Universitat de Lleida

TREBALL FINAL DE MÀSTER

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Estudiant: Óscar López Luna

Titulació: Màster en Enginyeria Informàtica

Títol de Treball Final de Màster: Analysis and migration to Scala of the BLAST Algorithm

Director/a: Fernando Cores Prado

Presentació

Mes: Setembre

Any: 2022

Índice

Capítulo 1. Introducción del proyecto

Este capítulo ofrece la explicación a todo lo que envuelve el proyecto, como de este mismo. Además se habla también de los objetivos y de la motivación que me ha llevado a escoger este tema. Para concluir este capítulo tenemos la planificación temporal junto a las tareas a realizar.

1.1. Introducción

Este proyecto consiste en la implementación y análisis del algoritmo BLAST [1] (Basic Local Alignment Search Tool) usando tecnología big data (Spark [3]).

Antes de empezar, hace falta saber que el algoritmo BLAST o Basic Local Alignment Search Tool [1], es un algoritmo heurístico, lo que significa que no puede garantizar una solución correcta, y programa para mapear una secuencia problema (query) contra una gran cantidad de secuencias que se encuentren en una base de datos. Al finalizar esa comparación, el algoritmo ha encontrado todas las secuencias parecidas a la del problema y a través de un parámetro nos permitirá juzgar los resultados obtenidos.

Es importante saber que este algoritmo, es el más usado de todos en cuanto a anotación y predicción funcional de genes o secuencias proteicas. A causa de eso se han creado muchos programas y variantes [2]. Blastn [16], Blastp [16], BlastX [16], son ejemplos de programas de la familia BLAST y por otro lado, tenemos Gapped Blast [17], PsiBlast [17] y el WU BLAST [18] que son las variantes creadas a partir del algoritmo base.

Al analizar BLAST, se nos aparece la necesidad de manejar grandes cantidades de datos porque si hacemos uso de BLAST en un ordenador personal se visualizará como los recursos escalan considerablemente cuando tenemos multiples genomas de referencia, cada uno con un tamaño de más de 1 GB, y además, también se tendría que tener en cuenta que habrán múltiples queries procesandose concurrentemente, por lo tanto, tenemos delante una problemática de Big Data [4], (parte de la informática que se encarga desde el análisis de los grandes cantidades datos incluyendo tanto el tratamiento y visualización de estos). Eso implica, que necesitaremos una solución que no use los recursos locales sino que haga uso de la nube, es decir, recursos preparados para el manejo de tal cantidad de información. En consecuencia, se hará uso de Spark juntamente con el algoritmo Blast, ya que el hecho de aplicar tecnologías Big Data como lo es Spark, nos permitirá mejorar las prestaciones, el rendimiento y hasta la propia productividad, a

parte de así, evitar la problemática de falta de recursos cuando tenemos multiples genomas de referencia.

Para poder resolver la problemática planteada anteriormente, se ha de usar una arquitectura adecuada. En este caso, implicaría el uso de un cluster hadoop junto a spark, para mejorar, como se ha comentado en el párrafo anterior, las prestaciones, la productividad y el rendimiento. Aunque para que estas mejoras se han más efectivas se tiene que usar una base de datos adecuada para este tipo de problemáticas, ya que las típicas SQL DB suelen ir muy lentas, cuando es necesario consultar grandes cantidades de datos (MegaBytes/GigaBytes). En este escenario es más recomendable las bases de datos que basan su funcionamiento en el manejo de grandes cantidades datos (suelen ser NoSQL), como Cassandra [5], ya que facilitan la manipulación de cantidades de datos tan grandes en una sola consulta. Haciendo uso de las tecnologías mencionadas en el párrafo anterior, se creó el proyecto Sparky Blast [6], este consiste en una nueva implementación del algoritmo de la herramienta básica de búsqueda de alineación local (BLAST) que utiliza la base de datos Cassandra para almacenar los diferentes conjuntos de datos de referencia y el marco de procesamiento de Spark para calcular los índices. y procesar consultas. Sparky Blast es capaz de utilizar los recursos distribuidos de un Big Data Cluster para procesar consultas en paralelo, mejorando tanto el tiempo de respuesta como el rendimiento del sistema. Al mismo tiempo, el uso de una arquitectura distribuida como Hadoop dota a la herramienta de una escalabilidad ilimitada, hecho que implica mejoras tanto a nivel de infraestructura como de rendimiento.

La idea detrás de este proyecto recae en la necesidad de mejorar la falta de integración con Cassandra que tiene la solución actual (Sparky Blast [6]), esto es debido a que el conector de Spark con Cassandra para Python, está menos optimizado que dicho conector para Scala (lenguaje nativo de Spark)[7]. Por ello, para poder maximizar las prestaciones de Cassandra es necesario migrar la implementación actual de Sparky Blast a Cassandra. Este cambio, mejorará no solo la falta de integración con la base de datos sino que además se deberían obtener mejores prestaciones y mejoras de rendimiento.

Dicha migración usará Scala [7], lenguaje de programación funcional el cual nos garantiza todo lo que buscamos, desde mayor integración con Cassandra hasta mejores prestaciones, hecho que nos lleva a poder afirmar que se obtendrá una mejora a nivel de rendimiento.

Todo seguido, se explicarán detalladamente los objetivos de este proyecto.

1.2. Objetivos del proyecto

Los objetivos de este proyecto, tal como se menciona en la introducción, consisten en el análisis y migración de una solución previamente realizada en Python y Cassandra, que actúa como base de datos para las cadenas de ADN, para obtener mayor integración, rendimiento y prestaciones.

El objetivo principal será, como se ha comentado en el párrafo superior, la mejora de rendimiento y para ello se realizará una migración a Scala (lenguaje de programación funcional, pionero en el área de Big Data) y se comprobará este hecho llevando a cabo un análisis de la “performance” de ambas soluciones.

El rendimiento se verá afectado debido a que la migración supondría poder aprovechar mejor la conexión con Cassandra, ya que nos permite un uso más preciso del conector [8] (estándar de acceso a las bases de datos). En otras palabras, accesos más rápidos debido a la reducción en el tiempo de conexión con la base de datos y sobretodo, en las consultas realizadas.

Una vez realizados el análisis y la migración, se proseguirá con una explicación de los resultados en base a los tiempos obtenidos en ambos casos.

En conclusión, la migración a Scala, debería aportar una mejora sustancial de rendimiento, prestaciones y productividad debido a que el conector tiene más afinidad con el nuevo lenguaje en el que estará implementado.

1.3. Planificación temporal de las tareas

En este apartado se realizará una breve explicación de cada uno de les tareas y por consiguiente, se llevará a cabo una estimación de la duración de cada una de ellas, en términos globales, analizaremos la duración estimada del proyecto.

1.3.1 Tareas a realizar

En este subapartado, se enumerarán y explicaran las tareas que componen el proyecto a realizar. Las tareas que se llevarán a cabo son las siguientes:

1. **Análisis de la problemática.** Esta tarea se centra en la explicación e introducción del motivo por el cual se realiza este trabajo, así como todas las tecnologias empleadas para su realización. También incluye la planificación y el presupuesto del proyecto.
2. **Estado del arte.** El objetivo de esta tarea sería realizar una introducción a todos aquellos aspectos que, desde el punto de vista de la tecnología y del trabajo previo, son necesarios para poder entender el proyecto.
3. **Desarrollo.** En esta tarea es dónde se realizará la migración de lenguaje de Python a Scala, teniendo en cuenta que previamente se haya puesto en funcionamiento Sparky Blast, el proyecto Python del cual se migra, y analizado el funcionamiento para realizar el proceso de migración correctamente y que este no cambie el funcionamiento del código, ya que ese no es el objetivo.
4. **Validación.** Esta tarea verifica que el funcionamiento entre ambas implementaciones sea el mismo, es decir, se valida que con una misma consulta ambas lleguen al mismo resultado y elucita las diferencias en base al tiempo de ejecución de cada una de las soluciones.

1.3.2 Duración estimada del proyecto

En este subapartado se detallará la duración de cada una de las tareas y se mostrará en la siguiente figura (Tabla 1).

Etapa	Feb.		Marzo		Abril		Mayo		Junio		Julio		Agosto	
	M1	M2	M1	M2	M1	M2	M1	M2	M1	M2	M1	M2	M1	M2
Análisis														
Estado del Arte														
Desarrollo														
Validación														
Análisis de prestaciones														

Tabla 1. Planificación temporal de las tareas a realizar

- Cada tarea incluye también el tiempo que se ha tardado en documentarla.

1.4. Presupuesto del proyecto

En este apartado, se van a tener en cuenta todos los elementos implicados en el proyecto, las horas necesarias y las dedicadas, y los recursos, ya sea hardware, cloud o software.

El principal objetivo del proyecto es la obtención de una solución con mayor rendimiento, capaz de devolver los resultados con mayor velocidad y con mejor rendimiento en cuanto, no solo a tiempo, sino a la relación tiempo-recursos usados. En términos del desarrollo planteado, las vacantes para crear la solución serían las siguientes:

- **1 Director de proyecto o Project Manager:** Una figura imprescindible para las primeras etapas del desarrollo, mantener el contacto con el cliente, formular dudas y llevar un seguimiento de la evolución del trabajo.

- **1 Software Developer:** Figura introducida como principal encargado del todo lo relacionado con el código y de realizar la migración al completo, junto al QA Tester, realizara la mayor parte del trabajo.

- **1 QA Tester:** Figura introducida para comprobar que ambas soluciones funcionen igual i será el encargado de comprobar el rendimiento del proyecto y de comentar sus fallos con el Software Developer y también realizará el análisis de todos los cambios para obtener un resultado de la problemática.

Cabe remarcar que la contratación o formación de un trabajador en más de una de las especialidades anteriores abarataría el costo total del proyecto. Sin embargo, en cuanto a este análisis económico de la solución, cada rol se evaluará de forma individual y toman como referencia el sueldo medio del cargo que ocupa.

Todos estos elementos, servirán para la computación del presupuesto de este proyecto que se mostrará a continuación (Tabla 2).

Despesa	Valor económico/mes*
Project Manager	4.187€
Software Developer	3.654€
QA Tester	3.500€
Total creación del projecte (9 meses)	102.069 € - 141.016 € **
Coste de mantenimiento anual[1]	75.532 €
(*) Sueldos extraídos del portal web Glassdoor por la región de España. (**) Coste variable dependiendo de la contratación separada o conjunta de un Software Developer & QA Tester y un poco más de dinero para posibles gastos inesperados.	

Tabla 2. Presupuesto del proyecto

Como se puede ver en la Tabla 2, el presupuesto se puede llegar a reducir si tenemos en cuenta que el rol de Software Developer, QA Tester se pueden juntar en una única persona porque el QA Tester, no estará en activo hasta mediados o finales del desarrollo del proyecto porque no tendría nada que probar ni ningún funcionamiento que evaluar. Por lo tanto, si hacemos la unión de ambos roles en la misma persona, entonces el presupuesto quedaría como en la Tabla 3.

[1] **Mantenimiento anual:** El coste de mantenimiento anual tan sólo involucra el trabajo del Software Developer y del QA Tester para seguir apoyando la solución una vez creada.

Despesa	Valor económico/mes*
Project Manager	4.187 - 4.500 €
Software Developer + QA Tester	3.933 - 4.122 €
Total creación del projecte (9 meses)	73.080 € - 77.598 € **
Coste de mantenimiento anual[2]	75.532 €
(*) Sueldos extraídos del portal web Glassdoor por la región de España. (**) Coste variable dependiendo de la contratación separada o conjunta de un Software Developer & QA Tester y un poco más de dinero para posibles gastos inesperados.	

Tabla 3. Presupuesto reducido del proyecto

1.5. Estructura de la memoria

Se detalla la estructura que tendrá el documento y un breve resumen de cada uno de los capítulos que compondran este documento.

1.5.1. Capítulo 1. Introducción al proyecto.

Este capítulo ofrece la explicación a todo lo que envuelve el proyecto, como de este mismo. Además se habla también de los objetivos y de la motivación que me ha llevado a escoger este tema. Para concluir este capítulo tenemos la planificación temporal junto a las tareas a realizar.

1.5.2. Capítulo 2. Estado del arte del proyecto.

En este capítulo se explican diferentes artículos que se han realizado sobre el mismo tema y sus aproximaciones. Se trata de establecer que se ha hecho recientemente sobre el tema seleccionado.

1.5.3. Capítulo 3. Desarrollo.

Este capítulo explica todo el proceso de montaje de la arquitectura usada y la migración a Scala para obtener las mejoras en rendimiento, prestaciones y productividad explicadas en el capítulo 1.

[2] **Mantenimiento anual:** El coste de mantenimiento anual tan sólo involucra el trabajo del Software Developer y del QA Tester para seguir apoyando la solución una vez creada.

1.5.4. Capítulo 4. Validación.

Este capítulo trata sobre la comprobación de funcionamiento entre ambas implementaciones, es decir, se valida que con una misma consulta ambas lleguen al mismo resultado. En el capítulo siguiente, se analizarán con detalle los temas de rendimiento y mejoras de prestaciones y productividad.

