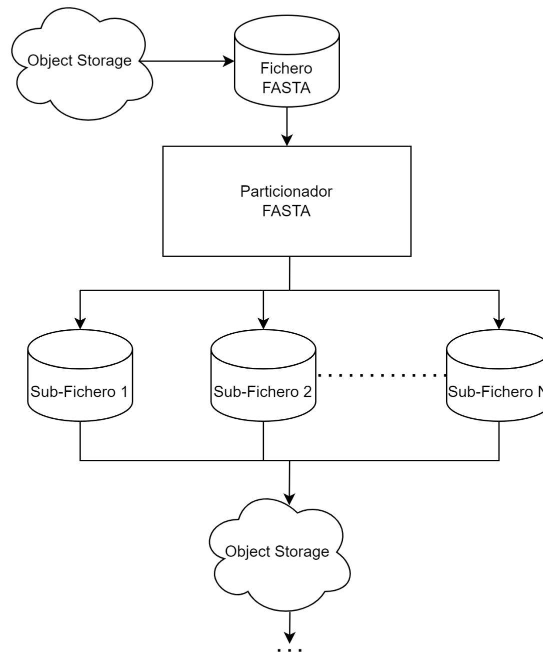

Figura 7: Particionador de ficheros FASTA.

4.2.2.2. Particionado de ficheros FASTQ

En el caso del particionador FASTQ, se sigue un proceso distinto al del particionador FASTA. En este caso, se utiliza la herramienta *gztool*[18] para generar índices con formato *.gz* a partir de un fichero comprimido en formato FASTQ.

Los índices generados se utilizan en la fase de alineamiento, donde se divide el fichero original en fragmentos, cada uno conteniendo un número determinado de secuencias y representando una fracción del fichero original. Durante la fase de alineamiento, estos fragmentos se procesan simultáneamente.

4.2.2.3. Generación de la iterdata

La iterdata consiste en datos previamente preprocesados con el objetivo de ser empleados en un sistema distribuido. En el contexto de este proyecto, Lithops es responsable de recolectar estos datos preprocesados y distribuirlos a las diferentes funciones que se lanzarán en la nube.

La fase de generación de iterdata se realiza después de la etapa de partición de archivos.

Durante esta fase, se asocia el identificador de cada fragmento del archivo FASTQ con el identificador correspondiente de un fragmento del archivo FASTA generado previamente.

Este proceso se encarga de preparar los datos de manera que sean adecuados para la fase de alineación de secuencias. Durante la fase de alineación, cada secuencia presente en el archivo FASTQ se alinea con una secuencia correspondiente en el archivo FASTA. Dado que se ejecutará una función por cada pareja FASTA-FASTQ, se desplegará en la nube un total de funciones igual al número de segmentos FASTA multiplicado por el número de segmentos FASTQ.

En resumen, el objetivo de esta fase es preparar los datos de entrada para un procesamiento distribuido y simultáneo en la fase de alineación de secuencias, donde cada función desplegada en la nube se encargará de una pareja FASTA-FASTQ.

4.2.3. Map

Una vez se han generado los datos a consumir, se puede proceder a la ejecución de las funciones Lambda. En esta implementación previa, toda la tarea de alineación se lleva a cabo de manera secuencial en una única función (ejecutada múltiples veces de manera paralela).

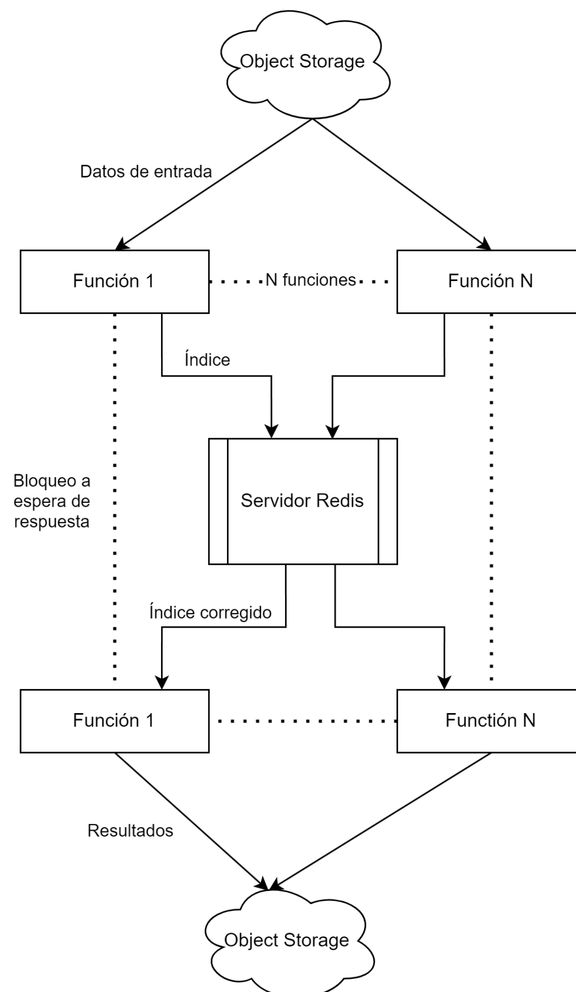

Figura 8: Esquema de la transferencia de datos en la anterior fase de Map.

En la figura 8 se muestran las diferentes transferencias de datos de esta fase. Como se puede observar, hay algunos datos que hacen uso de la base de datos de *Object Storage*, mientras que hay otros datos que necesitan ser procesados en conjunto que son enviados a un servidor centralizado de Redis. Esto resulta problemático, ya que este tipo de servidores suelen estar optimizados para la transferencia y almacenamiento de datos, y no para su procesado.

Cada función de estas puede ser dividida en las siguientes fases:

1. **Obtención de datos**: El segmento FASTA se obtiene de *Object Storage*, mientras que el segmento FASTQ se puede obtener tanto de *Object Storage* como de SRA (*Sequence Read Archive* [24]).
2. **Generación del índice GEM**: Para que algunos de los procesos posteriores puedan ejecutarse de manera óptima, es necesario que se genere un fichero *GEM* [20], el cual indexa el segmento FASTA.
3. **Generar alineamiento**: Usando la misma librería mencionada en el anterior paso, se procede a alinear el segmento FASTQ con el segmento FASTA. Esto genera dos resultados, un fichero *MAP*, el cual contiene los resultados del alineamiento en sí, y un fichero de índice, el cual es necesario poner en común con el resto de funciones lanzadas que hagan referencia al mismo segmento FASTQ. Para realizar esta coordinación, la función envía este fichero a un servidor *Redis* [19] y procede a bloquearse hasta que devuelva el índice "corregido" de este mismo servidor. A nivel de cómputo, esta fase es la más costosa.
4. **Corrección del índice**: Cada vez que el servidor *Redis* tiene disponible todos los índices referentes a un segmento FASTQ, ejecuta un script de corrección y se envía el índice corregido de vuelta a esas funciones. Este índice corregido es común para todas las funciones con ese segmento FASTQ.
5. **Filtrado**: A medida que las funciones van recibiendo los índices corregidos, estas se desbloquean y proceden con el siguiente paso. Este paso consiste en aplicar el índice corregido al fichero *MAP* con otro script y así obtener un *MAP* corregido.
6. **Generación del fichero Mpileup (ver 4.2.3.1)**: Se convierte el fichero *MAP* a formato *.mpileup*, el cual muestra en un formato de texto los resultados del alineamiento.
7. **Conversión a CSV/Parquet**: Para que el fichero pueda ser procesado con la herramienta *S3 Select* en la fase de Reduce, es necesario convertir el fichero *Mpileup* a uno de estos dos formatos. Finalmente, se suben estos ficheros a Object Storage y la función termina.

4.2.3.1. Mpileup

La extensión Mpileup es creada por la librería SAMTools [22] y es altamente utilizada para el análisis de datos de secuenciación de alto rendimiento.

Este formato normalizado se trata realmente de una tabla con una serie de columnas establecidas, separadas por tabulaciones y acabadas en un salto de línea. Cada línea representa

las lecturas en una posición del genoma concreta. El resto de columnas representan valores numéricos codificados como caracteres ASCII[4].

El formato Mpileup es capaz de informar de la presencia de variantes en una posición determinada. Es por ello que es altamente usado en la bioinformática para identificar variantes y caracterización de la variabilidad de un genoma determinado.

Las diferentes columnas contienen los siguientes datos:

1. Nombre del cromosoma
2. Posición dentro de la secuencia
3. Referencia base de esta posición
4. Número de alineaciones leídas en esta posición
5. Resultados de la lectura
6. Calidad de la lectura

```
chr1 10001 T 10 ,,,,,,,,,, ??????????
chr1 10002 A 11 ,,,,,,,,,,, ???????????
chr1 10003 A 11 ,,,,,,,,,,, ???????????
chr1 10004 C 16 ,,,,,,,,,,,,.... ????????????????
chr1 10005 C 17 ,,,,,,,,,,,,,+1C.... ?????????????????
...
```

Cuadro 3: Ejemplo de 5 filas de un fichero mpileup.

4.2.4. *Reduce*

Una vez han acabado todas las funciones del Map, han quedado como resultado un mismo número de ficheros *.mpileup*. Aunque durante la fase de Map ya se ha hecho una corrección de los índices para descartar algunos resultados, es necesario aplicar una operación de unión de las líneas con el mismo índice (segunda columna del fichero *.mpileup*). En la figura 9 se observan las diferentes fases, la generación de los archivos y las conexiones con *Object Storage*.

Las fases que se pueden observar en la figura 9 consisten en lo siguiente:

1. **Creación de claves**: Es necesario crear una serie de "claves" que representarán en Object Storage los resultados finales. Estas claves se usarán para crear *MultiPartUploads*, un mecanismo que permite que diferentes funciones suban diferentes segmentos de un fichero, y dejar que sea el propio proveedor de Object Storage el que concatene esos segmentos en un único fichero.

2. **Distribución de índices**: Debido al gran volumen de índices a procesar, resulta óptimo dividir esta tarea de unión entre diferentes funciones, dejando que cada una procese un número aproximado de índices. Para eso se lanzan diversas funciones que se encargan de dividir los índices entre diversos *reducers*, asignando como mucho 20.000.000 índices a cada uno.

3. **Reducción**: Una vez se dispone de los índices que cada función o *reducer* debe procesar, se lanzan en un map, es decir, de manera paralela. Cada función descargará las filas concretas con los índices asignados a través de una operación S3 Select, de manera que se descargan solo los datos necesarios y no todos los ficheros enteros (esta operación solo está disponible en el Cloud de Amazon). Al final de esta fase resulta un número de ficheros equivalente al número de segmentos FASTA.

4. **Unión final**: En esta fase simplemente se concatenan los ficheros generados en la anterior fase en un único fichero final utilizando una *MultiPartUpload*. Esta fase se puede considerar opcional, ya que no se aplica ninguna operación o script, simplemente se concatenan los resultados anteriores uno detrás de otro.

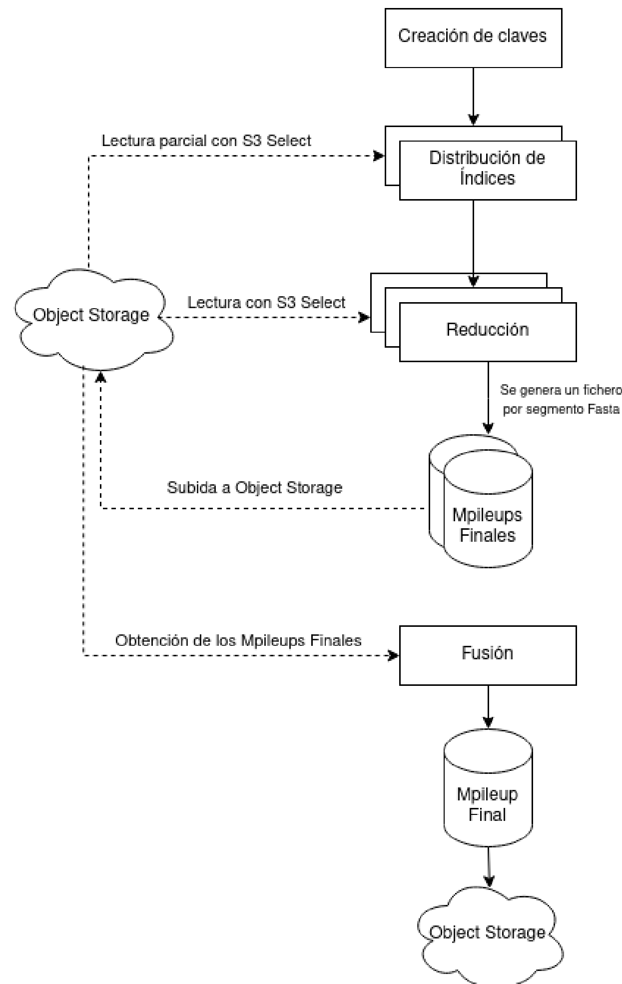

Figura 9: Esquema de la transferencia de datos en la anterior fase de Reduce.

4.3. Resumen

Como se ha mencionado en este apartado, el proyecto anterior presenta complejidad a nivel de arquitectura distribuida. Además, el código se encuentra mal estructurado y desorganizado, lo que dificulta su comprensión. Por lo tanto, el objetivo principal claro será mejorar la calidad de este código.

Además de eso, durante el análisis de las fases del código, se han identificado diversos puntos de optimización a nivel de arquitectura distribuida. Estas optimizaciones buscarán aprovechar de manera más eficiente los recursos disponibles, al mismo tiempo que se eliminan imperfecciones que obstaculizan la escalabilidad del programa. En el siguiente apartado, se explicarán en detalle estas optimizaciones.

5 Propuesta de la solución

En este apartado se va a hacer la propuesta teórica de mejora del proyecto previo. En primer lugar, se explicarán la visión de refactorizado implementada y a continuación las diferentes optimizaciones realizadas respecto al funcionamiento del proyecto.

5.1. Refactorización del código

El código original formaba parte de un repositorio en el cual habían trabajado muchas personas. La falta de coordinación y falta de experiencia llevó a un repositorio completamente desestructurado:

- A lo largo del repositorio original se pueden encontrar múltiples archivos versionados a través del nombre, es decir, ficheros como *codigov1, codigov2, codigov3...*. Esto es completamente innecesario, ya que se usa un repositorio gestionado con la herramienta *Git*, más concretamente el servicio *GitHub*. Este sistema permite producir cambios, pero manteniendo las versiones anteriores del código aparte, sin necesitar mantener todas las versiones de un archivo en el mismo proyecto.
- Ficheros y funciones de altas longitudes. Esto provoca que el código sea más difícil de mantener y comprender por otros usuarios que intenten trabajar sobre un proyecto.
- Mala documentación. El código existente carece de comentarios y explicaciones de los diferentes métodos y secciones de código.

El objetivo de esta parte del proyecto es reestructurar, limpiar y documentar el código para que cumpla con los estándares de calidad del código. Se proponen las siguientes mejoras:

- Eliminación de código antiguo, el cual quedará guardado en *commits* anteriores.
- División de los ficheros en paquetes o módulos con tareas o fases diferentes.
- División de las funciones en funciones auxiliares más pequeñas.
- Documentación del código extensiva que permita entender claramente todas las fases por las que pasa el proyecto.

5.2. Particionado on the fly para fichero FASTA

Como se ha expuesto en la sección anterior 4.2.2.1 acerca del particionado de archivos FASTA, la técnica previamente mencionada implicaba la división del archivo en fragmentos más pequeños, lo cual ocasionaba la duplicación de datos y dificultaba la capacidad de ampliación del conjunto de datos. Con el propósito de resolver este problema y mejorar la eficiencia en la gestión de conjuntos de datos extensos, se ha propuesto una nueva técnica de particionado que se basa en la generación de un archivo índice en lugar de múltiples archivos particionados.

Con la técnica anterior, se fragmentaba el archivo FASTA en secciones más reducidas, lo que generaba datos duplicados en la nube y obstaculizaba la escalabilidad de los datos. Además, cada vez que era necesario modificar el tamaño de los fragmentos, se debía crear nuevos archivos particionados.

En contraste, con la nueva técnica de particionado se genera un archivo índice que almacena información acerca de la ubicación del primer byte de cada encabezado y su base correspondiente. Esto elimina la necesidad de generar y mantener múltiples archivos particionados en la nube, permitiendo un uso más eficiente del almacenamiento.

La técnica propuesta se basa en el paquete *pyfaidx* [23], una herramienta de indexación para archivos FASTA. A diferencia de la herramienta original, la técnica propuesta ejecuta el proceso de indexación de manera paralela, lo que mejora significativamente el rendimiento y la velocidad del proceso.

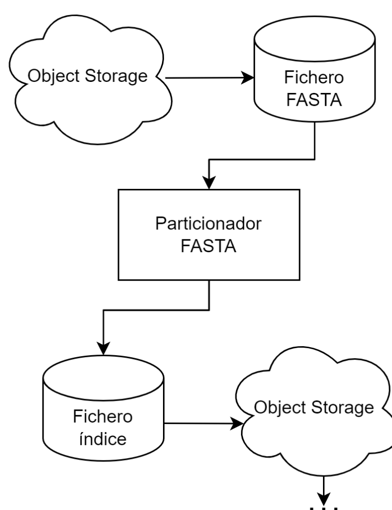

Figura 10: Particionador propuesto.

En conclusión, la nueva implementación del particionado mejora la eficiencia y escalabilidad en el manejo de grandes conjuntos de datos al eliminar la duplicación de datos y facilitar la gestión y el almacenamiento de la información.

5.3. Eliminación de bloqueos

En la anterior implementación de map, las funciones se bloquean cuando una vez generado el índice, ya que necesitan enviar dicho índice a un servidor de Redis y luego esperar la respuesta de este (tal y como se vio en la figura 8). Esto plantea los siguientes problemas:

- Debido a que el coste final de la ejecución se aplica sobre el tiempo total de ejecución de cada función, ese tiempo que la función se pasa bloqueada tiene un coste asociado. Es decir, se está aplicando el mismo coste por segundo de espera que si la función estuviese computando. Este problema puede agravarse si se dan alguno de los errores mencionados a continuación.
- En casos de datos masivos, puede ser necesario lanzar miles de funciones. Sin embargo, esto puede provocar un colapso del servidor de *Redis*, encargado de manejar las

solicitudes de transferencia de archivos y el procesamiento de los mismos para generar el índice corregido. Si el servidor se bloquea, las funciones quedarán bloqueadas hasta que se alcance el tiempo de espera *timeout*.

- Incluso en el caso de que el servidor no colapse, puede darse el caso de que no todas las funciones se lancen simultáneamente, o incluso que unas tarden mucho más que otras. Esto implica que las funciones de un mismo grupo tendrán siempre que esperar a que la última función de dicho grupo acabe.
- Es necesario tener un servidor dedicado a esta tarea, de manera que se pierde la idea de desacoplamiento entre funciones.

Debido a esto, es evidente la necesidad de encontrar una solución alternativa que permita una ejecución completamente paralela y sin comunicación directa entre funciones o procesos.

