UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Implementación de un backend serverless de código abierto en ROOT, una aplicación para la computación de física de altas energías

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Oliver Cortés, Pablo

Tutor/a: Alonso Jordá, Pedro

Cotutor/a externo: PADULANO, VINCENZO EDUARDO

CURSO ACADÉMICO: 2021/2022

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Implementación de un backend serverless de código abierto en ROOT, una aplicación para la computación de física de altas energías

TRABAJO FIN DE MÁSTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Pablo Oliver Cortés
Tutores: Vicenzo Eduardo Padulano
Pedro Alonso-Jordá

2021-2022

Resum

El CERN (Centre Europeen pour la Recherce Nucleaire) és el centre d'investigació per a la física d'altes energies (HEP) més gran del món. Ofereix desafiaments informàtics únics a conseqüència de la gran quantitat de dades que genera el Gran Colisionador d'-Hadrons (LHC). El CERN ha concebut i manté un programari denominat ROOT, que és l'estàndard de facto per a l'anàlisi de dades HEP. Aquest framework ofereix una interfície d'alt nivell i fàcil d'utilitzar, denominada *RDataFrame*, que permet gestionar i processar grans conjunts de dades. En els últims anys, aquesta interfície ha vist extendida la seua funcionalitat per a poder aprofitar les capacitats de còmput distribuïdes. Gràcies al seu model de programació declarativa, la API orientada a l'usuari es pot desacoblar del backend d'execució real. Aquest desacoblament permet escalar una anàlisi física a milers de nuclis computacionals, de manera automàtica, sobre diversos tipus de recursos distribuïts. De fet, el mòdul *RDataFrame* distribuït ja admet l'ús de motors establits de la indústria en general, com a Apache Spark o Dask. No obstant l'anterior, aquestes solucions actuals no seran suficients per a complir amb els requisits futurs quant a la quantitat de dades que generaran els nous acceleradors que estan projectats. És d'interés, per aqueix motiu, investigar un enfocament diferent, el que ofereix la computació *serverless*. Basantnos en un primer prototip que utilitza AWS Lambda, aquest treball presenta la creació d'un backend nou per a *RDataFrame* distribuït sobre l'eina OSCAR, un framework de codi obert que suporta la computació *serverless*. La implementació presenta noves formes, respecte al prototip basat en AWS Lambda, de sincronitzar el treball de les funcions.

Paraules clau: CERN, ROOT, OSCAR, Serverless Computing, RDataFrame, AWS Lambda

Resumen

El CERN (Centre Europeen pour la Recherce Nucleaire) es el centro de investigación para la física de altas energías (HEP) más grande del mundo. Ofrece desafíos informáticos únicos como consecuencia de la gran cantidad de datos que genera el Gran Colisionador de Hadrones (LHC). El CERN ha concebido y mantiene un software denominado ROOT, que es el estándar de facto para el análisis de datos HEP. Este framework ofrece una interfaz de alto nivel y fácil de utilizar, denominada *RDataFrame*, que permite gestionar y procesar grandes conjuntos de datos. En los últimos años, esta interfaz ha visto extendida su funcionalidad para poder aprovechar las capacidades de cómputo distribuidas. Gracias a su modelo de programación declarativa, la API orientada al usuario se puede desacoplar del backend de ejecución real. Este desacoplamiento permite escalar un análisis físico a miles de núcleos computacionales, de manera automática, sobre varios tipos de recursos distribuidos. De hecho, el módulo *RDataFrame* distribuido ya admite el uso de motores establecidos de la industria en general, como Apache Spark o Dask. No obstante lo anterior, estas soluciones actuales no van a ser suficientes para cumplir con los requisitos futuros en cuanto a la cantidad de datos que van a generar los nuevos aceleradores que están proyectados. Es de interés, por ese motivo, investigar un enfoque diferente, el que ofrece la computación *serverless*. Basándonos en un primer prototipo que utiliza AWS Lambda, este trabajo presenta la creación de un backend nuevo para *RDataFrame* distribuido sobre la herramienta *OSCAR*, un framework de código abierto

que soporta la computación *serverless*. La implementación presenta nuevas formas, con respecto al prototipo basado en AWS Lambda, de sincronizar el trabajo de las funciones.

Palabras clave: CERN, ROOT, OSCAR, Serverless Computing, RDataFrame, AWS Lambda

Abstract

CERN (Centre Europeen pour la Recherce Nucleaire) is the largest research centre for High Energy Physics (HEP). It offers unique computational challenges as a result of the large amount of data generated by the Large Hadron Collider (LHC). CERN has developed and supports a software called ROOT, which is the de facto standard for HEP data analysis. This framework offers a high-level and easy-to-use interface called *RDataFrame*, which allows managing and processing large data sets. In recent years, its functionality has been extended to take advantage of distributed computing capabilities. Thanks to its declarative programming model, the user-facing API can be decoupled from the actual execution backend. This decoupling allows physical analysis to scale automatically to thousands of computational cores over various types of distributed resources. In fact, the distributed *RDataFrame* module already supports the use of established general industry engines such as Apache Spark or Dask. Notwithstanding the foregoing, these current solutions will not be sufficient to meet future requirements in terms of the amount of data that the new projected accelerators will generate. It is of interest, for this reason, to investigate a different approach, the one offered by serverless computing. Based on a first prototype using AWS Lambda, this work presents the creation of a new backend for *RDataFrame* distributed over the *OSCAR* tool, an open source framework that supports serverless computing. The implementation introduces new ways, relative to the AWS Lambda-based prototype, to synchronize the work of functions.

Key words: CERN, ROOT, OSCAR, Serverless Computing, RDataFrame, AWS Lambda

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

CAPÍTULO 1
Introducción

1.1 Motivación

La Organización Europea para la Investigación Nuclear (CERN) es el centro de investigación para la Física de Altas Energías (HEP) más grande e importante del mundo. Uno de los principales instrumentos necesarios para la investigación y avance en este campo son los aceleradores de partículas, siendo el más grande de ellos el LHC. Este acelerador, construido y operado por el CERN, genera 1 Peta-Byte de datos por segundo cuando está encendido. Estos datos son recopilados por diferentes sensores diseñados para el estudio de diversos fenómenos físicos.

El LHC no funciona de forma constante, la generación de estos datos/eventos funciona por *runs*. Hace unos meses dio comienzo el tercer *run* del LHC, que durará hasta 2025. Además, está en desarrollo/construcción un nuevo acelerador denominado High Luminosity Large Hadron Collider (HL-LHC) [7] para el cual se estima que se necesiten entre 50 y 100 veces más recursos computacionales que los empleados actualmente y es sobre el que se ejecutarán sucesivos *runs* a partir de 2029.

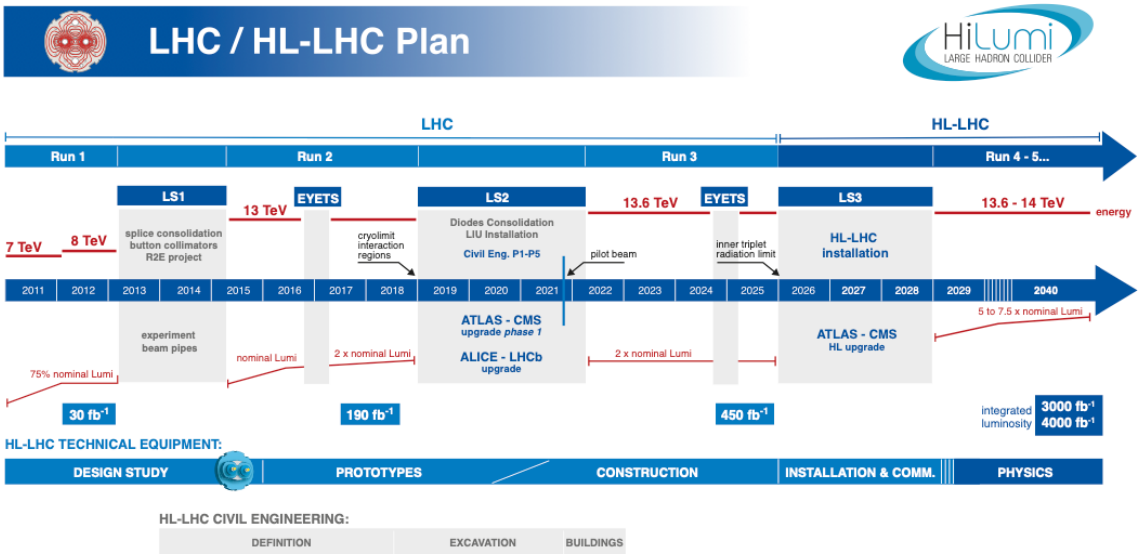


Figura 1.1: Planificación temporal del LHC y HL-LHC. Fuente: [8]

Como se puede apreciar en las Figuras 1.2 y 1.3, dos de los principales experimentos realizados en el LHC han visto sus necesidades computacionales satisfechas mediante aumentos progresivos del presupuesto sumado a los avances tecnológicos generacionales que recibe el hardware año a año. Sin embargo, una vez se ponga en marcha el HL-LHC

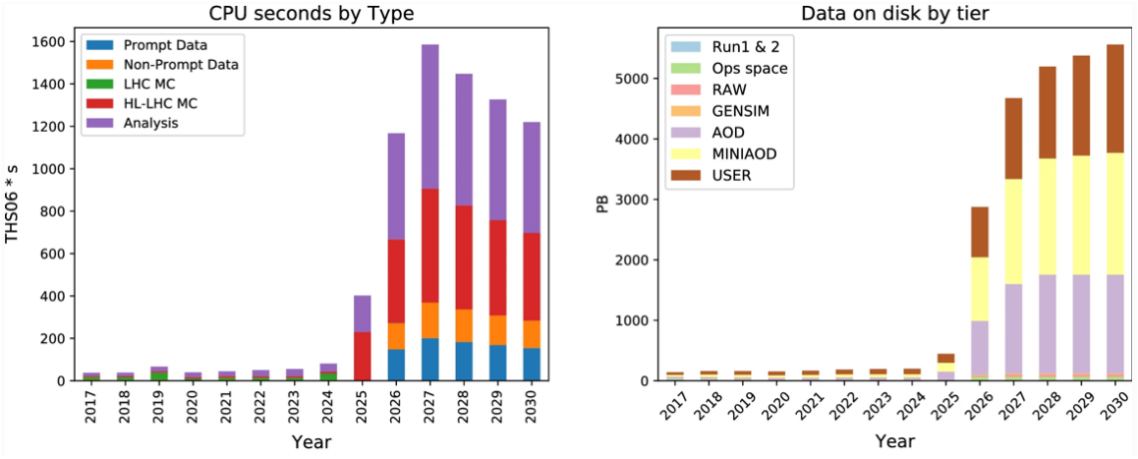
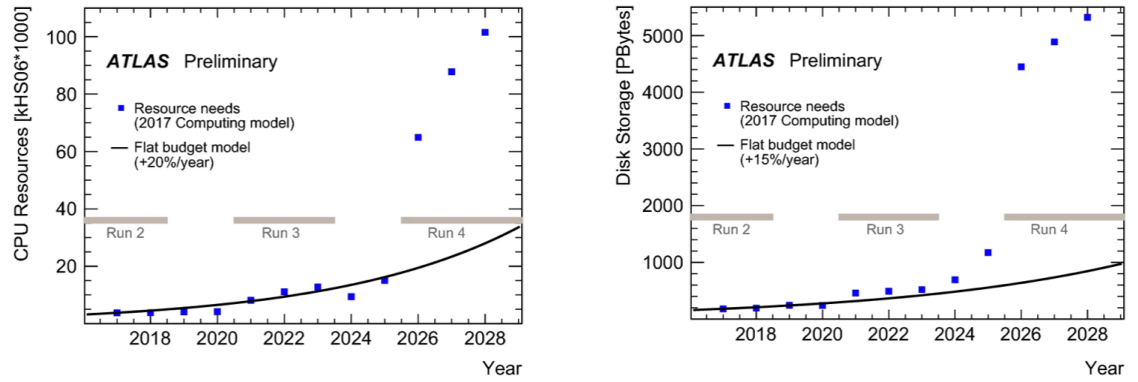


Figura 1.2: Estimación de requisitos computacionales y de almacenamiento del experimento CMS para el HL-LHC[1]. Fuente: [2]



(a) Requisitos computacionales

(b) Requisitos de almacenamiento

Figura 1.3: Estimación de requisitos computacionales y de almacenamiento del experimento ATLAS para el HL-LHC[1]. Fuente: [2]

estas capacidades de cómputo se quedarán muy por detrás de los requisitos necesarios. Concretamente, en la Figura 1.3 podemos ver cómo, si proyectamos el modelo presupuestario de los últimos años para mejorar las capacidades de cómputo del experimento, a partir de 2025 estas se quedarían muy por detrás de las necesidades reales.

