Ayman Bourramouss

FASTER DATA PARTITIONING OF GENOMIC SEQUENCES USING DATA LAKES

TREBALL DE FI DE GRAU

Pedro Garcia Lopez
Codirector: Lucio Marcello

Grau d'Enginyeria Informàtica/Telemàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2022

Resum.

En el següent Treball de final de grau es proposen diferents formes de particionament de dades (que podrien categoritzar-se com a big data o dades massives) per poder ser tractats posteriorment. L'objectiu és tenir un conjunt d'algorismes capaços de particionar dos tipus diferents de dades de forma eficient, això inclou també el posterior emmagatzematge i recuperació, utilitzant el paradigma serverless.

D'aquesta manera, es proporciona un conjunt d'algoritmes capaços de rebre unes dades senceres de determinat format i particionar-les perquè puguin ser consumides per diferents funcions lambda llançades en paral·lel, això conjuntament amb l'arquitectura MapReduce permeten identificar variants genòmiques dins del conjunt de dades proporcionat.

L'objectiu del projecte és particionar i proporcionar dades de la manera més eficient possible, explicant i raonant el perquè de l'arquitectura que proposo, així puc proporcionar dades al pipeline amb un gran throughtput gràcies a dades ja emmagatzemades a AWS.

Resumen.

En el siguiente Trabajo de final de grado se proponen distintes formas de particionado de datos (que podrían categorizarse como big data o datos masivos) para poder ser tratados posteriormente. El objetivo es tener un conjunto de algoritmos capaces de particionar dos tipos distintos de datos de forma eficiente, esto incluye también el posterior almacenamiento y recuperación, utilizando el paradigma *serverless.*

De este modo, se proporciona un conjunto de algoritmos capaces de recibir unos datos enteros de determinado formato y particionarlos para que puedan ser consumidos por distintas funciones lambda lanzadas en paralelo, esto, juntamente con la arquitectura MapReduce permiten identificar variantes genómicas dentro del conjunto de datos proporcionado.

El objetivo de mi proyecto es proporcionar y particionar datos de la forma más eficiente posible, explicando y razonando el porqué de la arquitectura que propongo, de este modo, puedo proporcionar datos al pipeline con un gran throughtput gracias a datos ya almacenados en AWS.

Abstract.

In the following Final Degree Project i propose different ways of partitioning data (which could be categorized as big data or massive data) in order to be able to be treated later. The objective is to have a set of algorithms capable of partitioning two different data types efficiently, including subsequent storage and retrieval, using the serverless paradigm.

In this way, I provide a set of algorithms capable of receiving data of a given format and partitioning it so that it can be consumed by different lambda functions launched in parallel, this, together with the MapReduce architecture allow to identify genomic variants within the provided dataset.

Resum

The goal of my tfg is to provide and partition data in the most efficient way possible, explaining and reasoning why the architecture I propose, in this way, I can provide data to the pipeline with a large throughput thanks to data already stored in AWS.

Índice

1 INTRODUCCIÓN Y OBJETIVOS 5
1.1 CONTEXTO 5
1.2 MOTIVACIÓN 6
1.3 REQUISITOS Y OBJETIVOS 7
1.3.1 Requisitos funcionales: 7
1.3.2 Requisitos no funcionales 8
1.3.3 Objetivos 8

2 CONCEPTOS BIOLÓGICOS 10
2.1 TAMAÑO DE LAS SECUENCIAS GENÓMICAS 10
2.2 FORMATO FASTQ 10
2.3 FORMATO FASTA 12

3 CONCEPTOS COMPUTACIONALES 15
3.1 CLOUD COMPUTING 15
3.2 SERVERLESS 15
3.3 AWS OPEN DATA Y LOS DATA LAKES 16
3.4 LITHOPS 17
3.5 MAPREDUCE 18

4 ARQUITECTURA 20
4.1 VISIÓN GENERAL DE LA ARQUITECTURA 20
4.2 MOTIVACIÓN DE LA ARQUITECTURA 22
4.3 COMPRENDIENDO *FASTQ-DUMP* 23
4.4 ALINEAMIENTO DE SECUENCIAS GENOMICAS 23
4.5 DISEÑO DE UN PARTICIONADOR DE SECUENCIAS MEDIANTE FASTQ-DUMP Y LITHOPS 24
4.5.1 Esearch API para obtener el número total de reads. 24
4.5.2 Preprocesamiento de la secuencia 26
4.5.3 Iterdata (Producto cartesiano entre la referencia y la secuencia) 27
4.5.4 Particionado de secuencias a partir de los metadatos 28
4.6 DISEÑO DE UN PARTICIONADOR DE REFERENCIAS FASTA CON LITHOPS Y EL MODELO MAPREDUCE 30
4.6.1 Introducción al problema 31
4.6.2 Lithops Partitioner 32
4.6.3 Indexado de la referencia (Fase de Map) 33
4.6.4 Generar nuevos títulos para los cromosomas (Etapa Reduce) 34
4.6.5 Generar las particiones de la referencia a partir de los metadatos 35
4.7 RELACIÓN ENTRE COMPONENTES. 36

5 IMPLEMENTACIÓN 38
5.1 PARTICIONADOR DE SECUENCIAS GENÓMICAS (*FASTQ*) 38
5.1.1 Obtener metadatos a partir de la API Esearch. 38
5.1.2 Procesamiento de los metadatos de la secuencia. 40
5.2 PARTICIONADOR DE REFERENCIAS GENÓMICAS (FASTA) 42
5.3 ITERDATA Y DESCARGA DE PARTICIONES DE LA SECUENCIA Y REFERENCIA. 44

6 EVALUACIÓN 48
6.1 ETAPA DE PREPROCESAMIENTO COMPARADA CON EL RESTO DEL PIPELINE, TIEMPO DE EJECUCIÓN Y DATOS QUE SE ALMACENAN. 48
6.2 TIEMPO DE DESCARGA DE PARTICIONES LAMBDA VS LOCAL 49
6.3 CANTIDAD DE TIEMPO NECESARIO PARA QUE EL PARTICIONADOR DE FASTA GENERE PARTICIONES EN S3 50
6.4 CANTIDAD DE TIEMPO NECESARIA PARA QUE SE HAGA EL PREPROCESAMIENTO DE LA SECUENCIA. 51

7 CONCLUSIONES 53
8 REFERENCIAS..... 54

Índice de tablas

TABLA 1. REQUISITOS FUNCIONALES....8
TABLA 2. REQUISITOS NO FUNCIONALES....8
TABLA 3. *OBJETIVOS*....9
TABLA 4. FORMATO DE LOS METADATOS DE LA SECUENCIA.....27
TABLA 5. INDEXACIÓN VERSIÓN DISTRIBUIDA VS SAMTOOLS FAIDX....34

Índice de figuras

FIGURA 1. CRECIMIENTO DE BASE DE DATOS SRA 7
FIGURA 2. COMPOSICIÓN READ FASTQ 11
FIGURA 3. ESTRUCTURA DE UNA REFERENCIA GENÓMICA. 12
FIGURA 4. RELACIÓN ENTRE SECUENCIAS FASTQ, REFERENCIAS FASTA Y MAPPING-ALINEAMIENTO. 13
FIGURA 5. TRADITIONAL APPROACH VS CLOUD APPROACH [10] 16
FIGURA 6. FUNCIONAMIENTO DE LITHOPS 18
FIGURA 7. MODELO MAPREDUCE, EJEMPLO DE CONTEO DE PALABRAS 19
FIGURA 8. RELACIONES ENTRE COMPONENTES. 20
FIGURA 9. COMUNICACIÓN VARIANT CALLER CON LA API DE NCBI (ESEARCH) 25
FIGURA 10. FORMATO XML METADATOS SECUENCIA EN ESEARCH, HTTP GET. 26
FIGURA 11.CABECERA FUNCIÓN PREPARE_FASTQ 26
FIGURA 12. CABECERA FUNCIÓN PREPROCESS_FASTQSRA 26
FIGURA 13. CABECERA FUNCIÓN GENERATE_FASTQ_CHUNK_LIST_FASTQ_SRA 27
FIGURA 14. CABECERA FUNCIÓN MAP_ALIGNMENT 28
FIGURA 15. ITERDATA Y LAMBDAS 29
FIGURA 16. CABECERA FASTQ_TO_MAPFUN 29
FIGURA 17. ARQUITECTURA DEL PARTICIONADOR DE SECUENCIAS GENÓMICAS CON TODOS SUS COMPONENTES 30
FIGURA 18. CÓDIGO PARTICIONADOR LITHOPS. 32
FIGURA 19. FUNCIÓN GENERATE_CHUNKS 34
FIGURA 20. CABECERA FUNCIÓN GENERATE_INDEX_FILE_TITLES 35
FIGURA 21. CABECERA GENERATE_CHUNKS_CORRECTED_INDEX 35
FIGURA 22. COMPONENTES DEL PARTICIONADOR DE REFERENCIAS 36
FIGURA 23. INSTANCIA CLASE SRAMETADATA Y LLAMADA A LA API 38
FIGURA 24. FUNCIÓN EFETCH_SRA_FROM_ACCESSIONS 40
FIGURA 25. FUNCIÓN EFETCH_METADATA_FROM_IDS 40
FIGURA 26. FUNCIÓN PREPARE_FASTQ 40
FIGURA 27. FUNCIÓN PREPROCESS_FASTQSRA 41
FIGURA 28. FUNCIÓN PREPARE_FASTA 42
FIGURA 29. FUNCIÓN SPLIT_FASTAFILE 42
FIGURA 30. LLAMADA AL SUBPROCESO QUE EJECUTA EL MAPREDUCE PARA PARTICIONAR LA REFERENCIA 42
FIGURA 31. FUNCIÓN SPLIT_FASTAFILE 43
FIGURA 32. FUNCIÓN LIST_KEYS LITHOPS API. 43
FIGURA 33. FUNCIÓN GENERATE_ALIGNMENT_ITERDATA 44
FIGURA 34. CÓDIGO FUNCIÓN GENERATE_ALIGNMENT_ITERDATA 44
FIGURA 35. CÓDIGO FUNCIÓN FASTQ_TO_MAPFUN QUE DESCARGA LOS CHUNKS A CADA UNO DE LOS WORKERS DE LA ETAPA DE MAP. 46
FIGURA 36. CÓDIGO FUNCIÓN COPY_TO_RUNTIME QUE DESCARGA LAS PARTICIONES DE LA REFERENCIA DESDE S3 A CADA UNO DE LOS WORKERS EN LA ETAPA DE MAP. 47
FIGURA 37. FUNCIÓN COPY_TO_RUNTIME 47
FIGURA 38. DATOS QUE SE ALMACENAN UTILIZANDO EL ALMACENAMIENTO EFIMERO DE LAMBDA 48
FIGURA 39. TIEMPO QUE TOMA CADA ETAPA DEL PIPELINE. 49
FIGURA 40. THROUGHPUT DESCARGA FASTQ-DUMP CLOUD VS LOCAL 49
FIGURA 41. TIEMPO NECESARIO PARA GENERAR PARTICIONES CON 97 WORKERS Y FASTA DE 3GB 51
FIGURA 42. TIEMPO NECESARIO PARA GENERAR PARTICIONES CON 23 WORKERS Y FASTA DE 3GB 51
FIGURA 43. TIEMPO VS TAMAÑO EN PROCESAMIENTO DE READS FASTQ 52

1 Introducción y objetivos

En este apartado se expone el problema que se pretende resolver, conceptos fundamentales para entender el problema y su posterior resolución.

Se introducirán los distintos conceptos que posteriormente permitirán la comprensión correcta de las ideas y soluciones tratadas, qué problemas se pretenden resolver en el campo de la genómica con las soluciones que he implementado y ayudar al lector a entender el porqué de algunas de estas soluciones.

Este proyecto ha sido desarrollado en colaboración con el CloudLab (URV), el Biomathematics & Statistics Scotland (BioSS) Research Institute y el James Hutton Interdisciplinary Scientific Research Institute en el proyecto conocido como CloudButton.

Con el objetivo de crear una metodología para el análisis genómico, en un proyeto que pertenece al EU Horizon 2020 Framework Programme (H2020) y cuyo objetivo es crear una plataforma de análisis de datos sin servidor.

De este modo y utilizando Lithops, un framework desarrollado en Python para el procesamiento distribuido de datos creado por el CloudLab procesar datos genómicos a gran escala no será una limitación.

1.1 Contexto

Es evidente que el cloud (también conocido como la nube) ofrece soluciones novedosas, parte de ello pasa por comprender qué es capaz de ofrecernos el cloud y qué ventajas puede aportarnos.

En este caso en particular, se trata el problema de cómo se pueden procesar grandes cantidades de datos de forma paralela, específicamente, como podemos ser capaces de proporcionar estos datos a nuestras aplicaciones utilizando el *cloud*. En términos laicos, de que maneras podríamos proporcionar a una aplicación datos de forma rápida y fácil, con un gran rendimiento y de forma escalable.

La principal ventaja del cloud respecto a un HPC o servidor es la capacidad de adaptarse a las necesidades computacionales de un momento concreto, los recursos se asignan cuando son necesarios y esto se traduce en un menor coste monetario [1], esto es conocido como “pay as you go”, ya que no es necesario contar con un HPC o una granja de servidores en todo momento, sino que el trabajo puede ser lanzado en un instante cualquiera, acabar y solamente se pagará el uso que hayamos hecho de este.

Esta es una ventaja del cloud, donde cualquier usuario es capaz de tener a su disposición miles de *cpu’s virtuales* en un momento concreto. Por lo tanto, la capacidad de *elasticidad* es importante si se considera que es necesario manejar una gran cantidad de datos y todavía más si estos varían de tamaño.

En el contexto del cloud, el uso de *FaaS* (Function as a Service) que permiten ejecutar código en forma de funciones en entornos de ejecución aislados (Normalmente MicroVM’s), que no guardan el estado es lo que nos permite el particionado y posterior almacenamiento de los resultados.

En concreto utilizamos Lambda que es un servicio proporcionado por AWS (Amazon Web Services), que es un servicio de cómputo (nuevamente, funciones) que nos permite ejecutar código sin aprovisionar ni administrar servidores. Esto permite que el volumen de datos sea *elástico*, capaz de variar de cantidad, y estas funciones ser capaces de adaptarse al volumen de datos.

1.2 Motivación

Por otro lado, una tendencia cada vez más creciente es almacenar datos comúnmente accedidos en el cloud para su posterior uso (Normalmente, los consumidores de estos datos suelen ser empresas, investigadores, proyectos de diversa índole etc....), de esta forma, el cloud actúa como un intermediario entre un cliente y una entidad. La entidad almacena datos en el cloud público en lo conocido como Data Lakes [2] (ya sea AWS, IBM Cloud, GCP) y permite que los clientes tengan acceso a estos. La principal ventaja de esta visión es que los clientes ya no tienen la necesidad de almacenar datos de forma privada en un *bucket* (Contenedor de objetos que nos permite almacenar, leer y modificar ficheros, también conocido como object storage), sino que son accesibles directamente desde el cloud, sin la necesidad de tener que subirlos.

La ventaja de esta visión es un ahorro de tiempo substancial para los clientes, por un lado, en tiempos de acceso (Ya que no hay necesidad de acceder a servidores externos para obtener los datos), y en costes de almacenamiento, ya que los datos no pertenecen al cliente, sino al *cloud.*

Idealmente, nos gustaría poder acceder a los ficheros desde el servidor más cercano a nosotros, para poder realizar la descarga de la forma más rápida posible. Algo todavía mejor, es que los datos se encontraran directamente almacenados en nuestro dispositivo, pero no disponemos de almacenamiento infinito o *casi infinito*.

Pero en el cloud sí, y por lo tanto, es posible tener acceso ‘local’ a ficheros sin necesidad de que sean descargados.

Un ejemplo de esto es *Registry of Open Data* de AWS [3], donde permiten que se almacenen secuencias genómicas (Que son ficheros accedidos por bioinformáticos de forma común) para su posterior tratamiento. Es una medida que están adoptando distintos *cloud providers* con el objetivo de atraer a potenciales usuarios. Como es el caso de bioinformáticos en el proyecto que he estado trabajando. Pero no solamente se ciñe a la bioinformática o genomica, si no que se almacenan datos de varios tipos (datos satelitales, sobre crawling, espaciales…).