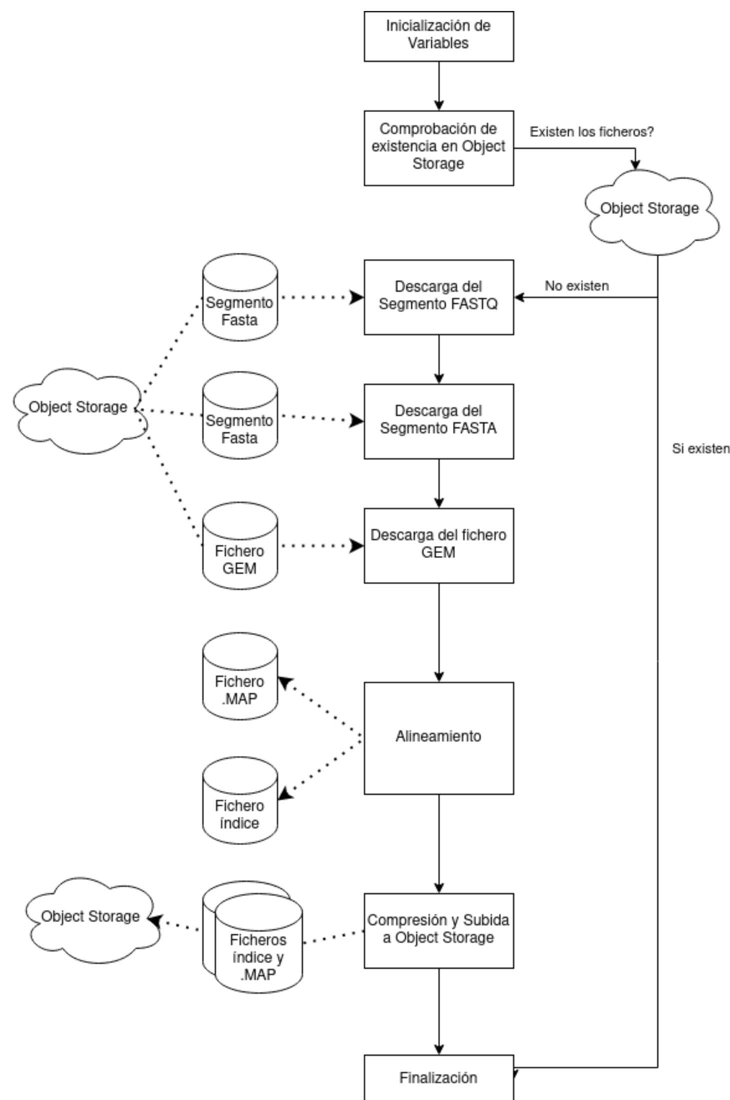

Figura 11: Pasos de la primera fase de alineamiento.

Se ha planteado una idea alternativa que consiste en que, una vez las funciones lleguen al segmento donde previamente quedaban bloqueadas, en lugar de subir el índice a *Redis* y

esperar la respuesta, suban los datos generados a Object Storage y finalicen su ejecución. Como han finalizado su ejecución, ya no hay un coste extra aplicado a las que acaban antes, de manera que solo habrá un coste por el tiempo que la función está realmente computando. De esta manera, se lanzaría una tanda de funciones paralelas que realizarían los pasos detallados en la figura 11.

Figura 12: Pasos de la corrección del índice.

Una vez las funciones han finalizado, hay que realizar la corrección de los índices, los cuales se encuentran en Object Storage. Para seguir con una arquitectura *Serverless*, se ha decidido que en lugar de realizarse la corrección en un servidor corriente, se haga también con funciones *Serverless*. Se lanzará una función por cada índice corregido necesario, es decir, un número de funciones igual al número de segmentos FASTQ. A medida que cada función genere ese índice corregido, como las funciones han terminado y no puede enviarles el resultado directamente, en su lugar se sube dicho índice a Object Storage. Las funciones lanzadas ejecutarían los pasos mostrados en la figura 12.

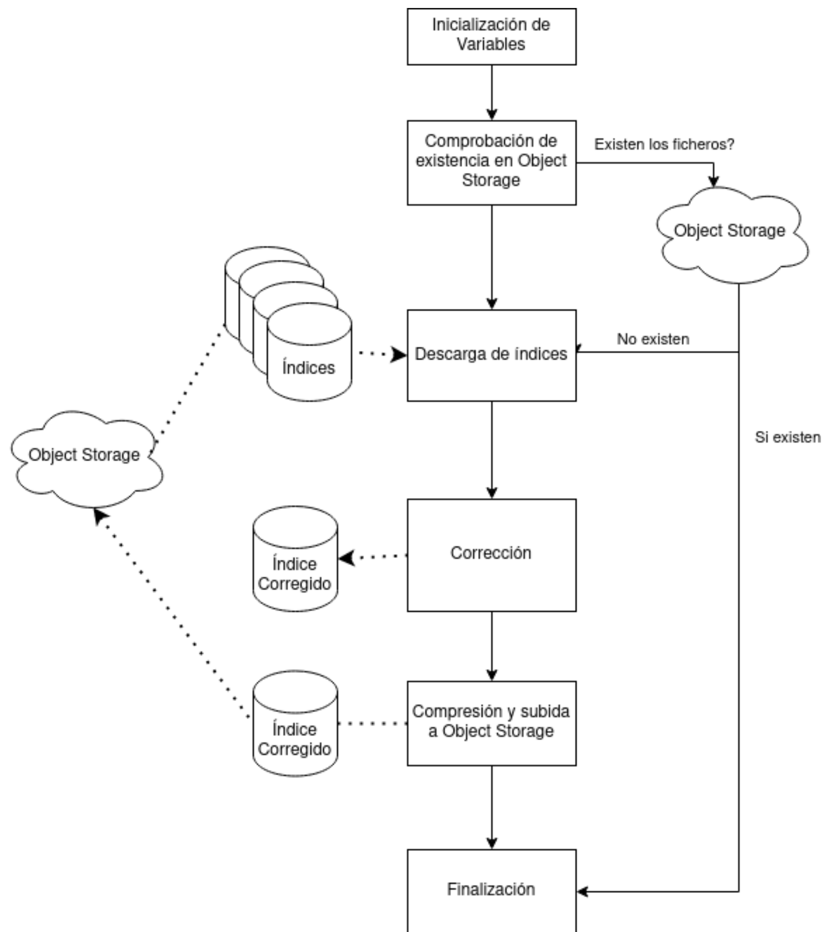

Figura 13: Pasos de la finalización de la fase de *Map*.

Por último, será necesario volver a lanzar un número de funciones igual que en la primera fase, pero en lugar de empezar otra vez de cero, se descargarán los resultados que se generaron, el índice corregido, y se ejecutará la porción de código que había a continuación del bloqueo. Una vez completado esto, la ejecución restante continuará de la misma manera que antes, siguiendo los pasos detallados en el diagrama 13.

5.4. Optimización de disco

En diversas secciones del código se guardan unos datos a disco, solo para ser leídos a continuación, es decir, cargados en memoria. Debido a que las funciones *Serverless* suelen tener poco espacio de disco (y en caso de necesitar más aumenta el coste), es conveniente optimizar el código para que no se guarden en disco los datos que no sean estrictamente necesarios.

5.5. Sistema de estadísticas y registros

Anteriormente, en el proceso de análisis, el tiempo y los registros del flujo de datos eran simplemente impresiones en la salida de la consola. Por ese motivo, se propuso una nueva metodología.

La nueva propuesta consiste en crear una estructura adecuada para almacenar los registros del flujo de datos y el tiempo de ejecución. En lugar de imprimir directamente estos datos durante el proceso, se acumulan en esta estructura. Una vez completado el proceso o una determinada fase del mismo, se genera un archivo que contiene todos los datos registrados. Posteriormente, este archivo se sube al almacenamiento de objetos (*Object Storage*).

Esta nueva metodología ofrece diversas ventajas:

1. En primer lugar, contribuye a tener un código más limpio y organizado, ya que se evita la impresión de datos directamente en la consola a lo largo del código. Además, al almacenar los registros de datos en una estructura durante la ejecución del programa, se mejora el control y seguimiento de los datos y el tiempo de ejecución en cada fase del proceso. Esto permite un análisis más detallado y un mejor entendimiento del comportamiento del programa.

2. En segundo lugar, facilita la generación de estadísticas a partir de la información obtenida en los registros de datos almacenados. Al contar con todos los datos registrados en un archivo, se simplifica la extracción de información relevante para la generación de métricas y análisis estadísticos. Esto es de gran utilidad para evaluar el rendimiento del proceso, identificar patrones o tendencias y tomar decisiones sobre qué modificar en el código.

3. Por último, los registros de datos ya no se pierden al ejecutar el proyecto nuevamente. Al almacenar los registros en una estructura y generar un archivo con todos los datos, se crea un registro persistente que se mantiene incluso después de que el programa finalice su ejecución. Esto significa que si se ejecuta el proyecto nuevamente, los registros previamente almacenados estarán disponibles para su análisis y procesamiento. De esta manera, se puede mantener un historial de datos o realizar comparaciones entre diferentes ejecuciones del proyecto.

En resumen, la adopción de esta nueva metodología propone reemplazar los simples *prints* de tiempos y registros de datos durante el proceso por una estructura de almacenamiento. Esto conlleva beneficios como un código más limpio, un mejor control del flujo de datos y tiempo en las diferentes fases, la posibilidad de crear estadísticas con la información obtenida y la conservación de los registros de datos para su análisis en futuras ejecuciones del proyecto.

5.6. Optimización de la generación y obtención del fichero GEM

En el sistema previo, el fichero GEM mencionado anteriormente se genera en las mismas funciones lanzadas de map. Esto resulta innecesario por diversos motivos.

En primer lugar, los segmentos FASTA idénticos generan el mismo fichero GEM. Eso quiere decir que si 20 funciones tienen asignado un mismo segmento FASTA, el mismo fichero GEM se generará 20 veces, cuando realmente solo es necesario generarlo una vez.

Además, la generación de este fichero GEM resulta computacionalmente intensa, por lo que puede llegar a tomar gran parte del tiempo de ejecución para segmentos FASTA grandes. La ventaja viene de que al depender únicamente del fichero FASTA, en futuras ejecuciones donde

se utilice el mismo FASTA se puede aprovechar el fichero GEM generado anteriormente, omitiendo esta generación completamente.

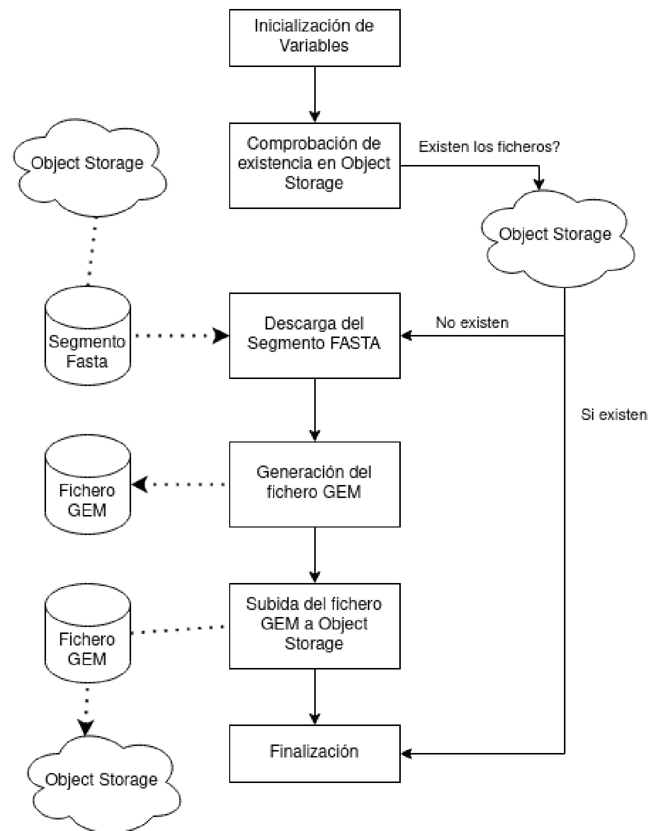

Figura 14: Pasos de la fase de generación del fichero GEM.

Por tanto, para solventar estos dos problemas se ha decidido separar la generación de este fichero a una fase previa al Map, de manera que se lancen solo las funciones necesarias y se guarden los datos en Object Storage. El objetivo consiste en tener una tanda de funciones que realicen cada una los pasos detallados en la figura 14.

5.7. Resumen

Como se puede observar, los cambios a realizar se pueden dividir en dos categorías. En lo que respecta a calidad de código, resulta necesario reescribir el código prácticamente desde cero y aplicar mecanismos de ingeniería y calidad de software. Junto a eso, también resulta necesario implementar una herramienta que permita obtener estadísticas y registros de las funciones para así poder analizar mejor su funcionamiento y rendimiento. A nivel de arquitectura, los cambios a realizar se pueden resumir en lo siguiente:

1. Mejora del tratamiento de datos FASTA.
2. Eliminación de sincronismo entre funciones.
3. Optimización del uso de disco.
4. Eliminación de ejecuciones redundantes.

6 Implementación

En este apartado se van a explicar los detalles técnicos de la implementación realizada siguiendo las ideas planteadas durante el apartado previo de propuesta.

6.1. Estructura principal

Con la nueva implementación del proyecto, las funciones y clases se organizan según su propósito dentro del programa. En lugar de tener todas las fases en un único fichero, el proyecto se divide en módulos y se reorganizan los ficheros que contienen las funciones auxiliares. La estructura del proyecto consiste en un repositorio principal que incluye el directorio asociado con *Docker*, el cual configura el entorno de ejecución del proyecto, el fichero para ejecutar el proyecto, códigos independientes para generar automáticamente estadísticas sobre el coste y tiempo de ejecución, y finalmente, el directorio que contiene el código principal del proyecto.

Dentro del último directorio mencionado, se encuentra el código principal del proyecto. La nueva implementación organiza la estructura principal para ejecutar las diferentes fases, enviar la información generada entre ellas, e inicializar y guardar los registros del tiempo de ejecución y los datos procesados. Además, las fases se dividen en tres subdirectorios: preprocesamiento, *map* y *reduce*. Cada subdirectorio se encarga de inicializar, configurar y ejecutar las diferentes fases.

En el directorio principal, además del código principal y los subdirectorios con las distintas fases, también se encuentran dos ficheros con funciones auxiliares, un fichero con funciones y clases que almacenan la configuración y parámetros necesarios para la ejecución del código, y un fichero con la clase encargada de almacenar y registrar los tiempos de ejecución y los datos procesados.

```
.
├── dockerfile/
│   ├── Dockerfile
│   └── scripts
└── serverlessgenomics/
    ├── mapping/
    │   ├── __init__.py
    │   ├── alignment_mapper.py
    │   ├── data_fetch.py
    │   └── map_caller.py
    ├── preprocessing/
    │   ├── __init__.py
    │   ├── preprocess_fasta.py
    │   └── preprocess_fastq.py
    ├── reducer/
    │   ├── __init__.py
    │   ├── reduce_caller.py
    │   └── reduce_functions.py
    ├── __init__.py
    ├── aux_functions.py
    ├── cachelithops.py
    ├── constants.py
    ├── parameters.py
    ├── pipeline.py
    ├── stats.py
    └── utils.py
```

Figura 15: Estructura del proyecto.

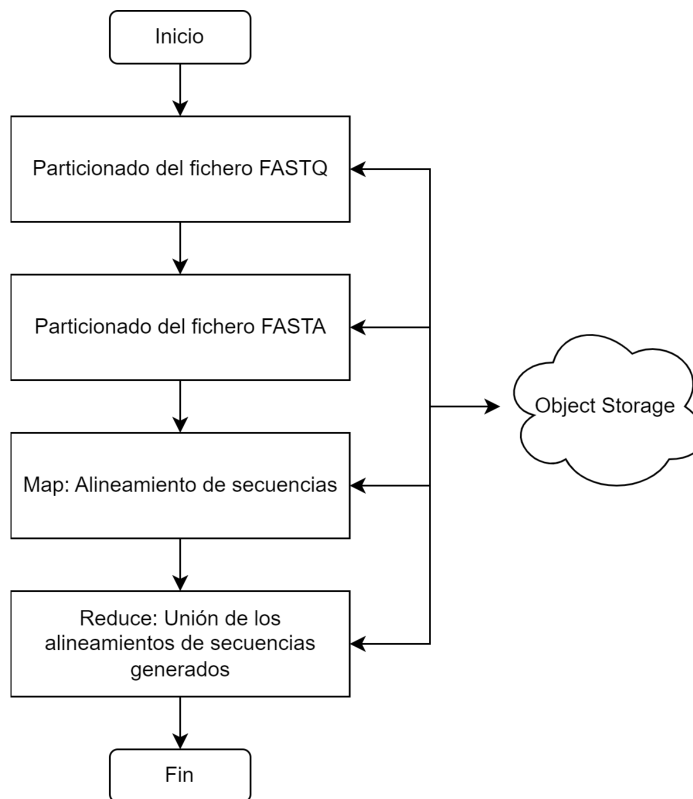

Figura 16: Fases principales del proyecto.

Siguiendo el orden de ejecución del programa, se ha realizado un cambio en que se realizan las llamadas y se pasan los parámetros de entrada. Anteriormente, el código se ejecutaba a través de la invocación del fichero principal, y se proporcionaban todos los parámetros de configuración requeridos como entrada. Actualmente, el programa es ejecutado mediante un fichero que realiza la llamada al código principal, proporcionando la configuración necesaria como parámetros de entrada.

```
1  import logging
2  from serverlessgenomics import VariantCallingPipeline
3
4  if __name__ == '__main__':
5    pipeline = VariantCallingPipeline(
6      fasta_path='s3://urv-bucket/fasta/TriTryp_Genome.fasta',  # FASTA file
      path
7      fasta_chunks=5,  # Number of partitions for the FASTA file
8      fastq_path='s3://urv-bucket/fastq/SRR6052133.fastq.gz',   # FASTQ file
      path
9      fastq_chunks=20, # Number of partitions for the FASTQ file
10     log_level=logging.DEBUG, # Amount of information printed on console
11     storage_bucket='urv-bucket', # Bucket storage name
12     override_id='8b3cead0-fd3f-4b1b-812e-618b6dbc4a28', # Execution
      identification id
13     log_stats=True # Statistics logging on/off
14   )
15
16   pipeline.run_pipeline()
```

Código 1: Ejemplo de llamada y configuración del programa

La nueva implementación del programa tiene como objetivo simplificar su ejecución y evitar

la necesidad de introducir la configuración requerida en cada ejecución. A diferencia de la implementación previa, en la que la configuración se pasaba como parámetro de entrada y se utilizaban variables globales, en la nueva implementación se ha diseñado una estructura específica para el proyecto que almacena toda la configuración necesaria para las diferentes fases del programa. Con esta estructura, es posible configurar por defecto aquellas opciones que no requieren modificación en cada ejecución del programa. La estructura contiene las siguientes configuraciones:

- Configuración general:
  - Ruta a los ficheros de entrada con formato FASTA y FASTQ.
  - Número de particiones a realizar en los ficheros FASTA y FASTQ.
  - Identificador SRA del fichero FASTQ.
  - Rango de particiones FASTA y FASTQ a ser procesados.
  - Nombre base del fichero FASTQ.
  - Indicadores booleanos que permiten determinar si una fase debe realizarse o no.
- Configuraciones de *Lithops*:
  - Número de procesos paralelos, por defecto, 1000.
  - Imagen que a utilizar durante el tiempo de ejecución, por defecto, ninguno.
  - Tamaño de memoria a utilizar durante el tiempo de ejecución, por defecto, 1024 MB.
  - Tiempo máximo por activación de función, por defecto, 2400 segundos.
  - Cantidad de información a mostrar durante la ejecución sobre el estado de las ejecuciones.
  - Indicador booleano que permite determinar si se debe hacer un registro sobre los tiempos de ejecuciones y el movimiento de datos.
- Prefijos de ficheros y nombres de directorios con los resultados intermedios y finales.

Una vez que se ha guardado toda la configuración en la estructura de datos, se procede a ejecutar el código principal del proyecto, el cual invoca a la función *run_pipeline* (ver Anexo A.1). Esta función desempeña dos tareas principales:

1. En primer lugar, se encarga de llevar a cabo la inicialización y el almacenamiento de los registros, en caso de que se requieran para comparaciones futuras.
2. En segundo lugar, invoca a tres subfunciones encargadas de la inicialización de los registros específicos correspondientes a cada fase del programa, así como de la ejecución de las tareas asignadas a cada una de ellas.