Desde la concepción del LHC, los requisitos computacionales que iban a ser necesarios para procesar las ingentes cantidades de datos generados suponían un reto que el CERN no podría gestionar por si mismo. Estos gastos computacionales ni siquiera habían sido tenidos en cuenta en los costes originales del LHC [21]. Para dar solución a este problema, desde 2001, se trabajó en el diseño de un *grid*, el *WLCG* (Worldwide LHC Computing Grid), en una colaboración entre el CERN y diversas instituciones académica. Las primeras pruebas se realizaron en 2003, y en 2006 ya estaría operativo y listo para procesar los datos una vez el LHC se pusiese en marcha. Desde su puesta en marcha hasta el día de hoy, el *WLCG* sigue siendo el *grid* más grande del que disponen los físicos para realizar sus análisis, tanto para los experimentos realizados en el CERN como para los experimentos realizados en otros aceleradores a lo largo del globo.

Dentro del WLCG, el modelo de recursos computacionales se divide en cuatro capas [9]. La capa 0 corresponde al centro de datos del propio CERN, dedicado principal-

[1]La unidad HS06 hace referencia a la puntuación obtenida por un sistema en el benchmark HEP-SPEC06 centrado en analizar el rendimiento de los sistemas para tareas similares a las que realizan los análisis HEP [22].

mente al almacenamiento de los datos recogidos por los sensores, a la reconstrucción de dichos datos y a dar salida a estos datos hacia la capa 1; esta capa supone al rededor de un 20 % de la capacidad de cómputo del *grid*. La capa 1 está compuesta por trece centros con suficiente capacidad de almacenamiento y disponibilidad de 24 horas, se encargan de almacenar los datos en crudo y reconstruidos, así como datos de simulaciones producidos en la capa 2, computacionalmente hacen tareas de reprocesado a gran escala. La capa 2 está compuesta principalmente por universidades e institutos científicos capaces de proporcionar poder computacional suficiente para determinadas tareas. Actualmente está compuesto por 160 centros distribuidos por todo el mundo. Por último, la capa 3 corresponde a recursos de cómputo locales, ya sea un departamento universitario o un ordenador personal, no existe una relación formal entre esta capa con el resto del *grid*.

Por un lado, la capa 0 se encarga de tareas que requieren de un hardware concreto para extraer todas las mediciones generadas por los sensores, almacenándolas a un ritmo de Peta-Bytes por segundo e imposibilitando el uso de hardware externo en esta primera capa. Por otro lado, las tres capas restantes podrían verse beneficiadas de las economías de escala proporcionadas por el *Cloud Computing* público, si se provee de herramientas que no requieran inversiones adicionales de recursos, algo que ya se mencionaba en un informe técnico de 2017 [4], tanto para el procesado de estos datos, como para el almacenamiento.

Dentro de estas necesidades, se definen múltiples frentes en los que se hace necesario realizar un esfuerzo de I+D [2, 4, 42] con el fin de mejorar todo el ecosistema HEP, y que comprenden, tanto la obtención y el procesado/análisis de los datos como la facilidad de uso de estas herramientas para el usuario final. Estos esfuerzos abarcan, desde la eficiencia, escalabilidad y rendimiento del software utilizado actualmente para aprovechar nuevas arquitecturas, hasta la adopción de nuevos paradigmas, como pueden ser el uso de la Inteligencia Artificial (IA) en ciertos aspectos de HEP o el uso de modelos *Big Data* para el análisis de los datos.

A pesar de los avances en estas áreas, los factores más limitantes para poder aprovechar todos los recursos disponibles siguen siendo la Entrada/Salida (I/O) y la red. Esto se debe a que los eventos físicos generados por el LHC son estadísticamente independientes, generando cargas de trabajo *embarrasingly parallel*. Esta característica de los datos a procesar debería implicar que si la implementación paralela es óptima el Speedup también lo debería ser. Pero, debido a las ingentes cantidades de datos que se generan, los cuales pueden no estar ubicados cerca de los recursos de cómputo, y a la reutilización de los mismo en distintos análisis hacen que el camino hacia esa escalabilidad ideal no sea sencillo.

Es por todo esto que se hace necesaria la investigación y el desarrollo de herramientas que faciliten el uso de recursos computacionales a escala, tanto para el propio CERN como para actores externos involucrados en el estudio de estos eventos, ya sea para su cómputo en clústeres on-premises, en *clouds* públicos o en sistemas federados de recursos, como pueda ser la Fundación EGI, con la cual colabora el *WLCG* desde hace más de diez años [27].

Dentro de la parte correspondiente al análisis de los datos, en los últimos años el CERN ha estado expandiendo la funcionalidad de *ROOT* (el principal software para análisis de datos HEP), tanto para simplificar su uso mediante una interfaz en Python que enmascara el código C++, como proporcionando nuevas estructuras de datos que simplifican el desarrollo de los análisis, esto es, *RDataFrame*. Además de extender todo lo anterior, simplificando también la computación distribuida mediante diversos *backends*, como puedan ser Apache Spark o Dask.

1.2 Objetivos

El principal objetivo de este trabajo se centra en extender *Distributed RDataFframe* [32] con un nuevo *backend* que permita la utilización de *OSCAR* [35], una solución *serverless* desarrollada por el grupo de investigación GRyCAP[1], centrada en computación HTC en base a eventos y que está integrada con la herramienta Infraestructure Manager [17] permitiendo la escalabilidad de la solución en múltiples *clouds* públicos. Este objetivo se ha llevado a cabo mediante la realización en base a los pasos siguientes:

1. Afianzar el conocimiento sobre computación distribuida, computación en la nube y el paradigma *serverless* centrado en el paradigma *event-driven* empleado por la herramienta *OSCAR*.
2. Analizar y comprender las herramientas necesarias para el análisis de datos HEP distribuido utilizando el software *ROOT*. En este caso concreto, nos centraremos en *RDataFrame* y *Distributed RDataFrame*.
3. Estudiar como se podría integrar el modelo de computación distribuida empleado por *Distributed RDataFrame* con el modelo *event-driven* ofrecido por *OSCAR*.
4. Implementación del *backend* para *OSCAR* para el procesado distribuido de los análisis de datos HEP con *ROOT*.
5. Estudio del rendimiento ofrecido por la/s implementación/es y, si procede, realizar propuestas de posibles mejoras.

1.3 Estructura de la memoria

La memoria de este trabajo de fin de máster está estructurada de la siguiente forma:

- *Capítulo 2. Herramientas y Estado del Arte.* En este capítulo se explican las principales herramientas que se utilizan durante el resto del documento así como se habla brevemente de la evolución de estas y su estado de desarrollo actual.
- *Capítulo 3. Implementación del backend* Sentadas las bases de las herramientas a utilizar, se estudian y describen diversas opciones de implementación.
- *Capítulo 4. Análisis de rendimiento.* Se estudian las prestaciones de las distintas arquitecturas propuestas y se valoran las ventajas y desventajas de cada una de ellas.
- *Capítulo 5. Conclusiones y Trabajo Futuro.* Para finalizar, se discute sobre las contribuciones principales de este trabajo y direcciones sobre las que seguir trabajando.

[1] Grupo de investigación perteneciente a la UPV centrando en la computación grid y cloud https://www.grycap.upv.es

CAPÍTULO 2
Herramientas y Estado del Arte

En este capítulo se detalla el funcionamiento de las principales herramientas que se han utilizado durante el desarrollo del trabajo junto con sus características, con el fin de proporcionar un marco de trabajo que sirva para justificar las decisiones de diseño del *backend* propuestas. Junto con la descripción de estas herramientas, se ofrece un vistazo al estado del arte de los análisis de física de altas energías y a las aplicaciones *serverless*.

2.1 ROOT

ROOT [5] es un software desarrollado en el CERN cuyo objetivo es el análisis de datos de física de altas energías. Este software se ha ido actualizando desde su concepción inicial en 1994, ampliando su funcionalidad y adaptándose a nuevas tecnologías. Está compuesto por diversos *frameworks* Orientados a Objetos que permiten el manejo y análisis de grandes volúmenes de datos de forma eficiente.

Por lo general, los datos son almacenados en un formato binario comprimido llamado *fichero ROOT* o *TFile*. Este *TFile* permite escribir en él cualquier tipo de objeto C++. En cuanto al acceso de los datos, gracias al uso de objetos, por ejemplo, mediante el uso de la clase *TTree* se pueden manipular múltiples *ficheros ROOT*, tanto locales como remotos, encadenándolos y mostrándolos como un único objeto. *TTree* es una interfaz de bajo de nivel de E/S de *ROOT* usada por otras clases.

También se incluyen funcionalidades para el análisis estadístico y la minería de datos. Además, proporciona herramientas para la visualización e impresión en múltiples formatos de los gráficos generados a partir de los análisis. Para facilitar la creación de los análisis se desarrolló un intérprete de C++ integrado en *ROOT*, Cling [45], que permite realizar los análisis de forma interactiva agilizándolos y evitando el costoso proceso de compilación y enlazado que conlleva el uso de lenguajes compilados. De forma adicional. *ROOT* también ofrece integración con otros lenguajes como Python o R, que se utilizan simplemente como interfaces de alto nivel que por debajo utilizando las librerías C++ de *ROOT*.

El ecosistema de *ROOT* no se queda solamente en el propio software, también existen herramientas desarrolladas sobre este. Por ejemplo, una de las herramientas más utilizadas es *SWAN* [10] (Service for Web based ANalysis), una plataforma online para realizar análisis de datos de forma interactiva construida sobre *ROOT*.

Todo ello construye una herramienta que es utilizada por miles de físicos a diario y no solo eso, si no que también se utiliza en otros ámbitos fuera de HEP.

Esta herramienta utiliza el paralelismo a diferentes niveles. En sus inicios, *ROOT* no contaba con ningún tipo de paralelismo, quedando relegado al usuario si este quería implementarlo de alguna forma. La principal forma de trabajo se centraba, y se sigue centrando, en dividir los datos de entrada y enviar múltiples trabajos a sistemas de colas como puedan ser HTCondor [30] o SLURM [1]. A lo largo de los años se ha ido extendiendo la funcionalidad de *ROOT* y recientemente se ha añadido soporte para el uso de multithreading. Este uso de multithreading está implementado utilizando Intel Threading Building Blocks [23] y gracias a *RDataFrame* se puede utilizar de forma transparente para el usuario. También se han desarrollado capacidades para la computación multiproceso, el desarrollo de esta tecnología es interna del CERN y sigue un modelo similar al modelo de paso de mensajes, aunque este modelo de programación no es muy utilizado.

A pesar de estos avances, tras lo expuesto en el capítulo anterior, se hace necesario seguir explorando nuevas vías para expandir y facilitar la escalabilidad de los análisis realizados con *ROOT*.