Los datos que se han utilizado para el proyecto son datos de secuencias genómicas (*FASTQ*) [4] y referencias genómicas (*FASTA*) [5], provenientes de una base de datos conocida como *SRA* o *Sequence Read Archive [6],* que es la mayor base de datos pública para secuencias genómicas. Estos datos son replicados en el Registry of Open Data para que puedan ser accedidos desde el cloud. La cantidad de secuencias genómicas que hay almacenadas en SRA es del orden de miles de TeraBytes.

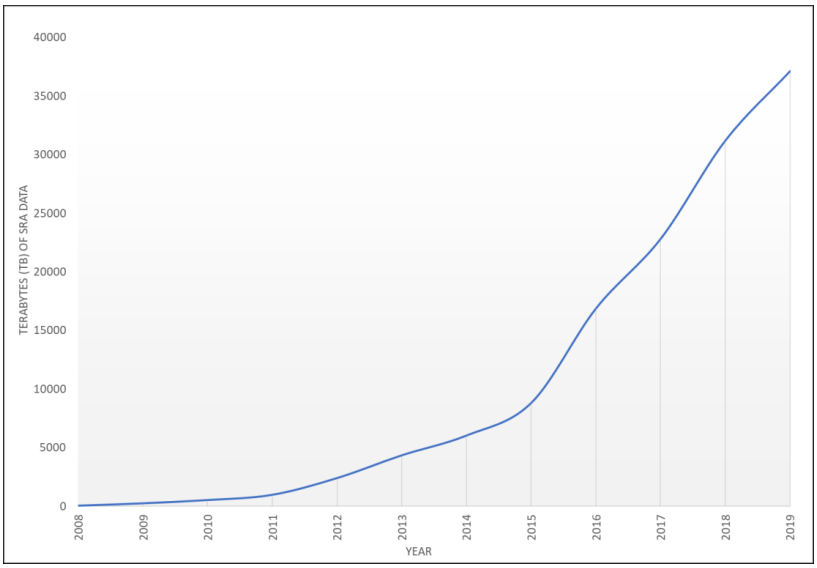

Figura 1. Crecimiento de base de datos SRA

Es una tendencia creciente y que ha permitido obtener un speed-up substancial respecto a la implementación que ya existía, donde evidentemente es necesario almacenar el fichero (de cientos de Gigabytes) en el propio bucket, lo que implica costes temporales y de almacenamiento importantes.

Por lo tanto, la motivación de este proyecto es de proveer grandes cantidades de datos a un pipeline complejo, teniendo en cuenta que el objetivo principal es que sea implementado sobre el *cloud, serverless, escalable, elástico* y de gran *rendimiento.*

1.3 Requisitos y Objetivos

1.3.1 Requisitos funcionales:

A continuación, en la siguiente tabla se presentan los requisitos funcionales los cuales so declaraciones de servicios que prestará el sistema, Es decir, las funcionalidades que necesita el cliente.

Requisito	**Descripción**
Buscar secuencias sobre una base de datos	El sistema debe ser capaz de buscar una secuencia sobre la base de datos.
Descargar secuencias en el cloud	El sistema debe ser capaz de utilizar la base de datos para descargar las secuencias en el cloud.
Obtener metadatos sobre la secuencia	El sistema debe ser capaz de buscar los metadatos relacionados a la secuencia, para poder particionarla.

Particionar secuencias	El sistema debe particionar secuencias basado en los metadatos relacionados a esta.
Particionar referencias basado en los inputs del usuario.	El sistema debe ser capaz de particionar referencias genómicas basado en los inputs que proporcione el usuario.
Mostrar los metadatos relacionados con la secuencia.	Que el sistema muestre los metadatos e información relacionados con la secuencia
Mostrar un error si la secuencia no se encuentra o no está disponible	Si no se encuentran la secuencia, que muestre un error por consola.

Tabla 1. Requisitos funcionales

1.3.2 Requisitos no funcionales

Requisito	**Descripción**
Facilidad de uso	Dado que será utilizado por personas con poco conocimiento informático, la facilidad de uso es una prioridad.
Rápido	Mediante el uso de data lakes, el objetivo es proporcionar secuencias particionadas de forma rápida.
Escalable y elástico	Que los particionadores se adapten basándose en el tamaño de las secuencias y referencias a particionar
Conexión a internet	Conexión a internet necesaria para ejecutar el pipeline, hasta entrada a la etapa de map.

Tabla 2. Requisitos no funcionales

1.3.3 Objetivos

Datos variados	Conectado a una base de datos capaz de proveer datos al pipeline

Acceso rápido a la información	Capaz de proveer información con un gran rendimiento sin alterar las necesidades del momento, mediante el uso de Data Lakes.
No alterar el flujo de trabajo	Que la persona que ejecute el pipeline se olvide de tener que proporcionar metadatos o subir los datos a su cloud provider, sino que simplemente especifique el nombre de una secuencia y referencia.
Pay-as-you-go	Pago por ejecución del del particionador, es decir, por pago por el uso que se haga de los particionadores.
Diseño serverless	Sin necesidad de mantener una infraestructura o servidor.
Adecuado para flujos de trabajos científicos	Dado que los usuarios serán personas dedicadas a la biología o bioinformático, el uso debe ser adecuado para ellos. Idealmente con que el usuario especifique el nombre de una secuencia la plataforma debería ser capaz de identificarla y proporcionar información sobre esta.

Tabla 3. ***Objetivos***

2 Conceptos biológicos

2.1 Tamaño de las secuencias genómicas

Las tecnologías de secuenciación de nueva generación generan datos genómicos de gran magnitud y se desconoce qué tan grandes pueden llegar a ser. Existen casos en los que una secuencia genomica es capaz de llegar a pesar cientos de GigaBytes, aunque no es un límite impuesto, sino que depende del organismo secuenciado.

Por ejemplo y para que sea visible la magnitud del tamaño de las secuencias genómicas, en el mejor de los casos teniendo la secuencia perfecta sin errores tecnológicos, eliminando información como la calidad o posicionamiento de la secuencia y utilizando strings de letras (A, T, C y G), la secuencia del ser humano por referencia seria de 3.2 millón de millones de *base pairs* (letras) o 700 mb aproximadamente. Realmente son necesarios el doble de *base pairs* para representar el genoma de una persona, ya que 3.2 millón de millones de base pairs serian para la referencia donde se elimina información redundante entre personas.

Si codificamos los caracteres A, T, G y C de forma que su conversión es en 2 bits y

Pasan a ser 00, 01, 10 y 11, nos encontramos que necesitamos unos 6000000000 bits o lo que es lo mismo, 750 mb.

Estos datos pueden ser encontrados en servidores de grandes compañías de secuenciación como pueden ser Illumina® o Roche454® que producen grandes cantidades de datasets genómicos, ya sean referencias o secuencias.

Algunos de estos datasets pueden llegar a tener un tamaño considerable, de cientos de gigabytes que hace que no puedan ser almacenados en los ordenadores de los bioinformáticos, y es por ello por lo que se recurre a los sistemas distribuidos, donde se intentan paralelizar estos flujos de trabajo para hacerlos lo más rápidos y eficientes posibles.

Por otro lado, no es extraño que hoy en día los bioinformáticos cuenten con HPCs o granjas de servidores, la desventaja de esto es que es necesaria la descarga de las secuencias genómicas para su posterior uso en los distintos flujos de trabajo, y esto conlleva tiempo en el que no se hace uso del recurso, sino que el científico está a la espera de que la secuencia sea descargada para poder hacer uso de esta.

Dado el tamaño excesivo de los ficheros fastq, estos se acaban comprimiendo para que ocupen menor espacio. Aunque esto ayuda a reducir el tamaño del fastq, el problema sigue siendo el mismo.

2.2 Formato FASTQ

El formato utilizado para las secuencias genómicas es el denominado *FASTQ*. Es un formato de texto para almacenar secuencias biológicas (normalmente nucleoides) y sus puntuaciones de calidad, tanto las puntuaciones de calidad como el contenido de las secuencias esta codificado con un único símbolo ASCII.

El formato fastq está compuesto por *reads* que varían en longitud, se pueden denominar cómo la unidad atómica del fichero fastq. Cada read está compuesta de 4 líneas.

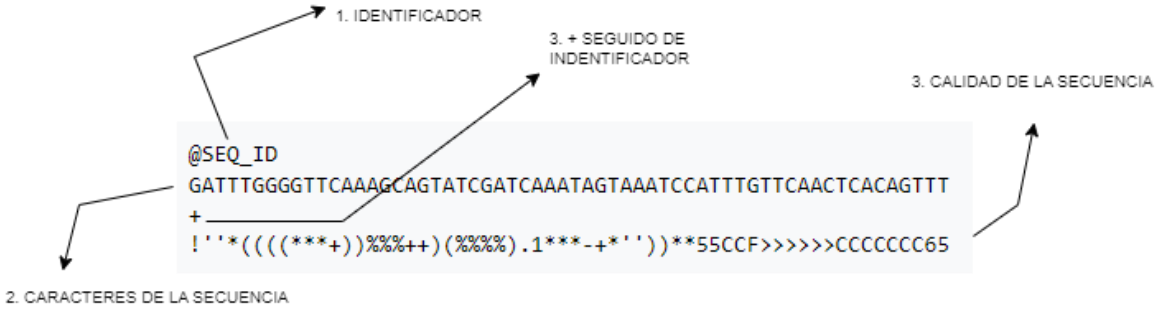

Figura 2. Composición read fastq

1. El identificador de la secuencia, opcionalmente y dependiendo de la base de datos de la cual lo hayamos descargado puede contener el atributo *length*, para saber el tamaño que ocupa esa *read*
2. Los datos de la secuencia (los base calls: A, C, T, G y N)
3. Separador '+' que puede contener a su lado el id de la secuencia (De forma opcional)
4. La calidad de los *base call*, cuando se hace la secuenciación hay margen de error, y cada carácter implica la probabilidad de error que puede haberse dado

Cada fichero fastq puede contener cientos y miles de *reads*, que se corresponden a la secuenciación de un organismo. Normalmente los mámales tienden a tener un menor número de *reads* pero más largos, mientras que las plantas suelen tener mayor cantidad de *reads* pero de longitud más corta.

En cuanto al almacenamiento de estos ficheros, que es lo que más interesa para la parte computacional, suelen existir diversas fuentes o bases de datos donde se almacenan secuencias de forma constante. Además, los usuarios son capaces de subir sus propias secuencias a estos repositorios para que puedan ser compartidos con la comunidad científica. Existen diversos repositorios de secuencias entre los cuales podríamos nombrar a Illumina®, *Sequence Read Archive*, *European Nucletoide Archive [7]* etc…

Las secuencias normalmente vienen en formato FASTQ o FASTQ.GZ (Comprimido), pero existen diversos formatos y no hay un estándar como tal. En mi caso me he centrado en el FASTQ que es el formato más utilizado.

Dicho esto, el *Sequence Read Archive* está haciendo el esfuerzo constante de almacenar las secuencias en cloud providers, ya que se han dado cuenta de que tener almacenadas las secuencias con distintos cloud providers permite tener acceso ultra rápido a estas. Esto permite que el científico o bioinformático que desee trabajar sobre una secuencia, se salte el paso de tener que descargar y subir la secuencia al cloud provider, lo que evita redundancia.

2.3 Formato FASTA

El formato utilizado para las referencias genómicas es el conocido como *FASTA*, es un formato de texto que representa secuencias de nucleoides o aminoácidos utilizando letras. El formato permite poner el nombre de la secuencia y comentarios.

A diferencia del FASTQ, el FASTA se trata de la referencia, por ponerlo en perspectiva, el FASTQ es una secuencia de una persona en concreto, mientras que el FASTA es la referencia, lo que comparten las personas en común.

Las referencias tienden a ser de menor tamaño y no contienen quality scores, además son de menor tamaño dado que se trata de una referencia.

El formato FASTA es el siguiente:

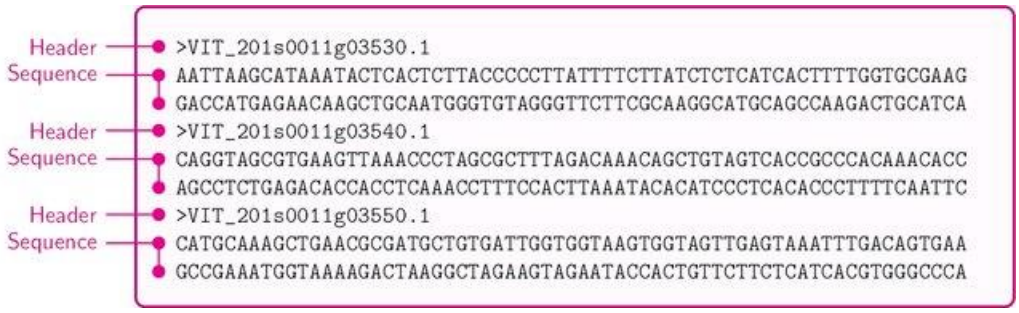

Figura 3. Estructura de una referencia genómica.

https://www.researchgate.net/publication/309134977_A_Survey_on_Data_Compression_Methods_for_Biological_Sequences/figures?lo=1

1. Header: La cabecera línea de cabecera, que empieza con >, da un identificador único a la secuencia y en algunos casos información adicional como puede ser la longitud de la secuencia.
2. Secuencia: Tras el Header, está la secuencia como tal, cada línea de la secuencia tiene 80 caracteres. Las secuencias pueden ser secuencias de proteínas, ácidos nucleicos o caracteres de alineamiento.

2.4 Variant Calling

El variant calling es la manera que tienen los científicos de identificar las variantes asociadas a una población. La bioinformática es la clave en cada parte del proceso y esencial para manejar datos genómicos.

El proceso de variant calling es el proceso el cual nos permite identificar variantes a partir de datos de la secuencia (FASTQ) y una referencia (FASTA).

El proceso se lleva a cabo de la siguiente forma:

1. Obtener los datos de secuenciación del genoma
2. Alineamiento de las secuencias a un genoma de referencia (formato BAM, CRAM, FASTA)
3. Identificar donde las lecturas se diferencian del genoma de referencia.

Los puntos 2 y 3 son los procesos de coste computacional más alto.

El punto 2 consiste en 2 etapas. Se indexa el genoma de referencia para tener un acceso más rápido, de modo que se ahorra memoria y tiempo. Y alineamiento-mapping como tal, donde se alinean las los reads (FASTQ) junto a la referencia. Para explicar el proceso de mapping utilizaré la siguiente figura:

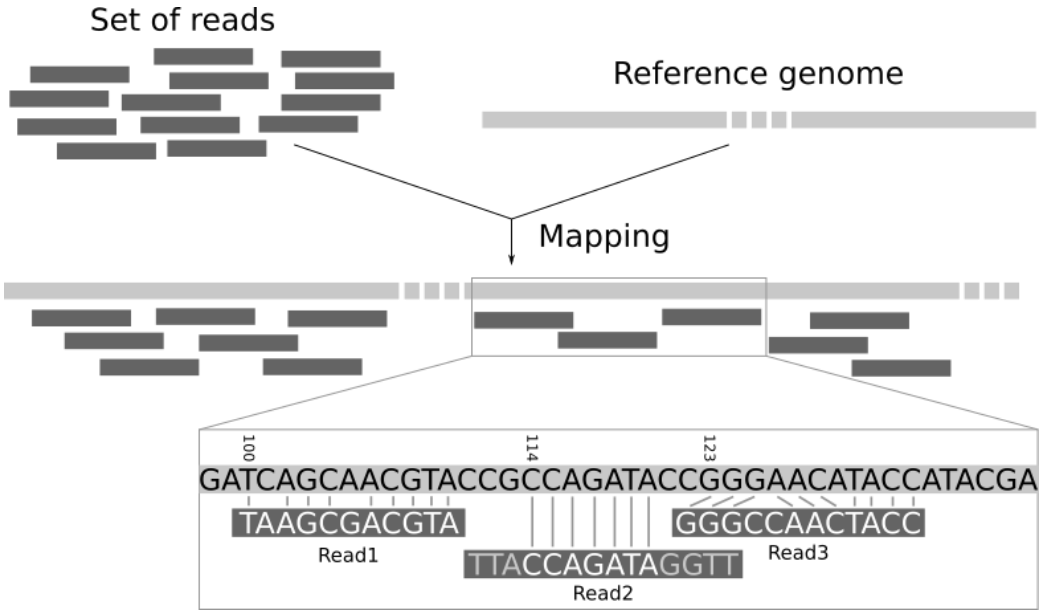

Figura 4. Relación entre secuencias FASTQ, referencias FASTA y mapping-alineamiento.