Como se ha mencionado previamente, esta función tiene la responsabilidad de preparar y ejecutar las diferentes fases del proyecto:

1. **Preprocesado**

```
1 def preprocess(self):
2   preprocessStat = Stats()
3 
4   preprocessStat.timer_start('preprocess')
5   self.fastq_chunks, subStatFastq = prepare_fastq_chunks(self.parameters
      , self.lithops)
6   self.fasta_chunks, subStatFasta = prepare_fasta_chunks(self.parameters
      , self.lithops)
7   preprocessStat.store_dictio(subStatFastq.get_stats(), "
      subprocesses_fastq", "preprocess")
8   preprocessStat.store_dictio(subStatFasta.get_stats(), "
      subprocesses_fasta", "preprocess")
9   return preprocessStat
```

Código 2: Preprocesado.

2. **Map**

```
1 def align_reads(self):
2   assert self.fasta_chunks is not None and self.fastq_chunks is not None
      , 'generate chunks first!'
3   alignReadsStat = Stats()
4 
5   alignReadsStat.timer_start('align_reads')
6   mapper_output, subStat = run_full_alignment(self.parameters,
7           self.lithops, self.fasta_chunks, self.fastq_chunks)
8   alignReadsStat.timer_stop('align_reads')
9   alignReadsStat.store_dictio(subStat.get_stats(), "phases",
10          "align_reads")
11  return mapper_output, alignReadsStat
```

Código 3: Map.

3. **Reduce**

```
1 def reduce(self, mapper_output):
2   reduceStat = Stats()
3 
4   reduceStat.timer_start('reduce')
5   subStat = run_reducer(self.parameters, self.lithops, mapper_output)
6   reduceStat.timer_stop('reduce')
7   reduceStat.store_dictio(subStat.get_stats(), "phases", "reduce")
8   return reduceStat
```

Código 4: Reduce.

6.2. Preprocesamiento

En la reimplementación del proceso de preprocesamiento se han realizado dos cambios importantes. En primer lugar, se ha trasladado el proceso de generación de iterdata a la fase de *map*. En segundo lugar, se ha modificado el código del particionador FASTA para evitar la generación de datos duplicados. En lugar de generar particiones del fichero original, ahora se crea un único fichero índice que se utiliza para generar las particiones necesarias.

La fase de preprocesamiento ahora está en un módulo llamado *preprocessing*, el cual contiene dos ficheros que se encargan del particionamiento de ficheros FASTA y FASTQ.

6.2.1. Particionador FASTA: preprocess_fasta.py

Durante la fase de particionamiento del fichero FASTA, se realizan dos etapas diferentes. En la primera, se crea el fichero de índice si no se ha generado anteriormente. En la segunda etapa, se producen las particiones de secuencias que se emplearán en la fase de alineación de secuencias, utilizando el fichero de índice generado previamente.

```
1 def prepare_fasta_chunks(pipeline_params: PipelineRun, lithops: Lithops):
2   """
3   Calculate fasta byte ranges and metadata for chunks of a pipeline run,
      generate faidx index if needed
4   """
5   subStat = Stats()
6
7   # Get number of sequences from fasta file, generate faidx file if needed
8   subStat.timer_start('prepare_fasta_chunks')
9   num_sequences = generate_faidx_from_s3(pipeline_params, lithops, subStat)
10  fasta_chunks = get_fasta_byte_ranges(pipeline_params, lithops,
      num_sequences)
11  subStat.timer_stop('prepare_fasta_chunks')
12
13  return fasta_chunks, subStat
```

Código 5: Función *prepare_fasta_chunks*.

En el código 5, se observa que la subfunción *generate_faidx_from_s3* es la encargada de la primera etapa. Por otro lado, la función *get_fasta_byte_ranges* se encarga de la segunda etapa.

6.2.1.1. Generación del fichero índice

En la primera etapa, se verifica si el fichero índice ha sido generado previamente. En caso contrario, la función *generate_faidx_from_s3* (ver Anexo B.1) se encarga de preparar la ejecución para generar dicho fichero índice. Esta ejecución se divide en dos fases: la fase de *map* (*create_index_chunked*, ver Anexo B.2) y la fase de *reduce* (*reduce_chunked_indexes*, ver Anexo B.3).

1. **Fase *map*:** Consiste en dividir un fichero FASTA en fragmentos para realizar una búsqueda paralela de las diferentes secuencias dentro del fichero original. Cada proceso paralelo se encarga de buscar el identificador de cada secuencia, la ubicación del primer byte de su encabezado y su correspondiente base, para luego almacenar esta información en una lista

   Antes de comenzar la lectura del fragmento asignado a cada proceso, se realiza una verificación para asegurarse de que la primera secuencia no ha sido cortada a mitad de su cabecera o base durante la partición del fichero original. Al final de la lectura, se realiza otra verificación para asegurarse de que la última secuencia no ha sido cortada a mitad de su cabecera.

En caso de que se verifique que se ha cortado una secuencia en un lugar no deseado, se agregará la información reconocida de la secuencia fragmentada junto con un identificador que indica la falta de información. De esta forma, en la siguiente fase del proceso se podrá tratar el problema y completar la información faltante de la secuencia.

2. **Fase *reduce*:** Se recopilan todas las listas generadas por los diferentes procesos y las une en una única lista. Durante esta unión, se corrigen las secuencias que se hayan fragmentado por la mitad y que, por lo tanto, no contengan toda la información necesaria.

   Una vez que se tiene la lista completa de secuencias corregidas, se combinan todas las secuencias en una única cadena de caracteres. Durante esta combinación, las secuencias se codifican en formato binario para facilitar su almacenamiento y procesamiento posterior.

   Posteriormente, se comprime la cadena de caracteres binaria para reducir su tamaño y optimizar el uso del espacio de almacenamiento. Finalmente, el fichero comprimido se almacena en el almacenamiento correspondiente.

6.2.1.2. Generación de las particiones

En la segunda etapa del proceso, la función *get_fasta_byte_ranges* (ver Anexo B.4) crea un número determinado de particiones del fichero FASTA según el número de procesos paralelos deseados. Para ello, se emplea el fichero índice generado en la etapa previa para definir los límites de cada partición.

Durante esta etapa, se genera una lista que almacena la ubicación del primer y último byte de cada partición. También se verifica que el inicio o final de cada partición no coincida con la mitad de la cabecera de una secuencia. Esta verificación se realiza mediante la identificación de los bytes que indican el inicio de una cabecera o la base del fichero índice.

6.2.2. Particionador FASTQ: preprocess_fastq.py

Se han realizado modificaciones y mejoras significativas en el particionador FASTQ, enfocadas en la optimización, refactorización y adaptación a la nueva estructura principal.

El particionador FASTQ, implementado en la función *prepare_fastq_chunks* (ver Anexo B.5), al igual que el particionador FASTA, consta de dos etapas fundamentales. La primera etapa se centra en la generación de un fichero de índice en caso de que no haya sido previamente creado. La segunda etapa utiliza dicho fichero índice, generado en la fase anterior, para llevar a cabo la partición de las secuencias.

Durante la primera etapa del proceso, se realiza una verificación para determinar si el fichero de índice ha sido generado previamente. En caso de no existir, se invoca a la función *generate_fastqgz_index_from_s3* (ver Anexo B.6), la cual se encarga de generar el fichero índice a partir del fichero FASTQ comprimido, utilizando la herramienta *gztool*.

En la segunda etapa del proceso, se utiliza la función *get_ranges_from_line_pairs* (ver Anexo B.7) para generar los rangos de partición basados en el número de particiones especificado por el usuario y las ventanas identificadas en el fichero FASTQ.

6.3. Map

Aunque realmente se realizan múltiples llamadas de tipo *map* (acción de lanzar múltiples funciones paralelas) a lo largo del pipeline, cuando se menciona como una fase se hace referencia a la parte del programa donde se realiza el procesamiento y computación principal de los ficheros de entrada.

Todo aquello relacionado con esta fase se ha empaquetado en un módulo llamado *mapping* a excepción de los propios scripts de alineación, los cuales se encuentran ya en el propio entorno de ejecución Docker. Estos scripts han sido provistos por Paolo Ribeca y Lucio Marcello.

6.3.1. Coordinador: map_caller.py

Este fichero actúa como coordinador de las diferentes subfases que intervienen en el procesado de los segmentos FASTA y FASTQ, junto a la generación de la iterdata. En la figura 6 se puede observar como la función *run_full_allignment* crea los datos de entrada de las diferentes funciones y realiza las invocaciones de AWS Lambda.

```
1 def run_full_alignment(pipeline_params: PipelineRun, lithops: Lithops,
     fasta_chunks, fastq_chunks):
2   # MAP: Stage 0.1
3   iterdata = generate_gem_generator_iterdata(pipeline_params, fasta_chunks)
4   lithops.invoker.map(gem_generator, iterdata)
5
6   # MAP: Stage 1
7   iterdata = generate_aligner_indexer_iterdata(pipeline_params, fasta_chunks,
      fastq_chunks)
8   aligner_indexer_result = lithops.invoker.map(aligner_indexer, iterdata)
9
10  # MAP: Index correction
11  iterdata = generate_index_correction_iterdata(pipeline_params,
     aligner_indexer_result)
12  index_correction_result = lithops.invoker.map(index_correction, iterdata)
13
14  # Map: Stage 2
15  iterdata = generate_index_to_mpileup_iterdata(pipeline_params, fasta_chunks
     , fastq_chunks,
16                         aligner_indexer_result, index_correction_result)
17  alignment_output = lithops.invoker.map(filter_index_to_mpileup, iterdata)
18
19  return alignment_output
```

Código 6: Coordinación de las diferentes fases del Map

En este fichero, también encontramos las funciones que generan la iterdata que consumirán las funciones lambda de cada subfase. Cada iterdata se encuentra dividida en diversas subfunciones que son llamadas desde la función principal coordinadora, con tal de seguir con la orientación modular.

1. **Generación del fichero GEM.** Tal y como se mencionaba en la propuesta, se ha separado esta subfase de las otras.

2. **Map Fase uno:** Generación del fichero *.MAP* e Índice.

3. **Corrección de índices:** Se corrige el índice con funciones serverless en lugar de con un servidor Redis alojado en una estancia EC2.
4. **Map Fase dos:** Aplicación del índice corregido y generación del Mpileup.

Como se puede observar, las funciones son parecidas entre sí, pero cada una empaqueta un vector estrictamente, los datos necesarios que va a necesitar cada función de cada fase. En algunos casos se puede tratar de una cadena de caracteres o a veces de un diccionario, como los metadatos de los segmentos FASTA y FASTQ.

1. **Iterdata de la fase GEM:** En la figura 7 se generan los datos de entrada para la creación del fichero GEM.

```
1 def generate_gem_generator_iterdata(pipeline_params: PipelineRun,
     fasta_chunks):
2   iterdata = []
3   for fa_i, fa_ch in enumerate(fasta_chunks):
4     if (pipeline_params.fasta_chunk_range is None) or (fa_i in
     pipeline_params.fasta_chunk_range):
5       params = {'pipeline_params': pipeline_params,
6           'fasta_chunk_id': fa_i, 'fasta_chunk': fa_ch}
7       iterdata.append(params)
8   return iterdata
```

Código 7: Iterdata de la fase GEM.

2. **Iterdata de la fase Mapper:** En la figura 8 se generan los datos de entrada para la alineación inicial de los ficheros FASTQ y FASTA.

```
1 def generate_aligner_indexer_iterdata(pipeline_params: PipelineRun,
     fasta_chunks, fastq_chunks):
2   iterdata = []
3   for fq_i, fq_ch in enumerate(fastq_chunks):
4     if (pipeline_params.fastq_chunk_range is None) or (fq_i in
     pipeline_params.fastq_chunk_range):
5       for fa_i, fa_ch in enumerate(fasta_chunks):
6         if (pipeline_params.fasta_chunk_range is None) or (fa_i in
     pipeline_params.fasta_chunk_range):
7           params = {'pipeline_params': pipeline_params,
8               'fasta_chunk_id': fa_i, 'fasta_chunk': fa_ch,
9               'fastq_chunk': fq_ch, 'fastq_chunk_id': fq_i}
10          iterdata.append(params)
11  return iterdata
```

Código 8: Iterdata de la fase Mapper.

3. **Iterdata de la fase Corrección de Índice:** En la figura 9 se generan los datos de entrada para la corrección de los índices.

```
1 def generate_index_correction_iterdata(pipeline_params,
     gem_mapper_output):
2   # Group gem mapper output by fastq chunk id
3   fq_groups = collections.defaultdict(list)
4   for fq, _, map_key, _ in gem_mapper_output:
5     fq_groups[fq].append(map_key)
6
7   iterdata = [{'pipeline_params': pipeline_params,
```

```
8            'fastq_chunk_id': fq_id,
9            'map_index_keys': map_keys} for fq_id, map_keys in fq_groups.
    items()]
10
11   return iterdata
```

Código 9: Iterdata de la fase Corrección de Índice.

4. **Iterdata de la fase Filtrado:** En la figura 10 se generan los datos de entrada para la aplicación del índice corregido y la generación del fichero Mpileup.

```
1 def generate_index_to_mpileup_iterdata(pipeline_params, fasta_chunks,
    fastq_chunks, gem_mapper_output, corrected_indexes):
2     iterdata = []
3     # Convert corrected index output (list of tuples) to sorted list by
    fastq chunk id
4     corrected_indexes_fq = [tup[1] for tup in sorted(corrected_indexes,
    key=lambda tup: tup[0])]
5     for fq_i, fa_i, _, filter_map_index in gem_mapper_output:
6         iterdata.append({'pipeline_params': pipeline_params,
7                          'fasta_chunk_id': fa_i,
8                          'fasta_chunk': fasta_chunks[fa_i],
9                          'fastq_chunk_id': fq_i,
10                         'fastq_chunk': fastq_chunks[fq_i],
11                         'filtered_map_key': filter_map_index,
12                         'corrected_index_key': corrected_indexes_fq[
    fq_i]})
13    return iterdata
```

Código 10: Iterdata de la fase Filtrado.

Una vez terminadas estas subfases de map, el código continuaría hacia la fase de Reduce.

6.3.2. Funciones de alineamiento: alignment_mapper.py

Este fichero contiene las funciones en sí que se van a ejecutar en las funciones lambda, así como sus dependencias. Esto incluye la generación del fichero GEM, el alineamiento inicial, la corrección del índice generado durante el primer alineamiento y el filtrado final.

6.3.2.1. Generación del fichero GEM

En primer lugar, está la función *gem_generator*, la cual dado un segmento FASTA se encarga de generar el fichero GEM. Este fichero GEM es necesario a la hora de alinear la muestra del fichero FASTQ con el fichero FASTA. Esta función se puede dividir en los siguientes pasos:

1. **Inicialización de variables:** Se inicializa el sistema de ficheros y los nombres y claves de los ficheros.

2. **Comprobación de existencia en Object Storage:** Antes de proceder a la generación de este fichero, se comprueba si este ya existe en Object Storage. En caso afirmativo, el fichero no se vuelve a generar, y en su lugar se devuelve la clave a este fichero ya existente.

3. **Descarga del segmento FASTA:** Como el fichero GEM no existe en la base de datos, se procede a descargar el segmento FASTA necesario para generarlo. Se obtiene a través de una descarga de rangos de bytes.

4. **Generación del fichero GEM (figura 11):** Se genera el fichero GEM a través de la librería *gem-indexer*.

5. **Subida del fichero GEM a Object Storage:** Antes de acabar se sube el fichero a Object Storage, donde será obtenido por las siguientes etapas del Map y en otras ejecuciones posteriores (siempre y cuando no se borre).

6. **Finalización:** Se borran los ficheros locales.

```
1 cmd = ['gem-indexer',
2     '--input', fasta_chunk_filename,
3     '--threads', str(multiprocessing.cpu_count()),
4     '-o', gem_index_filename.replace('.gem', '')]
5 out = sp.run(cmd, capture_output=True)
```

Código 11: Generación del fichero GEM en la función *gem_generator*.

Una vez hayan finalizado todas las funciones lanzadas, los ficheros GEM asignados a cada segmento FASTA estarán disponibles en Object Storage.

6.3.2.2. Alineación inicial

La siguiente fase consiste en la función *aligner_indexer*. En esta fase se genera el fichero con el mapeo entre los ficheros FASTQ y GEM, lo cual genera dos resultados: un fichero *.MAP* que contendrá la alineación y un fichero índice que deberá ser corregido en una fase posterior. Esta función se puede dividir en los siguientes pasos:

1. **Inicialización de variables:** Se inicializa el sistema de ficheros y los nombres y claves de los ficheros.

2. **Comprobación de existencia en Object Storage:** Si los ficheros que debe generar esta función ya existen en Object Storage debido a una ejecución previa, se devuelve la clave a estos ficheros.

3. **Descarga del segmento FASTQ:** Se obtiene el segmento FASTQ de *Object Storage* a través de una descarga de rangos de bytes.

4. **Descarga del segmento FASTA:** Se obtiene el segmento FASTQ de *Object Storage* a través de una descarga de rangos de bytes.

5. **Descarga del fichero GEM asignado al segmento FASTA.**

6. **Alineamiento (figura 12):** Se genera el fichero intermedio *.map* y el índice a corregir. Este punto es de los más intensos computacionalmente.

7. **Compresión:** Se comprimen ambos ficheros para agilizar la transferencia y ocupar menos en Object Storage.

8. **Subida a Object Storage:** Como la función va a borrar todo al finalizar, hace falta subir todos los ficheros que sean necesarios en una fase posterior a Object Storage.

9. **Finalización:** Se borran los ficheros locales.

```
1 cmd = ['/function/bin/map_index_and_filter_map_file_cmd_awsruntime.sh',
2         gem_index_filename,
3         fastq_chunk_filename,
4         "not-used", pipeline_params.base_name,
5         "s3",
6         "single-end"]
7 out = sp.run(cmd, capture_output=True)
```

Código 12: Alineación entre el fichero GEM y el fichero FASTQ en la función *aligner_indexer*.

En la implementación antigua, la función quedaba bloqueada una vez generado el alineamiento, pero en la nueva implementación finaliza su ejecución completamente una vez se han subido los resultados a *Object Storage*.

6.3.2.3. Corrección de índices

La siguiente fase consiste en corregir el índice generado. Para esto se lanzarán un número de funciones equivalente al número de segmentos FASTQ. El código encargado de esta tarea se encuentra en la función *index_correction*. Esta función consta de las siguientes fases:

1. **Inicialización de variables:** Se inicializa el sistema de ficheros y los nombres y claves de los ficheros.

2. **Comprobación de existencia en Object Storage:** Si los ficheros que debe generar esta función ya existen en Object Storage debido a una ejecución previa, se devuelve la clave a estos ficheros.

3. **Descarga de índices:** La función descargará todos los índices referentes a un segmento FASTQ concreto. Por tanto, el número total de ficheros a descargar será equivalente al número de segmentos FASTA (cada segmento FASTQ está alineado a cada segmento FASTA).

4. **Ejecución de los scripts de corrección (figura 13)**: A través de 2 scripts previamente proporcionados se realizan una serie de operaciones que tienen como resultado el índice corregido.

5. **Compresión:** Se comprime el resultado para agilizar la transferencia y ocupar menos en Object Storage.

6. **Subida a Object Storage:** Como la función va a borrar todo al finalizar, hace falta subir el índice corregido a Object Storage.