2.1.1. RDataFrame

Recientemente, ROOT, en su versión 6.14 lanzada en 2018 [36], introdujo una nueva forma de trabajar con los datos generados por los distintos experimentos en el CERN denominada *RDataFrame* [33], que simplifica el manejo de los datos exponiendo una API declarativa al estilo de Apache Spark [46] o de la librería de Python *pandas* [28]. Antes de la existencia de *RDataFrame* todas las estructuras de datos requerían de un modelo de programación imperativo. En este modelo imperativo es necesario que el usuario defina tanto qué operaciones han de llevarse a cabo como la forma en la que han de realizarse esas operaciones. Con la API declarativa que ofrece *RDataFrame* solamente se han de indicar qué operaciones se requieren para el análisis y *RDataFrame* se encargará de determinar la mejor forma de realizar las operaciones definidas por el usuario. Esta nueva interfaz mantiene toda la funcionalidad y eficiencia de *ROOT* para este tipo de computaciones.

Antes de la existencia de *RDataFrame* una de las estructuras de datos más utilizadas era, y sigue siendo, los *TTree*. Un árbol, *TTree*, consiste en una lista de columnas independientes llamadas ramas, *branches*, representadas por la clase *TBranch*. En un *TTree* la única información en memoria es una única entrada del árbol mientras que el resto reside en el sistema de almacenamiento, para trabajar sobre un *TTree* se va iterando sobre las entradas, idealmente de forma secuencial. Las columnas son independientes y pueden contener cualquier tipo de dato y a la hora de trabajar con un *TTree* se pueden seleccionar solamente algunas de ellas. Mientras que la clase *TBranch* representa la estructura, los datos son accedidos mediante las hojas, *TLeaf*, de la rama.

Todas las ramas y las hojas almacenan los datos de sus entradas en buffers de un tamaño determinado. Cuando un buffer se llena, se comprime. Este buffer comprimido se denomina *basket*, los cuales son escritos al *fichero ROOT*. Dependiendo de la información que se almacene en cada entrada las ramas almacenarán más o menos *baskets*, pudiendo contener una única entrada, o varias.

Estos *baskets* se agrupan en *clusters* para proporcionar un acceso eficiente y balanceado de las entradas del *TTree*. Para ello, un *cluster* contiene todos los datos de un rango de entradas determinado. Los árboles pueden cerrar los *baskets* antes de que estén llenos si las entradas alcanzan los límites del *cluster*.

Para que un usuario pueda trabajar con *TTrees* necesita conocer toda esta complejidad de la estructura de datos subyacente y trabajar sobre ella para crear, modificar o analizar

los datos almacenados en el *TTree*. *RDataFrame* también simplifica este proceso para el usuario, ocultándole gran parte de esta complejidad a la hora de trabajar.

En los ejemplos de Código 2.1 y 2.2 se pueden ver las diferencias entre el uso de *TTree*, una estructura de datos que opera bajo el modelo imperativo, frente a *RDataFrame* y cómo *RDataFrame* simplifica el proceso para la realización de un análisis sobre los eventos almacenados en un fichero ROOT utilizando tres variables del mismo, en este caso las variables son (A, B, C). El análisis es sencillo y consiste en seleccionar tres variables de los eventos, aplicarles un filtro, IsGoodEvent, y realizar alguna operación sobre los eventos que pasen el filtro. Cuando se habla de filtro se hace referencia a seleccionar únicamente aquellos eventos físicos (o registros si queremos centrarnos en una terminología puramente de datos) que cumplen una o varias condiciones definidas por el usuario, como pueda ser, que el valor de una variable del evento se encuentre por encima de una valor determinado, mediante condiciones ya definidas en *ROOT*, o la selección de un subconjunto de los eventos en base a un rango, por ejemplo, los cien primeros eventos de una fuente de datos.

```
1 TTreeReader reader("myTree", file);
2 TTreeReaderValue<A_t> a(reader, "A");
3 TTreeReaderValue<B_t> b(reader, "B");
4 TTreeReaderValue<C_t> c(reader, "C");
5 while(reader.Next()) {
6    if(IsGoodEvent(*a, *b, *c))
7       DoStuff(*a, *b, *c);
8 }
```

Código 2.1: TTreeReader

```
1 ROOT::RDataFrame d("myTree", file,
      {"A", "B", "C"});
2 d.Filter(IsGoodEvent).Foreach(
      DoStuff);
```

Código 2.2: RDataFrame

El flujo de trabajo en *RDataFrame* se puede resumir en los siguientes pasos:

1. Se especifican las fuentes de los datos, ya sean locales o en remoto.
2. Se definen una serie de operaciones sobre estos datos, como pueden ser filtrados o definiciones de nuevas variables.
3. Se define cómo agregar el resultado de estas transformaciones para generar los resultados deseados, ya sea un gráfico o cualquier otra estructura de datos deseada.

En la Figura 2.1 se muestra un ejemplo sencillo de código junto con el grafo de computación interna que contiene la lógica asociada al análisis. Cabe mencionar que, *RDataFrame* realiza una aproximación "perezosa" para los análisis, por lo tanto, durante el proceso de definición del análisis, no se realizará ninguna operación hasta que se incluya un nodo en el grafo que genere algún tipo de salida, ya sea gráfica o de otro tipo. En la figura serían los nodos azules *TGraph* y *TH1D* y en el código se correspondería con las líneas 10 y 11 que son las que piden obtener el contenido de esas variables.

Esto simplifica la forma de trabajar tradicional de la computación *grid* para HEP en la que un usuario tenía que programar, tanto cómo se procesaban las distintas fuentes de datos como los procesos de reducción, dejando al usuario solamente la tarea de definir que operaciones se han de realizar sobre los datos.

Como se comentaba anteriormente, los datos generados por los eventos físicos observados en los distintos experimentos son estadísticamente independientes, por lo que *RDataFrame* es capaz de abstraerse de las distintas estructuras de datos subyacentes en la que estos eventos se encuentren almacenados, y proporcionar una interfaz de alto nivel en la que son interpretados como un conjunto de datos homogéneo *columnar*.

La Figura 2.2 muestra una representación de cómo *RDataFrame*, de forma lógica, agrega diversas fuentes de datos mostrándolo al usuario como un único objeto, sobre los que

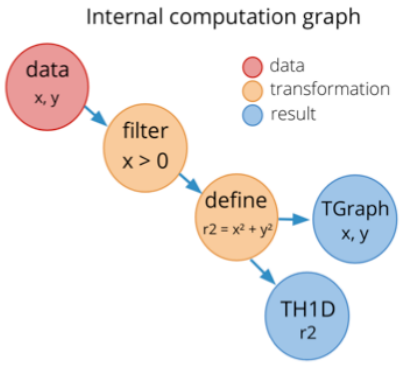
```python
from ROOT import RDataFrame

df  = RDataFrame(dataset)
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y")

rHist = df2.Histo1D("r2")
g = df2.Graph("x", "y")

rHist.GetValue()
g.GetValue()
```

(a) Código para el análisis en RDataFrame.



(b) Grafo computacional generado por el código.

Figura 2.1: Ejemplo de análisis sencillo utilizando *RDataFrame*. Fuente: [32]

realizar operaciones como definir nuevas variables o realizar operaciones de filtrado. Estas fuentes de datos pueden ser estructuras de datos de *ROOT*, CSVs o datos almacenados en una base de datos relacional, entre otros. En la figura se define una nueva variable llamada *myvar* que sería la columna de color rosa en la que indica *Define*. La línea discontinua y las tijeras representan una operación de filtrado por rango, descartando todas las entradas del *RDataFrame* que queden por debajo de la línea. Una vez realizadas las modificaciones pertinentes al *RDataFrame* se pasaría a generar los resultados requeridos por el usuario, como puedan ser histogramas o informes.

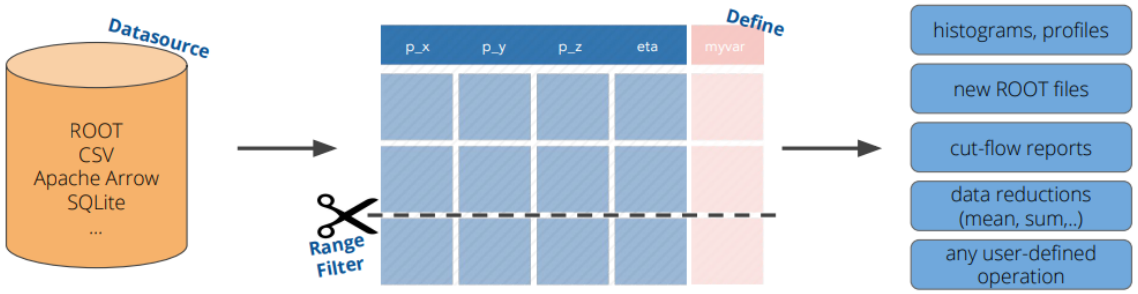


Figura 2.2: Abstracción de *RDataFrame*. Fuente: [33]

Una última característica de *RDataFrame* que cabe destacar es el paralelismo implícito que ofrece, facilitando el máximo aprovechamiento de todos los recursos disponibles de la máquina en la que se vaya a realizar el análisis. Este paralelismo se aprovecha de la independencia entre los eventos y se centra en asignar a cada hilo hardware disponible un subconjunto de los datos para repartir la carga de la forma más homogénea posible, actualmente el *scheduler* interno para este reparto de carga queda delegado en Intel Threading Building Blocks.

Para la distribución de la carga se realiza una lectura inicial de todos las fuentes especificadas en la construcción del *RDataFrame*, se contabiliza el número de *clusters* en cada una de ellas y estos *clusters* son los que se van asignando a los hilos hardware disponibles.

A nivel técnico *RDataFrame* se implementa como un clase C++ que encapsula todas las funcionalidades mencionadas.

2.1.2. Distributed RDataFrame

RDataFrame proporciona paralelismo implícito en los análisis realizados sobre este, facilitando el uso de todos los recursos de procesado de una sola máquina. Sin embargo,

carece de las capacidades necesarias para poder repartir la carga a múltiples máquinas ya sean clústeres tradicionales disponibles para el usuario o el uso de computación *cloud* ya sea público u *on-premises*. Para solucionar esto, se desarrolló *Distributed RDataFrame* [32] (*DistRDF* de ahora en adelante) con el fin de proporcionar al usuario, de forma sencilla, la capacidad de realizar sus análisis de forma distribuida sin apenas modificar cómo se realizan los análisis con *RDataFrame*. En los códigos 2.3 y 2.4 se pueden ver cuales serían los cambios necesarios para migrar un análisis realizado en *RDataFrame* a *DistRDF* utilizando como backend distribuido Dask [37].

```
if __name__ == "__main__":
    # Import ROOT.
    import ROOT

    # Initialize a RDataFrame object with 1024 events.
    df = ROOT.RDataFrame(1024)

    #Work on the RDataFrame
    h = df.Define("x", "gRandom->Rndm()").Histo1D("x")

    h.Draw()
```

Código 2.3: Iniciaclización de RDataFrame.

```
if __name__ == "__main__":
    # Import ROOT and required libraries.
    from dask.distributed import LocalCluster, Client
    import ROOT

    # Configure access to remote resources.
    RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame

    dask_client = Client(LocalCluster(n_workers=4,
                                      threads_per_wroker=1,
                                      processes=true))

    # Initialize a Distributed RDataFrame object with 1024 events.
    df = RDataFrame(1024, dask_client)

    # Continue working on the RDataFrame object as if it was a normal
        RDataFrame object.
    h = df.Define("x", "gRandom->Rndm()").Histo1D("x")

    h.Draw()
```

Código 2.4: Inicialización de RDataFrame distribuido con Dask.

Como se puede apreciar, para que un usuario pueda usar estas capacidades distribuidas solo necesita añadir algunas líneas al código necesarias para permitir la comunicación con los recursos remotos a los que quiera acceder. Para el ejemplo mostrado se solicita que el clúster distribuido de Dask se cree en la propia máquina bajo demanda, pero podrían ser solamente las credenciales necesarias para acceder al clúster que se quiera utilizar.