1.5.5. Capítulo 5. Conclusiones y trabajo futuro.

Este capítulo explica el conocimiento que nos aporta la solución creada y las tareas que quedan pendientes para realizar, las cuales formarán parte de la parte llamada trabajo futuro.

Capítulo 2. Estado del arte del proyecto.

El objetivo de este capítulo consiste en la realización de una introducción a todos aquellos aspectos necesarios, que desde el punto de vista de la tecnología y del trabajo previo, nos permitan facilitar el entendimiento del proyecto. Por consiguiente, se realizará la explicación del algoritmo Blast y de soluciones previas y por otro lado, se introducirán varias tecnologías del ámbito del Big Data, como son Hadoop, MapReduce, Spark y las bases de datos NoSql, entre las cuales nos focalizaremos en Cassandra.

2.1. Blast

Blast [2] es una herramienta básica de alineación local y se ha convertido en el estándar por defecto en las herramientas de búsqueda y de alineación genética. Este algoritmo todavía se está desarrollando activamente y es uno de los artículos más citados jamás escritos en el campo de la bioinformática. Muchos investigadores usan BLAST como una evaluación inicial de sus datos de secuencia del laboratorio y para obtener una idea de lo que están trabajando. BLAST está lejos de ser básico como su nombre indica, realmente, es un algoritmo muy avanzado que se ha vuelto muy popular debido a su disponibilidad, velocidad y precisión. En resumen, una búsqueda usando BLAST permite identificar secuencias homólogas buscando una o más bases de datos sobre la secuencia de consulta de interés.
Blast es un programa *open-source* (de código abierto), y cualquiera puede cambiar el código del programa. Es necesario destacar que este algoritmo, ha dado lugar a una serie de derivados, como WU-BLAST [18] que, probablemente, es el más comunmente utilizado.
BLAST es altamente escalable y está preparado para su integración en diferentes plataformas informáticas diferentes lo que hace posible su uso tanto en pequeñas computadoras de escritorio como en grandes grupos de computadoras. Una vez introducido el algortimo, se procederá a explicar en que consiste y en las diferentes variantes que han ido apareciendo de este.

2.1.1 En què consiste Blast ?

Blast es una herramienta de alineamiento local que utilitza un algoritmo cómo base, también se podría decir que es un método huerístico, lo que significa que es una algortimo de programación dinámico que es más rápido y eficiente pero relativamente menos sensitivo (menos preocupado sobre la velocidad de respuesta, es decir, menos focalizado en responder rápido, sino que se centra en responder correctamente). Esta herramienta, se usa para alinear múltiples secuencias y encontrar similitudes o diferencias entre varios tipos

de especies, genéticamente hablando. Seguidamente, se explicaran brevemente, los diferentes tipos de BLAST y su funcionamiento.

2.1.2 Tipos de Blast

Blast tiene diferentes tipos de programas para la alineación de secuencias de nucleótidos, proteínas, etc. Se explicarán, brevemente, las variantes básicas [16] de este algoritmo:

- **Blastn.** El blastn, es una variante de blast donde la secuencia a consultar es un nucleótido y la secuencia objetivo con la que se contrastará, también es un nucleótido, en resumen, es nucleótido contra nucleótido.

- **Blastp.** En el blastp, es lo mismo que el blastn lo que en lugar de tratarse de nucleótidos, consiste en proteínas. En otras palabras, tanto la secuencia a consultar como la secuencia objetivo con la que se contrastará, son proteínas.

- **Blastx**. En este tipo de blast la sequencia de consulta es una sequencia de nucleótidos y se ataca a una secuencia o base de datos de proteínas. Primero que todo, se convierte la secuencia de nucleótidos en la secuencia de proteínas que le corresponde en tres partes de solo lectura, las cuales se contrastarán con una búsqueda contra la proteina almacenada en la base de datos o con la secuencia, en caso de que no se haya usado sistema de almacenamiento.

También se hará una breve explicación de una variante más específica creada por la universidad de Washington, esta variante se llama **Wu Blast** [18]. Esta variante de Blast, más que una variante, es un poderoso paquete de software para la identificación de genes y proteínas mediante búsquedas de similitud sensibles, selectivas y rápidas de bases de datos de secuencias de proteínas y nucleótidos. Se ha de destacar que lo más importante de este es su licencia, ya que es un software propietario y es gratuito solo para uso académico.

2.1.3 Funcionamiento del algoritmo de Blast

El funcionamiento del algoritmo de esta herramienta, consiste en la búsqueda indexada de todas las cadenas de caracteres de cierta longitud dentro de la “consulta” dentro de la consulta inicial. Es importante destacar que se puede configurar la longitud de la cadena a

indexar, denominada “tamaño de palabra”, para poder llevar a cabo una configuración correcta hay que saber que el rango permitido varía según el programa BLAST utilizado, aunque, los valores típicos son:

- Búsqueda de secuencia de proteína a proteína → 3
- Búsqueda de nucleótido a nucleótido → 11

Una vez toda la configuración ha sido realizada, BLAST escanea la base de datos en busca de coincidencia entre las “palabras” indexadas en la “consulta” y las cadenas que se encuentran dentro de las secuencias de la base de datos. Se tiene que diferenciar las búsquedas nucleótido a nucleótido de las de proteína a proteína porque en el primer caso, se buscarán coincidencias exactas y en el segundo, se determinará la puntuación de coincidencia que debe superar el umbral especificado. Cuando se encuentra una coincidencia de palabra, dos palabras cercanas en el caso de búsquedas de proteínas, BLAST intenta extenderse tanto hacia adelante como hacia atrás desde la coincidencia para producir una alineación. BLAST continuará con esta extensión mientras la puntuación del alineamiento continúe aumentando o hasta que disminuya en una cantidad crítica debido a puntajes negativos dados por discrepancias. Esta cantidad crítica se conoce como el "retorno" [19].

A continuación, se explicaran los tres pasos básicos que sigue Blast y se ilustra con una imagen (Fig.1) el funcionamiento general de Blast para un mayor entendimiento.

Blast lleva a cabo tres pasos básicos:

1. Blast aplica la búsqueda de palabras en la cuál elimina las regiones de alta complejidad y entonces busca por las extensiones cortas de medida fija de la secuencia consultada.
2. Blast identifica el número exacto de palabras emparejables. Esas palabras, las cuales han devuelto la misma puntuación o superior al umbral, se escogen para el alineamiento. Estos alineamientos obtenidos se llaman "hits".
3. Blast extiende el alineamiento en ambas direcciones, como "ungapped alignment" (alineamiento sin huecos), y esa extensión se detendrá cuando llega a la máxima puntuación y entonces insertará un "gap" (hueco).

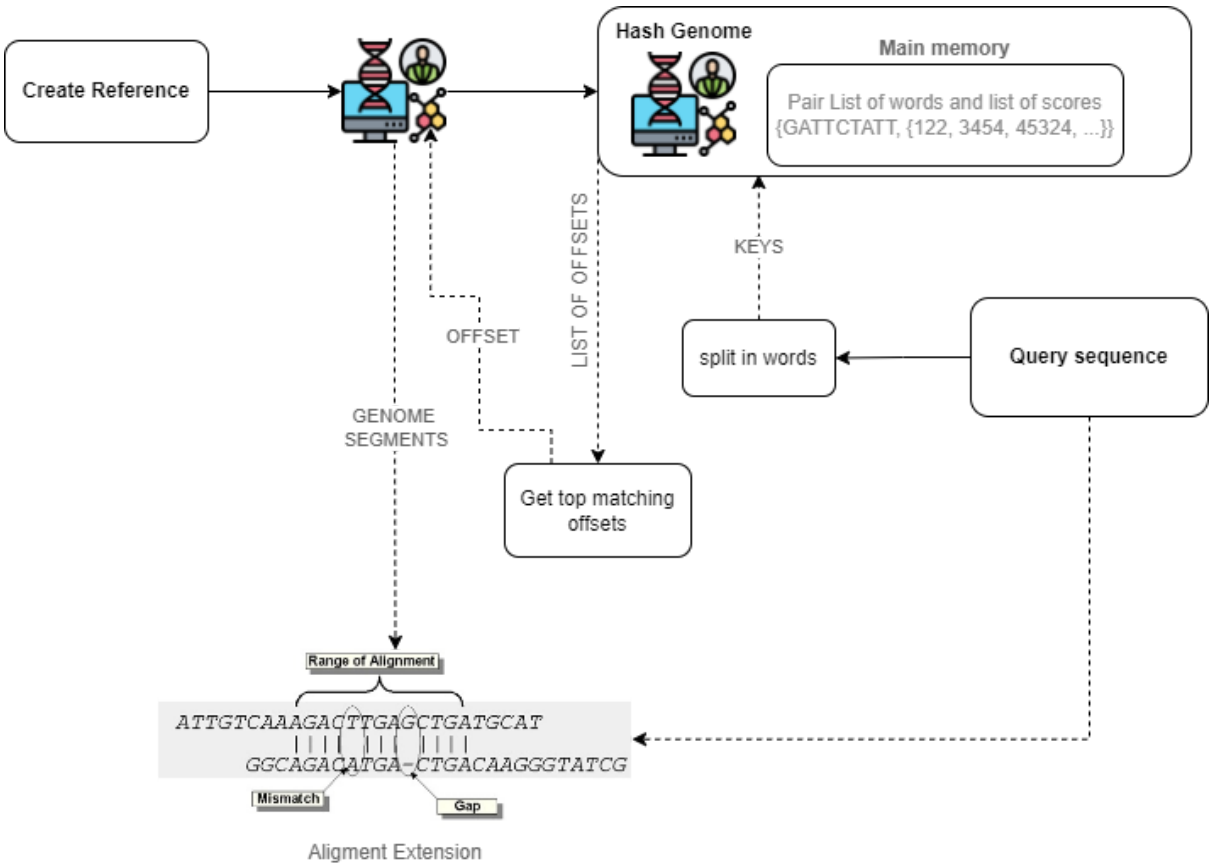

Fig. 1 *Funcionamiento del algoritmo BLAST*

A continuación se explicará la creación de un nuevo proyecto usando el algoritmo BLAST junto a tecnologías Big Data, como Spark o Hadoop, que serán explicados en la sección 2.3.

2.2. SparkyBlast

En esta sección, se explicará la herramienta SparkyBlast, junto a los objetivos que se pretenden conseguir con su uso. Además, también se tratarán, su funcionamiento y la arquitectura de esta solución.

La herramienta SparkyBlast es una implementación de Blast basada exclusivamente en un paradigma Spark, tal como se puede ver en la Figura 2. Soluciones previas a esta, tales como Cloud Blast [14] y SparkBlast [13], estos usan tecnología Big Data, exclusivamente, para dividir la consulta para que esta sea procesada en múltiples tarea distribuidas entre los nodos del cluster. En cada nodo, se incia una instancia del proceso original para procesar su parte del trabajo. De este modo, el rendimiento de la aplicación es mejorado cuando se usan múltiples procesadores.

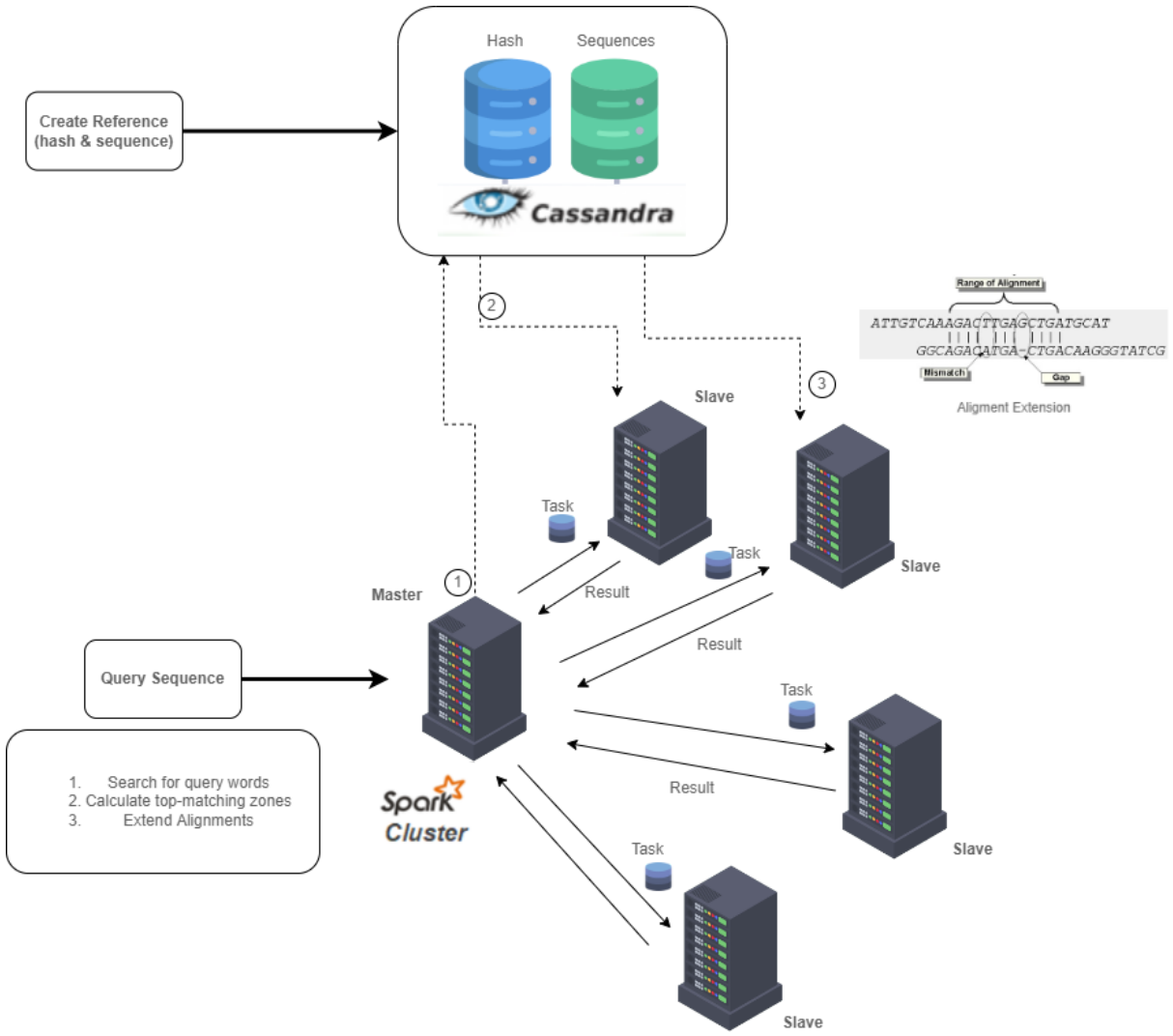

Fig. 2 *Arquitectura de SparkyBlast*

En el siguiente subapartado, se explicará más en detalle el diseño de SparkyBlast tal y como se muestra en la Figura 2.