7. **Finalización:** Se borran los ficheros locales.

```
cmd = f'/function/bin/binary_reducer.sh \
        /function/bin/merge_gem_alignment_metrics.sh \
        4 \
        {input_temp_dir}/* \
        > {intermediate_file}'
proc = sp.run(cmd, shell=True, universal_newlines=True, capture_output=True)

cmd = f'/function/bin/filter_merged_index.sh \
        {intermediate_file} \
        {output_file}'
proc = sp.run(cmd, shell=True, check=True, universal_newlines=True)
```

Código 13: Corrección de los índices de un set de segmentos FASTQ en la función *index_correction*

6.3.2.4. Filtrado

Para terminar el alineamiento hace falta aplicar el índice corregido al fichero MAP generado anteriormente. Para ello se llama a la función *filter_index_to_mpileup*. Esta porción de código sería el equivalente a nivel funcional a lo que realizaba la implementación antigua una vez se desbloqueaba. Consta de las siguientes fases:

1. **Inicialización de variables:** Se inicializa el sistema de ficheros y los nombres y claves de los ficheros.
2. **Comprobación de existencia en Object Storage:** Si los ficheros que debe generar esta función ya existen en Object Storage debido a una ejecución previa, se devuelve la clave a estos ficheros.
3. **Descarga del segmento FASTA.**
4. **Descarga del fichero MAP:** Este fichero intermedio se había generado durante la función *aligner_indexer* y ahora se recupera de Object Storage.
5. **Descarga del índice corregido:** Se obtiene el índice corregido. Todas las funciones asociadas a un mismo segmento FASTQ recibirán el mismo índice.
6. **Filtrado (figura 14):** Se utiliza el índice corregido sobre el fichero MAP para filtrar resultados no deseados.
7. **Generación de Mpileup:** Se convierte el resultado filtrado al formato estándar Mpileup.
8. **Subida a Object Storage:** Se sube el resultado final a Object Storage para luego ser procesado por la fase de Reduce.
9. **Finalización:** Se borran los ficheros locales.

```
#Filtrado
cmd = ['/function/bin/map_file_index_correction.sh',
        corrected_index_filename,
        filt_map_filename,
        str(pipeline_params.tolerance)]
proc = sp.run(cmd, capture_output=True)
```

```
7
8 #Generacion Mpileup
9 cmd = ['/function/bin/gempileup_run.sh',
10         corrected_map_file,
11         fasta_chunk_filename]
12 proc = sp.run(cmd, capture_output=True)
```

Código 14: Filtrado con el índice corregido y generación del fichero Mpileup *filter_index_to_mpileup*.

6.3.3. Funciones Auxiliares de transferencia de datos: data_fetch.py

Este fichero contiene las dos funciones utilizadas para recibir el segmento FASTQ o FASTA asignado a una función.

En primer lugar, la función *fetch_fastq_chunk* (ver Anexo D.1) se encarga de obtener el segmento FASTQ. Esta función es un poco más compleja que su contra-parte FASTA, ya que el fichero FASTQ se encuentra comprimido.

Para poder obtener dicho segmento es necesario primero descargar el índice generado en la fase de *pre-processing*. Una vez descargado es posible obtener los bytes concretos del fichero original.

A continuación está la función de obtención del segmento FASTA (figura 15). Esta es más simple, ya que recibe directamente por parámetro los bytes concretos a coger del fichero original.

```
1 def fetch_fasta_chunk(fasta_chunk: dict, target_filename: str, storage:
    lithops.Storage, fasta_path: S3Path):
2   # Get header data
3   extra_args = {'Range': f"bytes={fasta_chunk['offset_head']}-{fasta_chunk['
    offset_base']}"}
4   header_body = storage.get_object(bucket=fasta_path.bucket, key=fasta_path.
    key, extra_get_args=extra_args)
5   chunk_body = list(re.finditer(r">.+\n", header_body.decode('utf-8')))[0].
    group()
6
7   # Get chunk body and append to header
8   extra_args = {'Range': f"bytes={fasta_chunk['offset_base']}-{fasta_chunk['
    last_byte']}"}
9   base = storage.get_object(bucket=fasta_path.bucket, key=fasta_path.key,
    extra_get_args=extra_args).decode('utf-8')
10  chunk_body += base[1::] if base[0:1] == '\n' else base
11
12  with open(target_filename, 'w') as target_file:
13    target_file.writelines(chunk_body)
```

Código 15: Función auxiliar *fetch_fasta_chunk*.

6.3.4. Finalización de la fase de Map

Con esto puede dar por terminada la fase de Map. Como se puede observar, se ha pasado de tener una sola llamada para ejecutar la fase a tener 4 llamadas a diferentes funciones, lo cual permite tener más control sobre las diferentes fases. Esto permite también poder omitir más fácilmente las subfases que no sean necesarias y en caso de necesitar hacer modificaciones, solo ejecutar la fase que se ha modificado.

Aunque la nueva implementación soluciona los problemas de escalabilidad de la implementación antigua (tal y como se comprobará en la fase de evaluación) y se elimina la necesidad de un servidor dedicado para una de las fases, se añade el inconveniente de la transferencia de datos. En la implementación anterior, como las funciones *aligner_indexer* y *filter_index_to_mpileup* se ejecutaban en una única función ininterrumpida, no era necesario transferir ningún resultado. Sin embargo, al realizar esta separación se añade la necesidad de subir y posteriormente descargar los datos intermedios. De hecho, el segmento FASTA es descargado 2 veces, ya que en el filtrado es necesario. Esto añade una sobrecarga a las funciones, la cual, a pesar de no ser un tiempo relativamente pequeño comparado con el tiempo de computación, sigue siendo un inconveniente a la nueva implementación. A pesar de ello, se trata de un incremento menor con el que existía con los bloqueos causados por el servidor de Redis.

Ahora es necesario poner en común todos los resultados finales y juntarlos en un único fichero final, tal y como si se hubiese realizado el filtraje en una sola máquina de manera secuencial. Esto se realiza en la fase de Reduce.

6.4. Reduce

Esta fase consiste en extraer todos los ficheros generados en la fase de Map y juntarlos en un único fichero. Estor resulta más complejo que simplemente concatenar dichos ficheros. Es necesario realizar una operación de fusión a todas las líneas que tengan el mismo índice de posición.

Para esta tarea se ha creado un módulo llamado *reducer* que contiene un fichero con las funciones de coordinación y de *iterdata* y otro con las funciones lambda a ejecutar. Al igual que con la fase de Map, los scripts necesarios otorgados por Paolo Ribeca y Lucio Marcello se encuentran ya en la imagen Docker.

6.4.1. Coordinador: reduce_caller.py

Este fichero actúa como coordinador de las diferentes subfases que intervienen en la fusión de los ficheros Mpileup generados en la fase de Map. La gestión de estas subfases se hace a través de la función *run_reducer* (ver Anexo E.1). Estas subfases se pueden resumir en:

1. **Organización de los Mpileups:** Las claves de los ficheros Mpileup guardados en *Object Storage* son separadas en grupos según el segmento FASTA.
2. **Definición de nuevas claves:** Se definen los nombres que tendrán los ficheros finales generados por los reducers.
3. **Inicialización de las subidas parciales:** Como cada *reducer* creará una porción de un fichero final, se puede crear una subida parcial, de manera que cada *reducer* subirá una porción del fichero y el proveedor de *Object Storage* se encargará de concatenar dichas partes.
4. **Distribución de índices**: Se distribuyen los índices a fusionar entre diferentes *reducers* con tal de distribuir la carga de manera más o menos equitativa. Esta parte se realiza

con funciones Lambda, ya que se deben contar todos los índices de los Mpileups, lo cual se puede hacer de manera más óptima a través de múltiples funciones.

5. **Reducción**: Se lanzan los diferentes *reducers* los cuales procesan los índices que les han estado asignados en la anterior subfase.
6. **Concatenación final**: En la anterior subfase se han generado un número de ficheros equivalente al número de segmentos FASTA. Estos ficheros finales no requieren de ninguna operación como tal, sino que se pueden concatenar unos detrás de otros.

6.4.2. Funciones de reducción y distribución: reduce_functions.py

Este fichero contiene el código que se van a lanzar en funciones lambdas, así como algunas funciones auxiliares que se pueden ejecutar de manera local. Este código se trata de dos funciones, una para distribuir los índices y otra para la reducción donde se procesan dichos índices.

6.4.2.1. Distribución de índices

Esta etapa se encarga de analizar los ficheros Mpileup generados en la anterior fase para decidir los índices que cada *reducer* va a procesar. Esto resulta necesario hacerlo con funciones *lambda* por la gran cantidad de datos que se procesan.

De todos los archivos, solo se requiere la segunda columna, que contiene el índice de la posición de la lectura. Cada uno de estos índices puede aparecer solo una vez en un archivo, pero no es necesario que aparezca en todos ellos. Estos índices se obtienen mediante una consulta especial llamada *S3 Select* en el servicio de almacenamiento de objetos de AWS. Estas consultas son similares a las consultas SQL y permiten aplicar filtros a la consulta. En este caso, se utilizará para obtener solo la segunda columna y omitir el resto, lo que reduce significativamente los datos a descargar y procesar (ver figura 16).

En la implementación antigua era necesario convertir el Mpileup a CSV o Parquet para realizar esta conversión, pero a través de unos parámetros de configuración en la consulta se ha hecho posible realizar la consulta directamente sobre el Mpileup.

```
1 expression = "SELECT cast(s._2 as int) FROM s3object s"
2 input_serialization = {'CSV': {'RecordDelimiter': '\n', 'FieldDelimiter': '\t
    '}, 'CompressionType': 'NONE'}
```

Código 16: Configuración de S3 Select para la distribución de índices.

Una vez se han obtenido estos índices se cuenta cuantas veces aparece cada uno, y entonces se van dividiendo de manera que cada *reducer* procese cierta cantidad como máximo (de 20 a 50 millones de índices normalmente).

Se llevan a cabo los siguientes pasos:

1. **Consultas S3 Select:** Se obtienen los ficheros Mpileup asignados a un segmento FASTA descargando exactamente la segunda columna. Cada vez que aparece un índice se aumenta su contador en un diccionario.

2. **Distribución (figura 17):** Se distribuyen los índices entre diferentes *reducers*. Si no hay suficientes índices para llegar al máximo asignado, es posible que solo se lance un *reducer* por segmento.

```
1 MAX_INDEXES = 20_000_000
2 workers_data = []
3 indexes = 0
4
5 for key in count_indexes:
6   if indexes + count_indexes[key] < MAX_INDEXES:
7     indexes += count_indexes[key]
8     index = key
9   else:
10    indexes = 0
11    workers_data.append(index)
12 workers_data.append(key)
```

Código 17: Distribución de los índices.

6.4.2.2. Reducción

En este punto cada *reducer* descarga los índices asignados y aplica el script SiNPle-0.5, el cual se encarga de fusionar los índices de mismo valor. Al final de esta etapa se generan un número de archivos equivalente al número total de segmentos FASTA.

```
1 expression = "SELECT * FROM s3object s WHERE cast(s._2 as int) BETWEEN %s AND
      %s" % (range['start'], range['end'])
2 input_serialization = {'CSV': {'RecordDelimiter': '\n', 'FieldDelimiter': '\t
     '}, 'CompressionType': 'NONE'}
```

Código 18: Configuración de S3 Select para la obtención de rangos de Mpileup.

Para descargar solo los índices asignados se hace uso de la misma herramienta de la anterior sub-fase, el S3 Select. Esta vez, en lugar de descargar solo la segunda columna, se descargarán todas, pero solo aquellas donde el valor de la segunda columna caiga dentro del rango asignado a este *reducer* (figura 18). Esta operación se ejecutará a diversos ficheros, por lo que solo descargar una porción de los datos en lugar de todo el fichero resulta bastante óptimo.

La función que se encarga de todo esto, *reduce_function*, consta de los siguientes pasos:

1. **Inicialización de variables:** Se inicializa el sistema de ficheros y los nombres y claves de los ficheros.

2. **Consulta S3 Select:** Se realiza la consulta de S3 Select mostrada en la figura 18 para obtener las entradas de los ficheros con el índice dentro del rango asignado a este *reducer*.

3. **Reducción:** Se aplica el script de corrección, el cual devuelve el resultado a través de un PIPE para evitar pasar el resultado por fichero.

4. **Subida del resultado:** Se sube la parte del fichero correspondiente a *Object Storage* a través de una *MultiPartUpload*.

6.4.2.3. *Fusión final*

En esta pequeña fase se descargan los ficheros resultantes de la fase de reducción y se concatenan unos detrás de otros a través de una *MultiPartUpload*. Esta se hace a través de la función *final_merge* (figura 19) y se lanza una por cada fichero creado en la fase de reducción.

```
1 def final_merge(mpu_id: str, mpu_key: str, key: str, n_part: int,
     pipeline_params: PipelineRun, storage: Storage) -> dict:
2   sinple_out = storage.get_object(bucket=pipeline_params.storage_bucket, key=
     key)
3   s3 = storage.get_client()
4   part = s3.upload_part(
5     Body = sinple_out,
6     Bucket = pipeline_params.storage_bucket,
7     Key = mpu_key,
8     UploadId = mpu_id,
9     PartNumber = n_part)
10  return {"PartNumber" : n_part, "ETag" : part["ETag"], "mpu_id": mpu_id}
```

Código 19: Fusión final de archivos.

6.5. Monitorización

Con el propósito de registrar de manera precisa el flujo de datos y el tiempo de ejecución, se ha implementado una clase, llamada *stats* que almacena esta información de manera estructurada y legible. Esta estructura permite almacenar todos los datos relevantes de manera persistente, lo que facilita el análisis y la comparación con otras ejecuciones del programa. Esta estructura de registro tiene diferentes funciones, entre las que se incluyen:

- **Cronometrar el tiempo de ejecución:** Se han integrado funciones de temporización (*timer_start* y *timer_stop*, ver código 20) que permiten medir el tiempo de ejecución de las diferentes partes del programa.

```
1 def timer_start(self, script, extra_time=None):
2   if script in self.__tmp_registrer:
3     print(f'WARNING: the counter of the timer \"{script}\" was already
      running, it will restart.')
4   elif script in self.__stats and "execution_time" in self.__stats[
      script]:
5     raise Exception(f'The timer of \"{script}\" already existed, choose
      another name or remove the existing timer first.')
6   self.__tmp_registrer[script] = time.perf_counter() if extra_time is
      None else extra_time + time.perf_counter()
7
8 def timer_stop(self, script):
9   end_time = time.perf_counter()
10  if script in self.__tmp_registrer:
11    if script not in self.__stats:
12      self.__stats[script] = {}
13    self.__stats[script]["execution_time"] = end_time - self.
      __tmp_registrer[script]
14    del self.__tmp_registrer[script]
15  else:
16    raise Exception(f'You can not execute this function before "
      timer_start".')
```

Código 20: Función *timer_start* y *timer_stop*.

- **Almacenar diferentes tipos de datos:** Como se puede observar del código 21, se ha creado una función la cual permite almacenar en una estructura diferentes tipos de datos, incluyendo números, cadenas de caracteres, etc.

```
1 def store_size_data(self, name_data, size, script=None):
2   if script is None:
3     dictionary = self.__stats
4   else:
5     if script not in self.__stats:
6     self.__stats[script] = {}
7     dictionary = self.__stats[script]
8
9   if name_data in dictionary:
10     raise Exception(f'The key \"{name_data}\" contains data, choose
    another name or remove the key first.')
11   else:
12     dictionary[name_data] = size
```

Código 21: Función *store_size_data*.

- **Unir registros:** Se ha implementado una función (Código 22) para unir registros.

```
1 def store_dictio(self, dictio, name_dictio=None, script=None):
2   if isinstance(dictio, dict) and name_dictio is None:
3     raise Exception(f'The first parameter is not a dictionary, you must
    write a name for it (second parameter).')
4   if dictio is not None and dictio: # Store if it is not None or empty
5     if script is None:
6
7       dictionary = self.__stats
8     else:
9       if script not in self.__stats:
10         self.__stats[script] = {}
11       dictionary = self.__stats[script]
12
13     if name_dictio is not None:
14       if name_dictio in dictionary:
15         raise Exception(f'The key \"{name_dictio}\" contains data,
    choose another name or remove the key first.')
16       else:
17         dictionary[name_dictio] = dictio
18     elif not any(key in self.__stats for key in dictio):
19       dictionary.update(dictio)
```

Código 22: Función *store_dictio*.

- **Eliminar registros:** Se ha integrado una función (Código 23) para eliminar registros.

```
1  def delete_stat(self, stat):
2   if stat in self.__stats:
3     stat_data = deepcopy(self.__stats.get(stat))
4     del self.__stats[stat]
5     return stat_data
6   else:
7     print("WARNING: The state that was attempted to be deleted does not
    exist.")
8     return None
```

Código 23: Función *delete_stat*.

- **Obtener registros específicos o todo el registro:** La estructura tiene una función (Código 24) que permite acceder a registros específicos o a todo el registro.

```
1 def get_stats(self, stats=None):
2   if stats is None:
3     return deepcopy(self.__stats)
4
5   dictio = deepcopy(self.__stats.get(stats))
6   if dictio is None:
7     raise Exception(f'The key \"{stats}\" not exist.')
8   return dictio
```

Código 24: Función *get_stats*.

- **Almacenar los registros en el almacenamiento:** Se ha implementado una función (Código 25) que almacena el registro en una ubicación designada del almacenamiento.

```
1 def load_stats_to_json(self, bucket, name_file='log_stats'):
2   storage = Storage()
3   storage.put_object(bucket=bucket, key=f'stats/{name_file}.json',
4                  body=str(json.dumps(self.__stats,indent=2)))
```

Código 25: Función *load_stats_to_json*.

6.6. Resumen

En este apartado se han podido ver las secciones más significativas del código, así como las partes en las cuales han sido desglosadas todas las fases del programa. El código total resulta más extenso, en unas 3.300 líneas de código.

Aunque la implementación de los cambios vistos consigue la finalización correcta del programa, principalmente con el uso de juegos de prueba consistentes en pocos datos, es necesario realizar una ejecución del programa haciendo uso de datos más masivos para garantizar que el programa aguante la carga de trabajo que esto implica, tal y como se verá en el próximo apartado.

Aparte de los cambios a nivel de arquitectura, la implementación del sistema de registros ha jugado un papel importante en la implementación del resto de cambios, ya que ha permitido depurar de manera efectiva el resto de fases del programa.

7 Validación

En este apartado se busca validar la nueva implementación del pipeline para asegurarse de que sea correcta y funcional. Así mismo, también se busca demostrar que los nuevos cambios introducidos aporten una mejora respecto a la implementación previa, ya sea en términos de coste o de tiempo.

Por tanto, este apartado resulta crucial para el desarrollo de este proyecto, ya que permitirá garantizar que el trabajo realizado ha aportado una mejora al sistema previo.

A continuación están detallados los diferentes puntos que se van a evaluar:

- Calidad de código: Se evalúa como ha quedado el código resultante en términos de líneas de código, comentarios, limpieza...
- Evaluación de rendimiento: Se evaluará a través de una serie de experimentos el coste y el tiempo de las diferentes fases del proyecto.

7.1. Calidad del código

En primer lugar, se va a comparar el número de líneas de código del repositorio entero. Aunque este no es un medidor definitivo de la calidad del código, ayuda a hacerse a la idea de la cantidad de código redundante que se ha eliminado.

Lenguaje	Ficheros	Código	Comentarios
Python	22	451	2173
Bourne Shell	11	257	922
Dockerfile	3	73	290
TOTAL	**36**	**781**	**3385**

Cuadro 4: Análisis del repositorio nuevo.

Lenguaje	Ficheros	Código	Comentarios
Python	34	1188	5138
Bourne Shell	32	568	2218
OCaml	2	174	1827
R	3	90	305
Markdown	4	0	269
Dockerfile	3	65	229
JSON	1	0	101
TOTAL	**79**	**2085**	**10087**

Cuadro 5: Análisis del repositorio antiguo.

Otra métrica a evaluar es la cantidad y calidad de los comentarios en el código. En la nueva implementación se han comentado todas las funciones con los parámetros que estas necesitan y los resultados que emiten. Además, se han comentado las diferentes partes del código de manera que sea lo más fácil de entender en cuanto a funcionamiento.