Los inputs o entradas consisten en un set de reads y una referencia genomica y el proceso consiste en alinear los reads a la referencia. El proceso de mapping o alineamiento mostrado en la figura consiste en encontrar las posiciones en el genoma de referencia donde las reads se pueden alinear.

Los casos de uso del variant calling [8] pueden ser distintos y muy variados, desde asociaciones genéticas a una enfermedad hasta mutaciones de cáncer. Básicamente el variant calling lo que permite es observar diferencias en un genoma por referencia, comparando carácter por carácter distintos cromosomas.

Es decir, el variant calling se basa en el proceso de comparar, comparaciones entre distintos cromosomas. Es un proceso de gran coste computacional que requiere de distintas etapas como son alineamiento, filtrado, eliminación de errores etc… Para ello se requieren herramientas como SAM tools o GEM Mapper.

Es un proceso altamente paralelizable, donde podemos tomar porciones de nuestra secuencia y referencia (Estos han de ser inputs enteros) y trabajar sobre ellos como si de una sola comparación se tratase.

Si nuestra secuencia es M y nuestra referencia es N, hacen falta M x N comparaciones, donde cada M es comparada con todas las N, esto es un producto cartesiano.

3 Conceptos computacionales

Explicados los conceptos sobre biología y dados ciertos antecedentes, es de especial importancia entender los conceptos sobre computación y más relacionados al cloud computing, frameworks y herramientas.

3.1 Cloud computing

En los años recientes, el tamaño y la cantidad de datos a analizar ha incrementado de una forma exponencial, es tal que la capacidad de almacenamiento y cómputo se ha visto afectada. Disponer de un clúster de tamaño fijo puede volverse un problema si desean analizarse grandes cantidades de datos.

Cloud computing [9] nos provee de herramientas capaces de solventar este problema, básicamente el cloud nos ofrece capacidad de cómputo y almacenamiento escalable y distribuido a través de internet. Pero una tendencia creciente, es almacenar datos de forma distribuida en el cloud. Donde los propios usuarios pueden almacenar y compartirlos. Por otro lado, los cloud providers están adaptándose para ofrecer datos directamente desde el cloud, sin necesidad de subirlos para que posteriormente sean utilizados.

Esto ofrece una ventaja substancial respecto a un clúster, ya que tenemos acceso a datos sin necesidad de descargas. Esto es lo que ofrece *el Registry of Open Data* de AWS.

3.2 Serverless

Es evidente que el cloud computing ofrece gran cantidad de ventajas respecto a un clúster, pero otro problema radica en que, con el cloud, necesitamos administrar la infraestructura con una cantidad ridícula de parámetros para poder sacar el máximo provecho.

Serverless nació para solventar este problema, y es que el problema de configuración y administración radica sobre el cloud provider, mientras que el programador solamente se dedica a desarrollar la aplicación. Por lo tanto, se eliminan las distintas problemáticas que implican la administración de la infraestructura.

Existen diversos servicios en el cloud, pero en este proyecto me centro explícitamente en FaaS o Function as a Service, la idea detrás de FaaS es ejecutar aplicaciones a través de contenedores efímeros y pequeños bloques de código. Centrándose sobre todo en la funcionalidad. Ejemplos de los servicios FaaS son Cloud Functions o AWS Lambda.

La configuración es relativamente simple, establecer parámetros de uso de memoria, tanto efímera como ram y una imagen para el contenedor, donde se establecen las dependencias que necesitará la función.

La ejecución de funciones se establece normalmente mediante eventos, donde al suceder cierta acción una función se ejecuta bajo cierto backend para realizar cierta tarea.

3.3 AWS Open Data y Los Data Lakes

Open Data son datos que pueden ser utilizados de forma abierta, reusados y redistribuidos por cualquiera. Open data son datos estructurados, de licencia abierta y bien mantenidos.

Open data es una forma de incrementar la transparencia, donde instituciones gubernamentales, empresas u organizaciones de investigación pueden compartir datos de forma masiva y publica. Estos datos no pertenecen a AWS, sino que se utiliza AWS para compartirlos.

No son más que un conjunto de repositorios públicos desde los que podemos acceder a cantidades enormes de información de distinta índole, en nuestro caso información sobre genómica.

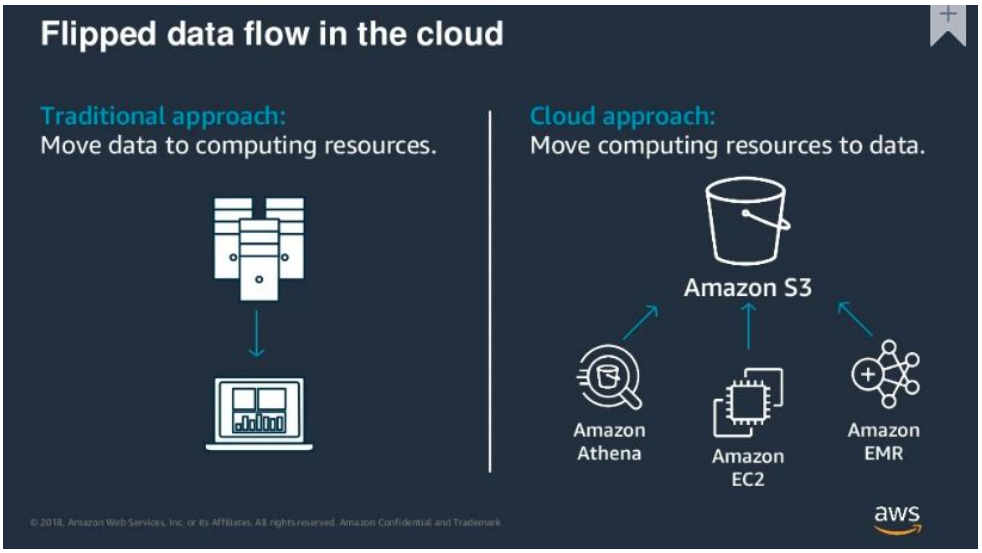

Figura 5. Traditional approach vs Cloud approach [10]

La ventaja de esto es que nos permite acceder a información de forma rápida y a coste cero.

Desde una Lambda, es posible acceder a secuencias genómicas utilizando mediante Open Data utilizando un middleware conocido como *fastq-dump*. Lo que permite *fastq-dump* es descargar secuencias genómicas especificando el nombre de esta y el rango de reads que necesitamos. Utilizando el middleware de forma local, este accede a los servidores de NCBI para descargar la secuencia a nuestro ordenador.

Pero desde el cloud, es capaz de descargar estas secuencias de forma ultra rápida y a coste cero.

Para entender esto, me gustaría utilizar una analogía.

Para realizar una descarga desde internet, básicamente tenemos que acceder a un servidor que está en una localización remota para realizar la descarga hacía nuestro ordenador, esto será más o menos rápido dependiendo de la localización en la que se encuentre el servidor, la cantidad de datos que necesitemos descargar y nuestra velocidad de descarga.

La combinación de Open Data y Serverless lo que permite es acceder al servidor directamente, y sobre él realizar nuestras operaciones. Por lo que ya no se produce una descarga, sino que tenemos acceso total al servidor y a los datos.

Por lo tanto, consiste en ***trasladar la capacidad de computación junto a los datos*.** De esta forma se soluciona uno de los mayores problemas de los clústeres y Big Data.

Se evitan pasos como la subida y descarga de datos, junto a la compresión y descompresión, lo que resulta en una aplicación mucho más rápida y eficiente en la etapa de preprocesamiento.

Esto es especialmente útil si el pipeline a desarrollar necesita una gran cantidad de datos, como es el caso.

3.4 Lithops

Lithops es un multi-cloud serverless computing framework [11] que permite ejecutar código local de forma masiva en los principales cloud providers. Lithops mueve el código del usuario al cloud sin la necesidad de que el usuario sepa cómo se despliega y ejecuta. Además, su arquitectura multi-agnóstica permite portabilidad a través de múltiples cloud providers.

Lo especialmente interesante para el caso de uso es que Lithops está diseñado para programas altamente paralelizables que requieren poca o ninguna comunicación entre procesos. Además, se ha utilizado de forma exitosa en diferentes campos como puede ser bioinformática (genómica y metabolómica) y datos geoespaciales (LiDAR, satelital).

Algunas características de lithops son las siguientes:

- **Facilidad de uso:** Es posible ejecutar hasta 1000 funciones *serverless* en paralelo sin necesidad de configuración
- **Fácil de aprender:** Hay documentación variada, por lo que es fácil iniciarse en cualquier backend y ejecutar diferentes ejemplos
- **Escalado dinámico:** Lithops permite escalar hasta miles de cores a demanda. El escalado es dinámico y depende de la paralelización de la aplicación
- **Multiplataforma:** Lithops soporta múltiples plataformas, cloud providers y back ends.
- **Detección de errores:** Permite detectar errores y trazarlos como si hubiese ocurrido localmente.

Lithops permite programar en alto nivel con varios métodos que permiten la ejecución de distintos modelos. Desde una llamada asíncrona a un método a MapReduce para procesar grandes cantidades de datos.

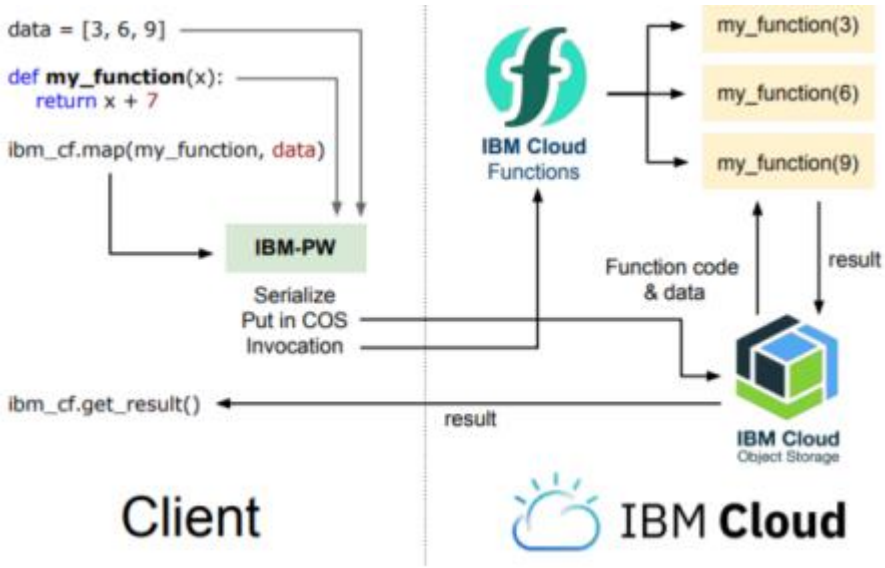

Figura 6. Funcionamiento de Lithops

El diagrama anterior muestra el funcionamiento de Lithops. Aunque el diagrama explica el caso de IBM, es exactamente el mismo funcionamiento que el de AWS pero con distintos productos.

1. El código local y los datos se serializa y se almacena en nuestro Bucket de preferencia.
2. Se ejecuta la función a través de IBM Cloud Functions, AWS Lambda o cualquier cloud provider.
3. El framework del cloud provider ejecuta las funciones en paralelo
4. El cliente hace pull de los resultados cuando están listos.

3.5 MapReduce

MapReduce [12] es un modelo de programación que consiste en facilitar el procesamiento simultáneo de grandes cantidades de datos. El MapReduce consiste en la técnica de *divide and conquer*, donde un input de gran tamaño se divide en cientos o miles de cpus para que pueda ser procesado en paralelo, donde cada porción es procesada por una o más cpu distintas.

La idea es obtener un speed up paralelizando los datos que se van a procesar.

Para ello, se requiere particionar los datos de forma correcta. Como ya he explicado anteriormente, siempre es más interesante utilizar *Data Lakes* que proveen datasets para el caso de uso. Con un Data Lake el problema se traslada a como particionar las secuencias.

Por lo tanto, no es extraño que MapReduce sea uno de los modelos preferidos a la hora de procesar grandes cantidades de datos.

Las fases del Map Reduce son las siguientes:

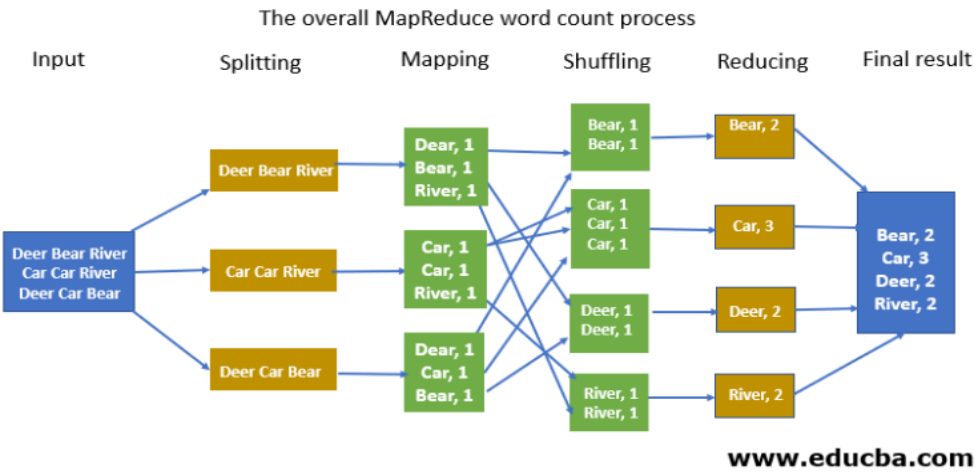

Figura 7. Modelo MapReduce, ejemplo de conteo de palabras

1. **Input:** Un input que puede ser un dataset de formatos diversos. Normalmente suelen ser datasets de tamaño considerable (Existen aplicaciones del modelo donde se usan datasets del orden de PetaBytes).
2. **Particionado:** Una vez se obtienen los datos, estos han de ser particionados para que a cada *mapper* le pertenezca un tamaño, este tamaño ha de ser igual o similar entre *mappers*, cada partición es asignada a un *mapper*, al final un mapper no deja de ser una función o método con un propósito concreto.
3. **Mapping:** Esta es la primera etapa de ejecución como tal, donde se hacen las operaciones necesarias y se crean procesos para poder transformar los datos. Cada *mapper* es independiente y depende de los datos que le haya enviado el particionador, en el ejemplo anterior, los *mappers* se encargan de contar el número de palabras por cada partición creada.
4. **Shuffling:** El shuffling es una etapa opcional, donde se filtran las salidas de los *mappers*, en el ejemplo anterior, el shuffling se encarga de agrupar las salidas de los *mappers* por la misma palabra.
5. **Reduce:** En esta etapa los resultados son agregados, donde cada uno de los resultados de los mappers o shufflers (en caso de que los hubiera) se agregan y generan una salida única. En el ejemplo de la figura, el reduce es el resultado del conteo de las palabras, el algoritmo puede ser cualquiera y depende del caso de uso.

Algo destacable del modelo map reduce es la capacidad de escalabilidad y elasticidad que ofrece, donde el modelo es capaz de escalar horizontalmente sin complicaciones. Dado el desacoplamiento y la independencia de las distintas particiones y mappers.

4 Arquitectura

En esta sección se explica el diseño y arquitectura del sistema de particionado de dos subsistemas dentro del Pipeline.

Por un lado, el particionador de *FASTQ*, que utiliza un Data Lake para proveer datos a la aplicación y que será lo que explicaré a continuación, y, por otro lado, el particionador de *FASTA*, que consiste en el modelo MapReduce.

4.1 Visión general de la arquitectura.

Este capítulo pretende dar una visión general sobre la arquitectura, ya que es un capítulo un poco denso y puede resultar un tanto complicado de entender.

La idea principal es desarrollar dos particionadores, uno que hará particiones de secuencias genómicas (FASTQ) y otro que hará particiones de referencias genómicas (FASTA). Particionar se refiere a dividir un fichero grande en porciones pequeñas conocidas como chunks o particiones. Estos son los dos inputs que necesita el pipeline como se ha explicado en el apartado de variant calling para que el pipeline pueda funcionar.