2.2.1. Funcionamiento

Tal como se puede ver la Figura 2, nos presenta la arquitectura de la solución SparkyBlast. Esta tiene como base de datos Cassandra [5], una base de datos distribuida NoSQL y un cluster de Spark [3]. Ambos residen en los mismos nodos del cluster. Cassandra se encarga de guardar las secuencias de referencia y el índice requerido para realizar la consulta. El cluster de Spark, es usado para el cálculo del índice invertido, ya que nos permite la paralelización de este y también, se utilizará para procesar las consultas, por el mismo motivo.

El funcionamiento básico de la herramienta, consiste en dos pasos, el primero se basa en realizar el procesamiento de las consultas y durante este procesamiento, dividir la consulta en múltiples palabras, de medida K) y luego, acceder al índice de Cassandra para obtener los “offsets” (desplazamientos dentro de la referencia de las coincidencias encontradas) en la referencia cuando estas palabras aparezcan. El segundo paso, consiste en el cálculo de las “top-matching zones” (zonas superiores de coincidencia) en las cuales se realiza la extensión del alineamiento.

La herramienta SparkyBlast está compuesta por dos módulos totalmente diferentes[3], el *CreateReference*, responsable de la creación y construcción de las dos tablas de referencia en Cassandra para las secuencia y de la creación del índice invertido. Por otro lado, el segundo, *DoQuery,* se usa para procesar la secuenca de consulta contra la referencia y devolver las mejores coincidencias de alineamientos posible. En las dos subsecciones siguientes, se explicarán estos dos módulos de forma breve.

2.2.1.1 Modulo 1: *CreateReference*

Crear la base de datos de referencia y su índice invertido asociado es una operación altamente computacional, especialmente para referencias de genoma completo. La razón de esto es el enorme tamaño de un genoma completo, que puede alcanzar decenas de gigabytes una vez desplegado en memoria. El cálculo y almacenamiento del índice invertido es incluso mucho más costoso, ya que su construcción requiere multiplicar el tamaño del genoma de referencia por K (el tamaño de la palabra), lo que da como resultado una gran cantidad de escrituras en la base de datos. Esta es la razón por la que es muy importante que, además de implementar procesamiento de consultas en Spark, también se realiza con la construcción del referencia del genoma (índice invertido).

Debido a varios inconvenientes con Spark, la creación de la tabla de secuencias finalmente se realizó utilizando un programa concurrente de Python. La razón para no usar Spark en este método es que Spark no garantiza la creación de bloques con un tamaño homogéneo a menos que se procesen secuencialmente. En esta situación, es más eficiente crear tablas de secuencia con un programa de Python que pueda realizar accesos aleatorios dentro del archivo de secuencia para leer y procesar los bloques simultáneamente.

[3] Repositori Github con los modulos y ficheros necesarios para el uso de la herramienta.

2.2.1.2 Modulo 2: *DoQuery*

En este módulo se implementan dos modos de procesamiento para adminstriar consultas, el de consulta individual o el de consulta múltiple. El algoritmo de ambos métodos es bastante parecido, pero el segundo método comparte los resultados de Cassandra entre las diversas consultas para reducir el número de accesos a la base de datos.
El algoritmo de multiple procesamiento de ficheros, se divide en 6 partes, y seguidamente, se explicarán cada una de ellas de forma breve y concisa:

1. Leer el fichero de queries y almacenar los datos en un dataframe con las columnas de queryId, sequence (secuencia) y (size) tamaño de la secuencia. Seguidamente, se calculan los k-words para todas las secuencias de consulta.

2. Acceder al índice invertido de Cassandra (tabla hash) y obtener los desplazamientos, “offsets”, junto a la referencia donde aparecen todos k-words de la consulta. Esta operación se implementa con el objetivo de minimizar el número de accesos a Cassandra.

3. Integrar la información con las queries y lo guardamos todo en un nuevo dataframe.

4. Calcular las mejores coincidencias entre todos los alineamientos, usando siempre el dataframe creado en el paso anterior, y guardamos los datos en un dataframe que se usará posteriormente.

5. Se realizan las extensiones de alineamientos a partir del dataframe creado en el paso anterior.

6. Una vez obtenidos los alineamientos que se consideran buenos (que superan un determinado umbral de calidad), los guardamos en un fichero de texto.

En resumen, los autores presentan Sparky-Blast, la reimplementación del algoritmo de la Herramienta básica de búsqueda de alineación local (BLAST) aplicando paradigmas y infraestructuras de Big-Data. Esta nueva versión es capaz de utilizar los recursos distribuidos de un Big-Data Spark Cluster para procesar consultas en paralelo, mejorando tanto el tiempo de respuesta como el rendimiento del sistema. Al mismo tiempo, el uso de una arquitectura distribuida como Hadoop otorga a la herramienta una escalabilidad

ilimitada desde el punto de vista tanto de la infraestructura de hardware como del rendimiento de la aplicación.

En el siguiente apartado, se explicará el paradigma Big Data y las tecnologías usadas en este.

2.3. Tecnologías Big Data

En esta subsección, se centrará en las diferentes tecnologías Big Data que se usan en este proyecto, para facilitar el entendimiento de esta solución y de sus antecedentes. Las tecnologías Big Data, como su nombre indica, son todas aquellas herramientas, programas, bases de datos, etc. que nos permiten manejar grandes cantidades de datos de forma distribuida/paralela. Se ha de destacar que, el uso de estás tecnologías, nos facilita la manipulación de esas grandes cantidades de datos y nos permite obtener los resultados mucho más rápido que si no se usarán, eso sí, nos genera la necesidad de tener un entorno preparado para tener tal cantidad de datos y poder tratarlos, ya que en un ordenador personal, no debería ser posible hacerlo.

En las siguientes subsecciones, se explicarán, desde las plataformas de software que nos permiten guardar ficheros de gran tamaño, como el paradigma más importante que usan. Además, se hará énfasis en el framework de procesamiento de grandes cantidades de datos Spark y en las bases de datos NoSQL, entre las cuáles, la explicación se centrará en Cassandra.

2.3.1 Hadoop

Hadoop [21] es una herramienta distribuida de procesamiento de grandes cantidades de datos y de código abierto, que se encaraga de administrar el procesamiento y guardado de estos datos para aplicaciones Big Data en servidores que contengan clusters escalables. Es la base de todo ecosistema Big Data, sobre todo para los que se utilizan principalmente para respaldar iniciativas de análisis avanzado, incluido el análisis predictivo, la extracción de datos y el aprendizaje automático. Los sistemas Hadoop permiten el uso de datos estructurados y no estructurados, hecho que nos da más flexibilidad para todos las fases de la manipulación de datos.

La capacidad de Hadoop para procesar y almacenar diferentes tipos de datos lo convierte en una opción particularmente adecuada para entornos de big data. Este, se ejecuta en servidores que se puede escalar para admitir miles de nodos de hardware y aporta el

sistema de archivos distribuidos de Hadoop (HDFS) [23] está diseñado para proporcionar acceso rápido a los datos de los nodos en un clúster, junto con capacidades tolerantes a fallos para que las aplicaciones puedan continuar ejecutándose si fallan los nodos individuales.
A continuación se explicará el paradigma más usado para crear soluciones que usen Hadoop, el Map Reduce.

2.3.2 Map Reduce

El paradigma map reduce [22] permite una escalabilidad masiva en los servidores en un cluster Hadoop. Como se comentaba anteriormente, se considera a map reduce como una parte indispensable para Hadoop.

El término “Map Reduce” hace referencia a dos tareas distintas que los programas Hadoop llevan a cabo. El primero, es el map, que consiste en la el procesamiento de un conjunto de datos y su conversión a otro grupo de datos, donde los elementos individuales se rompen en tuplas (parejas clave/valor). El segundo, es el reduce, y siempre se lleva a cabo después del map. Este coge los valores que devuelve el map como datos de entrada y combina esas tuplas de datos con grupos de tuplas más pequeños.

El paradigma Map Reduce nos ofrece varios beneficios para obtener mayor valor de tus datos:

- **Escalabilidad.** Procesamiento de muchas gigabytes de datos fácilmente a través de HDFS.
- **Fleixibilidad.** Hadoop permite un acceso más fácil a múltiples fuentes de datos y múltiples tipos de datos.
- **Velocidad.** Con un procesamiento paralelo y un movimiento de datos mínimo, Hadoop ofrece un procesamiento rápido de cantidades masivas de datos.
- **Simplicidad.** Los desarrolladores pueden escribir código en varios lenguajes, incluidos Java, C++ y Python.

2.3.3 Spark

Apache Spark es un motor o marco de procesamiento de datos que puede realizar rápidamente tareas de procesamiento en conjuntos de datos muy grandes y también puede distribuir tareas de procesamiento de datos en varias computadoras, ya sea solo o en conjunto con otras herramientas informáticas distribuidas. Estas dos cualidades son clave para los mundos de los grandes datos y el aprendizaje automático, que requieren la

organización de una potencia informática masiva para procesar grandes almacenes de datos. Spark también quita algunas de las cargas de programación de estas tareas de los hombros de los desarrolladores con una API fácil de usar que abstrae gran parte del trabajo duro de la computación distribuida y el procesamiento de big data.
A continuación, se argumentará porque se ha usado Spark que Hadoop Map Reduce y porque es mejor.

2.3.3.1 Spark en lugar de Hadoop Map Reduce

Actualmente, se encontrará Spark en la mayoría de distribuciones Hadoop, porque este nos aporta dos grandes ventajas que nos permiten superar al paradigma Map Reduce.

La primera ventaja es la **velocidad**. El motor de datos en memoria de Spark significa que puede realizar tareas hasta cien veces más rápido que MapReduce en ciertas situaciones, especialmente cuando se compara con trabajos de varias etapas que requieren la escritura de estado en el disco entre etapas.

La segunda ventaja es la API de la que nos provee Spark, que facilita mucho el desarrollo de aplicaciones Big Data.

En el siguiente apartado, se hablará de las bases de datos NoSQL y Cassandra, ya que es la que se usa en este proyecto.

2.3.4 Bases de datos NoSql - Cassandra

Las bases de datos NoSQL son bases de datos no relacionales que almacenan datos en documentos en lugar de tablas relacionales. En consecuencia, los clasificamos como "no solo SQL" y los subdividimos en una variedad de modelos de datos flexibles. Los tipos de bases de datos NoSQL incluyen bases de datos orientadas a documentos, de clave-valor, bases de datos orientadas a columnas, como Cassandra, y bases de datos orientadas a grafos. A continuación, se expicarán brevemente cada uno de estos tipos:

- **Bases de datos Clave-Valor.** Tienen el modelo de datos más sencillo de todos, una clave indexada asociada a un valor, que desde el punto de vista de la base de datos es información opaca que simplemente almacena y recupera asociada a la clave.
- **Orientadas a documentos.** Utilizan el modelo de documento, mayoritariamente en formato JSON, para almacenar y consultar información, como por ejemplo,

MongoDB. Estas permiten gestionar información con complejas estructuras jerárquicas, y ofrecen índices secundarios y completos lenguajes de consulta y agregación de datos.