En *DistRDF*, la distribución de la carga se realiza utilizando el esquema MapReduce [11], que, de nuevo, es transparente para el usuario. *DistRDF* se encargará de ir combinando todas las operaciones definidas por el usuario que no generen un resultado final, incluyéndolas en el grafo de computaciones para posteriormente transformarlas en una función de nivel superior que será aplicada a los eventos físicos que sean necesarios. Cuando en el análisis se defina una operación que sea de salida, esta será análoga a la fase de reducción del modelo MapReduce, y todas las operaciones anteriores a la de salida se corresponderían con la fase de mapeo, todo ello gestionado por *DistRDF*.

Al igual que con *RDataFrame*, *DistRDF* también sigue ese enfoque "perezoso" y las tareas no se reparten por el clúster hasta que se definan operaciones de salida en el análisis.

En la Figura 2.3 se muestra un grafo de computación en el que se puede visualizar este proceso de MapReduce. Los nodos naranjas se comprimirían en una única función que se aplicaría a cada evento y los nodos azules serán los correspondiente a la parte de agregación de los resultados generados por la fase de mapeo.

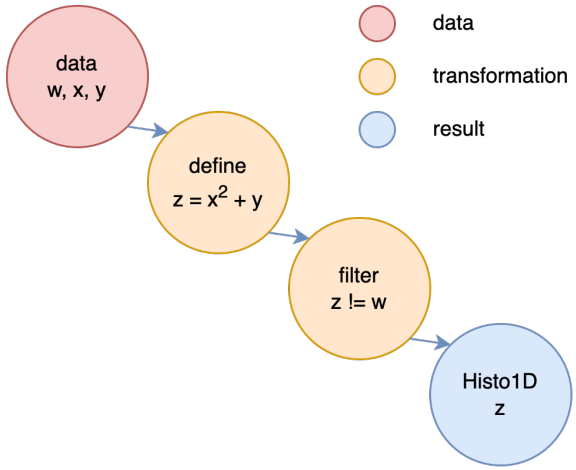


Figura 2.3: Grafo computacional de un análisis.

Un aspecto importante a considerar es cómo se dividen estos datos para repartirlos entre todos los *cores* del clúster en el que se vayan a procesar. La única información de la que dispone *RDataFrame* para realizar esta división de trabajo es el número de ficheros que el usuario ha indicado para realizar el análisis. De forma general, para la generación de tareas se utiliza un número determinado particiones. Este número puede ser proporcionado por el usuario o, en su defecto, se decide en base a un comportamiento por defecto definido en *DistRDF*, el cual trata de obtener el número de *cores* del clúster. Se crean tantas "tareas aproximadas" como número de particiones se hayan indicado, las tareas son aproximadas pues los ficheros únicamente son accedidos por los trabajadores/*cores* del clúster, nunca por el cliente.

Las tareas son enviadas al clúster y cada trabajador/*core* leerá la tarea que le ha sido asignada, abrirá el fichero correspondiente y leerá de los metadatos el número de entradas de ese fichero. Posteriormente, convierte la aproximación proporcionada en la tarea en el rango de entradas real a procesar.

2.1.3. Backends disponibles para Distributed RDataFrame

DistRDF ya cuenta con algunos backends implementados que han sido utilizados como referencia para el desarrollo de este trabajo. A día de hoy, *RDataFrame* ofrece soporte para dos backends distribuidos:

1. Apache Spark [46]. Un framework cuyo objetivo es el procesado de datos a gran escala que emplea MapReduce. Se puede montar sobre clústeres Apache Hadoop ya existentes o por sí solo, y la principal ventaja sobre Hadoop es que el almacenamiento de datos intermedios se realiza en memoria en vez de disco proporcionando una velocidad muy superior a Hadoop.

2. Dask [37]. Una librería de Python para computación paralela, formada por dos elementos principales: un planificador de tareas dinámico optimizado para cargas de

trabajo interactivas y el soporte a estructuras de datos *Big Data*. Además, también ofrece capacidades para la computación distribuida en Python.

También existe una implementación *serverless* que hace uso de *AWS Lambda* [26]. En esta implementación el *backend* se divide en dos partes. Por un lado la parte del cliente, que se encarga de establecer las conexiones pertinentes con *AWS* y de lanzar las funciones *lambda* que se requiera. Por otro lado, la parte "*worker*" encargada del despliegue y desmantelamiento de la infraestructura. Esta parte "*worker*" está construida sobre *Terraform* [20] para permitir, en un futuro, poder utilizar otros proveedores *cloud* de forma transparente.

En esta aproximación, el principal modo de trabajar reside en un enfoque síncrono en el que el lado cliente invoca las funciones *lambda* de forma síncrona, para ello se lanzan tantos hilos como funciones se quieran invocar. El motivo por el que se utiliza este tipo de invocación es para pode controlar el estado de las funciones *lambda* y en caso de que devuelva un error, reintentar la ejecución de la función.

En estas invocaciones se envía toda la información necesaria a cada función: la función de mapeo, el rango de los datos sobre los que esa función ha de trabajar y otros elementos. El cliente, una vez ha invocado todas las funciones, se queda a la espera de que estas invocaciones terminen, consultando el estado de los hilos utilizados para lanzar las funciones. Una vez todas las funciones han terminado de forma satisfactoria se procede a realizar la reducción de de los resultados de forma local.

Este *backend* sirve como punto de partida de este trabajo de fin de máster, el cuál se centra en aportar una implementación en un nuevo backend, *OSCAR*, y, además, aporta diversas mejoras al modelo *serverless* presentado para *AWS Lambda* entre las que se encuentran el desacople total del lado cliente con la arquitectura *serverless* evitando esa necesidad de llamadas síncronas a las funciones o la realización de la fase de reducción también en remoto.

Aunque estas mejoras se plantean y desarrollan para *OSCAR*, podrían ser implementadas en otros entornos *serverless*.

2.2 OSCAR

Desde su llegada a los grandes proveedores *cloud*, inicialmente por parte de Amazon Web Services en 2014 [41], el modelo de computación *Function as a Service*, o *FaaS*, no ha hecho más que ganar popularidad debido a la flexibilidad que ofrece. Este modelo abstrae al usuario de gran parte de costes asociados a la gestión de la infraestructura, ya sea *on-premises* o en el *cloud* y sus configuraciones. El usuario solamente necesita proveer el código de las funciones que serán ejecutadas sin necesidad de pre-aprovisionar explícitamente ningún tipo de infraestructura por parte del usuario. Para este tipo de servicios la unidad de escalado es la función.

Una de las principales características de este modelo es que han de ser *stateless* (aunque desde hace unos años Amazon Web Services ofrece la posibilidad de que las funciones *AWS Lambda* tengan un sistema de ficheros compartido y persistente [3]), lo que permite un alto nivel de paralelismo. Esto quiere decir que durante la ejecución de una función, estas no pueden depender de forma interna de la memoria, discos montados o elementos similares que en otros modelos proporcione información del estado del sistema distribuido, dependiendo única y exclusivamente de la información del evento que las ha invocado. Es decir, las ejecuciones de las funciones han de ser independientes entre sí, algo que se adapta muy bien a los análisis de datos HEP explicado anteriormente. De

esta forma se consigue que la escalabilidad de un sistema distribuido que pueda aprovechar este modelo de computación sea, teóricamente, óptima.

Otra de las características de este modelo, fundamental para este trabajo, es que las funciones pueden ejecutarse dentro de un contenedor. Esto da una flexibilidad máxima en cuanto al entorno de ejecución que necesiten las funciones. Por ejemplo, para los análisis con *ROOT*, se puede disponer de forma sencilla de un entorno con *ROOT* ya compilado y de cualquier otra librería que sean necesarias.

Las funciones por si solas no tienen mucha utilidad, y necesitan integrarse con otros servicios para poder crear lo que se conoce como computación *serverless*. Esta integración puede realizarse con sistemas que generen eventos, siendo estos eventos los que lancen las funciones a ejecutar. También pueden integrarse con bases de datos o sistemas de almacenamiento basados en objetos, como puedan ser *AWS S3* [40], o *MINIO* [29] para extraer los datos a procesar o almacenar los resultados.

Todo lo anterior nos lleva a *OSCAR* (*Open Source Computing for Data-Processing Applications*), una solución *open source* que proporciona un entorno para la computación *serverless* auto-escalable. *OSCAR* se enfoca en el procesamiento de ficheros dando soporte a un modelo de *High Throughput Computing*. La ejecución de las funciones se realiza mediante el uso de contenedores Docker [12].

A parte de *OSCAR*, existen otros *frameworks* para la utilización del modelo *serverless* enfocados en computación distribuida. Estos *frameworks* suelen estar diseñados para utilizar, principalmente, la infraestructura de *clouds* públicos.

Uno de los primeros *frameworks* que apareció fue *PyWren* [24] que ofrece funcionalidad para computación distribuida en Python de forma arbitraria a *AWS Lambda* permitiendo escalarlos de forma masiva, aunque está descontinuado desde 2017. Esta implementación inicial fue extendida en *numpywren* [43] para dar soporte a la computación distribuida de problemas de álgebra lineal *serverless* en *AWS Lambda*, junto con el desarrollo de un lenguaje, *LAmbdaPACK*, diseñado para implementar algoritmos de álgebra lineal altamente paralelos en entornos *serverless*.

Una alternativa reciente a *numpywren* es *Wukong* [6] un *framework* que permite la ejecución de trabajos divididos en tareas que puedan formar grafos acíclico dirigido complejos mediante un planificador descentralizado que mejora considerablemente las prestaciones de *numpywren*.

Otros trabajos en esta línea de investigación son *MARLA* [14] (MApReduce on AWS Lambda) y *SCAR* (Serverless Container-aware ARchitectures).

MARLA es un *framework* que da soporte al modelo MapReduce en *AWS Lambda* para Python, una de las ventajas de *MARLA* frente a otros *frameworks* similares es que se encarga de gestionar todo el proceso MapReduce, desde el particionado de los datos hasta la generación del resultado final donde el usuario solamente ha de definir las funciones Map y Reduce del proceso.

SCAR [34] es un *framework* que ofrece la posibilidad de ejecutar cualquier lenguaje de programación en *AWS Lambda* mediante el uso de contenedores permitiendo la ejecución de cualquier tipo de aplicación en un entorno *FaaS*, que dio soporte a este modelo de ejecución antes que la propia *AWS*.