En la tabla 4 se puede observar un análisis de la cantidad de ficheros, las líneas de código y los comentarios del nuevo proyecto. En la tabla 5 se puede observar el mismo análisis pero respecto a la implementación previa. Como se puede observar, se ha reducido las líneas de código a 1/3 de las originales. Respecto a los comentarios, aunque puede parecer que hay una cantidad similar respecto al código total, un 23 %, en realidad los comentarios "reales" son muchos más en la nueva implementación que en la antigua, ya que en la anterior había muchos "comentarios" que eran simplemente secciones de código enteras comentadas para que no se ejecutasen. También se han reducido los lenguajes utilizados a los 3 principales:

1. **Python:** El lenguaje principal que coordina todo el programa.
2. **Shell:** Lenguaje usado en algunos de los *scripts* de alineamiento.
3. **Dockerfile:** Lenguaje usado para definir el entorno de ejecución en las funciones Lambda.

```
1 def create_iterdata_reducer(intermediate_keys: dict, distributed_indexes:
    Tuple[Tuple[str]], multipart_ids: Tuple[str],
2             multipart_keys: Tuple[str], pipeline_params: PipelineRun) ->
    Tuple[dict]:
3   """
4   Create the iterdata for the reduce stage.
5
6   Args:
7     intermediate_keys (Tuple[str]): Keys distributed by fasta split
8     distributed_indexes (Tuple[Tuple[str]]): Indexes that each reducer should
      process
9     multipart_ids (Tuple[str]): Multipart Upload IDs
10    multipart_keys (Tuple[str]): Multipart Upload Keys
11    pipeline_params (PipelineRun): Pipeline Parameters
12
13  Returns:
14    Tuple[dict]: Reduce stage iterdata
15  """
```

Código 26: Ejemplo de comentario en la implementación nueva

En la imagen 26 se puede apreciar el tipo de comentarios que se han puesto en todas las funciones. En comparación, la implementación previa no disponía casi nunca de este tipo de comentarios, y en caso de tenerlos eran como los vistos en la imagen 27. Otro añadido nuevo es el *Type Hinting*, el cual define el tipo de contenido que tiene cada variable. Aunque esto no es necesario en un lenguaje de tipado dinámico como es Python, ayuda al mantenimiento del código.

```
1 def map_alignment(id, fasta_chunk, fastq_chunk, storage):
2   """
3   map function to process fasta + fastq chunks to general sequence
    alignmments
4   final format: mpileup
5   """
```

Código 27: Ejemplo de comentario en la implementación antigua

7.2. Evaluación del rendimiento

En la sección de evaluación del rendimiento del proyecto, se presentarán los resultados obtenidos a partir de experimentos realizados para determinar el tiempo de ejecución y el coste asociado a las funciones cuando se ejecutan en la nube. En particular, se hablará sobre la fase de ejecución más costosa, la fase de *map*, así como la fase en la que se ha realizado una reimplementación completa, el particionador FASTA.

7.2.1. Particionador FASTA

Durante el desarrollo del nuevo particionador FASTA, se estimó que el tiempo de ejecución y, por lo tanto, el coste de las funciones serían considerablemente menores en comparación con la versión anterior. Esto se debe a que en lugar de generar particiones del fichero original, se crea un único fichero índice. Por lo tanto, la diferencia entre los dos particionadores se encuentra en el hecho de que el flujo de datos enviado al almacenamiento era mucho menor.

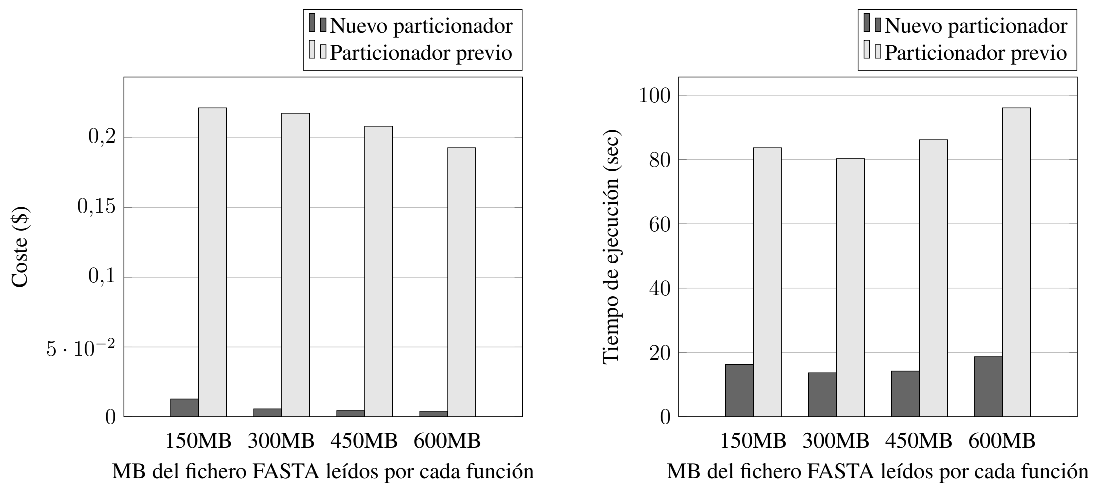

Figura 17: Fase de preprocesamiento: particiondado del fichero FASTA.

Como se puede observar en los diagramas (Figura 17), tanto el tiempo de ejecución como el costo del nuevo particionador son considerablemente menores en comparación con el particionador anterior. Como se mencionó anteriormente, al eliminar la subida de los fragmentos del fichero original al almacenamiento de objetos y cambiarlo por la subida de un fichero índice, se logró reducir el tiempo de ejecución y el costo total.

Por otro lado, se puede observar en los diagramas que a medida que aumenta el tamaño en megabytes (MB) que cada función lee del fichero original, el costo disminuye mientras que el tiempo de ejecución aumenta. Esto se debe a que, dado que el conjunto de datos es el mismo para todas las ejecuciones y lo único que varía es el tamaño de la partición del fichero original que cada función tiene, al aumentar el tamaño de la partición, se reduce el número de funciones lanzadas. Por lo tanto, al ejecutarse menos funciones, hay una menor paralelización, lo que lleva más tiempo realizar toda la ejecución, mientras que el costo disminuye, ya que el precio depende del número de funciones multiplicado por su tiempo de ejecución, entre otros factores, como el tamaño de la memoria asignada a cada función y el número de discos asignados.

7.2.2. *Fase map*

Con la eliminación del cuello de botella que generaba el *redis* en la fase de *map*, se teorizó que el tiempo de ejecución final y su costo disminuirían. Esto se debe a que las funciones lanzadas ya no necesitarían esperar a recibir un archivo índice generado en la fase de *map*. En su lugar, al adaptar dicha sección del código para utilizar funciones *Serverless* (como se propuso en la sección 5.3), evitando así la necesidad de utilizar *redis*, se esperaba observar una diferencia notable en el tiempo de ejecución y el costo de la fase de *map*.

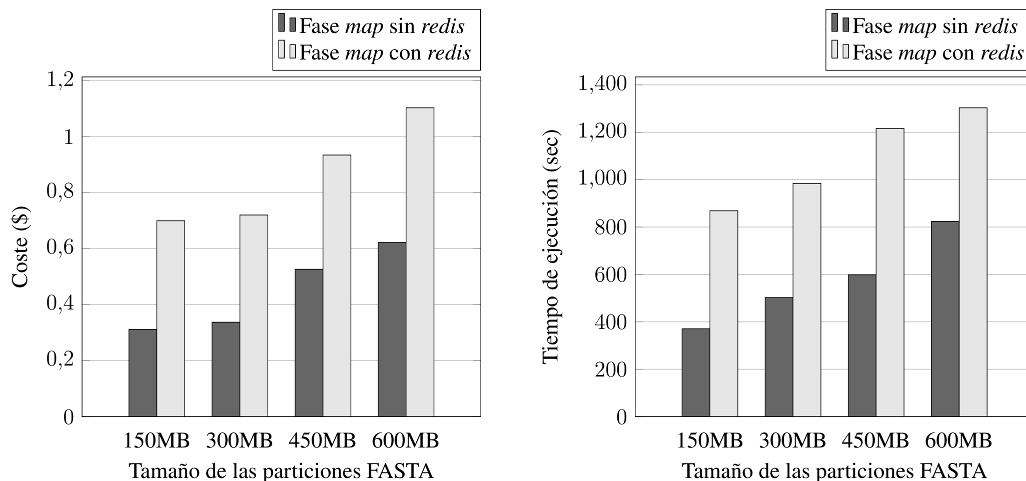

Figura 18: Comparativa con la anterior y actual fase *map*. El tamaño de las particiones FASTQ es fijo.

Al comparar los diagramas (Figura 18), se observa una reducción de casi el 50 % en el tiempo de ejecución y, por lo tanto, en el coste. Esto confirma que eliminar el uso de *redis* en la fase de *map* ha mejorado el tiempo y el coste total de la fase.

Después de completar el experimento anterior, surgió la idea de verificar si variar el tamaño de memoria y el número de CPUs en las subfases de la fase de *map* (*Gem*, *Mapper* y *Filter_mpileup*) podría mejorar el rendimiento general de la fase. Se planteó la posibilidad de que cada subfase pueda requerir un tamaño de memoria y un número de CPUs diferente para maximizar la eficiencia y el uso de los recursos en el sistema distribuido.

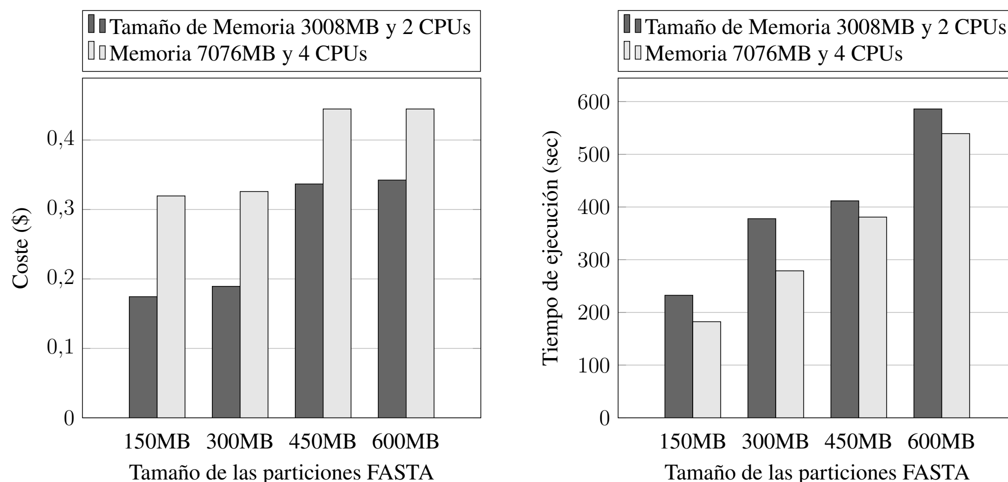

Figura 19: Fase *map*: *Gem*. El tamaño de las particiones FASTQ es fijo.

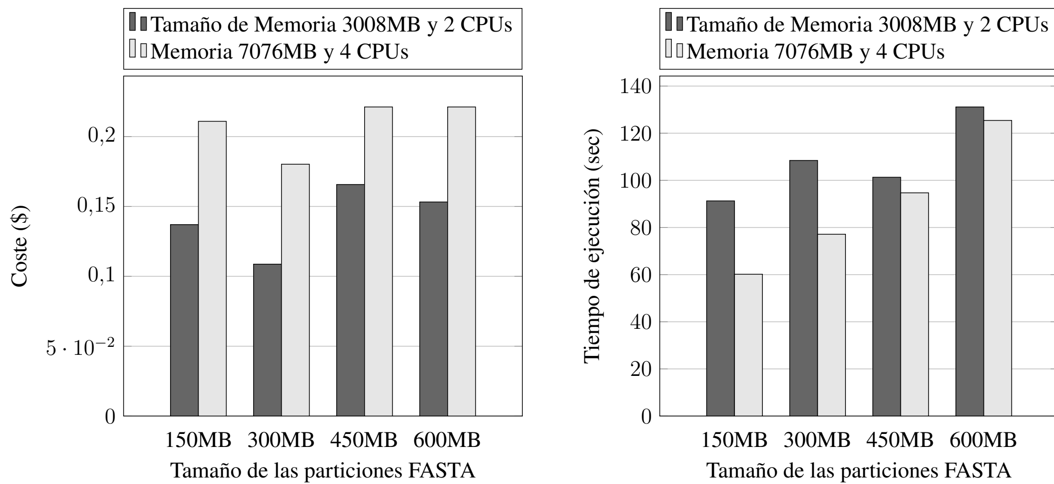

Figura 20: Fase *map*: *Mapper* (subfase 1). El tamaño de las particiones FASTQ es fijo.

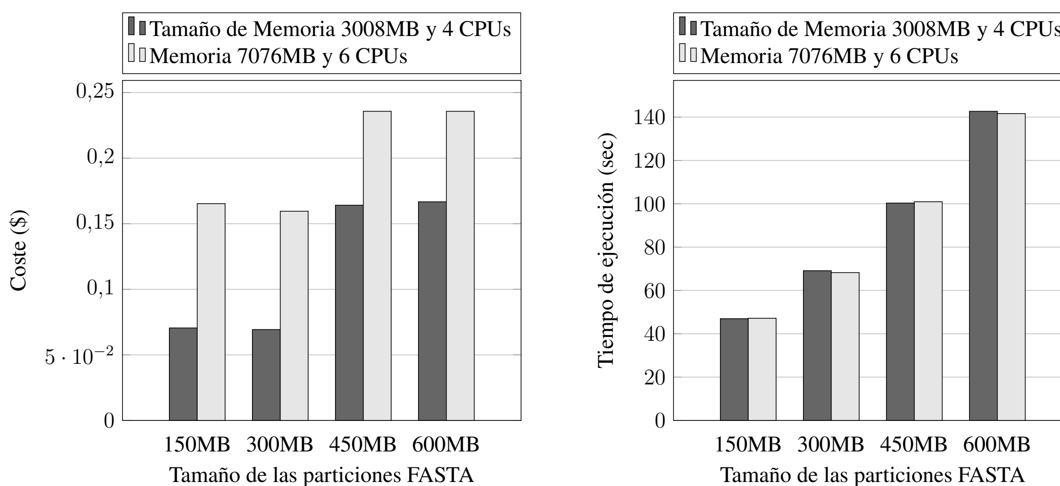

Figura 21: Fase *map*: *Filter_mpileup* (subfase 2). El tamaño de las particiones FASTQ es fijo.

Una vez realizadas las pruebas de las tres subfases (Figuras 19, 20 y 21), configurando los parámetros de memoria a 3008MB y 7076MB, y el número de CPUs a 2 y 4, se confirmaron dos cosas. Primero, cambiar el tamaño de memoria y el número de CPUs afecta el tiempo de ejecución y el coste final. Segundo, las diferentes subfases pueden ser configuradas con la misma configuración, ya que la diferencia en tiempo de ejecución y coste es mínima.

Además, se puede observar en los diagramas que cuanto más memoria y CPUs se utilizan, menor es el tiempo de ejecución y mayor es el coste. Esto se debe a que al aumentar la memoria y el número de CPUs, se puede procesar más trabajo en paralelo, lo que reduce el tiempo de ejecución. Sin embargo, al solicitar más recursos en la nube, el coste aumenta.

En este caso, dado que el tiempo de ejecución no varía significativamente entre 3008MB y 2 CPUs o 7076MB y 4 CPUs, la mejor configuración para minimizar el coste total en la fase de *map* sería utilizar un tamaño de memoria de 3008MB y 2 CPUs.

7.3. Prueba de escalabilidad

En esta sección se mostrarán los resultados de todas las llamadas a AWS Lambda y otros resultados adicionales que se realizaron con un gran volumen de datos de entrada. Esto se hizo para comprobar que partes de cada función tardan más en ejecutarse, ya que cada función está desglosada en diferentes trozos. El otro objetivo de este experimento es probar la escalabilidad general del nuevo programa.

7.3.1. Parámetros

Los parámetros utilizados para este experimento han estado los siguientes:

- **Fichero FASTA usado:** hg19.fa (3GB)
- **Tamaño de segmentos FASTA:** 150MB (20 segmentos en total)
- **Fichero FASTQ usado:** SRR15068323 (5.25GB)
- **Tamaño de segmentos FASTQ:** 150MB (35 segmentos en total)
- **Memoria general asignada:** 3GB
- **Memoria asignada a la fase de *Reduce*:** 8 GB

7.3.2. Fase de Map

En este apartado se van a evaluar las 4 subfases que forman la previamente explicada fase de *Map*.

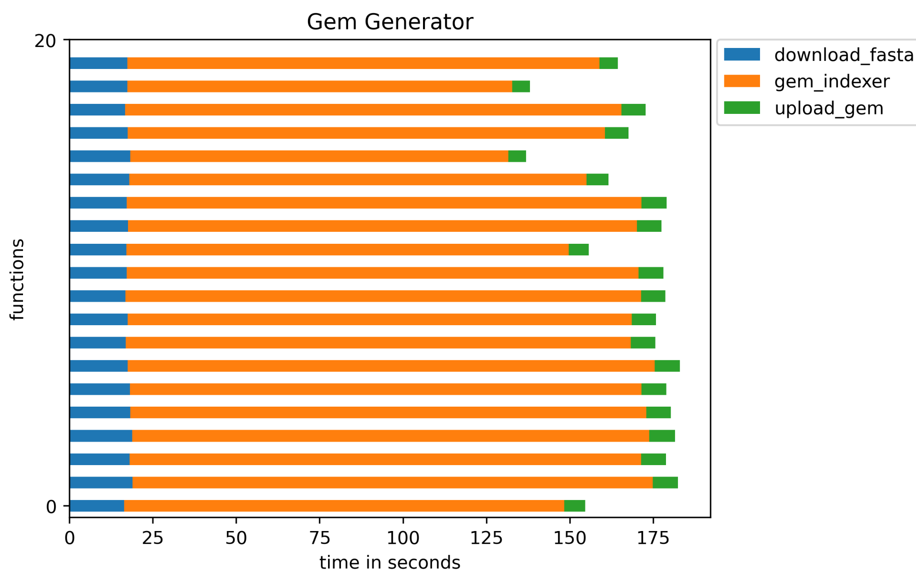

Figura 22: Generación del fichero GEM

En la figura 22 se pueden observar los tiempos de ejecución de las 20 funciones lanzadas para generar los ficheros GEM, una por cada segmento FASTA. Se puede observar que la fase de

descarga del segmento FASTA ocupa una parte importante, algo menos que luego subir el resultado. Esto resulta curioso, debido a que el fichero GEM suele ocupar aproximadamente 6 veces más que el segmento FASTA, y la velocidad de descarga y subida de las funciones es la misma. Esto es debido a que el segmento FASTA se obtiene a través de un mecanismo conocido como *Byte Range*, que permite descargar unos bytes concretos de un fichero. Esto es algo menos óptimo que una descarga normal, pero sigue siendo más rápido que descargar todo el fichero original. La mayor parte del tiempo de la función está dedicado a la computación del fichero GEM.