- **Orientadas a grafos.** El modelo de datos se centra en entidades y las relaciones entre éstas. Tanto las entidades (nodos del grafo) como las relaciones (aristas) pueden además tener atributos. Una entidad puede tener numerosas relaciones con cualquier otra entidad.
- **Orientadas a columnas.** Este tipo de bases de datos son similares a una tabla en bases de datos relacionales, pero un registro puede contener cualquier número de columnas (o familias de columnas). Son ideales para consultar y agregar grandes cantidades de datos cuando los datos pueden determinarse con antelación y no cambian con frecuencia. En este grupo encontramos ejemplos como Cassandra o HBase.

Si nos centramos en Cassandra, se ha saber que es una base de datos orientada a columnas, y eso significa que todo lo explicado sobre las bases de datos NoSQL orientadas a columnas se aplica a esta. Se ha elegido esta base de datos porque nos proporciona tolerancia a particiones y una alta disponibilidad. Además, que tiene como características principales el ser distribuida, que escala linearmente y de forma horizontal, y que implementa la estructura peer-to-peer [24], de esta manera cualquiera de los nodos puede tomar el rol de coordinador de una query, este rol no está fijo en el nodo master, porque no tiene la estructura, maestro-esclavo.

Capítulo 3. Desarrollo.

El objetivo de este capítulo consiste en la declaración de todos aquellos aspectos y cambios necesarios realizados en esta migración. Por consiguiente, se explicará desde los intentos de creación de una arquitectura con Docker [9], hasta la solución dada, estas explicaciones, iran acompañadas de una justificación de porque fallaron o de porque se llevaron a cabo. Seguidamente, se incluirá la explicación de la migración de los cambios realizados en el código para mejorar el rendimiento del código o porque era necesario al cambiarlo de lenguaje de programación. Para terminar, se dará a conocer, la dificultat de la tarea realizada.

3.1. Arquitectura

En este apartado, nos focalizaremos en los inicios de la migración donde primero que todo, necesitábamos un entorno para poder hacer funcionar el código recibido, ya que para usarlo, necesitabamos de un entorno con Python, Cassandra y Pyspark (Spark para Python). Antes de seguir con este nuevo entorno, es importante hablar de las pruebas que se realizaron usando Docker, que es una plataforma de software de código abierto para crear, implementar y administrar contenedores de aplicaciones virtualizados en un sistema operativo (SO) común, con un ecosistema de herramientas aliadas.

Se eligió Docker, porque se veía como la mejor opción para trabajar con un sistema distribuido de forma senzilla, para el uso de esta herramienta, es muy importante el sistemma operativo, el cual, en este caso, era Windows 10 [25]. Este sistema operativo, nos ofrecía el uso de la herramiento Docker Desktop, que es una aplicación fácil de instalar para su entorno Mac o Windows, en nuestro caso sería Windows, que le permite crear y compartir microservicios y aplicaciones en contenedores. Además, proporciona una interfaz simple que le permite administrar sus contenedores, aplicaciones e imágenes directamente desde su máquina sin tener que usar la CLI para realizar acciones principales.

Para usar esta herramienta, se requería descargar una máquina virtual de Ubuntu (Linux) y un Windows 10 Pro, porque necesitaba unos permisos exclusivos de esa versión de Windows. Una vez instaladas todas los complementos necesarios, debemos buscar una imagen que contenga lo que nosotros necesitamos, que en este caso era Cassandra con Pyspark, ya que el python vendría implícito al escoger Pyspark, porque tal como se ha dicho en el primer párrafo, esta es la versión de Spark para Python, y por lo tanto, si se decide

instalar pyspark, se debe de verificar si el hecho de instalarlo, incluye la version de Python correcta o no, en nuestro caso, necesitamos la 2.7., y por eso elegimos la 2.7.18.

Una vez realizada la instalación de todos los complementos y con la imagen ya creada, nos dispondremos a crear el contenedor de docker para poderlo ejecutar como un servicio en la nube. Este caso, no prospero i se decidió ejecutar en local el código premigración, para ver su funcionamiento y sus salidas por pantalla, para validar que en todo momento estaba realizando los mismos passos y que se obtenía el mismo resultado.

Por otro lado, se probó de realizar lo mismo, pero para migrarlo, es decir, se creo un entorno con Spark, Scala y Cassandra, y los passos seguidos para hacerlo fueron los siguientes, para ver los comandos con toda la guía de instalación detallada, visualizar mi repositorio de Github donde esta contenida esta información [28]:

1. Antes de empezar con el proceso de instalación, debemos descargar estas 3 herramientas:

    a. Docker [9]
    b. Docker Desktop [26]
    c. Git Bash [29]

2. Instalar Cassandra y Spark en Docker.

    a. Descarga (“Pull”) la imagen ya creada de spark y cassandra.
    b. Una vez que hemos sacado la imagen de Spark-cassandra, antes de ejecutarla. Debemos crear una red docker llamada sparkcassandra_default. Idealmente, lo ejecutaría usando docker compose (vea el archivo yml provisto). Esto garantizará que inicie el nodo principal y el trabajador en la misma red para que puedan verse entre sí.
    c. Comprobar si todo va bien usando la consola/terminal (cmd (git bash))

3. Escalado

    a. Se usará docker-compose para añadir más nodos y eso nos mejorará el rendimiento ya que podremos distribuir la carga de trabajo.
    b. Comprobar en el Docker Desktop que tenemos un cluster activo, en modo running.

4. Comprobar el despliegue

    a. Para comprobar el despliegue, se realizaran unas queries de testing en la db de cassandra usando CQLSH [30], es una interfaz de línea de comandos para interactuar con Cassandra usando CQL.

Esta aproximación, era prometedora, pero empezó a tener errores no identificados por los cuales, el clúster se cerraba de forma inesperada donde estabamos trabajando, y como eso ralentizaría aún más nuestro trabajo, se descarto esta solución y se optó por realizar una implementación en local, tanto para la versión en Scala (postmigración) como para la versión en Python (premigración).

Debido a esos inconvenientes, al final, se optó por la opción de crear una instalación local, tanto para SparkyBlast en Python como para Scala. La instalación local, tiene un nexo entre ambas versiones, la base de datos, ya que ambas van a usar Cassandra, en ese caso, solo debemos instalar la base de datos localmente en nuestro ordenador.

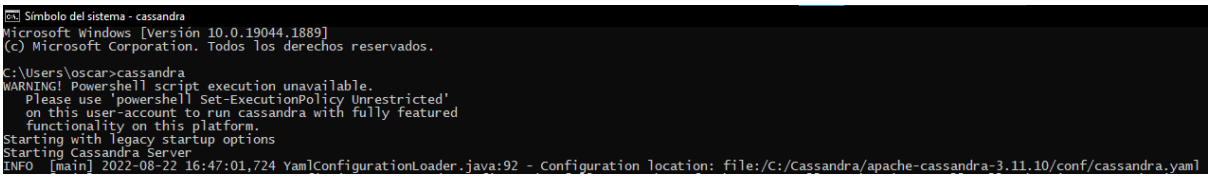

Fig. 3 *Cmd Init Cassandra*

Una vez instalada localmente, debemos abrir una terminal, y ejecutar el comando “cassandra” el cual nos iniciará una base de datos Cassandra en local, usando el fichero de configuración de cassandra, tal como se puede ver en la Figura 3. Es importante mencionar que para la conexión con Cassandra, es indispensable obtener el conector correspondiente, en este caso, se ha usado *“Datastax Connector Cassandra”* [34].

Una vez explicada la parte común, se pasará a explicar como se ha ejecutado en local el código Python (premigración). Para ello, se necesita el gestor de entornos virtuales, Anaconda [31] o Miniconda [32], en este caso se ha optado por el uso de Anaconda, ya que debido al sistema operativo usado, nos era mucho más fácil y además porque se estaba más familiarizado con el uso de este software.

El uso de Anaconda, nos ha permitido crear un entorno virtual de python en el cual se han instalado las dependencias principales, tales como son, cassandra driver, dependencia que nos permitirá la connexión con la base de datos y Pyspark, interfície de apache spark para python que nos permite el uso de spark en el código (paralelización, balanceo de carga, división de tareas de forma simple, manipulación de ficheros de gran tamaño…).

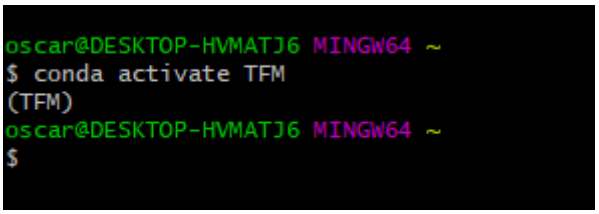

Fig. 4 Activate environment using Anaconda

Para usarlo de forma práctica, debemos activar el entorno tal como se ve en la Figura 4 y luego se hará uso de los comandos spark que nos permitan ejecutar nuestro código con las especificaciones que queramos.

Por otro lado, en la versión de scala, no es necesario crear un entorno virtual, ya que las dependencias se encapsularan mediante el uso del gestor de dependencias Maven [33].

Para ambas versiones, se usaran los mismos comandos para ejecutar el código, pero se cambiará el parámetro de fichero a ejecutar, y son los siguientes:

Creación de datos y del índice invertido.

```
spark-submit.cmd --name <name> --master <spark_master> --num-executors
<spark_num_executors> --executor-cores <spark_executors_cores>
--driver-memory <spark_driver_memory> --executor-memory
<spark_executor_memory>  --packages <conector_datastax_CassandraDB>
<fichero_a_ejecutar> <fichero/s_datos_de_entrada>  [Key_size=11]
[ReferenceName=blast] [Method=1] [BlockSize=1000]
[ContentBlockSize=10000] [HashGroupSize=12000000] [StatisticsFileName]
[Local=false]
```

Este comando ejecuta mediante spark, la funcionalidad de creación de un índice invertido y por lo tanto, el guardado de una estructura de datos con su correspondiente información. Los parámetros que se le pasan al comando, son modificables y cada uno de ellos, indica el elemento de la ejecución que se modifica, directamente relacionado con el nombre que se ve en el fragmento de código superior. Estos pueden especificar algunos atributos de hilo

para definir los recursos asignados al trabajo (número de ejecutores, número de núcleos por ejecutor y la memoria del controlador y del ejecutor). El programa recibe el archivo fasta de secuencia (este archivo debe almacenarse en HDFS), el tamaño de la clave (K), el nombre del espacio de claves de referencia en cassandra, el método utilizado para construir el índice invertido, el tamaño del bloque del archivo de entrada, el contenido el tamaño del bloque, el tamaño del grupo hash, el archivo de estadísticas de salida y si se ejecutará el código en local o en cluster.

Consulta de datos

```
spark-submit.cmd --name <name> --master <spark_master> --num-executors
<spark_num_executors> --executor-cores <spark_executors_cores>
--driver-memory <spark_driver_memory> --executor-memory
<spark_executor_memory>  --packages <conector_datastax_CassandraDB>
<fichero_a_ejecutar> --MQuery <fichero/s_datos_de_entrada>
[Key_size=11] [StatisticsFileName] [NumberOfPartitions] [HashName]
[Local=false]
```

Por consiguiente, este fragmento de codigo tiene las mismas propiedades por lo que a parámetros se refiere, lo único que es distinto, en este caso, es la funcionalidad, el orden y el número de parámetros. Esta instrucción, usa spark para realizar una consulta a la seqüencia de referencia que se ha guardado previamente en Cassandra. Los parámetros, pueden especificar algunos atributos de hilo para definir los recursos asignados al trabajo (número de ejecutores, número de núcleos por ejecutor y la memoria del controlador y del ejecutor). El programa recibe el método utilizado (procesamiento de consulta única o consulta múltiple), el archivo fasta de secuencia de consulta (este archivo debe almacenarse en HDFS), el nombre del espacio de claves de referencia de contenido en cassandra, el tamaño de clave (K) que debe coincidir con el uno utilizado para crear el índice invertido, el archivo de estadísticas de salida, el número de particiones para el procesamiento, el nombre del espacio de claves del índice invertido en Cassandra y si se ejecutará el código en local o en cluster.

Una vez explicada la arquitectura y como ejecutar el código en cada una de las versiones, se explicará el proceso de migración, desde los cambios más simples hasta grandes inconvenientes.

3.2. Migración

En este apartado, se explicará el motivo de la migración y también, se documentaran todos los cambios realizados durante el proceso. Estos, se dividiran en dos partes, una para cada fichero migrado, para eso tendran una introducción donde se explicará el conjunto de cambios realizado para que el código funcione localmente y otra donde se darán a conocer el resto de cambios realizados.

Como se ha comentado anteriormente, el motivo por el que se realiza esta migración es para tratar de mejorar el rendimiento y la optimización del código, mediante una mejor afinidad con el conector, es decir, tener más opciones para usar los métodos del conector en su más alto nivel y así aprovecharlo al máximo.