2.2.1. Arquitectura de OSCAR

OSCAR se presenta como un clúster de Kubernetes [15] elástico que se despliega mediante el uso de las siguientes herramientas: EC3 [18] para proporcionar elasticidad horizontal del clúster; Infrastructure Manager, IM [17], para dar soporte al despliegue mul-

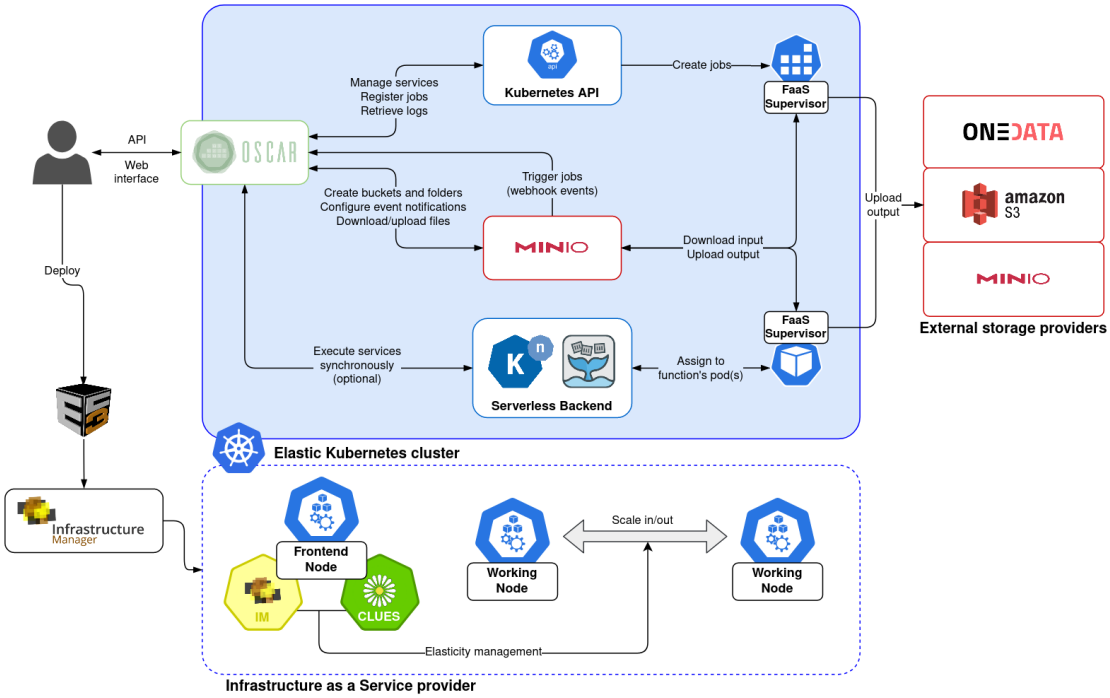


Figura 2.4: Arquitectura de *OSCAR*.

ticloud y CLUES [19] para gestionar la elasticidad del clúster encargándose del *scale in* y *scale out* de los nodos del clúster en base a la demanda de trabajos. En la Figura 2.4 se puede ver cómo interactúan estos servicios para el despliegue y escalado del clúster.

2.2.2. Componentes Principales

Por otro lado, se describen los componentes que forman *OSCAR* en sí, que definen cómo funciona internamente y son los que nos interesan, pues determinarán qué podemos y qué no podemos hacer a la hora de implementar el *backend* para *ROOT*.

- *OSCAR Manager*. El núcleo de *OSCAR*, encargado de gestionar los servicios y la interconexión con el resto de componentes. Proporciona una API REST a través de la cual crear los servicios. Este tipo de APIs se caracteriza por la carencia de estado en la comunicación entre cliente y servidor que se realiza utilizando el protocolo HTTP, mediante la cual, un cliente, con las credenciales necesarias, puede consultar los servicios disponibles así como crear, modificar o borrar servicios para los que esté autorizado. *OSCAR* también proporciona otras formas de interacción como una interfaz web y una herramienta de línea de comandos, OSCAR-CLI[1] .

- *Serverles Backend*. *OSCAR* ofrece dos posibilidades para la implementación de la parte *FaaS* del clúster Kubernetes, estas son, OpenFaaS [13] y Knative [25]. Que dan soporte para llamadas síncronas a los servicios e invocación mediante llamadas a una API REST. Para este trabajo no utilizaremos estas funcionalidades y nos centraremos en la funcionalidad por defecto de subida de ficheros que invocan a los servicios de forma asíncrona.

- *Servicio de almacenamiento interno*. El clúster utiliza un sistema de almacenamiento de objetos denominado *MINIO*. El funcionamiento de este tipo de sistemas se basa

[1] https://github.com/grycap/oscar-cli/.

en la creación de *buckets*, o *depósitos* en castellano, en los que se suben ficheros de cualquier tipo.

Los ficheros que se suben a los sistemas de objetos no tienen jerarquías de directorios como los sistemas de ficheros tradicionales, todo el almacenamiento se realiza en el mismo nivel, aunque, se suelen usar prefijos para simular una estructura de directorios y mantener cierto orden dentro de un *bucket*. El término directorio o carpeta se utiliza como concepto a efectos prácticos [39].

Estos *buckets* tiene asociadas ciertas características como control de acceso, auditoría o versionado, entre otros.

Las modificaciones realizadas sobre los ficheros (creación, modificación, eliminación) en este servicio serán enviadas a *OSCAR* como un evento y dependiendo de las reglas configuradas, *OSCAR*, lanzará, o no, un servicio asociado a ese objeto.

Este servicio interno puede ser utilizado como almacenamiento de los resultados finales del proceso, aunque también se ofrece la posibilidad de enviarlos de forma automática a proveedores externos como *MINIO*, *AWS S3* o *ONEDATA* [31], siempre y cuando se provea a *OSCAR* de las credenciales necesarias.

2.2.3. Flujo de Trabajo

Las unidades básicas de trabajo de *OSCAR* son las funciones, también denominadas *servicios* en la documentación. Para cada servicio se han de definir los recursos que necesita: CPU[2] y memoria; la imagen Docker que se usará como entorno de ejecución; el script que se ejecutará dentro del contenedor a modo de punto de entrada y, finalmente, sobre qué *bucket* debe escuchar la generación de eventos que se crean al subir, modificar o eliminar objetos. Para un *bucket* determinado se puede utilizar un prefijo o sufijo para filtrar qué objetos activarán la ejecución del servicio asociado, estos prefijos o sufijos no permiten la opción de utilizar *wildcards* y han de ser específicos.

Cuando se define un servicio, *OSCAR* se encarga de manera automática de la generación de los *buckets* en *MINIO* correspondientes, así como de configurar los tipos de eventos de los que quedará a la escucha.

Los resultados generados por los servicios pueden ser almacenados de nuevo en el servidor *MINIO* interno del clúster, permitiendo encadenar múltiples servicios si así se desea. Hay que tener especial cuidado de no generar bucles recursivos pues no se proporcionan herramientas para detectarlos y, en caso de un despliegue en *cloud* público, podría generar unos costes considerables.

OSCAR se centra en una aproximación *file-driven*. Cuando un usuario sube un fichero, esto genera un evento que es recogido por *OSCAR*. *OSCAR*, entonces, se encarga de lanzar un trabajo para ese fichero generando un *job* de Kubernetes, es decir, lanzando un contenedor en base a la imagen proporcionada por el cliente. En este *job* se inyectará el fichero que ha disparado el servicio además de ejecutarse un script que contiene las operaciones a realizar, también definido por el usuario.

Para ejemplificar este flujo vamos a definir un servicio sencillo que se encargue de aplicar un filtro a una imagen. Para ello debemos proporcionarle a *OSCAR* la definición del servicio mediante un fichero FDL (Function Definition Language) [16] utilizado por *OSCAR*.

```
1 functions:
2   oscar:
3   - oscar-cluster:
```

[2]CPU, en el contexto de recursos asignados a un servicio de *OSCAR*, hace referencia a una CPU lógica.

```
4       name: image-filter
5       memory: 250Mi
6       cpu: '1.0'
7       image: ghcr.io/break95/imagemagick
8       script: script.sh
9       input:
10      - storage_provider: minio.default
11        path: example-bucket/input
12      output:
13      - storage_provider: onedata.my_onedata
14        path: result-example-workflow
15
16  storage_providers:
17    onedata:
18      my_onedata:
19        oneprovider_host: my_provider.com
20        token: my_very_secret_token
21        space: my_onedata_space
```

Código 2.5: Ejemplo de definición de servicio.

El fichero FDL que se muestra en el Código 2.5 se divide en dos partes. Por un lado la definición de funciones, de línea 1 hasta la 14 y por otro las credenciales de almacenamientos para servicios externos al clúster, línea 16 en adelante.

La línea 3 hace referencia al nombre del clúster en el que será desplegado el servicio descrito. A continuación viene la definición del propio servicio con el nombre, recursos de memoria y CPU, para este caso 250 MB de memoria y 1 core, junto con la imagen Docker que se utilizará como entorno para la ejecución del servicio. La línea 8 hace referencia al script bash que *OSCAR* ejecutará dentro del contenedor cada vez que se invoque el servicio. Dependiendo de cómo se cree este servicio, ya sea mediante una llamada a la API o mediante el uso de la herramienta CLI de *OSCAR*, el script ha de enviarse directamente en el mensaje o estar en el mismo directorio en el que se encuentra el fichero de configuración, esto solamente es necesario para la creación.

En este caso queremos que los resultados se almacenen en un proveedor de almacenamiento externo que funciona bajo *ONEDATA* por lo que hemos de incluir dichas credenciales en el fichero de configuración del servicio.

Una vez definido y creado el servicio, cuando este sea invocado por la escritura de un fichero en la dirección de input especificada en las líneas 9-11, el script que se le ha pasado tendrá que encargase de leer el fichero que ha generado la invocación del servicio, la cual se conoce gracias a que se incluye una variable de entorno indicando la ubicación. Una vez realizadas las operaciones deseadas sobre el fichero de entrada se tendrá que escribir en el directorio temporal del sistema de ficheros del contenedor en el que se ejecuta. Una vez terminado el script, *OSCAR* se encarga de enviar el fichero escrito a las ubicaciones de salida especificadas en la definición del servicio, en este caso, a *ONEDATA*.

CAPÍTULO 3
Implementación del backend

Una vez explicado el funcionamiento de las herramientas a utilizar se procede a exponer el trabajo realizado durante el desarrollo del *backend*. La implementación del *backend* se puede dividir en dos bloques de igual importancia. Por un lado, se define cómo interactuará el usuario final con *DistRDF* para realizar los análisis en *OSCAR* y, por otro lado, se define cómo llevará a cabo el *backend* el procesado del análisis, distribuyendo la carga y recogiendo el resultado final. En este capítulo se comienza con la parte relativa al *backend*, se continúa con los servicios *OSCAR* necesarios y, una vez sean conocidas las necesidades que este tiene para la satisfactoria ejecución de los análisis, se procederá a determinar qué se le ha de requerir al usuario para utilizar *DistRDF* para utilizar el *backend* de *OSCAR*.

Todo el código e imágenes Docker desarrollados durante la elaboración de este trabajo están disponible en un repositorio de GitHub[1].

3.1 Backend ROOT

Cuando se realiza un análisis, todas las acciones definidas sobre el *RDataFrame* se van almacenando sin ser ejecutadas. Cuando se añade una operación que genera salida, por ejemplo, generar un histograma, *DistRDF* se encarga de analizar todo el código proporcionado por el usuario y pasar a una función interna, llamada *ProcessAndMerge*, los siguientes objetos:

***Mapper*:** es la función que se ha de aplicar a cada uno de los registros del conjunto de datos especificados. Recibe como entrada un objeto tipo *range* y genera como salida otro objeto correspondiente a la salida generada al aplicar la función *Mapper* a ese rango, al que se hará referencia como resultado parcial.

***Ranges*:** es una lista de igual tamaño que el número de particiones especificadas para el análisis. Cada elemento de la lista contiene la información correspondiente a los datos que ha de procesar, tanto la fuente de esos datos como las "tareas aproximadas" de cada una de las fuentes, como se explicaba en el capítulo anterior, estas tareas son utilizadas para obtener los eventos exactos a procesar.

***Reducer*:** es la función de agregación que se utilizará para ir reduciendo dos a dos los resultados parciales generados por la función *Mapper* de cara a obtener el resultado final que espera el usuario.

La implementación del *backend* correspondiente a la ejecución del análisis se ha de realizar dentro de la función *ProcessAndMerge* pues el valor que ha de devolver esta fun-

[1] https://github.com/Break95/TFM-ServerlessRoot.

ción es el resultado de agregar todos los resultados parciales generados y no tiene sentido realizarla en otro lugar.

De alguna forma, se ha de enviar toda esta información que recibe la función *ProcessAndMerge* al sistema de almacenamiento interno de *OSCAR*, o pasarlos a los servicios de forma directa. De tal forma que no solo un experimento se ejecute en paralelo en varios nodos si no que también se pueda dar soporte a que múltiples usuarios estén utilizando un clúster *OSCAR* de forma concurrente.