El diagrama de las relaciones entre componentes es el siguiente:

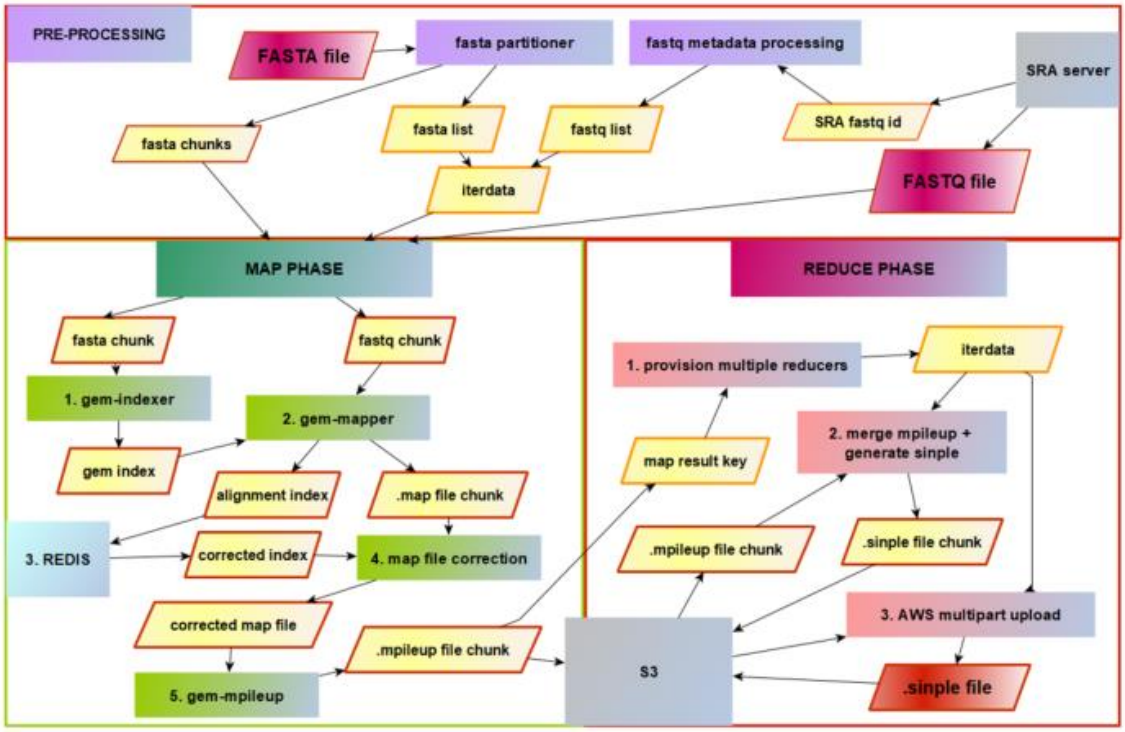

Figura 8. Relaciones entre componentes.

La a arquitectura que se propone trata sobre la etapa de **PRE-PROCESSING.** Esta etapa consiste en generar los datos necesarios para que pueda realizarse correctamente la **MAP PHASE o ETAPA DE MAP**. Dentro del modelo MapReduce.

Este apartado pretende explicar lo que se ve en el diagrama, para que el lector tenga ciertos antecedentes sobre lo que es necesario. A pesar de que algunos conceptos ya han sido introducidos anteriormente.

La **Figura 8** simboliza las relaciones entre componentes y estaré siempre referenciándome a ella para que el lector tenga una referencia visual y así le ayude a entender mejor las relaciones entre componentes.

El primer paso de todos es saber que secuencia de la base de datos de SRA y la referencia se quieren particionar, ya que ambos son los inputs necesarios para poder realizar el **variant calling.** Evidentemente, sin esos inputs el **variant calling** carece de sentido.

Estos inputs son introducidos mediante el intérprete de comandos del sistema operativo, el usuario introduce el nombre de la secuencia y el nombre de la referencia.

Aquí me gustaría parar un momento, y hacer un símil que quizá ayude al lector a entender que es una secuencia y que es una referencia.

Si lo utilizamos al ser humano, diríamos que una secuencia es cualquier persona con sus virtudes y defectos, de esta persona podríamos tomar sus características. De esta persona, tomamos todas sus características, por ejemplo, José:

- José tiene ojos.
- José respira.
- José tiene 2 brazos y 1 pierna.

  …

José es un individuo, una muestra. Tiene unas características, pero cada persona tiene las suyas propias.

La referencia es tomar las características que hacen al ser humano, aunque existen variaciones entre personas, se pone una referencia que son las características que cumplen la mayoría de personas.

- Las personas tienen 2 brazos y 2 piernas
- Las personas respiran
- Las personas tienen ojos

  …

El variant calling consiste en encontrar las diferencias entre José y las personas. Pero algo todavía más importante es que las referencias **no cambian a no ser que cambiemos de especie.**

Entonces, necesitamos un repositorio capaz de proveer datos de forma constante para las diferentes secuencias (FASTQ), ya que así lo exige el pipeline. Mientras que la referencia solamente se cambiará si se cambia de especie, ya que no podemos comparar manzanas con peras.

Aclarado esto, procedo a explicar las distintas componentes de la etapa de preprocesamiento, que es de lo que trata este proyecto.

Una vez se tiene el nombre de la secuencia y la localización de la secuencia, podemos proceder a obtener los metadatos de la secuencia. Este paso se refiere *a SRA Server de la Figura 8.*

Una vez se tienen los metadatos necesarios de la secuencia, hay que crear la lista de particiones en la que se dividirá, en *la Figura 8 eso es el recuadro denominado fastq metadata processing* y generará una lista de rangos de *reads.* Esto sería todo en cuanto a procesar secuencias se refiere.

En el caso de las referencias genómicas, a diferencia de las secuencias, no hay metadatos asociados a esta ni una API que nos permite obtener información sobre esta. Por lo que la forma de pre procesarlas es distinta.

La forma de pre procesar las referencias consiste, nuevamente, en la arquitectura MapReduce, y se explicará en capítulos posteriores. La diferencia es que la etapa conocida como *fasta partitioner en la Figura 8* generará una lista con los nombres de las particiones, ya que el fasta partitioner (particionador de referencias), sí genera particiones en esa etapa y se almacenan en object storage. Mientras que el fastq partitioner (particionador de secuencias), no guarda las secuencias en object storage. El argumento de este comportamiento es por lo explicado anteriormente. Las referencias son comunes entre especies, mientras que las secuencias dependen del individuo.

Entonces *fasta list de la Figura 8*, es el nombre de las particiones generadas.

Una vez tenemos **fasta list y fastq list** se genera **iterdata**, que es el producto cartesiano entre particiones de la referencia y particiones de la secuencia respectivamente. Cada partición de la secuencia se relaciona a con todas las particiones de la referencia en una lista de objetos. El iterdata es necesario para la etapa de map, ya que se llamarán a tantos workers como elementos haya en el iterdata.

Llegados hasta aquí, tenemos que se han generado las particiones de la referencia en object storage y el iterdata.

Eso sería todo en la etapa de **PRE-PROCESSING.**

En el **MAP PHASE,** se llaman a las funciones encargadas de procesar los datos, cada una de ellas es una Micro Máquina Virtual. Para que los datos puedan ser procesados, es necesario descargar el par de particiones en la Máquina Virtual y mediante los datos del elemento de iterdata que le haya tocado a esa función (Iterdata es una lista de objetos).

A la función 1, le tocaría la *partición fasta 1* y la *partición fastq 1*. La partición fasta se descarga en el sistema operativo mediante el método get_object de la API de Lithops desde object storage. Mientras que la partición de fastq se descarga utilizando Data Lakes y la aplicación *fastq-dump.* Esto se corresponde *a fasta-chunk y fastq-chunk de la Figura 8.*

4.2 Motivación de la arquitectura

Sabemos que es un problema proveer de datos a workflows que requieren de datos de forma constante, y más todavía si estos tratan con datos grandes *o big data*. Algunas soluciones pasan por la compresión, indexación y posterior descompresión, esto traslada el problema a una nueva dimensión, pero no lo erradica.

La solución que propongo consiste en el uso de Data Lakes junto con *cloud computing y Serverless*, de hecho, sin *cloud* esta solución no sería posible. Por lo tanto, se trata de un requisito indispensable.

La idea consiste en mover *la computación cerca de los datos* mediante uso de Data Lakes públicos. Por lo tanto, se trata de erradicar el problema de tener que proveer datos a un pipeline de forma constante mediante la descarga y posterior subida y tener que posteriormente indexar los ficheros para poder descomprimirlos.

El problema del uso de secuencias comprimidas es que requiere de indexación para su posterior descompresión, esto hace que el pipeline pase cierto tiempo en esta etapa, y más si se tratan de secuencias grandes. La idea del particionador de secuencias mediante SRA y *fastq-dump* es utilizar Data Lakes para que los usuarios del pipeline tengan una gran variedad de secuencias a su alcance sin perjudicar el rendimiento.

4.3 Comprendiendo *fastq-dump*

Fastq-dump [13] es una herramienta que permite la descarga de secuencias a partir del *NCBI Sequence Read Archive (SRA).* Estas secuencias son descargadas directamente como ficheros *fastq* que es el formato requerido por el pipeline. Existen diversos parámetrosd para *fastq-dump*. Aunque existe una versión actualizada de *fastq-dump* denominada *fasterq-dump[14]* tiene una desventaja, y es que *fasterq-dump* solamente permite la descarga entera de la secuencia, es decir, no permite descargar la secuencia por particiones.

Lo que sí permite *fasterq-dump* es la descarga paralela utilizando múltiples hilos, *fasterq-dump* utiliza múltiples hilos para realizar la descarga y descompresión de forma más rápida.

SRA utiliza su propio formato binario de compresión (que además es propietario y privado) para almacenar las secuencias. Cuando un fichero está siendo descargado vía *fastq-dump*, se hace la descompresión.

Por lo tanto, los pasos para descargar una secuencia utilizando SRA a nuestro ordenador son los siguientes:

1. Se llama a *fastq-dump* junto a los parámetros necesarios y el nombre de la secuencia (Por ejemplo, *fastq-dump nombre_secuencia*)
2. Fastq-dump accede al servidor de SRA y busca por el nombre de la secuencia
3. El fichero de formato *.sra* se descarga y se realiza la conversión a *.fastq* de forma local.

Puntualizar la distinción entre local y en el cloud. Las velocidades de descarga son mucho mayores en el cloud por el uso de Data Lakes (tal y como se verá en el capítulo de Evaluación), dado que tenemos la información más cerca de nosotros que si la estuviéramos descargando de forma local.

En el caso del uso de fastq-dump en el cloud sucede lo siguiente:

1. Se llama a *fastq-dump* junto a los parámetros necesarios y el nombre de la secuencia (Por ejemplo, *fastq-dump nombre_secuencia*)
2. Fastq-dump accede a los Data Lakes y recupera la secuencia.
3. El fichero de formato *.sra* se descarga y se realiza la conversión a *.fastq* en el cloud.

4.4 Alineamiento de secuencias genomicas

El alineamiento de secuencias es la base del variant calling, por lo tanto, tocaría explicar en qué consiste este proceso.

El alineamiento de secuencias es el proceso de comparar dos o más secuencias genómicas para resaltar sus zonas de similitud, al resaltar similitudes es posible conocer la evolución de estas y clasificar las distintas mutaciones.

En 1970 Needleman y Wunsch desarrollaron el primer algoritmo para pares de secuencias. Desde entonces, se han desarrollado múltiples algoritmos capaces de desarrollar esta tarea. Una característica de estos algoritmos es que han evolucionado para que puedan ser altamente paralelos.

El problema de alineamiento de secuencias es un problema algorítmicamente complejo, que está basado en la optimización y la programación dinámica.

4.5 Diseño de un particionador de secuencias mediante fastq-dump y Lithops

Mediante el uso de Lithops y *fastq-dump* se ha desarrollado un particionador de secuencias. Lo que permite el particionador es, que cuando el científico que quiera ejecutar el workflow pueda hacerlo solamente especificando el nombre de la secuencia y la cantidad de reads que ha de tener cada partición. Dado que una de las ventajas de serverless es la capacidad de elasticidad, el particionador se adaptará y ejecutarla tantas Lambdas o se crearan tantas particiones como:

$$P = \left\lceil \frac{totalreads}{readsperpartition} \right\rceil$$

Donde, *total reads* es el número total de reads de la secuencia.

Reads per partition es un parámetro especificado por el usuario, que denota el número de reads que ha de tener cada partición.

Y finalmente, P denota el número de particiones totales. Lambdas o map functions dentro del modelo Map Reduce que se ejecutan.

Por lo tanto, como **inputs** para el particionador de secuencias, tenemos la cantidad de reads que ha de tener cada partición y el nombre de la secuencia.

El particionador de secuencias se consiste en distintos componentes que lo hacen posible, realmente no utiliza el particionador de Lithops, sino que se llama a un proceso de fastq-dump por cada función de la etapa del map de forma síncrona, ya que no se puede seguir a la siguiente etapa (dentro de la función map) hasta que no se ha descargado la secuencia.

4.5.1 Esearch API para obtener el número total de reads.

La primera etapa consiste en obtener los metadatos de la secuencia, ya que desde *fastq-dump* no es posible conocer la cantidad total de reads que tiene la secuencia.

Para poder particionar la secuencia, necesitamos saber cuántas reads tiene en total, a partir de ello y las reads requeridas por partición, podemos generar todas las particiones necesarias.

Un requisito es que el número de reads por partición ha de ser introducido por el usuario que está utilizando el pipeline, por lo tanto, nos podemos olvidar de problemas de equilibrio de carga (Load balancing) en la etapa del map. La cantidad de reads es un argumento que se le pasa al pipeline por la línea de comandos del sistema operativo.

Este proceso ha pasado por dos iteraciones distintas, la primera donde se utiliza una dependencia dentro del contenedor y la segunda donde se elimina la dependencia y se utiliza un método Python que permite la obtención de los metadatos mediante la API de NCBI denominada Esearch [16].

Los metadatos solamente se obtienen una vez y al principio de la ejecución del pipeline. El método se ejecuta desde el sistema operativo del usuario, sin necesidad de utilizar el cloud para este proceso.

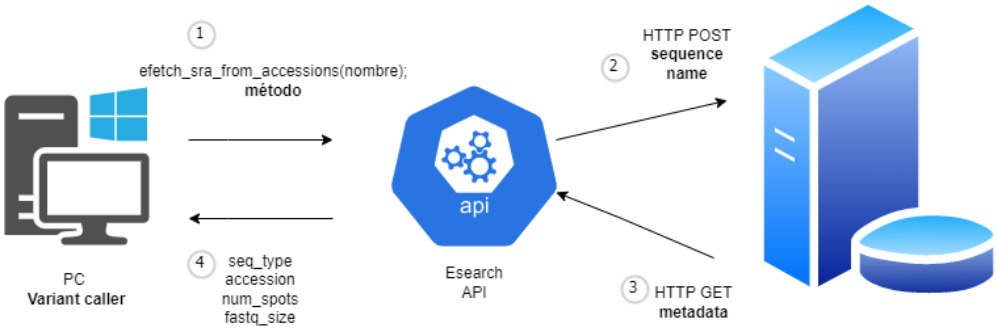

Figura 9. Comunicación variant caller con la API de NCBI (Esearch)

1. Se ejecuta el método efetch_sra_from_accessions(nombre de la secuencia).
2. Mediante el uso del api de *NCBI* denominada *Esearch* se utiliza un método en Python para ejecutar un HTTP Post. La api identifica las secuencias mediante el nombre que se ha pasado por parámetro en el método anterior
3. La api hace una consulta al servidor, y recupera los metadatos.
4. Los metadatos son retornados al variant caller mediante el método llamado en el punto 1.
    a. **Seq_type:** define el tipo de secuencia, existen 2 tipos distintos; *paired-end o single-end.* Es un dato necesario dado que se trata diferente la secuencia en el alineamiento dependiendo de si se trata **paired-end o single-end.**
    b. **Accession:** tupla donde se guardan todos los metadatos.
    c. **Num_spots:** Número de reads de la secuencia, se utiliza para generar tantas funciones en la fase de map como sean necesarias, de forma elástica.
    d. **Fastq_size:** Tamaño de la secuencia en Megabytes.