Para explicar los cambios, se usará la diferenciación por colores, siendo naranja el color del código en Python (premigración) y en azul el color del código en Scala (postmigración). Además se aplicará una nomenclatura para los de código en Python (P1, P2, P3, …) y otra para los de Scala (S1, S2, S3, …), representado con un hexágono en la parte superior derecha, con su inicial y el númeor de cambio correspondiente. Un cambio muy común y sin mucha importancia que nos servirás de ejemplo, es el del formato de printar las cosas por terminal.

P0

```
print("{}({}, {}, {}, {}, {}, {},
{}).".format(sys.argv[0],ReferenceFilename, KeySize, ReferenceName, Method,
BlockSize, ContentBlockSize, HashGroupsSize))
```

S0

```
println("Arg0: " + args(0) +
      "\nReferenceFilename: " + ReferenceFilename +
      "\nKeySize: " + KeySize +
      "\nReferenceName: " + ReferenceName +
      "\nMethod: " + Method +
      "\nBlockSize: " + BlockSize +
      "\nContentBlockSize: " + ContentBlockSize +
      "\nHashGroupsSize: " + HashGroupsSize)
```

Tal como se puede ver en el P0, el formato usado es que en cada {} se pondrá cada una de las variables siguiendo el orden de izquierda a derecha, en cambio en el S0, se ha optado

por un formato de nombre de la variable: valor de la variable, dicho formato, hace un código mucho más esclarecedor y es una forma mucho más entendible.

Una vez se ha explicado el ejemplo, pasamos a comentar los cambios.

Primeramente, se comentarán todos los cambios realizados en el fichero destinado a crear el índice invertido, és decir, crear la estructura y guardar los datos en la base de datos en Cassandra ("*Create Reference*").

3.2.1 Creación de índice invertido *("Create Reference")*

Los primeros cambios realizados en este fichero fueron los que se hicieron con la intención de poder ejecutar el fichero en local, estos cambios, se explicarán a continuación.

P1: Creación de la variable global Local, por si no nos llega de parámetro, tendremos el valor False por defecto, tal como se ve en P1, sino se quedará el que llegue por parámetro. Esta variable nos va a permitir diferenciar cuando el código vaya a acceder o utilizar el sistema de archivos local (windows/linux) si el valor es cierto; o bien a los archivos de datos distribuidos (HDFS) si el valor es falso.

P2: Se crea el condicional referente al sistema de ficheros distribuido HDFS, ya que si se ejecuta en local, este no existe y por lo tanto, simulamos tenerlo mediante la creación de carpetas vacias en local.

P3: Se crea el condicional referente a la funcionalidad de tirar un comando por terminal, para subir y unir los ficheros particionados en hdfs, hecho que en local, no se puede realizar porque como se ha explicado anteriormente en el P2, no hay acceso ni HDFS.

P1
```
Local = False
```

P2
```
if (Local):
    DHdfsHomePath =
"C:/Users/oscar/Desktop/SparkyBlastTest/SparkyBlast/src/hdfs/"
    DHdfsTmpPath = DHdfsHomePath + "Tmp/"
else :
    DHdfsHomePath = "hdfs://babel.udl.cat/user/nando/"
    DHdfsTmpPath = DHdfsHomePath + "Tmp/"
```

P3

```
if(not Local) :
    cmd = ['hdfs', 'dfs', '-getmerge',OutputFile+"/part-*",
"/tmp/prueba"]
    if (run_cmd(cmd)):
        print("Error getmerge")
    cmd = ['hdfs', 'dfs', '-appendToFile',"/tmp/prueba",
DHdfsHomePath+statisticsFileName]
    if (run_cmd(cmd)):
        print("Error appendToFile")
```

P4

```
if os.path.exists(OutputFilename):
   shutil.rmtree(OutputFilename)
      if Local :
          os.remove(OutputFilename)
      else :
          os.system("hdfs dfs -rm -R "+OutputFilename)
```

P5

```
def check_file(hdfs_file_path):
    import os.path
    os.path.exists(hdfs_file_path)

    if(not Local):
        cmd = ['hdfs', 'dfs', '-test', '-e', hdfs_file_path]
        code = run_cmd(cmd)
        return code


def remove_file(hdfs_file_path):
    import os.path
    if os.path.exists(hdfs_file_path):
        os.remove(hdfs_file_path)

    if(not Local):
        cmd = ['hdfs', 'dfs', '-rm', '-R', hdfs_file_path]
        code = run_cmd(cmd)
        return code
```

P6
```
if (len(sys.argv)>9):
    Local = sys.argv[9]
```

P4: Se crea el condicional referente a evitar que se lance por terminal un borrado de ficheros al sistema de fichero distribuido hdfs, ya que si se ejecuta en local, este no existe y por lo tanto, nos daría error.

P5: En el método “check_file” se añade el condicional para los casos que no sea en modo local, que se pueda usar el comando de comprobación de si un fichero está en HDFS, en local no tiene sentido hacerlo por el motivo expuesto anteriormente. En el otro método, en el “remove_file”, se añade el condicional en base a las mismas condiciones y evita que se use el borrado de ficheros de HDFS cuando no se está en local.

P6: Si el usuario envia el noveno argumento, este será asignado a Local, ya que en el noveno argumento, se enviará True o False.

Una vez realizados los cambios necesarios para poder ejecutar el código en local, se procederá a la migración de este. Las problemáticas y cambios destacados se comentarán a continuación comparando ambas versiones del código.

P7
```
    ## Configure Spark
    conf = SparkConf().setAppName(APP_NAME+ReferenceName)
    sc   = SparkContext(conf=conf)
    sqlContext = SQLContext(sc)
    random.seed()
```

S1
```
/**
* Configure Spark
*/
val spark = SparkSession
 .builder()
 .appName(APP_NAME + ReferenceName)
 .master("local[3]")
 .config("spark.executor.instances", 1)
 .config("spark.executor.cores", 2)
```

```
 .config("spark.executor.memory", "2gb")
 .getOrCreate()
```

En este caso, el cambio de P7 a S1, consiste en el uso de una forma no deprecada de crear el contexto y el spark sql, la cual es usando el SparkSession [35], ya que permite añadir todas las configuraciones mediante el patrón builder i al crear un objeto del tipo SparkSession, obtenemos tanto el SparkContext[4] como el SqlContext[5].

P8

```
#
# Create the sequences and hash tables for the Reference
#
def CreateCassandraReferenceTables(ReferenceName):
    cluster = Cluster(['127.0.0.1'])
    session = cluster.connect()
    session.execute("CREATE KEYSPACE IF NOT EXISTS "+ ReferenceName +" WITH replication
= {'class': 'SimpleStrategy', 'replication_factor': 1 };")
    session.execute("DROP TABLE IF EXISTS "+ ReferenceName +"."+ DReferenceHashTableName
+";")
    session.execute("CREATE TABLE "+ ReferenceName + "."+ DReferenceHashTableName +"
(seq text, block int, value list<bigint>, PRIMARY KEY(seq,block));")

    session.shutdown()
    cluster.shutdown()

    if (DDebug):
        print("Cassandra Database Created.")
```

S2

```
/**
* Create the sequence and hash tables for the Reference
*/
def CreateCassandraReferenceTables(referenceName: String): Unit = {

 val nodes : util.Collection[InetAddress] = DCassandraNodes.map(ip =>
InetAddress.getByName(ip)).asJavaCollection
 val cluster: Cluster = Cluster.builder().addContactPoint("127.0.0.1").build()
 implicit val session: Session = cluster.connect()
 session.execute("CREATE KEYSPACE IF NOT EXISTS " + referenceName + " WITH replication =
{'class': 'SimpleStrategy', 'replication_factor': 1 };")
 session.execute("DROP TABLE IF EXISTS " + referenceName + "." + DReferenceHashTableName
+ ";")
```

[4] Un SparkContext representa la conexión a un clúster de Spark y se puede usar para crear RDDs, acumuladores y variables de transmisión en ese clúster.
[5] El punto de entrada para trabajar con datos estructurados (filas y columnas) en Spark 1.x.
A partir de Spark 2.0, se reemplaza por SparkSession. Sin embargo, mantenemos la clase aquí por compatibilidad con versiones anteriores.

```
 session.execute("CREATE TABLE " + referenceName + "." + DReferenceHashTableName + "
(seq text, block int, value list<bigint>, PRIMARY KEY(seq,block));")

 session.close()
 cluster.close()

 if (DDebug) {
   println("Cassandra Database Created.")
 }
}
```

Este es uno de los cambios más importantes de toda la migración (de P8 a S2), porque afecta directamente al conector de Cassandra (toda la funcionalidad usada es lógica que nos da el conector “Datastax Cassandra” mencionado anteriormente) y el hecho de usar scala de la forma en que lo hacemos, para ello mirar el fragmento de codigo etiquetado con S2, permite un mínimo aumento de velocidad en cuanto a la ejecución del código cuando trata de crear conexiones con la BD. Se ha decidido usar la conexión mediante InetAddress, ya que era la única opción que nos permitia usar una o varias IPs, en este caso llamados “contact points”.

P9

```
# Read Reference file (offset,line) from hdfs
reference_rdd = sc.newAPIHadoopFile(
    ReferenceFilename,
    'org.apache.hadoop.mapreduce.lib.input.TextInputFormat',
    'org.apache.hadoop.io.LongWritable',
    'org.apache.hadoop.io.Text',
)
```

S3

```
/**
* Read Reference file (offset,line) from hdfs
*  Convert rdd to string rdd, to use it without exceptions
*/
var reference_rdd: RDD[(Long, String)] =
sc.newAPIHadoopFile(referenceFilename,
 classOf[TextInputFormat], classOf[LongWritable], classOf[Text], new
conf.Configuration())
 .map( row => {
   (row._1.get(), row._2.toString)
 })
```

En este cambio de P9 a S3, tal como se puede ver, el método del SparkContext, *sc*, *newAPIHadoopFile* pide unos parámetros completamente distintos y por lo tanto, se tuvo que adaptar a la nueva forma que pedía el método y por lo tanto, a parte de pasarle, el nombre del fichero y las tres classes de hadoop (*TextInputFormat, LongWritable, Text*), también se debe pasar una configuración y mapear el resultado devuelto a RDD mediante el uso de map. Todo eso para obtener el mismo resultado en S3, que en P9, porque a parte de los parámetros de entrada, también cambio el valor de retorno.

P10

```
def CreateDataFrame(reference_rdd):

    # Create DataFrame
    reference_df =
sqlContext.createDataFrame(reference_rdd,["file_offset","line"])
...
 if (Method==ECreate2LinesData):
        df = Create2LinesDataFrame(df1, header_size, BlockSize)
    elif (Method==ECreate1LineDataWithoutDependencies):
        df = Create1LineDataFrameWithoutDependencies(df1, header_size)
    elif (Method==ECreateBlocksData):
        df = CreateBlocksDataFrame(df1, header_size, BlockSize)

    return df
```

S3

```
def CreateDataFrame(reference_rdd: RDD[(Long,String)], spark: SparkSession):
DataFrame = {
...
/** Create Dataframe */
import spark.implicits._
implicit val enc: Encoder[data_GenClass] = Encoders.product[data_GenClass]
val reference_df : DataFrame = reference_rdd.toDF("file_offset", "line")
...
if( Method == ECreate2LinesData ) {
 df = Create2LinesDataFrame(df1, headerSize, BlockSize, spark)
}
else if( Method == ECreate1LineDataWithoutDependencies ) {
df = Create1LineDataFrameWithoutDependencies(df1, headerSize)
}
else if( Method == ECreateBlocksData ) {
 df = CreateBlocksDataFrame(df1, headerSize, BlockSize, spark)
}

df
}
```

S3.1

```
case class data_GenClass(size: Int, lines: String, offset: BigInt)
```

En este cambio de P10 a S3, se destacan dos partes, la primera, es la creación de un encoder del tipo "data_GenClass" eso implica que se ha creado una *"case class"* global, tal como podemos ver en la S3.1, esta clase contiene 3 variables, las cuales se usarán, a posteriori, para la creación de un Dataframe más fácilmente. La segunda, está dividida en dos, la primera división, es que en lugar de usar el método createDataframe como se hace en P10, se ha usado el rdd.toDF, porque para usar el createDataframe, en scala se necesita un schema y hacer uso del sparksession, ya que como hemos dicho, sustituye el sqlcontext en spark 2.x. La segunda división, consiste en darse cuenta que en la S3, en los métodos *"Create2LinesDataFrame,Create1LineDataFrameWithoutDependencies, CreateBlocksDataFrame"* se ha añadido el sparkSession como parámetro.

P11

```
if (DTiming):
      print("Time required for read and prepare dataframe in {}
seconds.\n".format(round(time() - gt0_bc.value,3)))
```

S4

```
if(DTiming) {
 val time: BigDecimal = BigDecimal((System.currentTimeMillis() / 1000) -
gt0_bc.value).setScale(3, RoundingMode.HALF_UP)
 println("Time required for read and prepare dataframe with " +
df1lines2.count() +
   " rows using " + df1lines2.rdd.getNumPartitions + " partitions in " +
time + " seconds.")
}
```

El cambio que se muestra de P11 a S4 que se produce en ambos ficheros, pero este es el primer ejemplo que se nos aparece y por ese motivo, se explica aquí que para poder llevar a cabo el round tal como se hace en la P11, en Scala, tal como se ve en S4, se uso el *BigDecimal* [36] para poder hacer el redondeo y se elige de la mitad para arriba (*half up*).