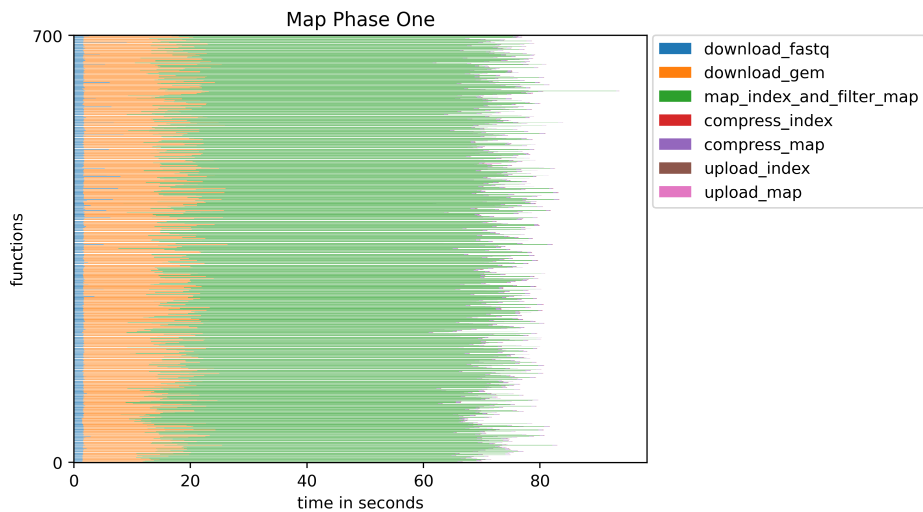

Figura 23: Alineamiento inicial

En la figura 23 se pueden observar las 700 funciones lanzadas para generar el primer alineamiento. Las nuevas partes del final, que consisten en subir a *Object Storage* los datos intermedios no suponen ningún problema, ya que consumen una cantidad de tiempo mínima. Los dos principales consumidores de tiempo son la descarga del fichero GEM y el alineamiento en sí, pero estos tienen poco margen de mejora.

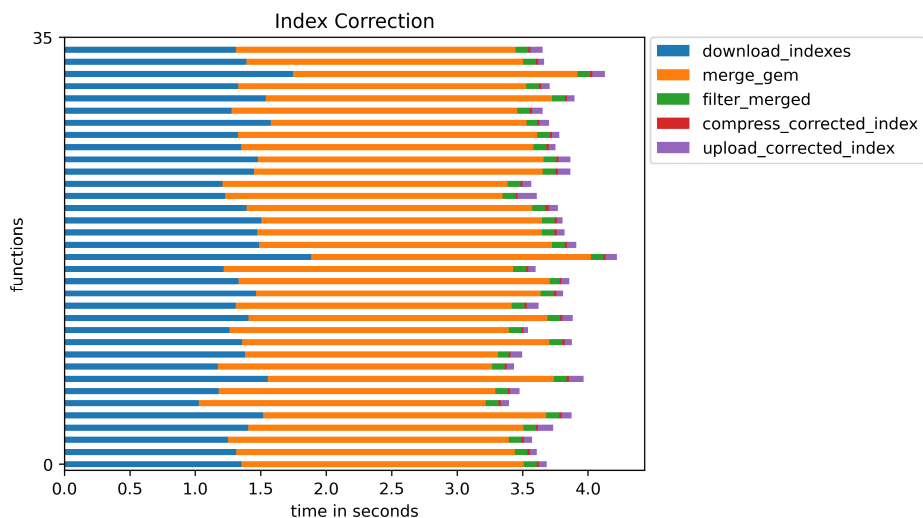

Figura 24: Corrección de los índices

En la figura 24 se muestra la nueva fase introducida para corregir los índices generados por la anterior fase. Como se puede observar, la cantidad de funciones lanzadas y el tiempo consumido son mínimos en comparación con las otras fases. Por tanto, dado que este experimento ha utilizado una gran cantidad de datos, se puede considerar que esta implementación ha escalado correctamente según lo previsto.

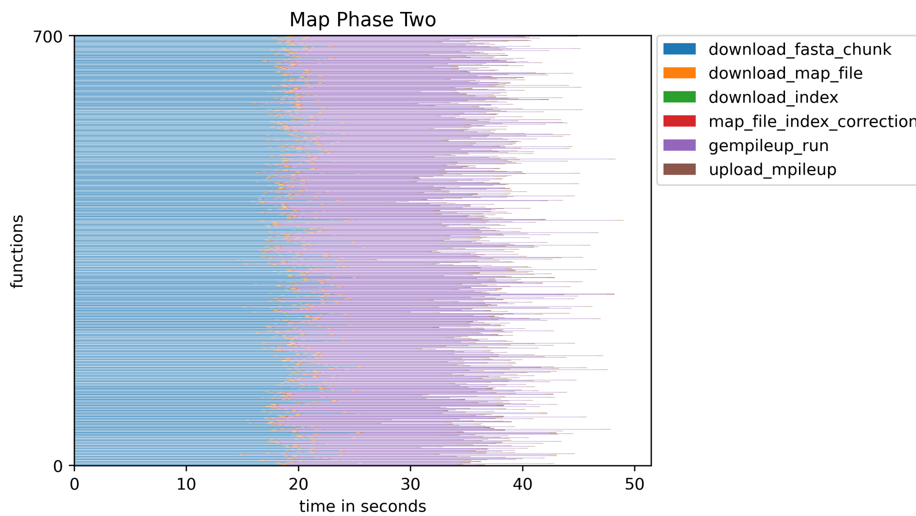

Figura 25: Aplicación del índice corregido y generación del fichero Mpileup

En la figura 25 se observa la última parte de esta fase, que consiste en filtrar los archivos intermedios y generar el archivo final. Como se puede observar, la mayor parte del tiempo consiste en descargar el segmento FASTA y en generar el archivo Mpileup. Las nuevas partes introducidas, que consisten en obtener los archivos intermedios de *Object Storage*, tienen un impacto mínimo en el tiempo de ejecución, por lo que se puede considerar que la nueva implementación que elimina el bloqueo entre funciones ha sido exitosa.

7.3.3. Fase de Reduce

En este apartado se van a evaluar las 2 subfases principales que forman la previamente explicada fase de *Reduce*.

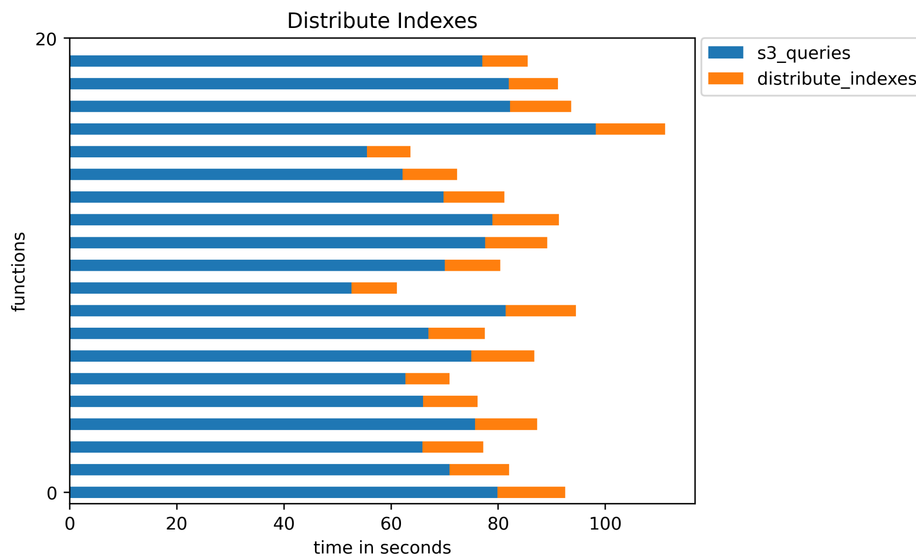

Figura 26: Distribución de índices

En la figura 26 se puede observar la fase de distribución de índices. Esta fase ha recibido pocos cambios respecto a su implementación anterior. Solo se ha implementado una pequeña mejora en la estructura de datos utilizada. Como se puede observar, la mayor parte del tiempo es tomada por la obtención de los datos a través de las peticiones *S3 Select*. Este se trata de una posible mejora posterior a la realización de este trabajo, la cual permitiría no solo mejorar los tiempos de descarga y costes, sino hacer más portable el programa, ya que este tipo de peticiones solo están disponibles en AWS.

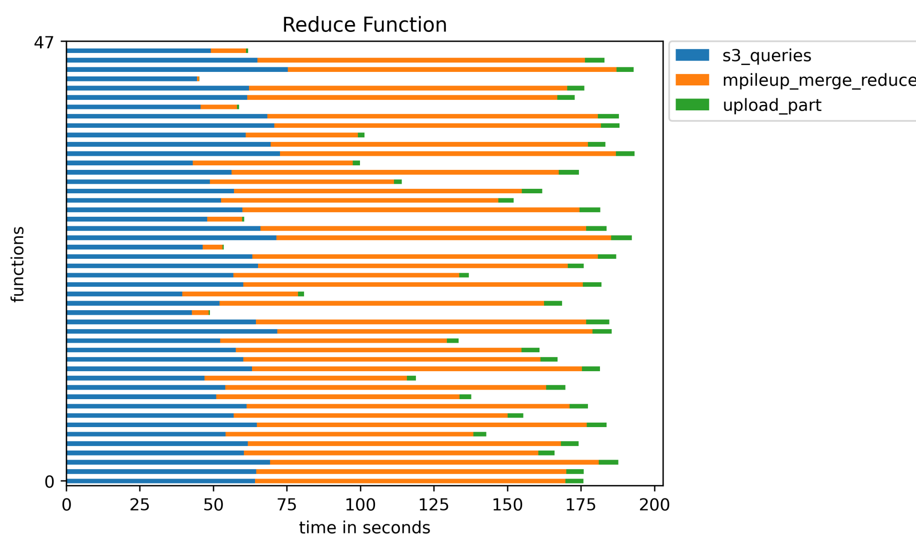

Figura 27: Reducción final

En la figura 27 se pueden observar las funciones lanzadas durante la reducción en sí. Igual que durante la distribución, solo se han implementado pequeños cambios a las estructuras de datos para así hacer un menor uso de disco. De igual manera, un posible cambio en el futuro sería la optimización de las consultas *S3 Select*, ya que estas siguen tomando una importante parte del tiempo de ejecución.

7.3.4. Resultados finales

En este apartado se van a analizar diferentes estadísticas generales resultantes de este experimento, como los tiempos por fase, las transferencias de datos y los costes.

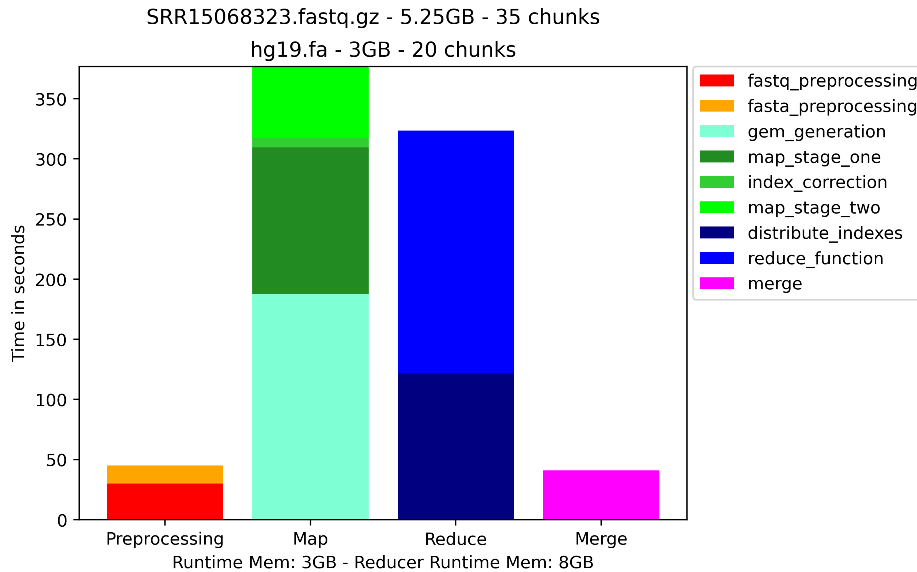

Figura 28: Tiempos por fase

En la figura 28 se puede observar el tiempo que ha tomado cada fase y subfase. Las 2 fases que toman más tiempo de ejecución son la generación del fichero GEM y la reducción. La generación del fichero GEM no es un problema, ya que este fichero solo debe ser generado una vez por fichero FASTA, y, por ende, en futuras ejecuciones donde solo cambie el fichero FASTQ no será necesario volver a generarlo gracias a que ahora se guarda este fichero en *Object Storage*. Respecto a la reducción, el tiempo de esta fase resulta fácilmente manipulable, ya que sería tan sencillo como asignar menos índices a cada reductor, de manera que se lanzarían más reductores que a su vez tardarían menos tiempo cada uno. Esto, sin embargo, tendría un impacto en el coste debido al funcionamiento de las consultas *S3 Select* (el coste de estas se explicará más adelante en este apartado).

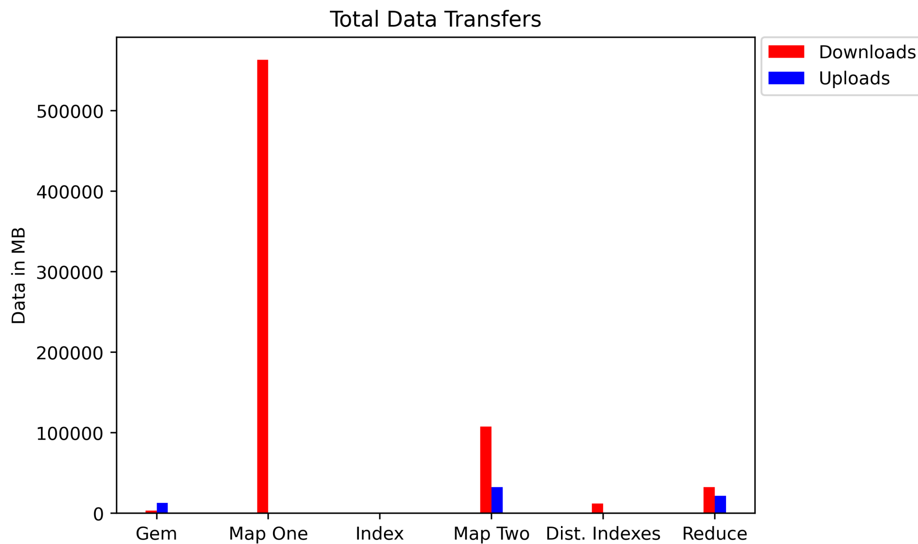

Figura 29: Transferencias de datos

En la figura 29 se pueden observar las transferencias de datos realizadas en total. La primera fase de alineamiento incurre en una increíble descarga de datos. Esto es debido a que se lanzan 700 funciones y cada una necesita descargar el fichero GEM, el cual pesa aproximadamente 6 veces más que el segmento FASTA, el cual se descarga en la fase "*Map Two*". Sin embargo, se puede observar que la nueva fase de corrección de índices apenas incurre en la transferencia de datos.

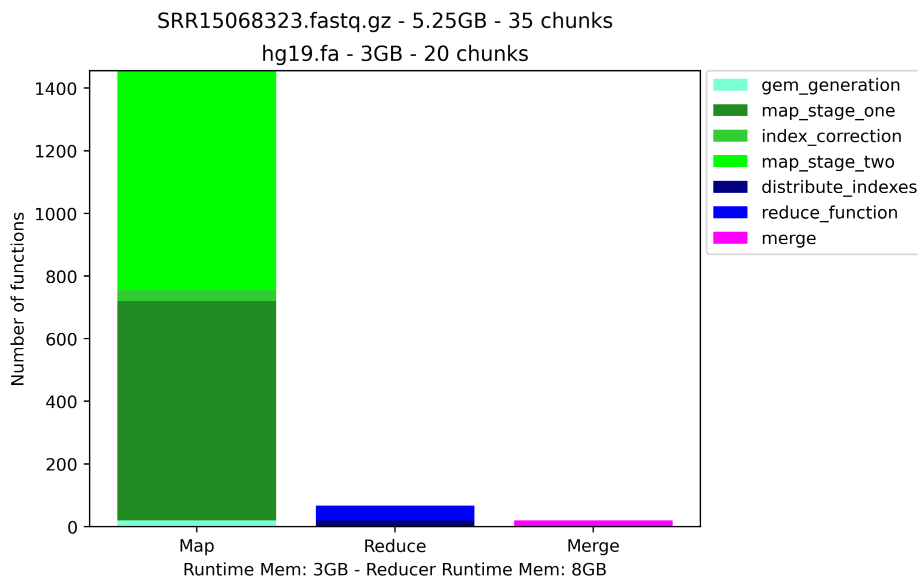

Figura 30: Funciones lanzadas

En la figura 30 se pueden observar el total de funciones lanzadas en cada fase. Las fases de alineamiento necesitan muchas más funciones, por lo que aunque en tiempo la fase de *Map* tenga una duración similar a la de *Reduce*, esta cuesta mucho más, tal y como se verá a continuación.

7.3.5. Análisis de costes

En la figura 6 se muestra la tabla de costes asociada a la ejecución de este experimento. Como se puede observar, las etapas más costosas son las de alineamiento: *map_one* (alineamiento inicial) y *map_two* (filtrado y mpileup). Por detrás también se acerca la etapa de *reduce*. Aunque es más reducido a la computación como tal, también es importante tener en cuenta los costes de las peticiones *S3 Select* tanto en la etapa de distribución de índices como en la de reducción.

Aunque es difícil realizar una comparación directa con el proyecto anterior debido a su baja escalabilidad, gracias a los datos proporcionados por Lucio Marcello sobre algunos experimentos antiguos se puede aproximar que tiene un tiempo de ejecución más o menos similar (dependiendo de los parámetros asignados), pero con una reducción de coste de aproximadamente un **15 %**.

Fases	Coste ($)
gem_generation	0.1708
map_one	2.5915
index_correction	0.0066
map_two	1.3271
dist_indexes	0.2224
reduce	0.9448
merge	0.0723
dist_indexes_select	0.0767
reduce_indexes_select	0.1872
total	**5.5994**

Cuadro 6: Costes por fase

Para calcular estos costes, se ha creado un script llamado "**cost_estimator.py**", el cual realiza el desglose de costes. Para calcular el coste de ejecución de las funciones Lambda, se ha hecho servir la tabla 31 como referencia. Una vez se sabe el coste por segundo de la memoria usada, es necesario sumar todos los segundos de computación de todas las funciones lanzadas y multiplicar ese tiempo por el coste. Respecto al disco usado, tiene un coste fijo de 0.0000000309$ por cada GB/s usado.

Memory (MB)	Price per 1ms
128	$0.0000000021
512	$0.0000000083
1024	$0.0000000167
1536	$0.0000000250
2048	$0.0000000333
3072	$0.0000000500
4096	$0.0000000667
5120	$0.0000000833
6144	$0.0000001000
7168	$0.0000001167
8192	$0.0000001333
9216	$0.0000001500
10240	$0.0000001667

Figura 31: Coste de las funciones Lambda según su memoria [8]

Respecto al coste de las operaciones *S3 Select*, cada una de estas tiene dos costes principales asociados, el de lectura y el de retorno de datos. Es decir, en un ejemplo donde el fichero original pesa 100MB y se retornan 5MB, habrá un coste asociado a los 100MB leídos y otro a los 5MB devueltos por la llamada. Estos dos costes están detallados en la tabla 7.

Tipo de coste	**Coste**
Datos leídos	0.002 $/GB
Datos devueltos	0.0007 $/GB

Cuadro 7: Tabla de costes de S3 Select.

7.4. Conclusiones

En resumen, como se ha podido observar en los experimentos realizados, los cambios hechos han mejorado el estado del código y de su rendimiento.

A nivel de calidad de código se han reducido las líneas de código totales a un 33 % de las originales y el código ha sido dividido y documentado con tal de hacerlo más mantenible y fácil de entender para otros programadores que decidan usar este proyecto.

Respecto a la arquitectura y el rendimiento del programa en sí, los nuevos cambios introducidos no solo han mejorado la escalabilidad a la hora de procesar un alto volumen de datos, sino que se ha reducido el coste de ejecución aproximadamente un 15 % manteniendo un tiempo de ejecución muy similar al proyecto anterior.

8 Futuro del proyecto

Con esto, concluimos nuestro trabajo en este proyecto. Sin embargo, es importante mencionar que todavía queda mucho futuro por delante.

Aún hay más optimizaciones que se pueden realizar, especialmente en lo que respecta a la transferencia de datos. Como se ha observado anteriormente, estas transferencias pueden ser masivas cuando se trabajan con un alto volumen de datos de entrada.