Estos datos se presentan al usuario mediante prints, realmente los datos necesarios para generar las particiones son el tipo de secuencia y el número de reads/spots.

El formato de los datos que retorna la API es de tipo XML, pero se convierten a una tupla utilizando Python.

Este paso es el que permite cumplir objetivos como “No alterar el flujo de trabajo”, ya que el usuario del pipeline no tiene que introducir la cantidad de reads totales o metadatos, sino que introduce el nombre de la secuencia y se olvida.

Figura 10. Formato XML, metadatos secuencia en Esearch, HTTP, GET.

4.5.2 Preprocesamiento de la secuencia

Una vez obtenidos los metadatos de la secuencia, es posible empezar la siguiente etapa que es la de preprocesamiento. El preprocesamiento consiste en preparar la cantidad de reads que irán por cada función dentro de la etapa de map. Básicamente que parte de la secuencia procesará cada función en la etapa de map.

En este caso, después de que se hayan recuperado los metadatos, se llama a la siguiente función:

```
prepare_fastq(cloud_adr, BUCKET_NAME, idx_folder, fastq_folder,
fastq_chunk_size, seq_type, fastq_file, seq_name,datasource,num_spots,
fastq_file2=None)
```

Figura 11. Cabecera función prepare_fastq

Esta función se encarga del procesamiento de la secuencia, pero se llaman a 2 funciones distintas dependiendo de si el fichero está comprimido (fastq.gz) o si el usuario ha decidido que prefiere utilizar SRA como su base de datos.

En el caso de SRA, se llama a la siguiente función:

```
preprocess_fastqsra(num_spots, fastq_chunk_size)
```

Figura 12. Cabecera función preprocess_fastqsra

Lo que hace la función **preprocess_fastqsra** es que basado en el **número de reads total** (recibidos de los metadatos) y la **cantidad de reads por particion** especificada por el usuario se genera un string.

El string está formado por 2 columnas, y cada fila representa una partición de la secuencia. Los valores inicio y fin depende de lo que haya insertado el usuario, si quiere que cada partición sea de tamaño 1000, entonces:

PARTICIÓN	INICIO	FIN
1	1	1000
2	1001	2000
…	…	…
N	2000	NUM_SPOTS

Tabla 4. Formato de los metadatos de la secuencia.

Es decir, se crea una tupla donde cada partición depende del tamaño que desee el usuario por partición. Para tener un formato uniforme entre la variante de secuencias comprimidas y por SRA, es necesario ejecutar la siguiente función:

```
generate_fastq_chunk_list_fastq_sra(data,seq_name)
```

Figura 13. Cabecera función generate_fastq_chunk_list_fastq_sra

Que toma la anterior tupla en forma de string y la transforma a un formato similar a json, cada partición va encabezada por el nombre de la secuencia. Por último, se retorna la lista en la función **prepare_fastq**.

Por lo tanto, la etapa de procesamiento se encarga de generar una tupla que contiene los distintos rangos de las distintas particiones, tomando como parámetro la cantidad total de reads que tiene la secuencia y la cantidad de reads por partición que el usuario desea.

4.5.3 Iterdata (Producto cartesiano entre la referencia y la secuencia)

Iterdata es la lista que se manda a las distintas funciones de map y donde se generan las particiones. Se trata de la información o metadatos necesarios para la etapa de map, que es donde se recuperan las particiones.

En este caso es una lista, donde en cada posición hay 2 objetos, uno representa a la partición de la referencia y el otro a la partición de la secuencia (con formatos fasta y fastq respectivamente.)

Cada elemento de la lista de iterdata generará una función Lambda, y sobre esas particiones de la referencia y la secuencia se ejecutará la función. Esto corresponde a la etapa del Map en el modelo MapReduce.

El iterdata es un producto cartesiano, donde se combinan todas las particiones de la referencia con todas las particiones de la secuencia.

El formato de la lista de iterdata es el siguiente:

1: {'fasta_chunk': 'fasta-chunks/hg19_640000000split_0.fasta', 'fastq_chunk': ('ERR9729866', {'number': 1, 'start_line': '0', 'end_line': '1000000'})}

Donde el fasta_chunk corresponde a la partición de la referencia, y fastq_chunk la particion de la secuencia.

En el iterdata hay tantos elementos como el producto entre particiones fasta y particiones fastq generadas, además, ese mismo número será el número total de funciones a las que se llamarán en la etapa de map. Para el alineamiento, que se ejecuta en la etapa de map, se prueban todas las combinaciones entre particiones.

4.5.4 Particionado de secuencias a partir de los metadatos

Una vez se han generado los datos para poder particionar la secuencia (preprocesamiento) ya se puede empezar la primera etapa del MapReduce, en este caso, el map.

Esta etapa consiste en llamar a la función map implementada en Lithops.

La función de map de lithops necesita como argumento *iterdata*, que es lo que se ha explicado en el apartado anterior, además de la función que va a ejecutar que se denomina

```
map_alignment(id, fasta_chunk, fastq_chunk, storage)
```

Figura 14. Cabecera función map_alignment

Dentro de esta función es donde se hará el particionamiento “on-the-fly” de la secuencia. Esto consiste en la descarga de cada partición de la secuencia dentro de las distintas funciones que se ejecutarán en la etapa del map. No se utiliza object storage, sino que las secuencias se descargan directamente en la MicroVM.

Para entender qué sucede en esta etapa, primero hay que entender que es la primera etapa que se ejecuta en el cloud mediante el método map() de la API de lithops.

El método map() llamará a tantas funciones como elementos tenga la lista de iterdata, cada elemento del iterdata será lo que es procesado por la función. Estas funciones que se ejecutarán son las denominadas serverless functions y en AWS se denominan Lambda.

Estas funciones Lambda son contenedores independientes entre ellos, con un sistema operativo que permite hacer cualquier cosa, aunque está pensado para cargas de trabajo livianas. Estos contenedores son máquinas virtuales, o más concretamente *MicroVMs* que utilizan Firecracker, Firecracker es un monitor de máquinas virtuales que utiliza un Kernel basado en Linux para crear las distintas *MicroVMs*.

Lambda al final es una *MicroVM* con cierto número de vCPUs (cpus virtuales), memoria efímera y ram. Estos parámetros pueden ser cambiados por el desarrollador de forma dinámica y entre ejecuciones.

Podemos configurar estas *MicroVMs* de Lambda mediante imágenes Docker, que són plantillas de aplicaciones para las *MicroVMs*.

Existe un pequeño inconveniente con lambda, y es que durante la ejecución de la función Lambda, solamente se tienen permisos de escritura en el directorio /tmp, es decir, que si hay algún fichero de configuración que se ha de modificar durante la ejecución de la fase de Map, hay que configurar la aplicación para que cree los ficheros de configuración en el directorio /tmp. De igual manera, las particiones han de descargarse en el directorio /tmp.

El problema de que se creen ficheros de configuración en tiempo de ejecución ha permitido crear una mejora sobre Lithops, y es que se permita modificar las variables de entorno desde la API de Lithops, sin tener que acceder a la interfaz del cloud provider para

modificarlas. Ya que esta opción no existía con anterioridad. Esto permite que se modifiquen los directorios donde se crean los ficheros de configuración, dado que ningún otro directorio tiene permiso de escritura, se crean en /tmp y luego se modifican.

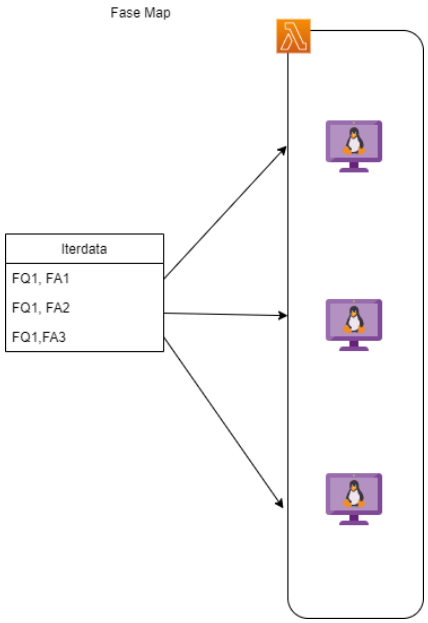

Figura 15. Iterdata y Lambdas

Dentro de la etapa de Map, que representa la función **map_alignment** se particionan las secuencias, también se alinean las secuencias con sus referencias, pero esto está fuera del alcance de este proyecto. Por lo que explicaré como funciona el particionado de secuencias que también sucede durante esta etapa.

La última etapa de particionado es la de descargar las secuencias en las MicroVM durante el tiempo de ejecución. La función encargada de esto es:

```
fastq_to_mapfun(fastq_n, fastq_file_key, fastq_chunk_data, BUCKET_NAME, fas
tq_folder, idx_folder, datasource,stage,id,debug)
```

Figura 16. Cabecera fastq_to_mapfun

Básicamente, se encarga de configurar el entorno en tiempo de ejecución para poder descargar las secuencias utilizando los Data Lakes de AWS y se descarga mediante fastq-dump y los datos proporcionados por la tabla de iterdata. Finalmente, la partición se almacena en el directorio /tmp para que se pueda realizar el alineamiento junto con la referencia.

Este proceso de descarga de particiones sucede en paralelo, donde cada MicroVM obtendrá su partición al mismo tiempo (teóricamente).

Además, este paso permite cumplir con un objetivo inicial y es el acceso rápido a la información. Mediante el uso de Data Lakes.

Por último, me gustaría finalizar con un diagrama de alto nivel de la arquitectura para el particionador de secuencias fastq mediante el uso de fastq-dump.

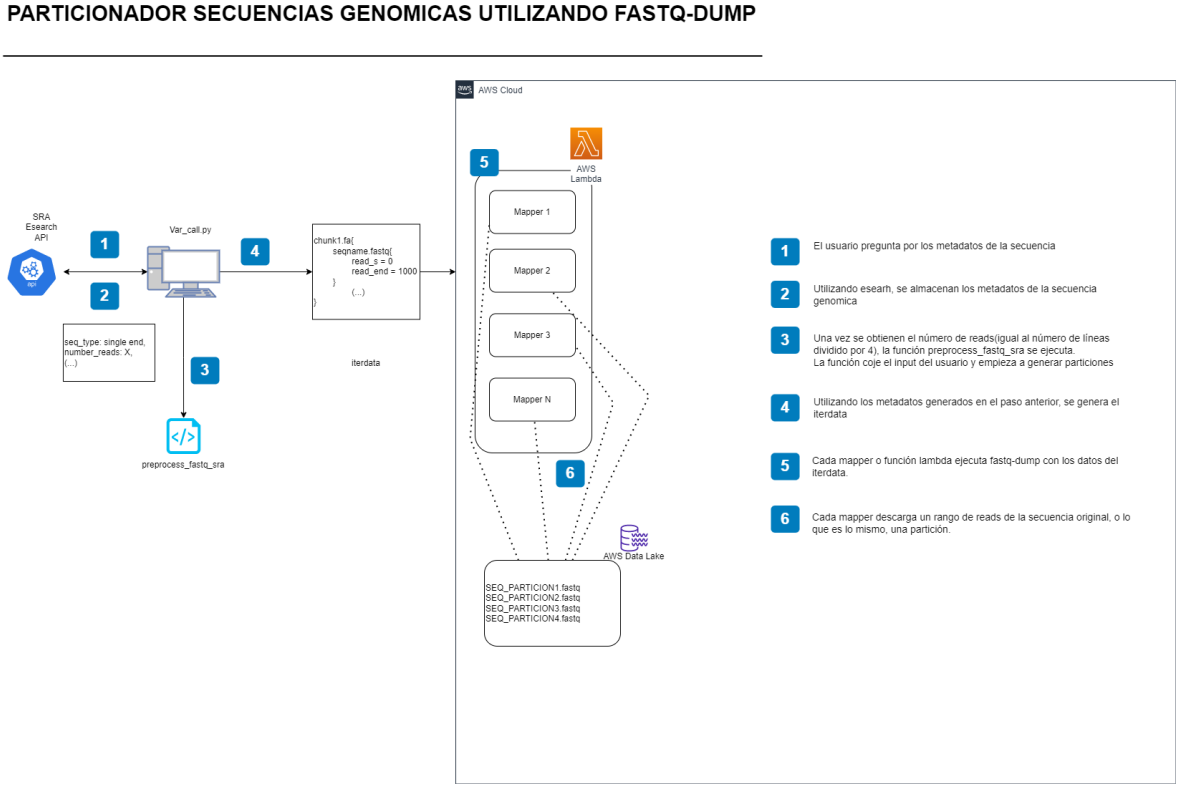

Figura 17. Arquitectura del particionador de secuencias genómicas con todos sus componentes

4.6 Diseño de un particionador de referencias fasta con Lithops y el modelo MapReduce

El siguiente particionador se trata del particionador de referencias genómicas (fasta), que junto al particionador de secuencias genómicas, forman la etapa de preprocesamiento.

Este particionador se basa en el modelo MapReduce explicado con anterioridad, pero sin utilizar el método map_reduce() de la API de Lithops, sino map() y una llamada asíncrona.

El particionador está compuesto por 3 etapas distintas, 2 forman el map_reduce y una última genera datos que son necesarios para la etapa del reduce.

El objetivo de esta etapa es conseguir particionar las referencias genómicas, un paso un tanto complejo, ya que hay ciertos prerrequisitos.

En este caso sí se utiliza el Lithops Partitioner, por lo tanto, empezaré explicando su funcionamiento.

4.6.1 Introducción al problema

Una vez acabado el particionador de secuencias (fastq), para probar el rendimiento, empecé a experimentar con el pipeline.

La etapa de preprocesamiento, que consiste en generar las particiones o chunks de la secuencia y referencia, tardaba un tiempo considerable y no era culpa del particionador de secuencias desarrollado.

Tras dejar que los experimentos corrieran por la noche, me percaté de un nuevo problema con la etapa de preprocesamiento, y esto era la forma en las que se generaban las particiones de la referencia (fasta). Para hacernos una idea, particionar una referencia de 3 GB podía llevar varias horas sin problema. Por lo que me puse a investigar el código que generaba las particiones para las referencias.

Cada vez que se particionaba una referencia con una longitud de partición diferente, se pasaba por el siguiente proceso.

1. La referencia completa, es decir, el fichero de formato fasta se guarda en object storage del cloud provider.
2. Se descarga la referencia al ordenador de usuario, si el fichero tenía un peso considerable se tenían que esperar un par de horas a que se descargara.
3. Una vez se descarga la referencia desde el cloud provider, se particiona de forma local, este proceso depende del hardware del usuario, pero era otro proceso que tardaba otras tantas horas.
4. Una vez particionada la referencia, que se particionaba mediante un script de bash, cada uno de los chunks o particiones se volvían a subir al object storage.

Por lo tanto, el proceso de partición de referencias genómicas era un proceso que dependía totalmente del hardware y recursos del usuario, en vez de utilizar las ventajas que proporciona el cloud. Realizar experimentos sobre el pipeline era cuestión de horas cada vez que se cambiaba la longitud de los chunks de la referencia genómica. Para poder evaluar el rendimiento se necesitaban horas e incluso días.

La idea es similar al particionador de secuencias, pero en este caso no se utiliza middleware (fastq-dump), sino que el particionador solamente necesita de Lithops para funcionar, además, se le puede proporcionar información desde un cloud storage ajeno y funcionaría perfectamente.

Por otro lado, hay un problema añadido respecto al particionador de secuencias. En el particionador de referencias los cromosomas difieren en tamaño, haciendo imposible dividir sin tener que indexar la referencia primero, ya que para particionar la referencia de forma correcta, se han de saber donde se encuentran los títulos y que partes forman de qué cromosoma.

Si, por ejemplo, un cromosoma fuese dividido en dos particiones, un requisito indispensable sería poder identificar el cromosoma en la segunda partición como parte del mismo cromosoma, esto supone un problema, porque no existen mecanismos como IPC (Interprocess communication) en el cloud, por lo tanto, no habría manera de comunicar a la segunda partición que forma parte del mismo cromosoma. Esto impone una serie de problemas a la hora de crear una arquitectura.