P12

```
    # Calculate keys
    t1 = time()
    result = df.rdd.flatMap(generateReferenceKeys)
```

S5

```
    /** Calculate keys */
    t1 = System.currentTimeMillis() / 1000
    df.show(3)
    val dataList: ListBuffer[List[((String, BigInt), BigInt)]] =
iterateDF(df, spark)
    //GEN REF. KEYS LIKE AN RDD Documentar cambios RDD df.rdd.flatmap to
list.flatten

//https://users.scala-lang.org/t/convert-a-list-list-string-into-rdd-str
ing/7828
    println(iterateDF(df, spark).toList)
    val result: RDD[((String, BigInt), BigInt)] =
spark.sparkContext.parallelize(dataList.flatten)
```

S5.1

```
def iterateDF(df : DataFrame, spark: SparkSession) :
ListBuffer[List[((String, BigInt), BigInt)]] = {
 import spark.implicits._
 val dataList = new ListBuffer[List[((String, BigInt), BigInt)]]
 df.as[data_GenClass].take(df.count.toInt).foreach(t => {
   dataList.append(generateReferenceKeys(mutable.Map(("size", t.size),
("lines", t.lines), ("offset", t.offset))).toList)
 })
 dataList
}
```

Este es el mayor cambio de todo este fichero, ya que contiene el cambio de un flatmap[6], tal como se ve en el fragmento de código P12, por un método, iterateDF (fragmento S5.1), que se ha creado porque el flatMap en scala funciona diferente y no nos permitía pasar una función de la forma que se hace en P12, por lo tanto, en el nuevo método, se ha usado la *case class,* mencionada anteriormente, para convertir el df en un dataset, con las columnas especificadas en esta e iterar cada una de sus rows, y llamar al método

[6] Como se puede adivinar por su nombre, es la combinación de un mapa y una operación plana. Eso significa que primero aplica una función a sus elementos y luego la aplana.

generateReferenceKeys pasándole una Lista con un Map dentro, para así facilitarnos luego el coger cada valor de la columna cuando iteremos la lista. Esta funcionalidad, simula el funcionamento de un flatMap de P12 pero de forma secuencial. Al devolver el resultado, se devolvía una Lista de Listas y para que se obtuviera la parte flat del flatMap, es decir, para obtener solo una lista de elementos, se usa el metodo flatten [37] (método perteneciente a las listas de scala).

P13

```
    # Calculate Reference keys and write to cassandra

#result.toDF().write.format("org.apache.spark.sql.cassandra").mode('append')
.options(table=DReferenceHashTableName, keyspace=ReferenceName).save()

result.toDF().write.format("org.apache.spark.sql.cassandra").mode('overwrite
').options(table=DReferenceHashTableName,
keyspace=ReferenceName).option("confirm.truncate","true").save()
    tc = time()
```

S6

```
val simpleSchema = new StructType()
 .add("seq", StringType)
 .add("block", IntegerType)
 .add("value", ArrayType(IntegerType))
...
/**
* Calculate Reference keys and write to cassandra
*
result.toDF().write.format("org.apache.spark.sql.cassandra").mode('appen
d').options(table=DReferenceHashTableName,
keyspace=ReferenceName).save()
**/
val resultDF = spark.createDataFrame(rowsResultRDD, simpleSchema)
resultDF.write.format("org.apache.spark.sql.cassandra")
 .mode("overwrite")
 .options(Map("table" -> DReferenceHashTableName, "keyspace" ->
ReferenceName))
 .option("confirm.truncate", "true")
 .save()
```

Este cambio de P13 a S6, es debido a la necesidad que tiene scala al usar el método “createDataframe”, el cual requiere de una schema para poder crear un Dataframe a partir de un RDD, y se ha usado esta forma en lugar de la de convertir a Dataframe mediante el método “toDF()”, porque no se puede usar si el RDD contiene objetos del tipo Row, y este

era nuestro caso. Por lo tanto, se realizo tal como se ve en S6, este cambio no afecta a nivel de rendimiento, al final, en ambos casos creamos un Dataframe, ya sea de una forma o de otra.

Una vez explicados todos los cambios importantes realizados y el motivo por el cual se hicieron, se procede a hacer el mismo procedimiento con el fichero de encargado de las consultas de los datos.

3.2.2 Consulta de datos *(“Do Query”)*

Para explicar los cambios, se usará el mismo método que en el apartado 3.2.1. En este fichero, se explicarán primero los cambios que se han considerado importante, pero se separan en dos partes, ya que este fichero tiene dos métodos principales. Aunque antes de argumentar esa parte, se redactarán los cambios generales de todo el fichero.

Es de suma importancia destacar que los cambios que aparezcan en este fichero y se hayan explicado en el otro, no se van a volver a explicar debido a que la problemática solo surgió la primera vez que se hizo.

P14

```
    if (len(sys.argv)<2):
        print("Error parametes. Usage: DoQuery [--MQuery] <Query_Files> <ReferenceName>
[Key_size=11] [StadisticsFile] [HashName] .\n")
        sys.exit(1)
    ...

    if (len(sys.argv)>1 and sys.argv[1].upper()=="--MQUERY"):
        DoMultipleQuery = True
        args=2
    else:
        DoMultipleQuery = False
        args=1

    QueryFilename = sys.argv[args]
    ReferenceName = sys.argv[args+1].lower()
    HashName=None
    NumberPartitions = DNumberPartitions
    if (len(sys.argv)>(args+2)):
        KeySize = int(sys.argv[args+2])
    if (len(sys.argv)>args+3):
        StatisticsFileName = sys.argv[args+3]
    if (len(sys.argv)>(args+4)):
        NumberPartitions = int(sys.argv[args+4])
    if (len(sys.argv)>(args+5)):
        HashName = sys.argv[args+5].lower()
```

S7

```
if(args.length < 2) {
 println("Error parameters. Usage: DoQuery [--MQuery] <Query_Files> <ReferenceName> [Key_size=11]
[StadisticsFile] [HashName] .")
 sys.exit(1)
}

...

if(args.length > 1 && args(1).toUpperCase.equals("--MQUERY")){
 DoMultipleQuery = true
 numArgs = 2
} else {
 DoMultipleQuery = false
 numArgs = 1
}

QueryFilename = args(numArgs)
ReferenceName = args(numArgs + 1).toLowerCase
HashName = ""
NumberPartitions = DNumberPartitions

if(args.length > numArgs + 2) {
 KeySize = args(numArgs + 2).toInt
}
if(args.length > numArgs + 3) {
 StatisticsFileName = args(numArgs + 3)
}
if(args.length > numArgs + 4) {
 NumberPartitions = args(numArgs + 4).toInt
}
if(args.length > numArgs + 5) {
 HashName = args(numArgs + 5).toLowerCase
}

if(args.length > numArgs + 6) {
 Local = args(numArgs + 6).toBoolean
}
```

Este cambio de P14 a S7, el elemento clave, es que se ha renombrado la variable args a numArgs, porque en Scala (S7) args es el simil de argv de python y por lo tanto, se ha cambiado el argv por args y luego, renombrar la variable que se llama de la misma forma.

3.2.2.1 Una sola consulta *(“Single Query”)*

Este subapartado describe y explica todos los cambios realizados para la migración del método “Query”.

P15

```
    # Create Cassandra Sesion
    for i in range(0,DCassandraRetriesNumber):
        while True:
            try:
```

```
                cluster = Cluster(DCassandraNodes)
                session = cluster.connect()
            except:
                # Print exception info.
                print("@@@@ Capatured Exception")
                exc_info = sys.exc_info()
                traceback.print_exception(*exc_info)
                del exc_info
                # Retry operation after a delay.
                sleep(DCassandraRetryTimeout)
                continue
            break
    if session is None:
        raise
```

S8

```
/** Create Cassandra Session */

var session : Session = null
var cluster: Cluster = null
for(i <- 0 until DCassandraRetriesNumber) {
 var times = 0
 val nodes : util.Collection[InetAddress] = DCassandraNodes.map(ip =>
InetAddress.getByName(ip)).asJavaCollection

 while (times < nodes.size()) {
     try {
       times += 1
       cluster = Cluster.builder().addContactPoints(nodes).build()
       session = cluster.connect()
       //TODO preguntar fcores bucle infinit

//https://alvinalexander.com/scala/break-continue-for-while-loops-in-sca
la-examples-how-to/
     } catch {
       case e: Exception => {
         /** Print except info */
         println("@@@@@ Captured Exception")
         println(e.getStackTrace.mkString("Array(", ", ", ")"))

         /** Retry operation after a delay. */
         Thread.sleep(DCassandraRetryTimeout.toLong)
       }
     }
 }
```

```
}
if (session == null) {
 throw new Exception("Null session")
}
```

De P15 a S8, se deben de destacar dos cambios importantes, a parte del cambio en la estructura de creación de sessión y conexión, que ya se comentó en el apartado 3.2.1.
El primer cambio, es que se ha cambiado la estructura de "try-except" usada en Python, por una estructura de "try-catch" que realiza la misma función, ya que en Scala es mejor realizarlo de esta forma debido a que el except no es un término normalizado para los grupos "try-catch" [38].

El segundo cambio, es referente a una mejora en el rendimiento, ya que se harán menos iteraciones en el bucle "while", a causa de que se ha cambiado el "`while True:`" de P15, itera siempre hasta que salimos de forma forzada de este, por un condicional "`while (times < nodes.size())`" de S8, que hace exactamente lo mismo pero sin tener que estar pendiente de forzar la salida del bucle i por lo tanto, es más ópitmo a nivel de código y también mejora el rendimiento ya que nos ahorramos una vuelta de bucle.

P16

```
    # Calculate Dataframe
    query_df, query_sequence = CreateDataFrame(query_rdd)
    query_df.show(3)
```

S9

```
/** Calculate Dataframe */
var resultTuple: (DataFrame, String) = createDataframe(queryRdd, spark)
var queryDf: DataFrame = resultTuple._1
var querySequence: String = resultTuple._2

queryDf.show(3)
```

El cambio de P16 a S9, incluye el cambio de método createDataframe para que este no devuelva dos valores, como en P16, sino que devuelva un tupla con esos valores, y para hacer que tuviera el mismo funcionamiento, se han creado luego las dos variables correspondientes ("queryDf", "querySequence") en base a los elementos de la tupla.

P17

```
def nonasciitoascii(unicodestring):
    return unicodestring.encode("ascii","ignore")
...
    convertedudf = udf(nonasciitoascii)
    df1 = df.withColumn("line", convertedudf(upper(df.line)))
```

S9

```
def nonAsciiToAscii(elemToParse: String): String ={
 // keep only valid ASCII characters
 elemToParse.toUpperCase.filter(_ <= 0x7f)
}
...
val stringToAsciiStringUDF = udf(nonAsciiToAscii _)
df2lines = initDf.withColumn("line",
stringToAsciiStringUDF(initDf("line")))
```

El cambio entre P17 y S9, se centra en la función "nonAsciiToAscii", encargada de parsear los caracteres a ASCII. La cuál se ha cambiado a un filter de los solo los caracteres ASCII válidos, el resultado sigue siendo el mismo, a pesar del cambio realizado. Por otro lado, tenemos la forma de crear una udf a partir de un método, que en scala pide el tipo de retorno, y por eso específicamos ese parámetro/s con "_" que signfica que al usar la udf en el dataframe, se le pasaran ya los parámetros que necesite esta udf. Un ejemplo de uso, lo tenemos en las últimas líneas de ambos fragmentos de código.

P18

```
def Create1LineDataFrameWithoutDependencies(df, header_size, blocksize):
...
   convertedudf = udf(nonasciitoascii)
    df = df.withColumn("lines", convertedudf(upper(df.line)))
    df3 = df.withColumn("size", F.length(df.lines)-DKeySize) \
             .withColumn("offset", df.file_offset-header_size) \
             .drop(df.file_offset)
...
    return df4
```

S10

```
def create1LineDataFrameWithoutDependencies(df: DataFrame, headerSize: Int, blockSize: Int, spark:
SparkSession): DataFrame = {
...
```

```
 val stringToAsciiStringUDF = udf(nonAsciiToAscii _)
 df1lines = initDf.withColumn("lines", stringToAsciiStringUDF(initDf("line")))
 df1lines2 = df1lines.withColumn("size", functions.length(df1lines("lines") - DKeySize))
   .withColumn("offset", df1lines("file_offset") - headerSize)
   .drop("file_offset")

...
 df1lines2
```

En este cambio, se corrige un error para el funcionamiento del código, debido a que en P18, el método nos devuelve “df4”, dicha variable, no existe en ningún momento i por lo tanto, se corrige en el código migrado, S10, y se devuelve, la última variable donde se ha guardado el resultado final después de todas la modificaciones (“df1lines”), que en el caso de P18 sería equivalente a devolver “df3”.