Partiendo de esta base, para la integración con *OSCAR* se puede indicar que un experimento se realice en un *bucket* o que se realice en la carpeta[2] de un *bucket* ya existente, perteneciente a un usuario en concreto. Se ha optado por la opción en la que un *bucket* representa un experimento por simplicidad, pero se utilizará el término *directorio raíz* como concepto genérico para demostrar que se podrían realizar ambas implementaciones, siendo el directorio raíz en el primer caso el nombre del *bucket* en sí y, en el segundo caso, el directorio raíz será una carpeta de un *bucket*. Para identificar un experimento se emplea un *UUID* generado durante la creación del *RDataFrame* y que es utilizado como el nombre del directorio raíz.

No existen diferencias de rendimiento entre utilizar un único *bucket* con una jerarquía de carpetas o emplear varios *buckets*. La diferencia entre estas dos aproximaciones es que cuando se dispone de varios *buckets* podemos configurar cada uno de ellos con ciertas propiedades y permisos. Por lo tanto, la decisión final sobre qué aproximación es más acertada debería quedar relegada a los requisitos que se definan en una integración real con el ecosistema de *ROOT*, como, por ejemplo, en *SWAN*.

Para poder enviar estos objetos desde el cliente a una máquina remota se necesita serializarlos. Para ello, se ha utilizado *cloudpickle*[3], una librería que extiende las funcionalidades básicas de la librería estándar de serialización de Python (`pickle`), permitiendo enviar funciones definidas durante la ejecución del *backend*, algo que con `pickle` no está soportado.

Por un lado, se serializa tanto la función *Mapper* como la función *Reducer*, y serán escritas en una carpeta del directorio raíz del sistema de almacenamiento de objetos a la que llamaremos *functions*. Estas escrituras no lanzan ningún servicio.

Con esta información ya en el clúster, por parte del *backend* de *ROOT* solamente quedaría definir una forma de invocar los servicios necesarios (esto se discute en la siguiente sección) y esperar al resultado final.

Una vez realizada la invocación de dichos servicios, la función queda bloqueada a la espera de que se genere el resultado final. Este resultado final puede ser escrito tanto en un proveedor externo como pudieran ser *MINIO*, *AWS S3*, o *ONEDATA*, o en el propio sistema interno del clúster. En la implementación se ha utilizado el sistema interno de *OSCAR* (*MINIO*) para almacenar el resultado final.

El *SDK* de *MINIO* ofrece una función que permite recibir notificaciones de determinados eventos, por ejemplo, escritura o modificación de ficheros en una ubicación concreta cuando estos se generan. Con esta funcionalidad el cliente se quedará a la espera, evitando hacer un polling constante al sistema, lo que podría suponer una sobrecarga innecesaria debido a esta consulta constante a *MINIO*.

Los pasos que el *backend* de *ROOT* lleva a cabo en el momento de inicializar un *RDataFrame* con un *backend* de *OSCAR* se resumen de la siguiente forma:

[2] Como se indicaba en el capítulo anterior, los sistemas de almacenamiento de objetos carecen de jerarquía. Durante el resto del documento se utilizará el término carpeta por simplicidad.

[3] https://github.com/cloudpipe/cloudpickle.

1. Comprobar que las credenciales necesarias han sido proporcionadas.
2. Generar un UUID.
3. Crear el directorio raíz en base al UUID.
4. Crear los servicios necesarios asociados al directorio raíz.

Para la creación de los servicios se usa el mismo esquema que el mostrado en el ejemplo de *OSCAR* visto en el capítulo anterior en la Figura 2.5. Una vez inicializado el objeto *RDataFrame* el control se devuelve al usuario de forma instantánea mientras se realiza la creación de servicios y *bucket* necesarios en paralelo. Antes de la ejecución del análisis se comprobará si estos servicios se han inicializado correctamente.

Con lo descrito, el proceso que sigue la función *ProcessAndMerge* quedaría como se muestra en el Código 3.1, el cual se ejecutaría cuando un usuario añade al análisis una operación que genere salida.

```
1 def ProcessAndMerge(Mapper, Ranges, Reducer)
2     Esperar a que los servicios OSCAR se hayan creado correctamente.
3     Enviar las funciones Mapper y Reducer al sistema de almacenamiento de
          objetos de el cluster.
4     Invocar los servicios necesarios.
5     Quedarse a la espera del resultado final.
```

Código 3.1: Pseudocódigo del Backend de ROOT.

3.2 Servicios OSCAR

Por el lado de *OSCAR* se necesitan dos o tres servicios dependiendo del tipo de reducción que se quiera realizar. Los servicios nucleares al problema serían un servicio *Mapper* y otro servicio *Reducer*. Dependiendo de cómo se vaya a realizar la reducción se necesitará disponer de un servicio adicional de coordinación.

Actualmente, la definición de los servicios de *OSCAR* no pueden tener como evento de entrada una ruta de MINIO con *wildcards*, a pesar de ser estos servicios lo suficientemente genéricos como para que puedan ser reutilizados en distintos análisis. Este ha sido uno de los obstáculos a solventar en el trabajo. Por lo que, para satisfacer el requisito de soportar múltiples usuarios o que un usuario realice múltiples análisis de forma concurrente, los servicios 1) han de ser creados bajo demanda, especificando las rutas exactas que generarán el disparo de las funciones; y 2) eliminados, junto con los datos enviados al sistema de objetos una vez el análisis haya terminado de forma satisfactoria.

3.2.1. Servicio Mapper

Para el servicio del Mapper se define un servicio que escuche los eventos de creación de ficheros en la carpeta *mapper-jobs/* del directorio raíz. Por otro lado, el *backend* escribirá en el sistema de objetos tantos ficheros como tenga la lista de *Ranges* y cada uno de estos ficheros contendrá la información, serializada, del *range* concreto, lanzando así la ejecución de tantos servicios Mapper simultáneos como se hayan considerado necesarios durante la inicialización del *RDataFrame*.

El trabajo de una invocación del servicio Mapper consistirá en leer del sistema de objetos la función Mapper deserializándola y convirtiéndola en un objeto y aplicar el *range* que le llega como parámetro de entrada también deserializándolo. Para obtener la

función Mapper es necesario tener acceso a las credenciales del sistema MINIO del clúster y la ruta al fichero que contiene la función Mapper. Esto es posible porque *OSCAR* inyecta en cada función, como variables de entorno, las credenciales del sistema de objetos y también porque se dispone de la ruta de dicho sistema que ha lanzado la función por lo que se puede obtener el directorio raíz del análisis pertinente y, a partir de ahí, obtener la función Mapper. En caso de que la función no tuviese esas credenciales, una alternativa sería incluir también la función Mapper en el fichero en el que enviamos el *range*, con la contraprestación de que se estarían enviando esa función tantas veces como invocaciones del servicio Mapper se realicen.

Cuando un servicio Mapper ha terminado de procesar la parte del análisis, escribe el resultado devuelto por la función Mapper en un fichero temporal del contenedor, siguiendo la filosofía de *OSCAR*, para que este se encargue de gestionar la subida de esos resultados a *MINIO*. Esta escritura por parte de *OSCAR* se realizará en la carpeta *partial-results*, que dará comienzo al proceso de reducción. Esta escritura lanzará o no directamente otro servicio dependiendo del modelo de reducción utilizado.

En el Código 3.2 se muestra el código empleado como punto en entrada al servicio Mapper; para los servicios de reducción y coordinación el esquema es similar.

```
1  #!/bin/bash
2
3  echo "SCRIPT: Invoked function"
4  echo "$INPUT_FILE_PATH"
5
6  # Needed for untrusted certificates.
7  echo 'Davix.GSI.CACheck: n' > .rootrc
8
9  # Get root directory and MINIO credentials.
10 mapper_dir=$(grep -m 1 path: /oscar/config/function_config.yaml | awk '{print
       $2}')
11 endpoint=$(grep endpoint /oscar/config/function_config.yaml | awk '{print $2}')
12 access_key=$(grep access_key /oscar/config/function_config.yaml | awk '{print
       $2}')
13 secret_key=$(grep secret_key /oscar/config/function_config.yaml | awk '{print
       $2}')
14
15 echo "Output will be stored in ${TMP_OUTPUT_DIR}"
16 echo "Creating partial results folder"
17 mkdir "$TMP_OUTPUT_DIR/partial-results"
18
19 python3 /opt/mapper.py "$INPUT_FILE_PATH" "$TMP_OUTPUT_DIR" "$mapper_dir" "
       $endpoint" "$access_key" "$secret_key"
20
21 echo "Python function ended"
22
23 echo "Exiting script."
```

Código 3.2: Código de entrada a la función.

En el código de entrada recopilamos las credenciales necesarias para la conexión con MINIO, líneas 10, 11, 12 y 13. La línea 7 del código es necesaria para poder conectarse a ficheros ROOT remoto en aquellos casos en los que los certificados de los servidores sean auto-firmados. El trabajo real del servicio Mapper se realiza dentro del script Python invocado en la línea 19 cuyo contenido se muestra en el Código 3.3. En este código Python en las líneas 10-13 se deserializa el objeto *Range* que ha invocado el servicio. Posteriormente, se establece la conexión con *MINIO*, líneas 16-23. En la línea 26 se obtiene el directorio raíz que se utiliza para descargar de *MINIO* la función Mapper, líneas 30-33. En la línea 37 se aplica la función Mapper al *Range*. Una vez se obtiene el resultado final, se genera el nombre del fichero que se subirá a *MINIO* con la información necesaria para

el proceso de reducción. Finalmente, se serializa el resultado y se escribe el fichero que *OSCAR* subirá a *MINIO*, líneas 44-47.

Los scripts para el resto de servicios, tanto de entrada al contenedor, como de Python, siguen una estructura similar, por lo que incluirlos y describirlos no aportaría ningún valor adicional. Por este motivo, para el resto de servicios, no se incluyen los códigos, pero se encuentra disponibles en el repositorio de GitHub para su consulta.

```
1  #!/usr/bin/env python3
2  import sys
3  import cloudpickle
4  import ROOT
5  import urllib3
6  from minio import Minio
7  import datetime
8
9  # Load the range object that has triggered the function and is found in the
       file system of the container.
10 rang = None
11 f = open(sys.argv[1], 'rb')
12 rang = cloudpickle.load(f)
13 f.close()
14
15 # Establish a connection with MINIO.
16 mc = Minio(endpoint=sys.argv[4][8:],
17            access_key=sys.argv[5],
18            secret_key=sys.argv[6],
19            secure=False,
20            http_client=urllib3.ProxyManager(
21                 sys.argv[4],
22                 cert_reqs='CERT_NONE')
23 )
24
25 # Get root directory
26 bucket_name = sys.argv[3].split('/')[0]
27 print(f'Bucket Name: {bucket_name}   -  {datetime.datetime.now()}')
28
29 # Get the Mapper function from the bucket.
30 mapper_response = mc.get_object(bucket_name, 'functions/mapper')
31 mapper_bytes = mapper_response.data
32 mapper_response.release_conn()
33 mapper = cloudpickle.loads(mapper_bytes)
34 print(f'Mapper: {mapper}  -  {datetime.datetime.now()}')
35
36 # Apply the Mapper function to the Range.
37 result = mapper(rang)
38
39 # Write the results in the local file system so that OSCAR will upload the
       result to MINIO.
40 file_name = f'{rang.id}_{rang.id}'
41 print(f'File Name: {file_name} - {datetime.datetime.now()}')
42
43
44 result_bytes = cloudpickle.dumps(result)
45 f = open(f'{sys.argv[2]}/partial-results/{file_name}', 'wb')
46 f.write(result_bytes)
47 f.close()
48
49 print(f'Result written to Bucket. - {datetime.datetime.now()}')
```

Código 3.3: Código de mapper.py.