4.6.2 Lithops Partitioner

Lithops incluye un particionador que permite la división de ficheros para poder ejecutar cada pequeña porción en distintas funciones, al igual que si quisiéramos paralelizar la lectura de un fichero, y dividiéramos la lectura en diferentes threads.

La idea del particionador de Lithops es que cuando se le proporcione un fichero, lo lean diversas funciones de modo que se haga una lectura en paralelo, es necesario que el fichero a particionar se encuentre en object storage propio o ajeno.

```
def line_counter_in_chunk(obj):
    counter = {}
    data = obj.data_stream.read()

    for line in data.decode('utf-8').split('\n'):
        if line not in counter:
            counter[line] = 1
        else:
            counter[line] += 1
    return counter


if __name__ == "__main__":
    data_location = 'cos://bucket_name/file_name.csv'
    size = int(6.7 * pow(2,20))  # ~6.7MiB - arbitrarily chosen chunk size in bytes

    fexec = lithops.FunctionExecutor()
    fexec.map(line_counter_in_chunk, data_location,obj_chunk_size=size)
    res = fexec.get_result()

    with open('logs/map_output', 'w') as f:
        f.write(str(res).replace('{','\n{'))
```

Figura 18. Código particionador lithops.

Como se ve en la figura anterior, es necesaria una función que procese las distintas particiones (line_counter_in_chunk). Esta función se ejecutará por cada partición generada, y el número de particiones depende del tamaño de estas.

Una vez se especifica el tamaño de cada partición y la dirección donde se encuentra, la función map de la API de Lithops se encarga de particionar el fichero.

Una ventaja de que sea Serverless es que de encontrarse el fichero en el cloud y este sea público (ya sea en el object storage de un usuario o en un Data Lake), no hace falta subirlo a nuestro object storage, un ejemplo de esto es *Board Genome Reference* [17], que nos proporciona direcciones de referencias genómicas comunmente usadas, de este modo, podemos particionar directamente sin tenerlo que subir a nuestro object storage privado.

4.6.3 Indexado de la referencia (Fase de Map)

Para poder particionar las referencias genómicas, primero hace falta saber dónde se encuentran las cabeceras en cada partición, por lo tanto, la primera etapa trata de generar metadatos a partir de las distintas particiones, utilizando el particionador de Lithops.

Por lo que la primera etapa de map es que cada función lea lo que hay en una cantidad de bytes que se recibe por input, entonces, se crean P particiones dependiendo de la cantidad de bytes que se leen por función. Básicamente, esta etapa se trata de indexar el *.fasta* para posteriormente poder modificar las cabeceras.

Una vez el particionador de Lithops actúa y divide la referencia en P particiones, se pasa una regex para ver donde se encuentran los títulos de los cromosomas y el contenido.

El matching se hace con la siguiente regex:

```
>.+\n
```

Además, se añade un overlap también definido por el usuario, el overlap consiste en añadir una cantidad de bytes al inicio y final, es un requisito que tienen las particiones de la referencia.

En esta fase, todos los datos para generar las particiones ya han sido creados, aunque habría que hacer modificaciones sobre los títulos de las particiones para reducir el tamaño del .mpileup.

Al final de esta etapa se guardan tantos ficheros con formato *.data* como particiones se crearan. A continuación el formato.

```
0,6x
6,33554732
```

La primera línea corresponde a un título, y nos dice que el título se encuentra en el rango de bytes del byte 0 al byte 6, los títulos se marcan con x. La siguiente línea corresponde al contenido del cromosoma, la secuencia de caracteres del cromosoma como tal. 33554732 es el tamaño de la partición.

Se generarán tantos *.data* como particiones creadas.

Esto es, cuanto a la fase de indexación, existe una herramienta que hace exactamente lo mismo, indexar los *.fasta*, pero en este caso se trata de la misma funcionalidad, pero distribuido, por lo que se pueden realizar ciertas comparaciones.

En el caso de la versión distribuida, se utilizan 47 workers o vCPUs y 2^25 bytes de tamaño de partición, la etapa de map incrementa la cantidad de workers basado en el tamaño de la partición, por lo tanto, funciona de forma elástica.

Nombre	Tamaño	Versión distribuida	Samtools faidx
Hg19	3gb	13.00 seconds	26.855 seconds

Tb927_01_v5.1	25.7 mb	9.4 seconds	0.285 seconds

Tabla 5. Indexación versión distribuida vs. samtools faidx

Para las referencias de menor tamaño, la indexación se hace mejor de forma local dado que no hay tiempo de invocación de funciones ni coldstart, pero para referencias más grandes del orden de los gigabytes las ventajas son obvias.

De este modo tenemos una herramienta capaz de hacer lo mismo que samtools faidx [18] pero mucho más rápido para referencias de mayor tamaño.

El objetivo de esta etapa es trasladar el problema a otro "plano", ya que particionar directamente sobre la referencia es complicado y toma más tiempo.

Esta etapa lo hace la función:

```
def generate_chunks(self,obj,total_obj_size,obj_size,overlap)
```

Figura 19. Función generate_chunks

Y se hace en paralelo.

4.6.4 Generar nuevos títulos para los cromosomas (Etapa Reduce)

Una vez se tienen los índices, hace falta enumerar los cromosomas. Normalmente, los cromosomas contienen nombres largos y esto se hace para ahorrar almacenamiento efímero en las etapas siguientes.

Para identificar las particiones, en vez de utilizar nombres que pueden ser largos y ocupar varios bytes, se utilizará el siguiente formato:

>indicecromosoma_subindice

El *indicecromosoma* sirve para saber si se trata de un nuevo cromosoma, un cromosoma distinto. El subíndice sirve para saber si parte del cromosoma anterior ha sido particionado por una función distinta. Dado que las particiones se hacen utilizando el Lithops Partitioner que funciona mediante una llamada a la función map(), es posible que un cromosoma esté dividido en diferentes workers-funciones-mappers, por lo que es necesario identificarlos.

Al final de la llamada síncrona, que es una función de la API de Lithops para ejecutar una sola función en el cloud, se recorrerán todos los ficheros de metadatos *.data* de forma iterativa y en orden y los .data se transformarán en el siguiente formato:

>1_1, >chr1

6,33554732

Donde el *>1_1* es el título nuevo y *>chr1* es el título antiguo, esto se hace porque ha de ser un cambio revertible, entonces no hay otra manera de hacerlo que no sea guardando el título del cromosoma anterior.

Esta etapa la hace la función:

```
generate_index_file_titles(total_obj_size)
```

Figura 20. Cabecera función generate_index_file_titles

Y se hace de forma secuencial.

4.6.5 Generar las particiones de la referencia a partir de los metadatos

Una vez tenemos los metadatos correctos, hace falta generar las particiones de la referencia a partir de los metadatos, esto se hace en paralelo utilizando la función map() de la API de Lithops.

Para poder generar las particiones de la referencia *.fasta*, se llama a la función:

```
generate_chunks_corrected_index(id,r,storage)
```

Figura 21. Cabecera generate_chunks_corrected_index

Desde el map, donde cada función leerá los metadatos en paralelo y generará un fichero .fasta que corresponde a una partición.

Cada función generará un fichero que se guardará en object storage.

La justificación de guardarlo en object storage es que en teoría, la partición solamente se hará cada vez que se quiera cambiar la referencia genómica, pero a modo experimental, se ejecuta cada vez que se cambia de tamaño de partición, para probar el rendimiento del pipeline según se cambie el tamaño de las particiones de la referencia genómica.

Por último, si existe la referencia genómica se particionó con anterioridad, dado que se almacena en object storage, el pipeline se saltará el proceso de particionado y simplemente utilizará las particiones que ya existen en object storage.

Finalmente, el particionador representa una clase totalmente independiente del pipeline denominada FastaPartitioner.py, y se llama a un subproceso mediante el pipeline cada vez que es necesario particionar, el subproceso ha de ser síncrono dado que sin particiones .fasta no se puede seguir al siguiente paso.

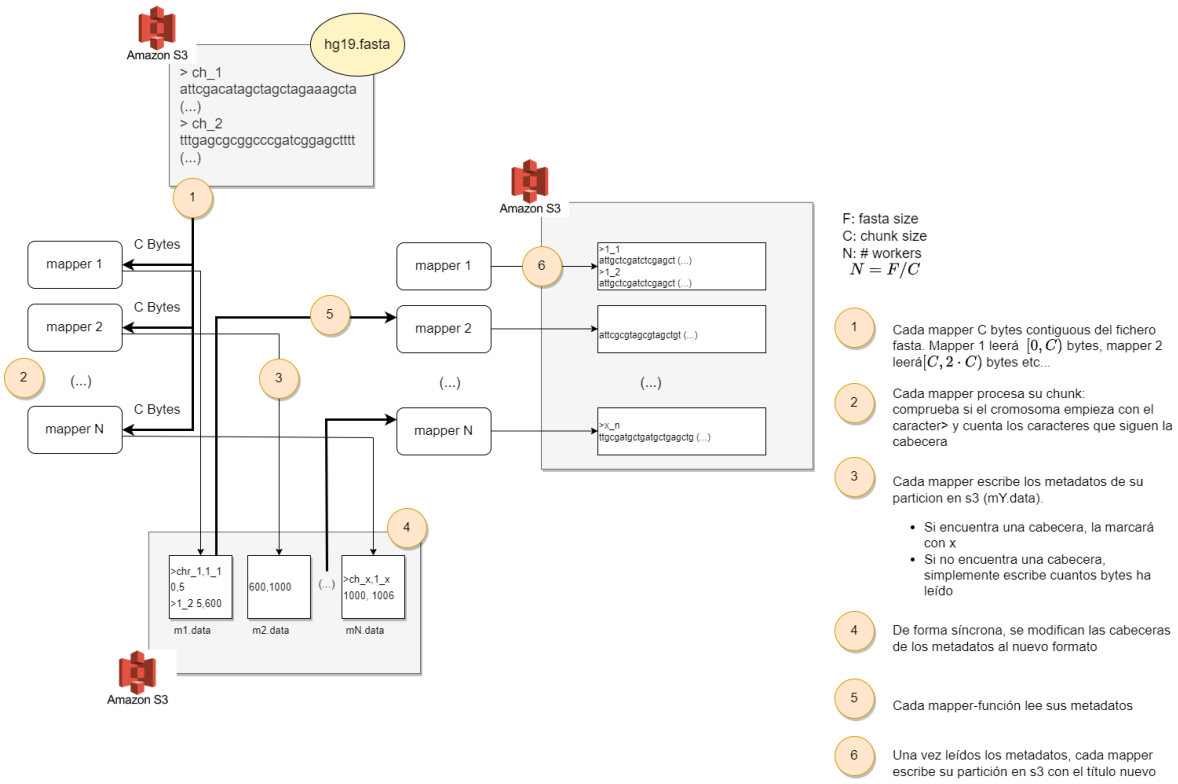

Figura 22. Componentes del particionador de referencias.

4.7 Relación entre componentes.

Por último, me gustaría aclarar la relación entre componentes del pipeline y sobre todo del preprocesamiento, a pesar de que ya se han ido aclarando a lo largo de los párrafos

Ambos particionadores (de referencias .fasta y de secuencias .fastq) acaban formando el iterdata, que son los ficheros necesarios para empezar la etapa del map_alignment del pipeline, que es donde se hace la alineación. Todas las componentes anteriormente mencionadas son de preprocesamiento y necesarias para poder empezar la etapa del map.

Hay una diferencia fundamental entre ambos particionadores, el particionador de secuencias no genera las particiones hasta que se ejecuta la etapa del map, y no se almacenan en object storage porque las secuencias normalmente se modifican.

En cambio, las referencias se mantienen entre especies, es decir, si el científico está ejecutando el pipeline con secuencias de humanos, utilizará una referencia humana. Mientras que irá modificando los individuos (las secuencias .fastq).

5 Implementación

Una vez explicada la arquitectura, tocaría dar paso al apartado de implementación. En este apartado se pretende explicar como se ha implementado la arquitectura, con el código correspondiente.

5.1 Particionador de secuencias genómicas (*fastq*)

El objetivo de este apartado es explicar la implementación del particionador de secuencias *fastq*.

5.1.1 Obtener metadatos a partir de la API Esearch.

Este primer paso sucede cada vez que se ejecuta el pipeline, donde es necesario obtener los metadatos para poder realizar las particiones de la secuencia.

Una vez el usuario del pipeline introduce los datos necesarios para ejecutar el pipeline, se realiza una llamada a la API de SRA denominada Esearch, junto al indentificador de la secuencia, y esta retorna una lista con todos los metadatos.

Sé instancia a la clase *SraMetadata* en el **main**, que se encuentra bajo *Metadata.py*

```
metadata = Metadata.SraMetadata()
    arr_seqs = [fq_seqname]
    if datasource == "SRA":
        accession = metadata.efetch_sra_from_accessions(arr_seqs)
        seq_type = accession['pairing'].to_string(index=False)
        num_spots = accession['spots'].to_string(index=False)
        fastq_size = accession['run_size'].to_string(index=False)
        print("Retrieving data from sra...")
        print("Sequence type: " + seq_type)
        print("Number of spots: " + num_spots)
        print("fastq size: " + fastq_size)
```

Figura 23. Instancia clase SraMetadata y llamada a la api

Una vez instanciada la clase *SraMetadata*, se llama al método *efetch_sra_from_accessions*, dentro de esta función se configura la api para hacer la llamada y se llama al método *efetch_metadata_from_ids*, que es el que realiza la llamada a la api y guarda los datos en una tabla.

```
 def efetch_sra_from_accessions(self, accessions):
        accessions = list(set(accessions))
        if len(accessions) == 0:
            return []
        logging.info("Querying NCBI esearch for {} distinct accessions e.g.
{}".format(
            len(accessions), accessions[0]))
        sra_ids = []

        webenv = None
```

```
        request_term = ' OR '.join(["{}[accn]".format(acc) for acc in
accessions])

        retmax = len(accessions)+10
        params=self.add_api_key({
            "db": "sra",
            "term": request_term,
            "email": "aymanbourramouss@gmail.com",
            "retmax": retmax,
            "usehistory": "y",
            })
        if webenv is None:
            params['WebEnv'] = webenv

        res = self._retry_request(
            "esearch from accessions",
            lambda: requests.post(
                url="https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.f
cgi",
                data=params))

        root = ET.fromstring(res.text)
        if webenv is None:
            webenv = root.find('WebEnv').text
        id_list_node = root.find('IdList')
        sra_ids = list(set([c.text for c in id_list_node]))

        if len(sra_ids) == 0:
            logging.warning("Unable to find any accessions, from the list:
{}".format(accessions))
            return None

        logging.info("Querying NCBI efetch for {} distinct IDs e.g.
{}".format(
            len(sra_ids), sra_ids[0]))
        metadata = self.efetch_metadata_from_ids(webenv, accessions,
len(sra_ids))

        # Ensure all hits are found, and trim results to just those that are
real hits
        if RUN_ACCESSION_KEY not in metadata.columns:
            raise Exception("No metadata could be retrieved")

        metadata.sort_values([STUDY_ACCESSION_KEY,RUN_ACCESSION_KEY],
inplace=True)

        if len(metadata) != len(accessions):
            found_runs = set(metadata[RUN_ACCESSION_KEY].to_list())
            not_found = list([a for a in accessions if a not in found_runs])
```

```
            logging.warning("Unable to find all accessions. The {} missing
ones were: {}".format(
                len(not_found), not_found
            ))

        return metadata
```

Figura 24. Función efetch_sra_from_accessions

El método anterior es un wrapper para la función que realiza la llamada a la api. El método encargado de hacer la llamada a la api y guardarlos en un diccionario es el siguiente:

```
def efetch_metadata_from_ids(self, webenv, accessions, num_ids)
```

Figura 25. Función efetch_metadata_from_ids

Este retorna un dataframe con todos los datos asociados a una secuencia (por si son necesarios en el futuro).

Este código se basa en *Kingfisher* [19], un programa de código abierto que permite la descarga de secuencias y los metadatos asociados de forma local. Se han hecho algunas modificaciones para que incluya los metadatos necesarios.

Una vez se obtienen los metadatos, el siguiente paso es el procesamiento de la secuencia.

5.1.2 Procesamiento de los metadatos de la secuencia.

Una vez obtenido los metadatos, estos se han de procesar para generar las particiones de forma correcta. Este paso consiste en generar una tabla de rangos de reads para que las particiones de la secuencia puedan ser generadas de forma correcta.