P19

```
    if hashName is None:
       querySelect = "SELECT * FROM " + referenceName + "." +
DReferenceHashTableName + " WHERE seq=%s"
    else:
      querySelect = "SELECT * FROM " + hashName + "." +
DReferenceHashTableName + " WHERE seq=%s"
```

S11

```
var querySelect: String = ""
if (hashName == null || hashName.equals("")) {
 querySelect = "SELECT * FROM " + referenceName + "." +
DReferenceHashTableName + " WHERE seq='" + key + "'"
} else {
 querySelect = "SELECT * FROM " + hashName + "." +
DReferenceHashTableName + " WHERE seq='" + key + "'"
}
```

Este cambio, es simplemente debido a que Scala no es capaz de detectar el “%s”, tal como se usa en P19, y por lo tanto, esta se ha añadido directamente en la concatenación del String de consulta, tal como se puede ver en S11.

P20

```
def ProcessCassandraQueryR(key, despl, session, referenceName, hashName):
...
 res = []
    for result_row in resultSelect:
        if (DDebug):
```

```
            print("{}.{}->{}".format(result_row.seq, result_row.block, list(map(lambda offset:
offset, result_row.value))))
        res.append(map(lambda offset: offset+int(despl), result_row.value))
        #res.append(map(lambda offset: int(offset)+int(despl), result_row.value))

    res = flattened_list = [y for x in res for y in x]


    if (DDebug):
        print("Processing Query Record result: {}.".format(res))

    return res
```

S12

```
def processCassandraQueryR(key: String, despl: Int, session: Session, referenceName: String,
hashName: String): ListBuffer[Long] = {
...
val result: ListBuffer[mutable.Buffer[Long]] = new ListBuffer[mutable.Buffer[Long]]()
import collection.JavaConversions._

resultSelect.foreach(resultRow => {
 println(resultRow.getList("value", classOf[java.lang.Long]))
 val valuesList: util.List[java.lang.Long] = resultRow.getList("value", classOf[java.lang.Long])
 if (DDebug) {
   println(resultRow.getString("seq") + "." + resultRow.getInt("block") + " -> " +
valuesList.map(offset => offset))
 }
 val modifValuesList: mutable.Buffer[Long] = valuesList.map(offset => offset + despl)
 result+=modifValuesList
})

println(result)

if (DDebug) {
 println("Processing Query Record result: " + result + ".")
}
val flattenedResult: ListBuffer[Long] = result.flatten

if (DDebug) {
 println("Processing Query Record result: " + flattenedResult + " .")
}

flattenedResult
```

El cambio que se muestra entre P20 y S12, consiste en el cambio de la forma de realizar el bucle y de sobretodo, de obtener los diferentes valores, ya que en Scala (S12), los tipos de los métodos han ido cambiando respecto a los de Python (P20). Lo más importante de esta parte, es que, en S12, había problemas para obtener el BigInt, que venía dentro del resultado de la consulta, porque no había manera de obtener ese valor a causa de que el tipo BigInt no era aceptado como válido y por lo tanto, se buscó una alternativa, la cuál fue, guardar ese valor tipo BigInt en un Long de Java, porque este encapsulaba perfectamente el valor devuelto porque un long tiene más capacidad que un BigInt, y mezclar Scala y Java,

es un práctica común debido a que Scala, para según que cosas, no tiene forma viable de llevarlo a cabo, este caso es uno de esos.

P21

```
def GetKeysOffsetsInReference(sc, session, referenceName,
keysdespl_rdd):
    global DDebug
    if (DDebug):
        print("GetKeysOffsetsInReference")

    if (DDebug & False):
        # Show Reference Table
        GenRef = load_and_get_table_df(referenceName,
DReferenceContentTableName)
        print("Reference Table:")
        GenRef.show()
...
 return matching_despl_rdd
```

S13

```
def getKeysOffsetsInReference(spark: SparkSession, referenceName:
String, keyDesplRdd : RDD[(String, Int)]): RDD[Long] ={
 if(DDebug) {
   //1
   println("GetKeysOffsetsInReferenceByPartition")
 }

 if(DDebug && false) {
   //Show reference table
   val genRef : DataFrame = loadAndGetTableDF(spark, referenceName,
DReferenceContentTableName)
   println("Reference Table:")
   genRef.show()
 }
...
matchingDesplRDD
}
```

El cambio entre P21 y S13, consiste en la sustitución del parámetro “sc” por el parámetro “spark” y de la eliminación del parámetro session, ya que este no se usaba. Además, se debe destacar que en el caso de S13, se ha añadido “spark” como parámetro al método “`loadAndGetTableDF`”, para poder usar el sqlcontext dentro de dicho método, a diferencia del P21 que no lo usa, sino que coge la variable global, práctica no muy buena en Scala.

P22

```
# 3. Calculate Top-matching offsets
    t3 = time()
    offsets_count = reference_keys.map(lambda off: (off,1)).reduceByKey(add)
```

S14

```
/** 3. Calculate Top-matching offsets */
val t3: Long = System.currentTimeMillis() / 1000
val offsetsCount: RDD[(Long, Int)] = referenceKeysRdd.map(off => (off,
1)).reduceByKey(_ + _).sortBy(kv => kv._1, true)
```

En este cambio de P22 a S14 se muestra como se ha adaptado el la lambda usada en P22 junto al método add del *reduceByKey*, a código Scala, hecho que implica usar *arrow function* i en el *reduceByKey* en lugar de ponerle add, se usa el "_+_", que hace exactamente lo mismo, su misión es ir sumando los valores.

P23

```
def DistributeAligments(sc, querySequence, referenceName, offsets_count,
contentBlockSize):
...
    #CassandraAligmentR(offsets_count.collect()[0][0])
    aligments = offsets_count.map(lambda off: CassandraAligmentR(off[0]))
...
    return aligments
```

S15

```
def distributeAlignments(spark: SparkSession, session: Session,
querySequence: String, referenceName: String, offsetsCount : RDD[(Long,
Int)], contentBlockSize: Int): RDD[Seq[Any]] ={
...
 //Changed RDD.map to for because spark cannot serialize that task
 val elems: ListBuffer[Seq[Any]] = ListBuffer[Seq[Any]]()
 for(off <- offsetsCount.collect()){
   elems.append(cassandraAligmentR(off._1.toInt, session, spark))
 }
...
 aligments
}
```

Este cambio de P23, recae justo donde se está usando el *"offsets_count.map"*, el cual pasará a ser un for porque sino se hacía ese cambio, se tenía una excepción de spark que nos alertaba de que spark no podía serializar la tarea. Además de eso, hay que resaltar que en el caso de S15 la función "*cassandraAligmentR*" tiene dos parámetros más, la *session* (sesión de la DB) y spark (sesión de spark donde esta el contexto y todo lo necesario).

P24

```
@static_vars(BlockCache = collections.defaultdict(list))
def GetRefereceContentBlocksCache(ses, referenceName, bbegin, bend):
    ...
    for block in range(bbegin, bend+1):
        if block in GetRefereceContentBlocksCache.BlockCache:
            if (DDebug):
                print("HIT block cache {} ->
{}.".format(block,GetRefereceContentBlocksCache.BlockCache[block][0]))
            contentSequence = contentSequence + GetRefereceContentBlocksCache.BlockCache[block][1]
            blocks_read +=1
            end = GetRefereceContentBlocksCache.BlockCache[block][0] +
len(GetRefereceContentBlocksCache.BlockCache[block][1])
        else:
       ...
```

S16

```
def getReferenceContentBlocksCache(session: Session, referenceName: String, bbegin: Int, bend:Int):
(Int, String) = {
    val blockCache: ListBuffer[(Int, String)] = new ListBuffer[(Int, String)]()
...

 for(block <- bbegin until bend + 1){
   if(blockCache.nonEmpty && blockCache.size+1 >= block) {
     if(DDebug){
       println("HIT block cache " + block + " -> " + blockCache.lift(block).get._1 + " .")
     }
     contentSequence = contentSequence + blockCache.lift(block).get._2
     blocksRead += 1
     end = blockCache.lift(block).get._1 + blockCache.lift(block).get._2.length
   } else {
     ...
}
```

En este cambio se han de destacar dos partes, por parte del P24, hay que resaltar el uso de "@static_vars" [40], que le da un valor por defecto al objeto en el que lo almacenes, en este caso, es la variable "*BlockCache*". Para poder implementar la misma lógica en Scala, se tuvo que añadir una variable local del tipo lista ya inicializada, y además, para poder obtener los valores de la misma forma que se hacía en P24, en S16 se ha usado la función "*lift*" [41], que nos permite obtener esos valores y encima nos evita las posibles excepciones que esos puedan lanzar.

P25

```
def DoAligment(querySequence, referenceSequence):
    ti=time()
    align_seq1,align_seq2,align_score,i_start = Aligment(querySequence,
referenceSequence)
…
    return align_seq1,align_seq2,align_score, i_start
```

S17

```
def doAlignment(querySequence: String, referenceSequence:String): Seq[Any] = {
 val ti = System.currentTimeMillis()/1000
 val aligmentValues: Seq[Any] = aligment(querySequence, referenceSequence)
…
 aligmentValues
}
```

Este cambio de P25 a S17, es importante comentar que cuando hay retornos como el de P25, en scala no hay posibildad de hacerlo de esas forma, y lo que se ha decidido hacer ha sido guardar todos esos valores en una seqüencia y devolverla como resultado final. Ese hecho hace que el funcionamiento no se vea alterado, solo que se tenga en cuenta, a la hora de tratar dichos valores.

3.2.2.2 Consulta múltiple *(“Multiple Query”)*

Este subapartado describe y explica todos los cambios realizados para la migración del método “MultipleQuery”.

P26

```
    if DProcessingByPartitions:
  matching_despl_rdd = offsets_rdd.glom().flatMap(lambda kv:
ProcessMultipleCassandraQueryByPartition(kv) if len(kv) > 0 else "")
    else:
 matching_despl_rdd = offsets_rdd.map(lambda kv:
ProcessMultipleCassandraQuery(kv))
```

S18

```
var matchingDesplRDD: RDD[(String, Seq[Long])] =
 if(DProcessingByPartitions) processMultipleCassandraQueries(offsetsRDD,
true)
 else processMultipleCassandraQueries(offsetsRDD, false)
```

S18.1

```
def processMultipleCassandraQueries(offsetsRDD: RDD[String], partition:
Boolean): RDD[(String, Seq[Long])] = {
 if(partition) {
   val matchingDesplRDD: RDD[(String, Seq[Long])] =
offsetsRDD.glom().flatMap(kv => {
     if (kv.length > 0) {
       val queryArray: Array[(String, Seq[Long])] =
processMultipleCassandraQueryByPartition(kv)
       queryArray
     } else {
       val emptyArray = Array[(String, Seq[Long])]()
       emptyArray
     }
   })
   println(matchingDesplRDD.collect().mkString("Array(", ", ", ")"))
   matchingDesplRDD
 } else {
   val matchingDesplRDD: RDD[(String, Seq[Long])] =
offsetsRDD.glom().flatMap(kv => {
     processMultipleCassandraQuery(kv)
   })
   println(matchingDesplRDD.collect().mkString("Array(", ", ", ")"))
   matchingDesplRDD
 }
}
```

En este cambio de P26 a S18, se ha cambiado la funcionalidad del condicional conteniendo toda la lógica en una función, la que se muestra en S18.1, es un cambio del flatMap de Python (P26) a una función que hace lo mismo (S18 and S18.1). La metodologia es la misma lo que en lugar de hacerlo en una línea como en P26, está hecho en un método auxiliar (S18 & S18.1).

Estos son todos los cambios importantes que se han decidido destacar en este documento, una vez explicados, se procederá a comentar la validación, es decir que ambas soluciones nos aporten el mismo resultado y también se compararan los tiempos de ejecución.

Capítulo 4. Validación.

En este capítulo, se comprobaran que los resultados de ambas versiones sean los mismos y también se hará una comprobación del tiempo de ejecución de cada una de estas. A continuación, se procederá con la comprovación de los resultados de ambos ficheros, el que crea la referencia ("*createReference*") y el que consulta los datos ("doQuery").