Aunque, a priori, agrupar los eventos a procesar al inicio del análisis y aplicarles la función Mapper al completo parezca buena idea, para trabajos largos padece de un

problema de balanceo. Y es que cuando se realizan operaciones tipo *Filter* sobre el conjunto de datos se van reduciendo el número de entradas que cada invocación del servicio Mapper procesa y podríamos llegar a tener casos en los que un servicio Mapper, tras una operación de filtrado, termina su trabajo mientras que el resto continúan ejecutándose durante largos periodos de tiempo, desaprovechando la capacidad de cómputo asignada a esa invocación del servicio Mapper. Actualmente, no hay forma de controlar esto. Por este motivo, una propuesta de futuro consistiría en modificar esta implementación inicial y fraccionarla en diversas etapas para redistribuir la carga y estudiar si se aprecia algún tipo de mejora.

Para entornos como *AWS Lambda* en los que el tiempo de ejecución de la función está limitado a 15 minutos, escalar el número de servicios Mapper para distribuir la carga puede solucionar el problema. Pero, si los datos con los que se está trabajando se encuentran en remoto y la red supone un cuello de botella esta forma de escalar el problema no funcionaría. Para solucionar el problema en este tipo de entornos, limitados por tiempo, tendríamos que dividir aún más la carga, añadiendo más servicios Mapper, para garantizar la ejecución dentro del tiempo límite y lanzar los trabajos en múltiples etapas para no saturar la conexión a los datos.

3.2.2. Servicio Reducer

Para la etapa de reducción de resultados el tiempo de procesado no supone un problema pues simplemente consiste en ir agregando los resultados de dos en dos hasta obtener un resultado final, y este proceso es rápido.

Pero se plantea el problema de cómo realizar esta reducción cuando las funciones no tienen estado. A continuación, se explican las dos propuestas realizadas en este trabajo y lo que supone la principal contribución frente a lo realizado en el trabajo de AWS [26].

Reducción sin coordinación

En este modelo de reducción, el resultado escrito por cada servicio Mapper lanzará un servicio de reducción, Reducer.

El problema de la reducción en un entorno *serverless* sin coordinación radica en cómo determinar el estado de dicha reducción. La parte del proceso de *mapping* generará una cantidad determinada de resultados parciales. Si establecemos un orden en los servicios Mappers invocados asignándoles un identificador a cada uno de ellos, como pueda ser, la posición en la lista de *ranges* (recordemos que invocamos un servicio Mapper por cada elemento en dicha lista) y limitamos que las reducciones se hagan en un determinado orden podemos realizar esta reducción sin necesidad de coordinador. Sin embargo, esta opción tiene también sus inconvenientes.

Se hace necesario proporcionar información adicional que se encontrará en una nueva carpeta, *reducer-jobs*, para que cada resultado parcial sepa con qué otro resultado parcial ha de reducirse. Para ello, se ha optado por escribir tantos ficheros como trabajos de reducción sean necesarios. Los nombres de los ficheros contendrán la información necesaria sobre la reducción, siendo el nombre la unión de los dos identificadores separados por un carácter seleccionado para este fin. Por ejemplo, si tenemos dos servicios Mapper, que se han de reducir entre sí, con identificadores `1_1` y `2_2`, respectivamente, el nombre de este fichero de información sobre la reducción se llamará `1_1-2_2`. Entonces, cuando un resultado parcial lance un servicio de reducción este consultará el nombre del archivo que lo ha lanzado, pues será el nombre de una de las partes a reducir, y consultará los nombres de los archivos del sistema de objetos buscando su otra parte a reducir. Una vez

se sepa cual es el nombre del otro resultado parcial con el que se ha de reducir, consultará la carpeta que contiene los resultados parciales para ver si esa parte ya está disponible. En caso de que no esté disponible la otra parte terminará y, cuando esa parte restante lance el servicio de reducción, se realizará la reducción. Este proceso queda resumido de la siguiente forma:

1. Obtener el id del resultado parcial que ha invocado al Reducer.
2. Obtener las credenciales de MINIO.
3. Obtener de MINIO el listado de trabajos de reducciones.
4. Buscar el resultado parcial restante con el que ha de reducirse.
    - Si no existe un trabajo de reducción es que es el resultado final y se termina el servicio Reducer sin escribir nada.
    - Si existe el trabajo de reducción pero el resultado con el que se ha reducir no está disponible se termina el servicio Reducer.
    - Si existe el trabajo de reducción y se encuentra disponible el otro resultado parcial, se ejecuta la reducción y se escribe el resultado de esta reducción en *partial-results*. Finalmente se termina el servicio Reducer.

Por otro lado, se necesita encontrar una forma de determinar la generación de estos nombres de forma determinista para que sea cual sea el número de servicios Mapper invocados la reducción se realice de forma satisfactoria. El algoritmo propuesto para solucionar este problema se centra en ir generando un árbol binario desde las hojas hasta la raíz, en el que las hojas son los identificadores de los Mappers, y en cada subida de nivel se van combinando los nombres de dos en dos hasta llegar a la raíz. La combinación de nombre se realiza cogiendo los identificadores de los extremos que coincidirán con los identificadores mínimos y máximos que se han reducido hasta ese punto. Por su parte los servicios Reducer deberán escribir en *partial-results* el resultado con el nombre correspondiente a la reducción de los dos trabajos reducidos para que el nuevo servicio Reducer que se invoque sea capaz de reducirse con otro resultado parcial. Esta generación de trabajos de reducción está integrada en el *backend* de *ROOT* pero se ha desarrollado en esta sección por conveniencia.

Aunque el árbol generado es binario podría cambiarse esta granularidad haciendo que se combinasen en cada salto de nivel 3 o más resultados parciales a la vez.

En el Código 3.4 se muestra el código del algoritmo.

```
def index_generator(ranges):
    # Calculate the max depth of the tree based on the reduction factor.

    max_depth = ceil(log(len(ranges))/log(2))
    init_ranges = [[rang.id] * 4 for rang in ranges]
    array_job = []
    array_job.append(init_ranges)
    depth = 1

    while max_depth >= depth:
        # Add an upper level to the tree.
        array_job.append([])

        tmp_len = len(array_job[depth-1])
        odd = False
        if tmp_len % 2 != 0:
            tmp_len -= 1
```

```
18              odd = True
19
20          # Generate job ids for the current tree level.
21          for rang in range(0, tmp_len,2):
22              t1 = array_job[depth-1][rang]
23              t2 = array_job[depth-1][rang+1]
24              array_job[depth].append([t1[0], t1[3], t2[0], t2[3]])
25
26          # In case of having an odd number of jobs in the current level, move
                the last job up one level.
27          if odd:
28              array_job[depth].append(array_job[depth-1].pop())
29
30          depth += 1
31
32      # Discard the deepest level as it is the same as the Mapper jobs.
33      reducers = array_job[1:]
34      flattened = [f'{job[0]}_{job[1]}-{job[2]}_{job[3]}' for elem in reducers
            for job in elem]
35      return flattened
```

Código 3.4: Algoritmo de generación de trabajos de reducción.

Para ejemplificar este proceso de reducción se procede a exponer un ejemplo, la Figura 3.1 es una ilustración de este proceso, suponiendo una división de la carga del análisis en 8 partes. Por lo tanto, invocaremos 8 servicios Mapper, cada uno con un identificador del 0 al 7 y los identificadores se irán combinando dos a dos subiendo hacia arriba en la estructura del árbol hasta llegar a la raíz.

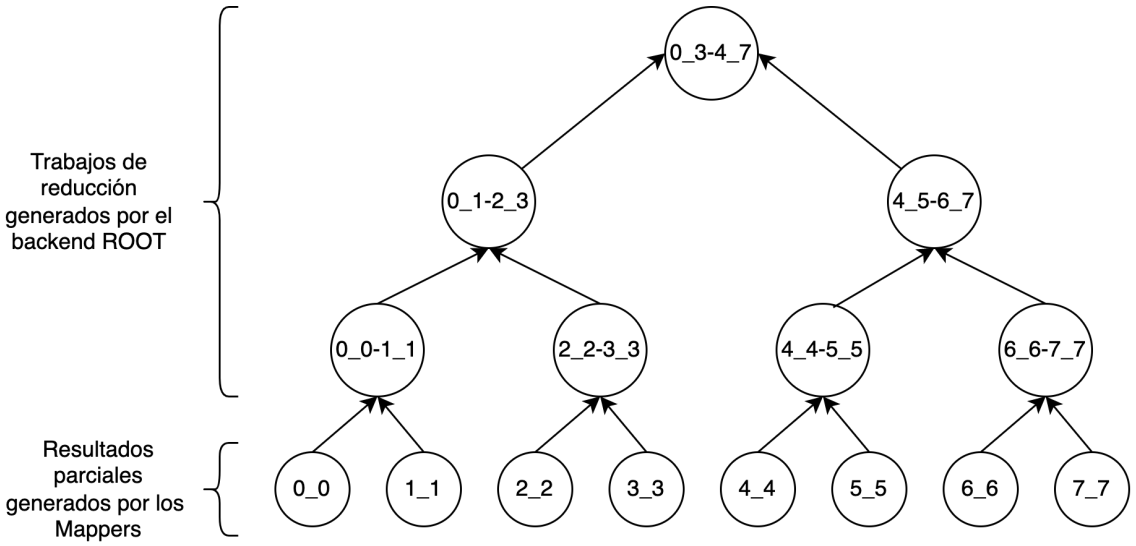


Figura 3.1: Representación del algoritmo de reducción.

La Figura 3.2 muestra como quedarían los trabajos de reducción para el caso de que en alguno de los niveles tengamos trabajos impares, como por ejemplo, para un análisis que invoque siete servicios Mapper.

A partir de estos árboles los nombres del nivel más profundo quedarán descartados, pues son los correspondientes a los servicios Mapper y se escribirán el nombre del resto de nodos a la carpeta correspondiente.

La principal ventaja de este enfoque es que no se requiere de ningún tipo de interacción con el proceso ni de una función de coordinación que tenga que estar ejecutándose desde el principio hasta el final, lo cual sería un gasto de recursos considerable especial-

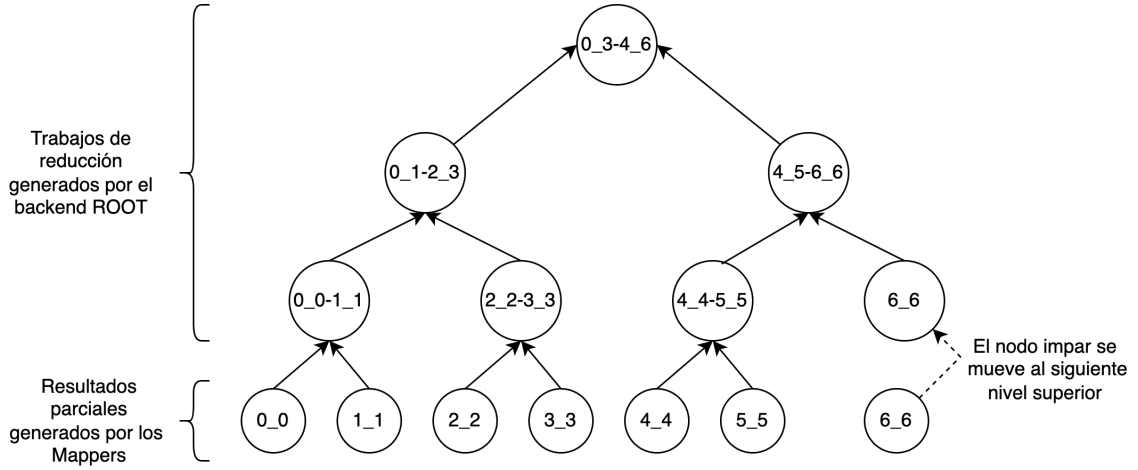


Figura 3.2: Representación del algoritmo de reducción para casos incompletos.

mente en entornos como *AWS Lambda*, donde la facturación de este tipo de funciones se lleva a cabo por milisegundos.