Desde el **main** después de obtener los metadatos sobre la secuencia se llama a la función encargada de procesar los metadatos de la secuencia.

```
    fastq_list = lithopsgenetics.prepare_fastq(cloud_adr, BUCKET_NAME,
idx_folder, fastq_folder, fastq_read_n, seq_type, fastq_file, fq_seqname,
datasource, num_spots, fastq_file2)
```

Figura 26. Función prepare_fastq

Esta función tiene 2 comportamientos distintos, uno en caso de secuencias comprimidas, que implementó un compañero del grupo. Y para secuencias provenientes de SRA.

Si el datasource es “SRA” e independientemente de si la secuencia es “single-end” o “paired-end” se llamará a la función *preprocess_fastqsra.*

La función *preprocess_fastqsra* se encarga de crear un string de los rangos de las particiones.

```
def preprocess_fastqsra( num_spots, chunk_size):
    print("start of *preprocessfastq_file* function")

    end = 0
    ini = 0

    mod =  num_spots % chunk_size
    data = ""
    while end < num_spots:
        if end == num_spots-(chunk_size+mod):
            ini = end
            end = num_spots
            data = data + str(ini)+" "+str(end) + '\n'
        else:
            ini = end
            end = end + chunk_size
            data = data + str(ini)+" "+str(end) + '\n'

    print("end of *preprocessfastq_file* function")
    return data
```

Figura 27. Función preprocess_fastqsra

Finalmente, la lista se convierte en una tabla de particiones. Mediante la función generate_fastq_chunk_list_fastq_sra. Esta función simplemente se encarga de estandarizar el formato respecto a la variante comprimida. Transformando la cadena de caracteres en el siguiente formato:

[

'ERR9729866', {'number': 1, 'start_line': '0', 'end_line': '1000000'},

...

'ERR9729866', {'number': N, 'start_line': 'X-1', 'end_line': 'X'},

]

5.2 Particionador de referencias genómicas (fasta)

Una vez realizada la llamada a la función de prepare_fastq, que procesa la secuencia, queda procesar la referencia fasta. Toda esta etapa forma parte de lo que se denomina preprocesamiento

Por lo tanto, de nuevo en el **main** se llama a la siguiente función:

```
    fasta_list = lithopsgenetics.prepare_fasta(cloud_adr, runtime_id,
FASTA_BUCKET, fasta_folder, fasta_file, split_fasta_folder,
fasta_chunk_size, fasta_chunks_prefix, fasta_char_overlap)
```

Figura 28. Función prepare_fasta

Esta función tiene 2 tipos de comportamientos distintos.

1. Las particiones de la referencia no están almacenadas en object storage
2. Las particiones sí que fueron creadas con anterioridad.

Si las particiones no existen en object storage, entonces se crean mediante el método

```
split_fastafile(cloud_adr, runtime, BUCKET_NAME, fasta_folder, fasta_file,
split_fasta_folder, fasta_chunk_size, fasta_line_overlap)
```

Figura 29. Función split_fastafile

Que es un wrapper para el particionador de referencias, dentro de esta función se llama al particionador de referencias mediante la creación de un subproceso en Python.

```
    args = "python FastaPartitioner.py --data_location
"+"s3"+"://"+bucket_name+"/"+fasta_folder+fasta_file+" --chunk_size
"+str(fasta_chunk_size)+" --overlap "+str(chunk_overlap)+" --mybucket
"+bucket_name+" --fasta_folder "+split_fasta_folder+" --runtime "+runtime
    args = args.split(" ")

    total_objects = sp.check_output(args).decode('utf-8')
```

Figura 30. Llamada al subproceso que ejecuta el MapReduce para particionar la referencia

Dentro de *FastaPartitoner.py* nos encontramos con el particionador de referencias que sigue la siguiente estructura:

```
        #Generate indexes for a fasta file, to know where all the titles and sequences are
located.
        #Outputs chunk{i}.data files in s3, containing byte-ranges of sequence and titles
            fexec.map(…)
            fexec.wait()
            iterdata = fexec.get_result()

        #Generate alternative titles for the genomic sequences
            fexec.call_async(…)
            fexec.wait()

        #Once all byte-range chunks and alternative titles are generated, generate the fasta
chunks in s3.
            fexec.map(generate_chunks_corrected_index, iterdata)
            fexec.wait()

            fexec.plot()
            fexec.clean()
```

Figura 31. Función Split_fastafile

Que sigue la arquitectura MapReduce, explicada en el apartado 4.5.

Al final de la ejecución del subproceso, se escriben todas las particiones en object storage, mediante el método put_object de la API de Lithops, esto sucede en el último map.

Si se crearon con anterioridad, simplemente se listan utilizando el método

```
 storage.list_keys(BUCKET_NAME, prefix=split_fasta_folder +
fasta_chunks_prefix)
```

Figura 32. Función list_keys **Lithops API.**

Que devuelve una lista de los objetos con un prefijo, el prefijo consiste en la combinación del identificador de la referencia y el tamaño de la partición.

Esto se hace a modo de caché, para no tener que volver a generar particiones de la referencia que ya fueron creadas con anterioridad, lo que ahorra tiempo de ejecución y recursos.

Tanto si se generan como si fueron creados, el subproceso devuelve el nombre de cada una de las particiones de la referencia creadas.

Por último, tocaría ver la implementación del iterdata y como se generan las particiones de la secuencia y la referencia.

5.3 Iterdata y descarga de particiones de la secuencia y referencia.

Una vez se tienen los metadatos de la secuencia y la lista de particiones de la referencia, se crea el Iterdata, esto lo hace la función:

```
    iterdata = lithopsgenetics.generate_alignment_iterdata(fastq_list,
fasta_list, iterdata_n)
```

Figura 33. Función generate_alignment_iterdata

Como ya se ha comentado varias veces, en esta fase se hace el producto cartesiano entre las particiones fasta y fastq (de nuevo, secuencia y referencia).

```
def generate_alignment_iterdata(list_fastq, list_fasta, iterdata_n):
    """
    Creates the lithops iterdata from the fasta and fastq chunk lists
    """

    os.chdir(CWD)

    iterdata = []
    for fastq_key in list_fastq:
        for fasta_key in list_fasta:
            #print(fastq_key)
            iterdata.append({'fasta_chunk': fasta_key, 'fastq_chunk':
fastq_key})


    if iterdata_n is not None:
        print("iterdata subset elements to be parsed: " + str(iterdata_n))
        iterdata = iterdata[0:int(iterdata_n)]
        print("returning first" + str(iterdata_n) + " elements")
        print("end of *create_iterdata_from_info_files* function - single
end sequencing")
        return iterdata
    else:
        print("returning all iterdata elements")
        #print(str(iterdata))
        print("end of *create_iterdata_from_info_files* function - single
end sequencing")
        return iterdata
```

Figura 34. Código función generate_alignment_iterdata

La función retorna el producto cartesiano, con el formato especificado en el capítulo 5, sobre la arquitectura.

Una vez hecha la llamada, se guarda en iterdata como una tabla de objetos, siguiendo el formato que se especificó en el capítulo 5.

La siguiente etapa se hace en el Map, ya no es preprocesamiento. Esta etapa de map se hace sobre los workers o funciones Lambda y consiste en la descarga de las particiones para que puedan ser procesadas.

La siguiente etapa consiste en que una vez se llega a la etapa de **Map**, se ejecuta fastq-dump para descargar la secuencia sobre la función.

```
def fastq_to_mapfun(fastq_n, fastq_file_key, fastq_chunk_data, BUCKET_NAME,
fastq_folder, idx_folder, datasource,stage,id,debug):
    '''
    Function executed within the map function to retrieve the relevant fastq
chunk from object storage or SRA
    '''

    storage = Storage()
    if datasource == None:
        af.printl(fastq_n + " file key: " +
str(fastq_file_key),stage,id,debug)
        af.printl(fastq_n + " chunk data: " +
str(fastq_chunk_data),stage,id,debug)
        byte_range = f"{int(fastq_chunk_data['start_byte'])-1}-
{int(fastq_chunk_data['end_byte'])}"
        temp_fastq_gz = af.copy_to_runtime(storage, BUCKET_NAME,
fastq_folder, fastq_file_key, byte_range)

        # getting index and decompressing fastq chunk
        temp_fastq = temp_fastq_gz.replace('.fastq.gz',
f'_chunk{fastq_chunk_data["number"]}.fastq')
        temp_fastq_i = af.copy_to_runtime(storage, BUCKET_NAME, idx_folder,
f'{fastq_file_key}i')
        block_length = str(int(fastq_chunk_data['end_line']) -
int(fastq_chunk_data['start_line']) + 1)
        cmd = f'gztool -I {temp_fastq_i} -n {fastq_chunk_data["start_byte"]}
-L {fastq_chunk_data["start_line"]} {temp_fastq_gz} | head -{block_length} >
{temp_fastq}'
        sp.run(cmd, shell=True, check=True, universal_newlines=True)
    elif datasource == "SRA":
        seq_name = fastq_file_key

        sp.call(['chmod','+x','fastq-dump'])
        # To supress the vdb-config warning
```

```
        sp.run(['vdb-config', '-i'])
        # Report cloud identity so it can take data from s3 needed to be
executed only once per vm
        sp.run(['vdb-config', '--report-cloud-identity', 'yes'],
capture_output=True)

        os.chdir("/tmp")
        temp_fastq =
'/tmp/'+seq_name+f'_chunk{fastq_chunk_data["number"]}.fastq'
        data_output = sp.run(['fastq-dump', str(seq_name), '-X',
str(int(fastq_chunk_data["start_line"])) , '-N',
str(int(fastq_chunk_data["end_line"])),'-O','/tmp'],
                             capture_output=True)
        af.printl("data_output contents: " +
str(data_output),stage,id,debug)
        os.rename('/tmp/'+seq_name+'.fastq', temp_fastq)


    return temp_fastq
```

Figura 35. Código función fastq_to_mapfun **que descarga los chunks a cada uno de los workers de la etapa de map.**

Esta función es ejecutada en paralelo y por cada uno de los workers. Forma parte de la etapa de Map. Lo que hace esta función es configurar el entorno mediante el uso de *vdb-config* y descarga una partición mediante *fastq-dump* y el *iterdata* generado. Este proceso se hace en paralelo, y en cada una de las funciones.

La partición se descarga en el directorio /tmp, ya que es el único directorio que permite escritura.

Una vez ejecutada esta función, se tienen las particiones de la secuencia descargadas sobre los workers, pero falta descargar las particiones ya creadas de las referencias *.fasta*.

Este último paso, que también se hace en paralelo, consiste en descargar las particiones de la referencia mediante el método get_object de la API de Lithops. A diferencia de la descarga de secuencias, que se hace mediante los Data Lakes de AWS y en el object storage del usuario no se guarda nada, las particiones de la referencia sí que están almacenadas en object storage (a modo de caché), y se recuperan mediante el método get_object de la API de Lithops.

La función encargada de la descarga de particiones en paralelo, sobre las funciones en la etapa de map es:

```
def copy_to_runtime(storage, bucket, folder, file_name, stage, id, debug,
byte_range=None):
    printl(f'Copying {file_name} to /tmp folder',stage,id,debug)
    extra_get_args = {'Range': f'bytes={byte_range}'} if byte_range else {}
```

```
    obj_stream = storage.get_object(bucket=bucket, key=folder+file_name,
stream=True, extra_get_args=extra_get_args)
    temp_file = "/tmp/" + file_name
    with open(temp_file, 'wb') as file:
        shutil.copyfileobj(obj_stream, file)
        printl(f'Finished copying {file_name} to /tmp
folder',stage,id,debug)
    return temp_file
```

Figura 36. Código función copy_to_runtime **que descarga las particiones de la referencia desde S3 a cada uno de los workers en la etapa de map.**

y se llama de la siguiente forma:

```
        fasta = af.copy_to_runtime(storage, FASTA_BUCKET,
fasta_chunk_folder_file[0]+"/", fasta_chunk_folder_file[1], stage, id,
debug)
```

Figura 37. Función copy_to_runtime

Una vez las dos particiones son descargadas en la función, ya se puede empezar la etapa de procesamiento, que consiste en la alineación de variantes. Esto ya está fuera del alcance de este proyecto.

6 Evaluación

6.1 Etapa de preprocesamiento comparada con el resto del pipeline, tiempo de ejecución y datos que se almacenan.

Este es un experimento para ver el tiempo que toma cada etapa, y los datos que se procesan o se descargan sobre el almacenamiento efímero de Lambda. El experimento se hace sobre 3 funciones en total.

En la Figura 38 se ve la cantidad de datos que hay en el directorio /tmp de la función.

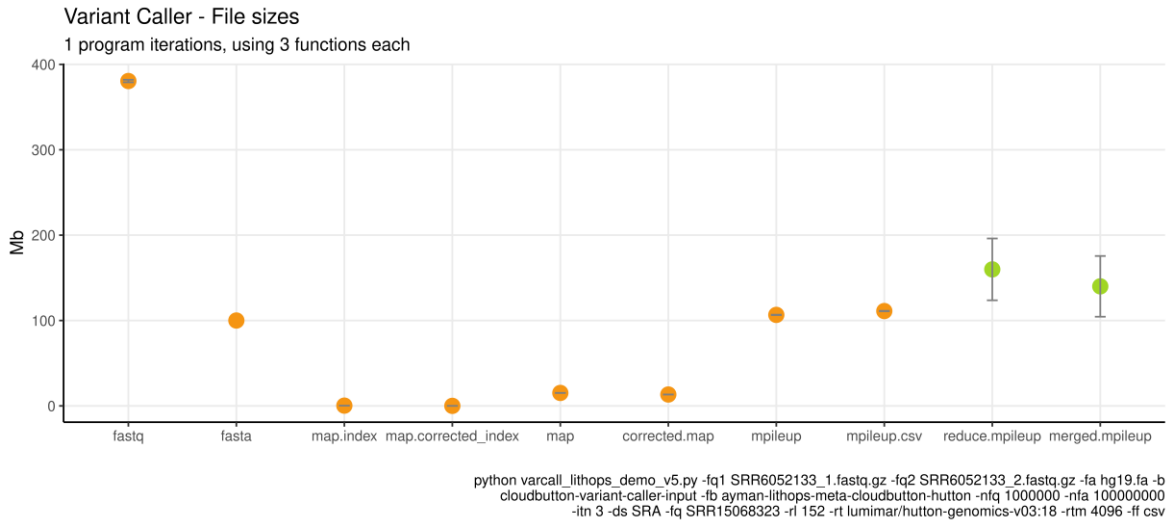

Figura 38. Datos que se almacenan utilizando el almacenamiento efimero de Lambda

En la Figura 39 se ve el tiempo que se tarda en generar o procesar estos datos. Si nos fijamos en Fastq chunk, la descarga tarda alrededor de 20 segundos para 400mb y 5 segundos para 100mb en el caso de Fasta chunk.

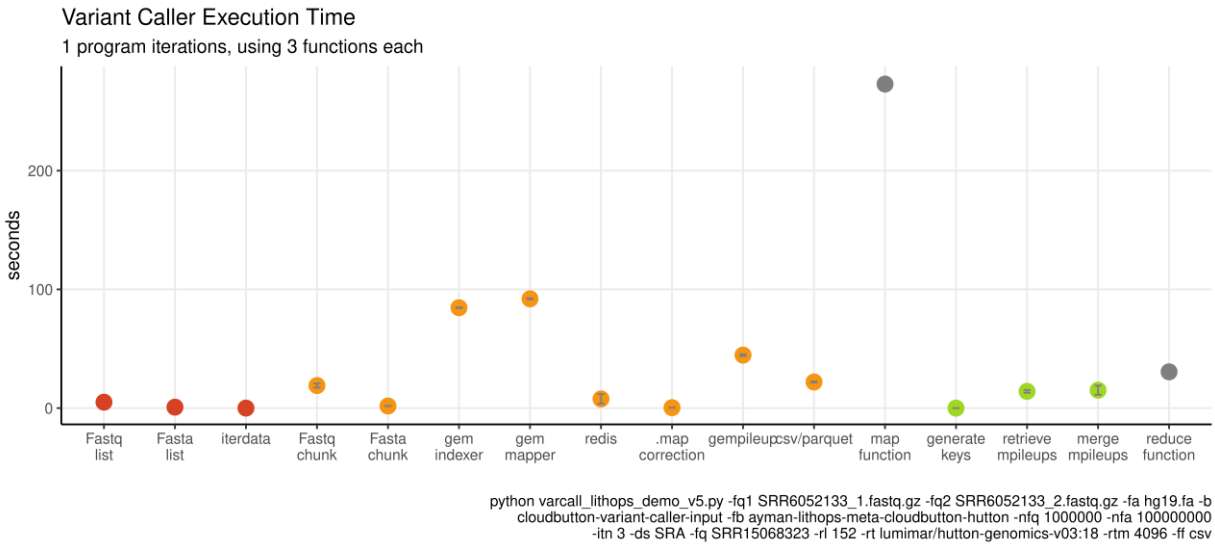

Figura 39. Tiempo que toma cada etapa del pipeline.