Además de mejorar estas transferencias de datos, también estamos buscando mejorar aún más el paralelismo en el programa en su conjunto para hacerlo más escalable.

Todo esto resulta complicado debido a la complejidad del Big Data y a los problemas asociados con él.

En cuanto a los resultados en sí, aún se pueden mejorar los *scripts* de alineamiento para obtener resultados más precisos. Tanto en la implementación previa como en la nueva, existe un pequeño porcentaje de discrepancias entre una ejecución completamente secuencial de las herramientas y la implementación distribuida.

También se pueden realizar más experimentos con conjuntos de datos aún más masivos para identificar las partes del proyecto que generan los mayores costos y tiempos de procesamiento, lo que nos permitirá planificar futuras mejoras y optimizaciones.

Además de todo esto, el proyecto también sienta las bases para otro tipo de implementaciones de herramientas de *variant calling* y herramientas en general de bioinformática.

9 Conclusión

En este Trabajo de Fin de Grado nos propusimos dos objetivos principales: la refactorización de un proyecto existente y su optimización. Tras muchos meses de trabajo, hemos logrado cumplir ambos objetivos, lo cual nos ha permitido mejorar nuestros conocimientos sobre la nube y el fenómeno conocido como Big Data.

Durante la tarea de refactorización resultó necesario asimilar un código de más de 10.000 líneas poco documentadas a lo largo de más de un mes de análisis. Se identificaron las diferentes fases, las decisiones tomadas sobre la arquitectura y, finalmente, se procedió a la reconstrucción. Esta refactorización se realizó prácticamente desde cero, utilizando diferentes técnicas y conceptos de ingeniería y calidad de software, lo que nos permitió reducir el código de alrededor de 10000 líneas a aproximadamente 3300, es decir, un tercio del original. Además, se ha documentado el código de este nuevo programa para facilitar futuros avances en el proyecto.

Respecto a la optimización, fue necesario comprender a la perfección el funcionamiento de las arquitecturas distribuidas, ya que el proyecto anterior presentaba errores de diseño que impedían un uso correcto de los recursos disponibles. Esto resultó complejo debido a la complejidad de la arquitectura distribuida y los grandes volúmenes de datos. Se optimizaron las transferencias y el manejo de datos, y además, se mejoró el código en sí mediante el uso de algoritmos y estructuras de datos más eficientes. Todas estas optimizaciones permitieron que el proyecto pueda escalar y procesar grandes volúmenes de datos de entrada, al tiempo que redujeron los costos en aproximadamente un 15 %.

Estos logros son de gran importancia, ya que son aspectos cruciales para el mantenimiento y la evolución de cualquier proyecto informático. No solo se mejoró la calidad del código, sino que también se sentaron las bases para un desarrollo futuro más eficiente y sostenible. Muchos de los conceptos aplicados en la refactorización del código fueron aprendidos en la asignatura de Técnicas Avanzadas de Programación, mientras que para la arquitectura del programa se aplicaron conceptos de la asignatura de Sistemas Distribuidos.

También es importante reconocer que todavía existe margen para mejorar este proyecto, aunque esas mejoras tienen una complejidad que ha quedado fuera de los objetivos de este trabajo.

Aunque nuestro trabajo se centró en la parte informática, es importante destacar que la herramienta desarrollada se enfoca en el campo de la genómica, y las mejoras realizadas permitirán un análisis de genomas no solo más rápido que los métodos tradicionales, sino también más económico.

Referencias

[1] *ADN*. URL: https://es.wikipedia.org/wiki/%C3%81cido_desoxirribonucleico (visitado 21-05-2023).

[2] *Alineamiento de secuencias*. URL: https://es.wikipedia.org/wiki/Alineamiento_de_secuencias (visitado 12-05-2023).

[3] *ARN*. URL: https://es.wikipedia.org/wiki/%C3%81cido_ribonucleico (visitado 21-05-2023).

[4] *ASCII*. URL: https://elcodigoascii.com.ar/ (visitado 21-04-2023).

[5] *AWS Lambda*. URL: https://aws.amazon.com/lambda/ (visitado 11-05-2023).

[6] *Cloudbutton*. URL: https://cloudbutton.eu/ (visitado 23-02-2023).

[7] *Cloudlab*. URL: https://cloudlab.urv.cat/ (visitado 23-02-2023).

[8] *Costes de las funciones Lambda*. URL: https://aws.amazon.com/lambda/pricing/ (visitado 25-05-2023).

[9] *Costes de las peticiones S3 Select*. URL: https://blog.clairvoyantsoft.com/accelerate-s3-data-querying-performance-with-s3-select-ff84dde1b8da (visitado 21-05-2023).

[10] *D2.3 CloudButton Architecture Specs and Early Prototypes*. URL: https://cloudbutton.eu/docs/deliverables/CloudButton_D2.3_Public.pdf (visitado 21-03-2023).

[11] *Docker*. URL: https://www.docker.com/ (visitado 05-04-2023).

[12] *Estructura y función del ADN y de los genes. I Tipos de alteraciones de la función del gen por mutaciones*. URL: https://www.elsevier.es/es-revista-medicina-familia-semergen-40-articulo-estructura-funcion-del-adn-genes--S1138359310000596 (visitado 21-05-2023).

[13] *Ficheros FASTA*. URL: https://en.wikipedia.org/wiki/FASTA_format (visitado 24-03-2023).

[14] *Ficheros FASTQ*. URL: https://en.wikipedia.org/wiki/FASTQ_format (visitado 24-03-2023).

[15] *Function as a Service*. URL: https://en.wikipedia.org/wiki/Function_as_a_service (visitado 28-02-2023).

[16] *Genómica*. URL: https://www.genome.gov/es/genetics-glossary/Genomica (visitado 12-05-2023).

[17] *Genómica*. URL: https://es.wikipedia.org/wiki/Gen%C3%B3mica (visitado 12-05-2023).

[18] *Gztool*. URL: https://manpages.ubuntu.com/manpages/impish/man1/gztool.1.html (visitado 27-03-2023).

[19] *Librería abierta de Redis*. URL: https://redis.io/ (visitado 26-04-2023).

[20] *Librería SAM-Tools*. URL: https://github.com/smarco/gem3-mapper (visitado 26-04-2023).

[21] *Lithops Middleware*. URL: https://lithops-cloud.github.io/ (visitado 04-04-2023).

[22] *Mpileup Format*. URL: https://en.wikipedia.org/wiki/Pileup_format (visitado 24-04-2023).

[23] *Pyfaidx*. URL: https://pypi.org/project/pyfaidx/ (visitado 10-04-2023).

[24] *Sequence Read Archive*. URL: https://www.ncbi.nlm.nih.gov/sra (visitado 26-04-2023).

[25] *Top Cloud Providers*. URL: https://allcode.com/cloud-providers/ (visitado 27-03-2023).

Anexo

A Fichero pipeline.py

A.1. Función Run_pipeline

```
1 def run_pipeline(self):
2     """
3     Execute all pipeline steps in order
4     """
5     stats: Stats = self.pipeline_stats()
6     stats.timer_start('pipeline')
7     # PreProcess Stage
8     if self.parameters.skip_prep is False:
9       preprocessStat = self.preprocess()
10
11    # Map Stage
12    if self.parameters.skip_map is False:
13      mapper_output, alignReadsStat = self.align_reads()
14
15    # Reduce Stage
16    if self.parameters.skip_reduce is False:
17      reduceStat = self.reduce(mapper_output)
18    stats.timer_stop('pipeline')
19
20    if self.parameters.skip_prep is False:
21      stats.store_dictio(preprocessStat.get_stats(), "preprocess_phase", "
    pipeline")
22    if self.parameters.skip_map is False:
23      stats.store_dictio(alignReadsStat.get_stats(), "alignReads_phase", "
    pipeline")
24    if self.parameters.skip_reduce is False:
25      stats.store_dictio(reduceStat.get_stats(), "reduce_phase", "pipeline")
26
27    if self.parameters.log_stats:
28      stats.load_stats_to_json(self.parameters.storage_bucket, self.
    parameters.log_stats_name)
```

Código 28: Función *run_pipeline*.

B Fichero preprocess_fasta.py

B.1. Función generate_faidx_from_s3

```
1 def generate_faidx_from_s3(pipeline_params: PipelineRun, lithops: Lithops,
    stats):
2   fasta_head = try_head_object(lithops.storage, pipeline_params.fasta_path.
    bucket, pipeline_params.fasta_path.key)
3   if fasta_head is None:
4   raise Exception(f'fasta file with key {pipeline_params.fastq_path} does not
    exists')
5
6   faidx_head = try_head_object(lithops.storage, pipeline_params.
    storage_bucket, pipeline_params.faidx_key)
7   if faidx_head is not None:
8   logger.debug('Faidx for %s found', pipeline_params.fasta_path.stem)
9   num_sequences = int(faidx_head['x-amz-meta-num_sequences'])
10  else:
11  logger.info('Faidx for %s not found, generating fasta index file',
    pipeline_params.fasta_path.stem)
```

```
12      stats.timer_start('generate_fasta_index')
13      fasta_head = lithops.storage.head_object(pipeline_params.fasta_path.
       bucket, pipeline_params.fasta_path.key)
14      fasta_file_sz = int(fasta_head['content-length'])
15      chunk_size = math.ceil(fasta_file_sz / pipeline_params.fasta_chunks)
16
17      map_iterdata = [{'fasta_path': pipeline_params.fasta_path}] *
       pipeline_params.fasta_chunks
18      extra_args = {'chunk_size': chunk_size,
19              'fasta_size': fasta_file_sz,
20              'num_chunks': pipeline_params.fasta_chunks}
21      extra_env = {'BUCKET': pipeline_params.storage_bucket, 'FAIDX_KEY':
       pipeline_params.faidx_key}
22      num_sequences = lithops.invoker.map_reduce(
23              map_function=create_index_chunked,
24              map_iterdata=map_iterdata,
25              extra_args=extra_args, extra_env=extra_env,
26              reduce_function=reduce_chunked_indexes)
27      stats.timer_stop('generate_fasta_index')
28
29      logger.info('Generated faidx for FASTA %s (read %d sequences)',
       pipeline_params.fasta_path.stem, num_sequences)
30
31    stats.store_size_data('size_fasta_file', lithops.storage.head_object(bucket
       =pipeline_params.fasta_path.bucket, key=pipeline_params.fasta_path.key)['
       content-length'])
32    stats.store_size_data('size_index_file', lithops.storage.head_object(bucket
       =pipeline_params.storage_bucket, key=pipeline_params.faidx_key)['content-
       length'])
33    logger.info('Read %d sequences from FASTA %s', num_sequences,
       pipeline_params.fasta_path.stem)
34    return num_sequences
```

Código 29: Función *generate_faidx_from_s3*.

B.2. Función create_index_chunked

```
1  def create_index_chunked(storage, id, fasta_path, chunk_size, fasta_size,
       num_chunks):
2    """
3    Lithops callee function (map)
4    Generate partial index of a chunk of a FASTA file
5    """
6    min_range = id * chunk_size
7    max_range = int(fasta_size) if id == num_chunks - 1 else (id + 1) *
       chunk_size
8    data = storage.get_object(bucket=fasta_path.bucket, key=fasta_path.key,
9                   extra_get_args={'Range': f'bytes={min_range}-{max_range - 1}'
       }).decode('utf-8')
10   content = []
11   # If it were '>' it would also find the ones inside the head information
12   ini_heads = list(re.finditer(r"\n>", data))
13   heads = list(re.finditer(r">.+\n", data))
14
15   if ini_heads or data[0] == '>':  # If the list is not empty or there is >
       in the first byte
16     first_sequence = True
17     prev = -1
18     for m in heads:
19       start = min_range + m.start()
20       end = min_range + m.end()
21       if first_sequence:
```

```
22        first_sequence = False
23        if id > 0 and start - 1 > min_range:
24          # If it is not the worker of the first part of the file and in
    addition it
25          # turns out that the partition begins in the middle of the base of
    a sequence.
26          # (start-1): avoid having a split sequence in the index that only
    has '\n'.
27          match_text = list(re.finditer('.*\n', data[0:m.start()]))
28          if match_text and len(match_text) > 1:
29            text = match_text[0].group().split(' ')[0].replace('\n', '')
30            offset = match_text[1].start() + min_range
31            # >> offset_head offset_bases_split ^first_line_before_space_or_\
    n^
32            content.append(f">> <Y> {str(offset)} ^{text}^")  # Split
    sequences
33          else:
34            # When the first header found is false, when in a split stream
    there is a split header
35            # that has a '>' inside (ex: >tr|...o-alpha-(1->5)-L-e...\n)
36            first_sequence = True
37      if prev != start:  # When if the current sequence base is not empty
38        # name_id offset_head offset_bases
39        id_name = m.group().replace('\n', '').split(' ')[0].replace('>', '')
40        content.append(f"{id_name} {str(start)} {str(end)}")
41      prev = end
42
43    # Check if the last head of the current one is cut. (ini_heads[-1].start
    () + 1): ignore '\n'
44    if len(heads) != 0 and len(ini_heads) != 0 and ini_heads[-1].start() + 1
    > heads[-1].start():
45      last_seq_start = ini_heads[-1].start() + min_range + 1  # (... + 1):
    ignore '\n'
46      text = data[last_seq_start - min_range::]
47      # [<->|<_>]name_id_split offset_head
48      # if '<->' there is all id
49      content.append(f"{'<-' if ' ' in text else '<_'}{text.split(' ')[0]} {
    str(last_seq_start)}")
50  return content
```

Código 30: Función *create_index_chunked*.

B.3. Función reduce_chunked_indexes

```
1 def reduce_chunked_indexes(results, storage):
2   """
3   Lithops callee function (reduce)
4   Reduce partial indexes of chunked FASTA file into a single index for the
      entire file
5   """
6   bucket = os.getenv('BUCKET')
7   faidx_key = os.getenv('FAIDX_KEY')
8
9   if len(results) > 1:
10    results = list(filter(None, results))
11    for i, list_seq in enumerate(results):
12      if i > 0:
13        list_prev = results[i - 1]
14        # If it is not empty the current and previous dictionary
15        if list_prev and list_seq:
16          param = list_seq[0].split(' ')
17          seq_prev = list_prev[-1]
```

```
18          param_seq_prev = seq_prev.split(' ')
19
20          # If the first sequence is split
21          if '>>' in list_seq[0]:
22            if '<->' in seq_prev or '<_>' in seq_prev:
23              # If the split was after a space, then there is all id
24              if '<->' in seq_prev:
25                name_id = param_seq_prev[0].replace('<->', '')
26              else:
27                name_id = param_seq_prev[0].replace('<_>', '') + param[3].
   replace('^', '')
28              list_seq[0] = rename_sequence(list_seq[0], param, name_id,
   param_seq_prev[1], param[2])
29            else:
30              list_seq[0] = seq_prev
31            list_prev.pop() # Remove previous sequence
32
33    num_sequences = reduce(lambda x, y: x + y, map(lambda r: len(r), results))
34    index_result = bz2.compress(b'\n'.join(s.encode('utf-8') for s in itertools
     .chain(*results)))
35    s3 = storage.get_client()
36    s3.put_object(Bucket=bucket, Key=faidx_key, Body=index_result, Metadata={'
     num_sequences': str(num_sequences)})
37    return num_sequences
38
39 def rename_sequence(sequence, param, name_id, offset_head, offset_base):
40    sequence = sequence.replace(f' {param[3]}', '')  # Remove 3rt param
41    sequence = sequence.replace(f' {param[2]} ', f' {offset_base} ')  #
     offset_base -> offset_base
42    sequence = sequence.replace(' <Y> ', f' {offset_head} ')  # Y -->
     offset_head
43    sequence = sequence.replace('>> ', f'{name_id} ')  # '>>' -> name_id
44    return sequence
```

Código 31: Función *reduce_chunked_indexes*.

B.4. Función get_fasta_byte_ranges

```
1 def get_fasta_byte_ranges(pipeline_params: PipelineRun, lithops: Lithops,
     num_sequences):
2    '''
3    Generate chunks according to the number of fasta chunks requested
4    '''
5    fasta_chunks = []
6    fasta_file_head = lithops.storage.head_object(pipeline_params.fasta_path.
     bucket, pipeline_params.fasta_path.key)
7    fasta_file_sz = int(fasta_file_head['content-length'])
8    fa_chunk_size = int(fasta_file_sz / int(pipeline_params.fasta_chunks))
9    compressed_faidx = lithops.storage.get_object(pipeline_params.
     storage_bucket, pipeline_params.faidx_key)
10   faidx = bz2.decompress(compressed_faidx).decode('utf-8').split('\n')
11   i = j = 0
12   min = fa_chunk_size * j
13   max = fa_chunk_size * (j + 1)
14   while max <= fasta_file_sz:
15     # Find first full/half sequence of the chunk
16     if int(faidx[i].split(' ')[1]) <= min < int(faidx[i].split(' ')[2]):  #
     In the head
17       fa_chunk = {'offset_head': int(faidx[i].split(' ')[1]), 'offset_base':
     int(faidx[i].split(' ')[2])}
18     elif i == num_sequences - 1 or min < int(faidx[i + 1].split(' ')[1]):  #
     In the base
```

```
19       fa_chunk = {'offset_head': int(faidx[i].split(' ')[1]), 'offset_base':
    min}
20     elif i < num_sequences:
21       i += 1
22       while i + 1 < num_sequences and min > int(faidx[i + 1].split(' ')[1]):
23         i += 1
24       if min < int(faidx[i].split(' ')[2]):
25         fa_chunk = {'offset_head': int(faidx[i].split(' ')[1]), 'offset_base'
    : int(faidx[i].split(' ')[2])}
26       else:
27         fa_chunk = {'offset_head': int(faidx[i].split(' ')[1]), 'offset_base'
    : min}
28     else:
29       raise Exception('ERROR: there was a problem getting the first byte of a
     fasta chunk.')
30 
31     # Find last full/half sequence of the chunk
32     if i == num_sequences - 1 or max < int(faidx[i + 1].split(' ')[1]):
33       fa_chunk['last_byte'] = max - 1 if fa_chunk_size * (j + 2) <=
    fasta_file_sz else fasta_file_sz - 1
34     else:
35       if max < int(faidx[i + 1].split(' ')[2]):  # Split in the middle of
    head
36         fa_chunk['last_byte'] = int(faidx[i + 1].split(' ')[1]) - 1
37         i += 1
38       elif i < num_sequences:
39         i += 1
40         while i + 1 < num_sequences and max > int(faidx[i + 1].split(' ')[1])
    :
41           i += 1
42         fa_chunk['last_byte'] = max - 1
43       else:
44         raise Exception('ERROR: there was a problem getting the last byte of
    a fasta chunk.')
45 
46     fa_chunk['chunk_id'] = j
47     fasta_chunks.append(fa_chunk)
48     j += 1
49     min = fa_chunk_size * j
50     max = fa_chunk_size * (j + 1)
51   return fasta_chunks
```

Código 32: Función *get_fasta_byte_ranges*.