4.1. Creación de referencia

Primero comprobaremos el resultado de la version de Python y el tiempo que esta tarda, para hacerlo, nos colocaremos en la carpeta donde tengamos el fichero de datos y el codigo Python y usaremos la siguiente instrucción:

```
spark-submit.cmd --name Hash_example2 --master local[3] --num-executors 4
--executor-cores 1  --driver-memory 2g --executor-memory 4g  --packages
com.datastax.spark:spark-cassandra-connector_2.11:2.4.1 ./SparkBlast_CreateReference.py
./Example2.txt 11 example2 1 1000 10000 12000000 ./Results/Scalability_GRCh38F.res true
```

Las estadísticas iniciales antes de inicar el programa, con los datos que se ven en la consulta anterior, son las siguientes:

```
++++++++++++ INITIAL STATISTICS 08/25/2022 01:04:05 ++++++++++++++++++++
+ Reference: example2   File: ./Example2.txt.
+ Key Size: 11          Method: 1.
@@@@@@ BROADCAST VAR @@@@@@ :::::::::::::: <pyspark.broadcast.Broadcast object at
0x0000000005771208>
+ Num Executors: None   Executors/cores: None   Executor Mem: None.
+ Hash Groups Size: 12000000    Partition Size: 1000    Content Block Size: 10000.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Y las estadísticas finales son estas:

```
++++++++++++++++++++ FINAL STATISTICS 08/25/2022 01:04:05 ++++++++++++++++++
+ Reference: example2   File: ./Example2.txt.
+ Key Size: 11                  Method: 1.
+ Num Executors: None           Executors/cores: None           Executor Mem: None.
+ Hash Groups Size: 12000000    Partition Size: 1000    Content Block Size: 10000.
+ Total Time: 28.447            Data Read Time: 13.865          Data Frame Time: 9.155.
+ Key Calc Time: 2.564          Cassandra Write Time: 2.863.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Esta ejecución ha tardado **28,447 segundos** con los parámetros que vemos en las estadísticas finales mostradas justo en el fragmento de arriba.

A continuación comprobaremos los resultados de la ejecución del código en Scala y el tiempo que tarda. Usaremos la misma instrucción pero cambiando el fichero Python por uno en Scala.

Las estadísticas iniciales antes de iniciar el programa, con los datos que se ven en la consulta anterior son:

```
++++++++++++ INITIAL STATISTICS ++++++++++++ 25/08/2022 19:58:42 ++++++
+ Reference: Sequences
File:C:/Users/oscar/Desktop/tfmTest/sparkScalaTest1/src/main/scala/org/tfm/sparkScala/do
cs/Example2.txt .
+ Key Size: 21        Method: 1 .
+ Num Executors: 1    Executors/cores: 2    Executor Mem: 2gb.
+ Hash Groups Size: 12000000        Partition Size: 131072       Content Block Size:
10000.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Las estadísticas finales después de ejecutar el programa, son las siguientes:

```
++++++++++++++ FINAL STATISTICS ++++++++++++ 25/08/2022 19:58:57 +++++++
+ Reference: Sequences
File:C:/Users/oscar/Desktop/tfmTest/sparkScalaTest1/src/main/scala/org/tfm/sparkScala/do
cs/Example2.txt .
+ Key Size: 21        Method: 1 .
+ Num Executors: 1    Executors/cores: 2    Executor Mem: 2gb.
+ Hash Groups Size: 12000000        Partition Size: 131072       Content Block Size:
10000.
+ Total Time: 15.000  Data Read Time: 5.000 Data Frame Time: 4.000 .
+ Key Calc Time: 1.000       Cassandra Write Time: 5.000 .
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Esta ejecución ha tardado **15,000 segundos** con los parámetros que vemos en las estadísticas finales mostradas justo en el fragmento de arriba.

Por lo tanto, se puede extraer de esta conclusión que el código scala va mucho más rápido que el de python debido a que tarda 13 segundos menos y por lo tanto, se puede ver que el hecho de haber migrado el código nos ha reportado una mejora muy sustancial en tiempo.

4.2. Consulta de datos

En este subapartado, se realizará la validación y comprobación del tiempo de la consulta de datos y por lo tanto, usaremos la misma estrategia que en el 4.1. Primero ejecutaremos el código Python y veremos cuanto tarda, para llevarlo a cabo, usaremos la siguiente instrucción:

```
spark-submit.cmd --name Query_example2 --master local[3] --num-executors 5
--executor-cores 4 --driver-memory 4g --executor-memory 10g --packages
com.datastax.spark:spark-cassandra-connector_2.11:2.4.1 ./SparkBlast_DoQuery.py --MQuery
./Query1.txt example2 11 ./Results/Scalability_DoQuery_GRCh38F.res 160 example2
```

Las estadísticas iniciales antes de iniciar el programa, con los datos que se ven en la consulta anterior son:

```
++++++++++++ INITIAL STATISTICS 08/25/2022 20:49:37 ++++++++++++++++++++
+ Reference: example2   Query file: ./Query1.txt.
+ Key Size: 11          Method: 1.
+ Num Executors: None   Executors/cores: None   Executor Mem: None.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Las estadísticas finales después de ejecutar el programa, son las siguientes:

```
# Total time required for processing the query with 7 keys: 590.23 seconds.
++++++++++++++ FINAL STATISTICS ++++++ 25/08/2022 20:59:12 ++++++++++++++
+ Reference: example2        File:./src/main/scala/org/tfm/sparkScala/docs/Query1.txt .
+ Key Size: 11        Method: 1 .
+ Num Executors: 1    Executors/cores: 2    Executor Mem: 2gb.
+ Total Time: 590.234 Data Read Time: 21.000       Data Frame Time: 9.000 .
+ Top Matching Time: 3.000   Alignment Extension Time: 0.000 .
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Esta ejecución ha tardado **590,23 segundos,** un tiempo muy elevado, con los parámetros que vemos en las estadísticas finales mostradas justo en el fragmento de arriba, hay que destacar que la consulta se ha realizado con 7 keys, hecho que influye en el tiempo total, cuantas más keys, más tiempo, porque la dificultad de la consulta aumenta.

A continuación comprobaremos los resultados de la ejecución del código en Scala y el tiempo que tarda. Usaremos la misma instrucción pero cambiando el fichero Python por uno en Scala.

Las estadísticas iniciales antes de iniciar el programa, con los datos que se ven en la consulta anterior son:

```
++++++++++++ INITIAL STATISTICS ++++++++++++++ 25/08/2022 20:44:32 ++++++
+ Reference: example2        Query
file:./src/main/scala/org/tfm/sparkScala/docs/Query1.txt .
+ Key Size: 11        Method: 1 .
+ Num Executors: 1    Executors/cores: 2    Executor Mem: 2gb.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Las estadísticas finales después de ejecutar el programa, son las siguientes:

```
# Total time required for processing the query with 7 keys: 37.000 seconds.
++++++++++++++ FINAL STATISTICS ++++++ 25/08/2022 20:45:12 ++++++++++++
+ Reference: example2        File:./src/main/scala/org/tfm/sparkScala/docs/Query1.txt .
+ Key Size: 11        Method: 1 .
+ Num Executors: 1    Executors/cores: 2    Executor Mem: 2gb.
+ Total Time: 37.000  Data Read Time: 21.000       Data Frame Time: 9.000 .
+ Top Matching Time: 3.000   Alignment Extension Time: 0.000 .
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Esta ejecución ha tardado **37,000 segundos** con los parámetros que vemos en las estadísticas finales mostradas justo en el fragmento de arriba, hay que destacar que la consulta se ha realizado con 7 keys, hecho que influye en el tiempo total, cuantas más keys, más tiempo, porque la dificultad de la consulta aumenta.

Como se puede apreciar, claramente, el código en scala va muchísimo más rápido que el hecho en Python y por lo tanto, se trata de una decisión clara de cuál de los dos códigos es mejor en cuanto a tiempos y rendimiento, porque con los mismos recursos, la consulta de los datos es mucho más rápida en Scala que en Python. Ese hecho, nos permite poder afirmar con firmeza que la migración del proyecto ha sido un cambio satisfactorio. Acto seguido, se llevaran a cabo la comprobación de resultados.

4.3. Comprobación de resultados

En este subapartado, se realizará la comprobación del resultado de la consulta de datos.
Los resultados del código Python son los siguientes:

```
# Python Result
List(55, 71, 0.8235294222831726, X-WVUTSRQPONMLKJI, XXWVUTSRQPONMLK-J)
List(81, 97, 0.8235294222831726, X-WVUTSRQPONMLKJI, XXWVUTSRQPONMLK-J)
```

Los resultados del código Scala son los siguientes:

```
// Scala Result
List(55, 71, 0.8235294222831726, X-WVUTSRQPONMLKJI, XXWVUTSRQPONMLK-J)
List(81, 97, 0.8235294222831726, X-WVUTSRQPONMLKJI, XXWVUTSRQPONMLK-J)
```

Tal como se puede ver en ambos fragmentos los resultados son exactamente idénticos y por lo tanto, se puede afirmar que la migración se ha realizado satisfactoriamente. Al obtener una validación satisfactoria, se llevaran a cabo las conclusiones y el trabajo futuro del proyecto.

Capítulo 5. Conclusiones y trabajo futuro.

Finalmente, podemos afirmar si la migración ha sido un cambio positivo o no, por lo visto en las ejecuciones en local, podemos ver que la migración ha sido un cambio más que positivo, porque nos ha permitido reducir mucho el tiempo de consulta y un poco el tiempo de creación de datos. Lo más importante de todo es que se ha podido aprovechar la afinidad de Scala con el conector para Cassandra ("*datastax"*), para acabar realizando una mejora sustancial en cuanto a tiempos de ejecución, eso si, se debe resaltar que la solución sigue siendo la misma.

En definitiva, la migración realizada ha sido un cambio más que satisfactorio, ya que hemos reducido mucho los tiempos de consulta de los datos, según las validaciones realizadas.

Como trabajo futuro se espera poder realizar pruebas con más recursos y en un entorno no local, por lo cual, se podrá validar mejor tanto el rendimiento, como el tiempo y los recursos empleados. Por otro lado, a parte de eso, se deberá revisar el código tanto Python como Scala, para encontrar posibles mejoras para poder conseguir un rendimiento mayor a causa de una mejor optimización tanto del código como de las prestaciones.

Bibliografía

[1] Blast Algorithm - "Blast Algorithm". (citado 28/04/2021)

[2] Blast Programs and Variants - "Blast Programs and Variants"

[3] Spark - "Spark". (citado 28/04/2021)

[4] Big Data - "Big Data". (citado 28/04/2021)

[5] Cassandra - "Cassandra". (citado 28/04/2021)

[6] F. Cores Prado - Sparky Blast- "Sparky Blast". (citado 28/04/2021)

[7] Scala- "Scala". (citado 28/04/2021)

[8] Cassandra Connector using Scala - "Cassandra connector using scala". (citado 28/04/2021)

[9] Docker - "Docker". (citado 28/04/2021)

[10] Spark Build Examples - "Spark Build Examples". (citado 28/04/2021)

[11] Insert Cassandra DB example Using Spark - "Insert Cassandra DB - Spark". (citado 28/04/2021)

[12] Spark Cassandra dockerized example - "Spark Cassandra Dockerized Example". (citado 28/04/2021)

[13] Spark Blast (High Throughput BLAST Algorithm using Spark and Cassandra) - Fernando Cores, Fernando Guirado y Josep Lluís Lerida. (citado 22/03/2022)

[14] Matsunaga, A., Tsugawa, M., Fortes, J.: Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In: 2008 IEEE Fourth International Conference on eScience, pp. 222–229. IEEE (2008) (citado 22/03/2022)

[15] NCBI BlastN - Nucleotide BlastN

[16] Blast Local Alignment Search Tool - Blastn, Blastp, BlastX (citado 03/04/2022)

[17] Altschul SF, Madden TL, Schäffer AA, et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. 1997;25(17):3389-3402. doi:10.1093/nar/25.17.3389 (citado 03/04/2022)

[18] Washington University Blast - Wu Blast (citado 03/04/2022)

[19] Altschul, S. F. (2001). BLAST algorithm. e LS. (citado 01/06/2022)

[20] Ncbi Blast Search - Explanation of the Database Search (citado 02/06/2022)

[21] Hadoop - Hadoop Relevance (citado 02/06/2022)

[22] Hadoop Map Reduce - Map Reduce (citado 02/06/2022)

[23] Hadoop Distributed File System - HDFS (citado 02/06/2022)

[24] Peer-to-peer Technology - P2P (citado 02/06/2022)

[25] Windows 10 - W10 (citado 19/07/2022)

[26] Docker Desktop Tool - Docker Desktop (citado 19/07/2022)

[27] Repository Custom Docker Scala Cassandra - Github Repo (citado 22/07/2022)

[28] Env Config Repo Docker Scala Cassandra - Github Repo Environment Document (citado 22/07/2022)

[29] Git For Windows - Git bash (citado 22/07/2022)

[30] The CQL shell - CQLSH (citado 22/07/2022)

[31] Gestor de entornos virtuales - Anaconda (citado 28/07/2022)

[32] Gestor de entornos virtuales - Conda Mini (citado 28/07/2022)

[33] Gestor de dependencias - Maven (citado 28/07/2022)

[34] Conector base de datos Cassandra - Datastax CassandraDB (citado 28/07/2022)

[35] Spark Session Information by Apache Spark - Spark Session (citado 28/07/2022)

[36] Big Decimal Java - Big Decimal (citado 28/07/2022)

[37] How to flatten a List of Lists in Scala - Scala List Flatten (citado 28/07/2022)

[38] Check cassandra session - Cassandra session in Scala (citado 20/08/2022)

[39] Stackoverflow schema type unit not supported - Udf wants map not foreach (citado 20/08/2022)

[40] Python default dict - default dict (citado 22/08/2022)

[41] What is Lift in Scala? - Lifting in Scala (citado 22/08/2022)