Lo que permite visualizar este experimento es que tan rápido es la etapa de **PRE-PROCESSING** comparada con el resto de las etapas y la cantidad de datos que se procesa. De este modo se puede ver en las gráficas cómo o que tan rápido es la etapa de preprocesamiento de forma visual. Realmente estamos viendo como se cumple el objetivo inicial de "Acceso rápido a la información". Esto demuestra como los particionadores descargan y procesan rápidamente la información, mediante el uso del propio object storage o los Data Lakes para las secuencias.

La etapa de preprocesamiento es la que más datos almacena en el directorio /tmp, mientras que es la más rápida comparada con el resto de etapas.

6.2 Tiempo de descarga de particiones Lambda vs Local

Dado que se ha hablado tanto de los Data Lakes, hay que comparar la diferencia de throughput entre local vs cloud. Este experimento consiste en ver cómo afecta el uso de *fastq-dump* en local (desde nuestro ordenador) *vs* el cloud.

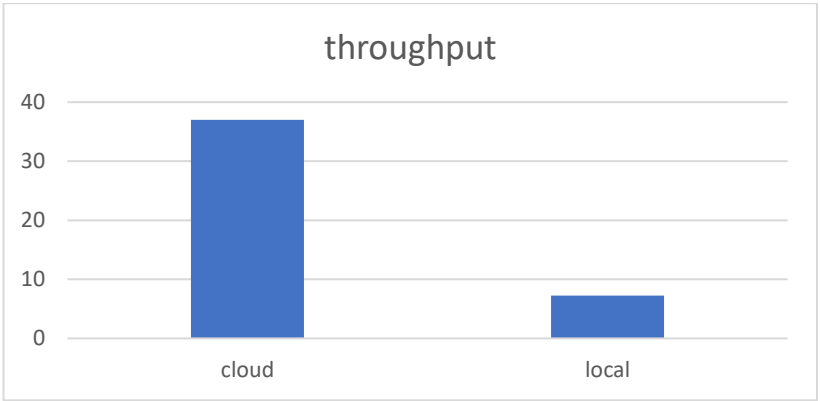

Figura 40. Throughput descarga fastq-dump cloud vs local

Como se ve en la *Figura 40*, obtenemos un throughput mayor utilizando el cloud. Además, el throughput local varía dependiendo de nuestra localización y ancho de banda, ya que la descarga se realiza mediante los servidores de SRA, mientras que en el cloud, se accede a los Data Lakes donde están almacenadas las secuencias.

El thoughput para el cloud es de 37 MB/s mientras que en local es de 7 MB/s.

6.3 Cantidad de tiempo necesario para que el particionador de fasta genere particiones en S3

En este caso, el experimento consiste ver la cantidad de tiempo que tarda en ejecutarse el particionador de referencias por si solo, ya que al final es una aplicación por si misma. En caso de ejecutarse el particionador de referencias, se tarda 13 segundos en generar las particiones de la referencia del ser humano (*hg19.fa*) mediante el uso de Serverless y Data Lakes (La referencia estaba en otro object storage distinto al del usuario, *s3://broad-references/hg19/v0/Homo_sapiens_assembly19.fasta*). Este mismo experimento, utilizando el procedimiento explicado en el capítulo *4.6.1* era cuestión de horas, mientras que, en el caso de la arquitectura implementada, es cuestión de 13 segundos.

Además, este grafico incluye todas las etapas, preprocesamiento de la referencia y particionado. Ya que el particionado de referencias se hace todo entero.

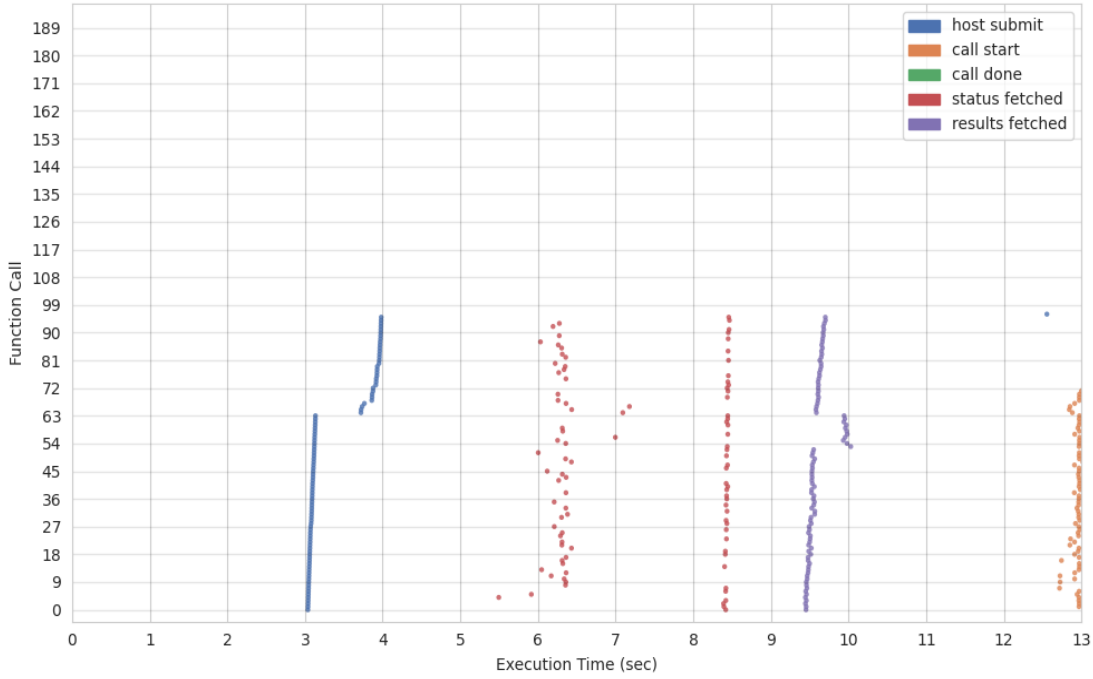

Figura 41. Tiempo necesario para generar particiones con 97 workers y fasta de 3gb

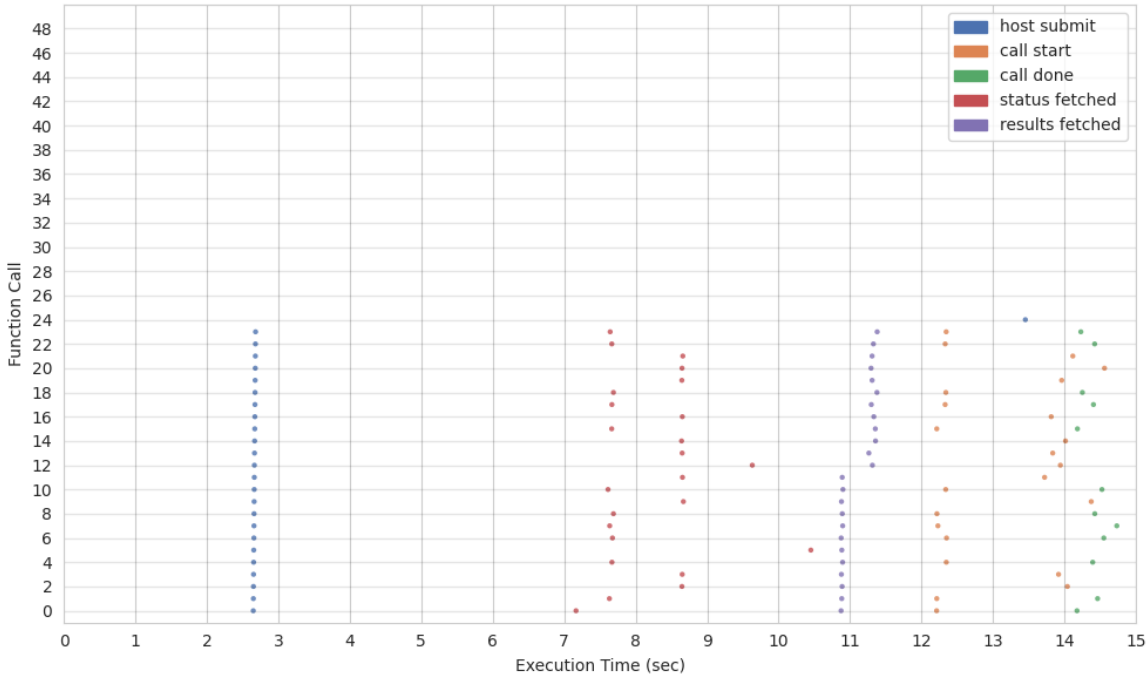

Figura 42. Tiempo necesario para generar particiones con 23 workers y fasta de 3gb

Por otro lado, si no es necesario generar particiones porque fueron creadas con anterioridad, el método *list_objects* de la API de Lithops tarda 0.6 segundos en obtener la lista de particiones de S3 (Esto se hace a modo de caché, para no tener que volver a crear particiones de una referencia ya particionada).

La cantidad de tiempo a particionar una referencia es *muy relativa.* Si bien es cierto que el tiempo necesario es mayor a medida que la referencia a particionar es mayor, se puede minimizar incrementando la cantidad de workers, de modo que cada worker procesa menor cantidad de información.

6.4 Cantidad de tiempo necesaria para que se haga el preprocesamiento de la secuencia.

En este caso se observa el tiempo medio para preprocesar las secuencias. Se está viendo el tiempo que se tarda en generar el fastq list de la **Figura 8.** Como se ve, es cuestión de 2 segundos para una secuencia de 10gb.

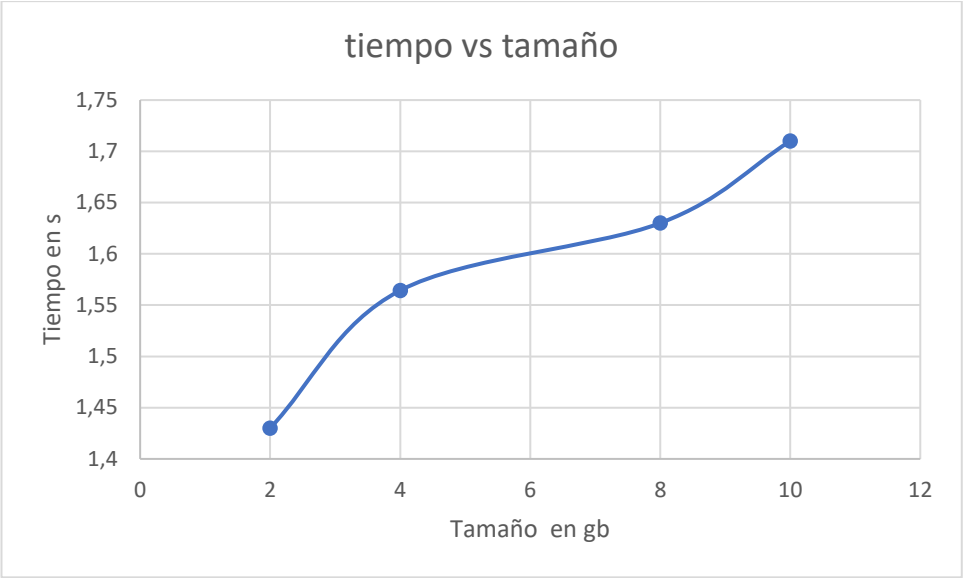

Figura 43. Tiempo vs. tamaño en procesamiento de reads fastq

En este gráfico se puede ver que a medida que crece el tamaño de la secuencia, crece el tiempo que se tarda en preprocesar. Se ha utilizado el mismo tamaño de partición en cada uno de los casos. A menor tamaño de partición, mayor el tiempo dada la naturaleza del algoritmo.

7 Conclusiones

Este proyecto ha supuesto un gran aprendizaje. He podido ver como se relacionan 2 campos completamente distintos y crean una intersección que desconocía, la bioinformática.

He podido colaborar con grandes profesionales y he sido capaz de responder a las necesidades del James Hutton Research Institute and Biomathematics & Statistics Scotland (BioSS). Un particionador de secuencias con un gran throughput y que está conectado a la mayor base de datos pública de secuencias genómicas existente y un particionador de referencias genómicas en el cloud, aprovechando tecnologías ya existentes como el Lithops Paritioner, reduciendo así un proceso que podía durar horas.

Como se ha visto en la evaluación y arquitectura, se han cumplido los objetivos propuestos, mediante el uso de API, en este caso *Esearch* he podido obtener todos los metadatos relacionados a una secuencia, sin afectar al flujo de trabajo, además acceso a datos variados mediante el uso de *fastq-dump*.

Gracias a los particionadores, los bioinformáticos tienen acceso a secuencias sin tener que realizar transferencias de datos entre sus ordenadores y el cloud. Pueden ejecutar sus experimentos simplemente insertando un nombre de una secuencia, olvidándose de tener que comprimir, indexar, realizar transferencias de datos y posteriormente descomprimir. Este es exactamente el beneficio que proporcionan los Data Lakes. El ahorro de tiempo y recursos.

El hecho de que existan Data Lakes cuyos datos son públicos y se actualizan como si de una base de datos se tratara es extremadamente beneficioso para los pipelines impulsados por datos como es el caso. Me atrevería a decir que veremos más pipelines de este tipo en el futuro**, donde la computación se mueve más cerca de los datos**.

Las ventajas que ofrecen los Data Lakes son muchas y dan sentido a nuevas arquitecturas en el cloud, cada vez más proveedores de datos comparten sus datasets con los cloud providers. No es raro encontrarse pipelines cuyos datos son obtenidos desde programas como AWS Open Data.

El presente es el cloud, y el futuro serán los Data Lakes públicos.

8 Referencias

[1] Eduardo Roloff, Matthias Diener, Alexandre Carissimi, Philippe O. A. Navaux , High Performance Computing in the cloud: Deployment, performance and cost efficiency.

[2] Pwint Phyu Khine and Zhao Shun Wang, Data lake: a new ideology in big data era

[3] Registry of open data (https://registry.opendata.aws/)

[4] FASTQ Format (https://en.wikipedia.org/wiki/FASTQ_format)

[5] FASTA Format (https://en.wikipedia.org/wiki/FASTA_format)

[6] Rasko Leinonen,[1,*] Hideaki Sugawara,[2] and Martin Shumway[3], on behalf of the International Nucleotide Sequence Database Collaboration, The Sequence Read Archive

[7] European Nucleotide Archive (https://www.ebi.ac.uk/ena/browser/)

[8] Variant calling (https://www.b-kl.eu/variant-calling/)

[9] E. Jonas et al., “Cloud Programming Simplified: A Berkeley View on Serverless Computing,” Feb. 2019.

[10] Working with open data on aws (https://www.slideshare.net/AmazonWebServices/working-with-open-data-on-aws)

[11] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López,

Outsourcing Data Processing Jobs with Lithops

[12] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters

[13] Fastq-dump (https://github.com/ncbi/sra-tools/wiki)

[14] Fasterq-dump (https://github.com/ncbi/sra-tools/wiki/HowTo:-fasterq-dump)

[15] Sequence alignment (https://en.wikipedia.org/wiki/Sequence_alignment)

[16] The E-utilities In-Depth: Parameters, Syntax and More, Eric Sayers, PhD.

[17] Board genome reference (https://s3.amazonaws.com/broad-references/broad-references-readme.html)

[18] Samtools-faidx (https://biolib.com/samtools/samtools-faidx)

[19] Kingfisher (https://biolib.com/samtools/samtools-faidx)