B.5. Función get_fasta_byte_ranges

```
1 def prepare_fastq_chunks(pipeline_params: PipelineRun, lithops: Lithops):
2   """
3   Calculate fastq byte ranges for chunks of a pipeline run, generate gzip
     index if needed
4   """
5   subStat = Stats()
6   subStat.timer_start('prepare_fastq_chunks')
7   if pipeline_params.fastq_path is not None:
8     res = generate_fastqgz_index_from_s3(pipeline_params, lithops)
9     num_lines = res[0]
10    subStat.store_dictio(res[1], 'generating_index_fastq_from_s3')
11  elif pipeline_params.fastq_sra is not None:
12    # TODO implement get fastq file from sra archive
13    raise NotImplementedError()
14  else:
15    raise Exception('fastq reference required')
```

```
16
17   # Split by number of reads per worker (each read is composed of 4 lines)
18   assert (num_lines % 4) == 0, 'fastq file total number of lines is not
       multiple of 4!'
19   num_reads = num_lines // 4
20   reads_batch = ceil(num_reads / pipeline_params.fastq_chunks)
21   read_pairs = [(reads_batch * i, (reads_batch * i) + reads_batch) for i in
       range(pipeline_params.fastq_chunks)]
22
23   # Convert read pairs back to line numbers (starting in 1)
24   line_pairs = [((l0 * 4) + 1, (l1 * 4) + 1) for l0, l1 in read_pairs]
25
26   # Adjust last pair for num batches not multiple of number of total reads (
       last batch will have fewer lines)
27   if line_pairs[-1][1] > num_lines:
28     l0, _ = line_pairs[-1]
29     line_pairs[-1] = (l0, num_lines + 1)
30
31   # Get byte ranges from line pairs using GZip index
32   process, compare if it is faster invoking remote lambda
33   byte_ranges = get_ranges_from_line_pairs(pipeline_params, lithops,
       line_pairs, subStat)
34   chunks = [{'chunk_id': i, 'line_0': line_0, 'line_1': line_1, 'range_0':
       range_0, 'range_1': range_1} for i, ((line_0, line_1), (range_0, range_1)
       ) in enumerate(zip(line_pairs, byte_ranges))]
35   subStat.timer_stop('prepare_fastq_chunks')
36   return chunks, subStat
```

Código 33: Función *get_fasta_byte_ranges*.

B.6. Funciones generate_fastqgz_index_from_s3 y generate_idx_from_gzip

```
1 def generate_fastqgz_index_from_s3(pipeline_params: PipelineRun, lithops:
      Lithops) -> int:
2   """
3   Generate gzip index file for FASTQ in storage if necessary, returns number
       of lines of FASTQ file
4   """
5   # Check if fastq file exists
6   fastq_head = try_head_object(lithops.storage, pipeline_params.fastq_path.
       bucket, pipeline_params.fastq_path.key)
7   if fastq_head is None:
8     raise Exception(f'fastq file with key {pipeline_params.fastq_path} does
       not exists')
9
10  # Check if fastqgz index file exists
11  index_key, tab_key = pipeline_params.fastqgz_idx_keys
12  fastq_idx_head = try_head_object(lithops.storage,
13                   pipeline_params.fastq_path.bucket, index_key)
14  fastq_tab_head = try_head_object(lithops.storage,
15                   pipeline_params.fastq_path.bucket, tab_key)
16  dictio = {}
17  if fastq_idx_head is None or fastq_tab_head is None:
18    # Generate gzip index file for compressed fastq input
19    logger.info('Generating gzip index file for FASTQ %s', pipeline_params.
       fastq_path.stem)
20    res = lithops.invoker.call(generate_idx_from_gzip, (pipeline_params,
       pipeline_params.fastq_path))
21    total_lines = res[0]
22    dictio = res[1]
23  else:
24    # Get total lines from header metadata
```

```
25      logger.debug('Fastqz index for %s found', pipeline_params.fastq_path.stem
        )
26      total_lines = int(fastq_tab_head['x-amz-meta-total_lines'])
27    logger.info('Read %d sequences from FASTQ %s', total_lines / 4,
        pipeline_params.fastq_path.stem)
28
29    return total_lines, dictio
30
31  def generate_idx_from_gzip(pipeline_params: PipelineRun, gzip_file_path:
        S3Path, storage: lithops.Storage):
32    """
33    Lithops callee function
34    Create index file from gzip archive using gztool (https://github.com/
        circulosmeos/gztool)
35    """
36    gztool = get_gztool_path()
37    tmp_index_file_name = tempfile.mktemp()
38    s3 = storage.get_client()
39    stats = Stats()
40    try:
41      stats.timer_start('get_fastq_file')
42      res = s3.get_object(Bucket=gzip_file_path.bucket, Key=gzip_file_path.key)
43      stats.timer_stop('get_fastq_file')
44      stats.store_size_data('size', storage.head_object(gzip_file_path.bucket,
        gzip_file_path.key)['content-length'], 'fastq_file')
45      data_stream = res['Body']
46      force_delete_local_path(tmp_index_file_name)
47      t0 = time.perf_counter()
48
49      # Create index and save to tmp file
50      index_proc = subprocess.Popen([gztool, '-i', '-x', '-I',
        tmp_index_file_name],
51                    stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=
        subprocess.PIPE)
52      chunk = data_stream.read(CHUNK_SIZE)
53      while chunk != b"":
54        index_proc.stdin.write(chunk)
55        chunk = data_stream.read(CHUNK_SIZE)
56      if hasattr(data_stream, 'close'):
57        data_stream.close()
58
59      stdout, stderr = index_proc.communicate()
60      if index_proc.returncode > 0:
61        logger.debug(stdout.decode('utf-8'))
62        logger.debug(stderr.decode('utf-8'))
63        raise Exception('Error creating gz index')
64
65      # Generate list of access windows from index
66      proc = subprocess.run([gztool, '-ell', '-I', tmp_index_file_name], check=
        True,
67                 capture_output=True, text=True)
68      output = proc.stdout
69
70      # Store index binary file
71      index_key, tab_key = pipeline_params.fastqgz_idx_keys
72      stats.timer_start('upload_index_binary_fastq')
73      s3.upload_file(Filename=tmp_index_file_name, Bucket=pipeline_params.
        storage_bucket, Key=index_key)
74      stats.timer_stop('upload_index_binary_fastq')
75      stats.store_size_data('size', storage.head_object(bucket=pipeline_params.
        storage_bucket, key=index_key)['content-length'], 'index_binary_fastq')
76
77      # Get the total number of lines
```

```
78     total_lines: int = int(RE_NUMS.findall(RE_NLINES.findall(output).pop()).
      pop())
79     logger.debug('Indexed gzipped text file with %d total lines', total_lines
      )
80     t1 = time.perf_counter()
81     logger.debug('Index generated in %.3f seconds', t1 - t0)
82 
83     # Generator function that parses output to avoid copying all window data
      as lists
84     def _lines_generator():
85       for f in RE_WINDOWS.finditer(output):
86         nums = [int(n) for n in RE_NUMS.findall(f.group())]
87         yield nums
88 
89     # Generate data frame that stores gzip index windows offsets
90     df = pd.DataFrame(_lines_generator(), columns=['window', 'compressed_byte
      ', 'uncompressed_byte', 'line_number', 'window_size', 'window_offset'])
91     df.set_index(['window'], inplace=True)
92 
93     # Store data frame as parquet
94     out_stream = io.BytesIO()
95     df.to_parquet(out_stream, engine='pyarrow')
96     out_stream.seek(0)
97     stats.timer_start('put_data_frame_parquet')
98     s3.put_object(Bucket=pipeline_params.storage_bucket, Key=tab_key, Body=
      out_stream, Metadata={'total_lines': str(total_lines)})
99     stats.timer_stop('put_data_frame_parquet')
100    stats.store_size_data('size', storage.head_object(pipeline_params.
      storage_bucket, tab_key)['content-length'], 'put_data_frame_parquet')
101    return total_lines, stats.get_stats()
102  finally:
103    force_delete_local_path(tmp_index_file_name)
```

Código 34: Función *generate_fastqgz_index_from_s3* y *generate_idx_from_gzip*.

B.7. Función get_ranges_from_line_pairs

```
1 def get_ranges_from_line_pairs(pipeline_params: PipelineRun, lithops: Lithops
    , pairs: List[Tuple[int, int]], stats):
2   _, tab_key = pipeline_params.fastqgz_idx_keys
3   stats.timer_start('get_data_frame_parquet')
4   meta_obj_body = lithops.storage.get_object(bucket=pipeline_params.
    storage_bucket, key=tab_key)
5   stats.timer_stop('get_data_frame_parquet')
6   stats.store_size_data('size', lithops.storage.head_object(pipeline_params.
    storage_bucket, tab_key)['content-length'], 'get_data_frame_parquet')
7 
8   meta_buff = io.BytesIO(meta_obj_body)
9   meta_buff.seek(0)
10  df = pd.read_parquet(meta_buff)
11  line_indexes = df['line_number'].to_numpy()
12  num_windows = df.shape[0]
13 
14  head_fastq = lithops.storage.head_object(bucket=pipeline_params.fastq_path.
    bucket,
15                        key=pipeline_params.fastq_path.key)
16  fastqgz_sz = int(head_fastq['content-length'])
17 
18  byte_ranges = [None] * len(pairs)
19  for i, (line_0, line_1) in enumerate(pairs):
20    # Find the closest window index for line_0
21    window_head_idx = (np.abs(line_indexes - line_0)).argmin()
```

```
22     # Check if window line entry pont is past requested line_0, if so, get
      previous window
23     window_head_line = df.iloc[window_head_idx]['line_number']
24     if window_head_line > line_0:
25       window_head_idx = window_head_idx - 1
26     # Get offset in compressed archive for window 0
27     window0_offset = df.iloc[window_head_idx]['compressed_byte']
28
29     # Find the closest window index for line_0
30     widow_tail_idx = (np.abs(line_indexes - line_1)).argmin()
31     # Check if window line entry pont is before requested line_1, if so, get
      next window
32     window_tail_line = df.iloc[widow_tail_idx]['line_number']
33     if window_tail_line < line_1:
34       widow_tail_idx = widow_tail_idx + 1
35     if widow_tail_idx >= num_windows:
36       # Adjust offset for lines inside last window, use end of compressed
      archive for 2nd offset
37       window1_offset = fastqgz_sz
38     else:
39       window1_offset = df.iloc[widow_tail_idx]['compressed_byte']
40
41     byte_ranges[i] = (window0_offset, window1_offset)
42
43   return byte_ranges
```

Código 35: Función *get_ranges_from_line_pairs*.

C Fichero stats.py

C.1. Clase stats

```
1  class Stats:
2    def __init__(self):
3      self.__tmp_registrer={}
4      self.__stats={}
5
6    def timer_start(self, script, extra_time=None):
7      if script in self.__tmp_registrer:
8        print(f'WARNING: the counter of the timer \"{script}\" was already
      running, it will restart.')
9      elif script in self.__stats and "execution_time" in self.__stats[script]:
10       raise Exception(f'The timer of \"{script}\" already existed, choose
      another name or remove the existing timer first.')
11
12     self.__tmp_registrer[script] = time.perf_counter() if extra_time is None
      else extra_time + time.perf_counter()
13
14   def timer_stop(self, script):
15     end_time = time.perf_counter()
16
17     if script in self.__tmp_registrer:
18       if script not in self.__stats:
19         self.__stats[script] = {}
20       self.__stats[script]["execution_time"] = end_time - self.
      __tmp_registrer[script]
21       del self.__tmp_registrer[script]
22     else:
23       raise Exception(f'You can not execute this function before "timer_start
      ".')
24
```

```
25  def store_size_data(self, name_data, size, script=None):
26    if script is None:
27        dictionary = self.__stats
28    else:
29      if script not in self.__stats:
30        self.__stats[script] = {}
31      dictionary = self.__stats[script]
32
33    if name_data in dictionary:
34      raise Exception(f'The key \"{name_data}\" contains data, choose another
       name or remove the key first.')
35    else:
36      dictionary[name_data] = size
37
38  def store_dictio(self, dictio, name_dictio=None, script=None):
39    if isinstance(dictio, dict) and name_dictio is None:
40      raise Exception(f'The first parameter is not a dictionary, you must
      write a name for it (second parameter).')
41    if dictio is not None and dictio: # Store if it is not None or empty
42      if script is None:
43
44        dictionary = self.__stats
45      else:
46        if script not in self.__stats:
47          self.__stats[script] = {}
48        dictionary = self.__stats[script]
49
50      if name_dictio is not None:
51        if name_dictio in dictionary:
52          raise Exception(f'The key \"{name_dictio}\" contains data, choose
      another name or remove the key first.')
53        else:
54          dictionary[name_dictio] = dictio
55      elif not any(key in self.__stats for key in dictio):
56        dictionary.update(dictio)
57
58  def delete_stat(self, stat):
59    if stat in self.__stats:
60      stat_data = deepcopy(self.__stats.get(stat))
61      del self.__stats[stat]
62      return stat_data
63    else:
64      print("WARNING: The state that was attempted to be deleted does not
      exist.")
65      return None
66
67  def get_stats(self, stats=None):
68    if stats is None:
69      return deepcopy(self.__stats)
70
71    dictio = deepcopy(self.__stats.get(stats))
72    if dictio is None:
73      raise Exception(f'The key \"{stats}\" not exist.')
74    return dictio
75
76  def load_stats_to_json(self, bucket, name_file='log_stats'):
77    storage = Storage()
78    storage.put_object(bucket=bucket, key=f'stats/{name_file}.json', body=str
      (json.dumps(self.__stats,indent=2)))
```

Código 36: Clase *stats*.

D Fichero data_fetch.py

D.1. Obtención del segmento FASTQ

```
def fetch_fastq_chunk(fastq_chunk: dict, target_filename: str, storage: lithops.Storage, fastq_path: S3Path,
                      storage_bucket: str, fastqgz_index_key: str):
    tmp_index_file = tempfile.mktemp()
    gztool = get_gztool_path()
    lines = []
    lines_to_read = fastq_chunk['line_1'] - fastq_chunk['line_0'] + 1

    try:
        t0 = time.perf_counter()

        # Get index and store it to temp file
        storage.download_file(bucket=storage_bucket, key=fastqgz_index_key, file_name=tmp_index_file)

        # Get compressed byte range
        extra_get_args = {'Range': f"bytes={fastq_chunk['range_0'] - 1}-{fastq_chunk['range_1'] - 1}"}
        body = storage.get_object(fastq_path.bucket, fastq_path.key, True, extra_get_args)

        cmd = [gztool, '-I', tmp_index_file, '-n', str(fastq_chunk['range_0']), '-L', str(fastq_chunk['line_0'])]
        proc = subprocess.Popen(cmd, stdin=subprocess.PIPE, stdout=subprocess.PIPE)

        # TODO program might get stuck if subprocess fails, blocking io should be done in a backgroun thread or using async/await
        def _writer_feeder():
            logger.debug('Writer thread started')
            input_chunk = body.read(CHUNK_SIZE)
            while input_chunk != b"":
                # logger.debug('Writing %d bytes to pipe', len(chunk))
                try:
                    proc.stdin.write(input_chunk)
                except BrokenPipeError:
                    break
                input_chunk = body.read(CHUNK_SIZE)
            try:
                proc.stdin.flush()
                proc.stdin.close()
            except BrokenPipeError:
                pass
            logger.debug('Writer thread finished')

        writer_thread = threading.Thread(target=_writer_feeder)
        writer_thread.start()

        output_chunk = proc.stdout.read(CHUNK_SIZE)
        last_line = None
        while output_chunk != b"":
            # logger.debug('Read %d bytes from pipe', len(chunk))
            text = output_chunk.decode('utf-8')
            chunk_lines = text.splitlines()
            if last_line is not None:
                last_line = last_line + chunk_lines.pop(0)
                lines.append(last_line)
                last_line = None
            if text[-1] != "\n":
```

```
53                 last_line = chunk_lines.pop()
54
55             lines.extend(chunk_lines)
56
57             # Stop decompressing lines if number of lines to read in this chunk
       is reached
58             if len(lines) > lines_to_read:
59                 proc.stdout.close()
60                 break
61
62             # Try to read next decompressed chunk
63             # a ValueError is raised if the pipe is closed, meaning the writer
      or the subprocess closed it
64             try:
65                 output_chunk = proc.stdout.read(CHUNK_SIZE)
66             except ValueError:
67                 output_chunk = b""
68
69         try:
70             proc.wait()
71         except ValueError as e:
72             logger.error(e)
73
74         writer_thread.join()
75
76         t1 = time.perf_counter()
77         logger.debug('Got partition in %.3f seconds', t1 - t0)
78
79         # TODO write lines to file as decompressed instead of saving them all
      in memory
80         with open(target_filename, 'w') as target_file:
81             target_file.writelines((line + '\n' for line in lines[:fastq_chunk[
      'line_1'] - fastq_chunk['line_0']]))
82
83     finally:
84         force_delete_local_path(tmp_index_file)
```

Código 37: Función fetch_fastq_chunk()

E Fichero reduce_caller.py

E.1. Coordinación de la fase de Reduce

```
1  def run_reducer(pipeline_params: PipelineRun, lithops: Lithops, mapper_output
    ):
2      subStat = Stats()
3      logger.debug("START OF REDUCE STAGE")
4
5      #TODO: Fix errors with lithops cache and multipart upload ids
6      lithops=deflithops.FunctionExecutor(runtime_memory=8192)
7
8      # 1 Organize the keys generated by the map phase by fasta split
9      intermediate_keys = keys_by_fasta_split(mapper_output)
10
11     # 2 Create the keys for the multipart uploads
12     multipart_keys = create_multipart_keys(pipeline_params)
13
14     # 3 Create the multipart uploads and get their IDs
15     multipart_ids = []
16     for key in multipart_keys:
```

```
17          multipart_ids.append(create_multipart(pipeline_params, key, lithops.
    storage))
18
19      # 4 Select the indexes that each reducer will process
20      indexes_iterdata = []
21      for fasta in intermediate_keys:
22          data = {
23              'pipeline_params': pipeline_params,
24              'keys': intermediate_keys[fasta]
25          }
26          indexes_iterdata.append(data)
27
28
29      logger.debug("DISTRIBUTING INDEXES BETWEEN REDUCERS")
30      subStat.timer_start('distribute_indexes')
31      distributed_indexes = lithops.map(distribute_indexes, indexes_iterdata).
    get_result()
32      subStat.timer_stop('distribute_indexes')
33      distributed_indexes, timers = split_data_result(distributed_indexes)
34      subStat.store_dictio(timers, "function_details", "distribute_indexes")
35
36
37      # 5 Launch the reducers
38      logger.debug("EXECUTING REDUCE FUNCTION")
39      reducer_iterdata = create_iterdata_reducer(intermediate_keys,
    distributed_indexes, multipart_ids, multipart_keys, pipeline_params)
40      subStat.timer_start('reduce_function')
41      reducer_output = lithops.map(reduce_function, reducer_iterdata).
    get_result()
42      subStat.timer_stop('reduce_function')
43      reducer_output, timers = split_data_result(reducer_output)
44      subStat.store_dictio(timers, "function_details", "reduce_function")
45
46      # 6 Complete the multipart uploads that the reducers created
47      complete_multipart(multipart_keys, multipart_ids, reducer_output,
    pipeline_params, lithops.storage.storage_handler.s3_client)
48
49      # 7 Create a multipart upload key and ID for the final file
50      final_sinple_key = f'tmp/{pipeline_params.run_id}/final.alignment'
51      final_id = create_multipart(pipeline_params, final_sinple_key, lithops.
    storage)
52
53      # 8 Merge files created in stage 6 into one single file
54      n_parts = len(multipart_keys)
55      part = 1
56      merge_iterdata = []
57      while part <= n_parts:
58          data = {
59              "mpu_id": final_id,
60              "mpu_key": final_sinple_key,
61              "key": multipart_keys[part-1],
62              "n_part": part,
63              "pipeline_params": pipeline_params
64          }
65          merge_iterdata.append(data)
66          part += 1
67
68
69      logger.debug("EXECUTING FINAL MERGE")
70      subStat.timer_start('final_merge')
71      final_merge_results = lithops.map(final_merge, merge_iterdata).get_result
    ()
72      subStat.timer_stop('final_merge')
```

```
73     final_merge_results, timers = split_data_result(final_merge_results)
74     subStat.store_dictio(timers, "function_details", "final_merge")
75
76
77     # 8 Complete the previous multipart upload
78     finish(final_sinple_key, final_id, final_merge_results, pipeline_params,
      lithops.storage.storage_handler.s3_client)
79
80     logger.debug("END OF REDUCE STAGE")
81     return subStat
```

Código 38: Función run_reducer()