Por contra, se presentan dos desventajas. Por un lado, si dos servicios Mapper terminan en un tiempo prácticamente idéntico, los servicios Reducer asociados se lanzarán también al mismo tiempo. Cuándo estos dos servicios Reducers vayan a consultar la parte con la que se han reducir, ambos la encontrarán disponible y ambos realizarán la reducción. Esto hará que los dos servicios Reducer escriban el resultado de la reducción, invocando dos nuevos servicios Reducer. Esta doble escritura no interfiere con el resultado final pero la invocación extra del servicio implica un uso ineficiente de los recursos disponibles. Además, este comportamiento podría propagarse a lo largo de todo árbol de reducción. Por otro lado, aunque avancemos algo del trabajo, el resultado del último servicio Mapper siempre tendrá que realizar todo el camino de reducciones para llegar hasta la raíz del árbol.

La Figura 3.3 muestra un resumen de la interconexión de todos los componentes descritos. Los identificadores numéricos hacen referencia a cuándo se produce un evento por primera vez.

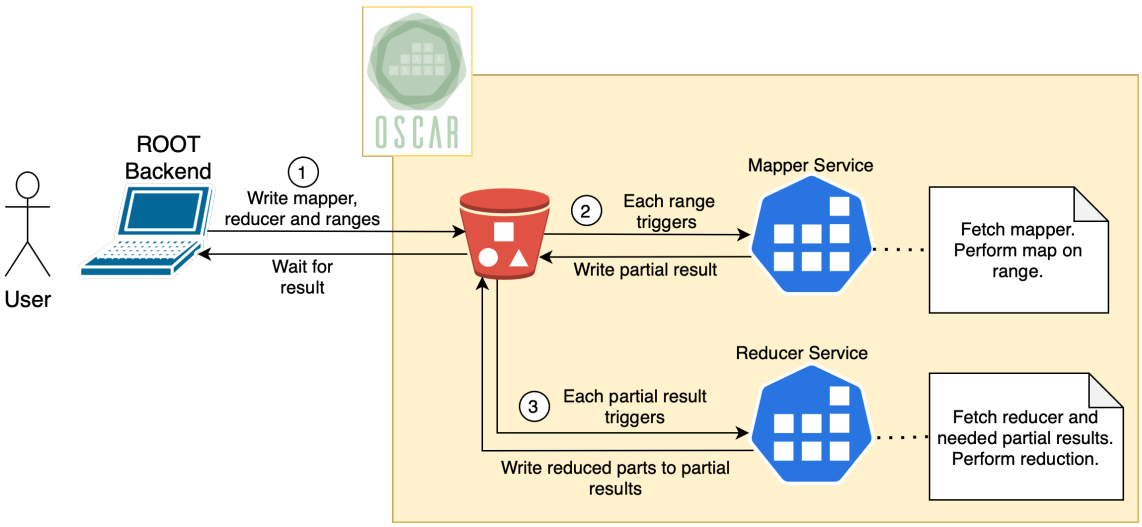


Figura 3.3: Interacción de los componentes para la reducción no coordinada.

Reducción coordinada

Otra alternativa para el proceso de reducción consiste en tener un servicio adicional encargado de la coordinación del proceso de reducción. El *backend* invocará una única vez este servicio antes de que se invoquen los servicios Mapper y este nuevo servicio se encargará de ir monitorizando la carpeta en la que los servicios Mapper generan sus resultados, *partial-results*. Esta monitorización se realiza de igual forma que la que realiza el cliente mientras espera el resultado final, recibiendo un evento cada vez que se escribe un resultado parcial, como se indicaba en la Sección 3.1.

El servicio de coordinación será responsable de ir lanzando trabajos de reducción cuando se estime necesario escribiendo en la carpeta *reducer-jobs* el trabajo de reducción a realizar.

Por un lado, el *backend* generará un fichero con la configuración necesaria del servicio coordinador. Esta configuración consiste en una lista de números enteros en la que cada valor representa el número de resultados parciales que ha de reducir una invocación del servicio Reducer. El coordinador al estar escuchando a la carpeta en la que se escriben los resultados parciales, cada vez que se escriba uno y reciba la notificación correspondiente, irá guardando el nombre de dicho fichero. Una vez alcance el número de resultados parciales especificados en el elemento actual de la lista de reducciones de la configuración, generará un trabajo de reducción que incluirá los nombres de todos los resultados parciales que ha de reducir. Hecho esto pasaría al siguiente elemento de la lista de reducciones hasta realizarlas todas. Una pequeña mejora que se ha realizado en base a este proceso es que la última reducción la realice el propio coordinador.

Cabe aclarar que, dado que la función Reducer solamente acepta dos parámetros de entrada estas reducciones "múltiples" se realizan iterando sobre los trabajos a reducir, pero evitan la invocación de múltiples servicios Reducer y algunas de las limitaciones que teníamos con la reducción no coordinada.

La escritura de los trabajos de reducción por parte del coordinador se hace directamente a *MINIO* sin la interferencia de *OSCAR*, pues este último solo sube los ficheros una vez el servicio haya terminado y el servicio coordinador no debe terminar hasta que se hayan reducido todos los resultados parciales.

Para este caso el servicio Reducer será invocado cuando se escriban ficheros en la carpeta *reducer-jobs* en vez de *partial-results*, como sucedía en el caso no coordinado. El servicio de reducción necesita una modificación adicional. Ya no necesita comprobar si existen otros resultados parciales, simplemente ha de deserializar el contenido del trabajo e ir obteniendo los resultados parciales desde MINIO y reduciéndolos todos. Una vez haya terminado de reducirlos escribirá en un fichero ese resultado parcial con el resultado que *OSCAR* subirá a *MINIO* y del cuál será notificado el servicio Coordinador.

Una ventaja de esta reducción coordinada es que ya no existe la necesidad de mantener un orden en el nombre de los ficheros que se escriben, el servicio coordinador se encarga de escribir los nombres de los resultados parciales a reducir en el trabajo de reducción que coincidirán con los resultados parciales disponibles.

Al disponer de un coordinador se puede estructurar el proceso de reducción de la forma que más convenga, como pueda ser partir la reducción en dos mitades o que una invocación inicial se encargue de reducir un 80 % de los resultados parciales y una segunda invocación se encargue de las reducciones restantes. Esto será algo a estudiar en el siguiente capítulo.

La Figura 3.4 muestra un resumen de cómo interactúan los distintos servicios para este nuevo sistema de reducción.

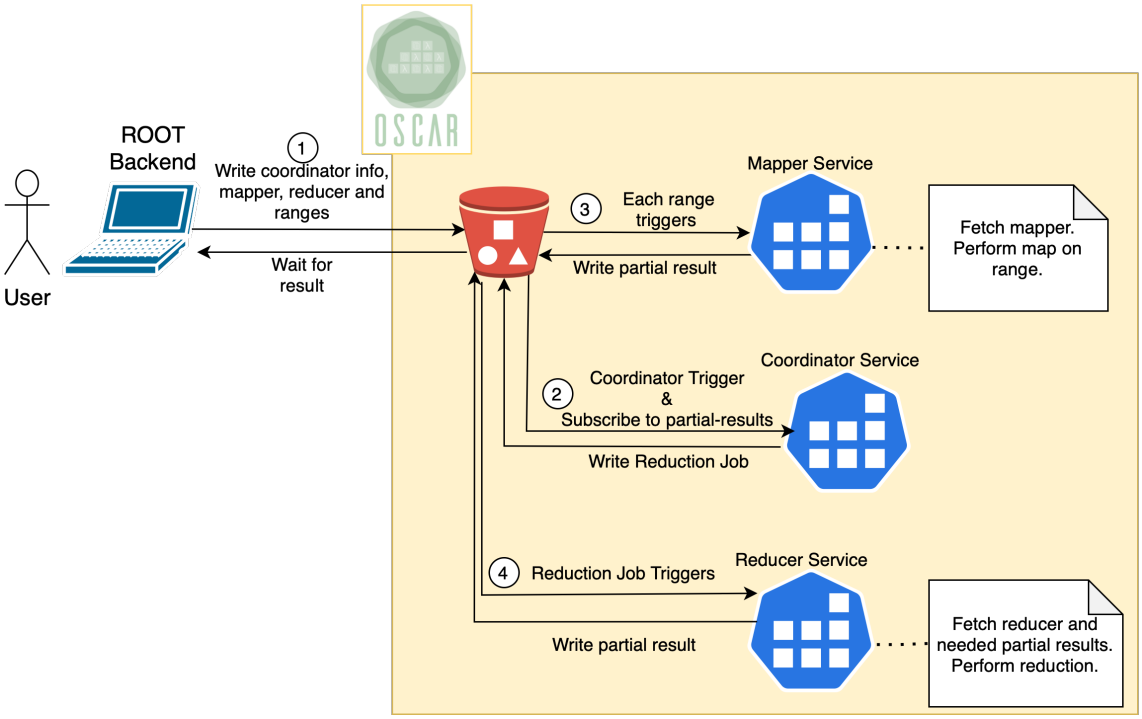


Figura 3.4: Interacción de los componentes para la reducción coordinada.

3.2.3. Resumen

Los primeros intentos de integración con *OSCAR* empleaban las llamadas directas a la API de *OSCAR* para invocar los servicios, tanto Mapper como Reducer, pasando en el *payload* de la petición la información necesaria. Una de las ventajas que esta forma de trabajar aportaba era una dependencia menor del disco en el que se encuentre MINIO manteniendo el uso de la red igual, pues la cantidad de datos que se enviaría por la red sería la misma. Pero presentaba algunas contraprestaciones. Una de las contraprestaciones de esta implementación inicial era que complicaba la lógica del código, pues se hacía necesario determinar que ficheros enviar en el *payload* y cuáles escribir a *MINIO* en el proceso de reducción, pues no se pueden comunicar entre ellos de forma directa. Además de necesitar enviar la información necesaria de *OSCAR* para que la nueva función invocada pudiese llamar a otra. La otra contraprestación era una dependencia de la versión del servicio. En *OSCAR* para poder llamar a un servicio a través de la API se necesita un *token* que se genera con la creación del servicio y cambia al actualizarlo o al volver a crearlo. Por lo que, también implicaba añadir el código necesario al *frontend* para controlar estos casos.

La aproximación utilizada finalmente, se centra exclusivamente en la escritura de ficheros para la generación de eventos, explotando al máximo *OSCAR*. Esta aproximación, a parte de no tener las contraprestaciones del uso de llamadas a la API, tiene una ventaja fundamental y es que, en caso de fallo de algunos de los servicios Mapper o Reducer, sería posible determinar en qué punto ha fallado y reintentar la ejecución de estas partes que han fallado. Esta recuperación se realizaría escaneando todos los ficheros generados en el directorio raíz junto con su información y reintentarlo a partir del punto en el que falló.

En la Figura 3.5 se puede ver un vistazo global cómo quedaría la estructura de MINIO para los dos casos estudiados. La principal diferencia entre ambos enfoques está en que en el escenario no coordinado los trabajos de reducción se escriben antes de que comien-

Figura 3.5: Estructura del sistema de almacenamiento interno dependiendo del modelo de reducción.

cen los servicios Mapper, mientras que en el coordinado los trabajos de reducción son escritos por el coordinador a medida que sea necesario.

3.3 Frontend ROOT

Como se indicaba en el capítulo anterior, las diferencias entre la utilización de *RDataFrame* y *DistRDF* deben ser mínimas para facilitar a usuarios que ya hagan uso de *RDataFrame* la migración a estos nuevos servicios distribuidos. Las modificaciones necesarias suelen realizarse a la hora de importar el módulo de Python correspondiente al *backend* deseado.

Conforme a lo expuesto a lo largo del capítulo, un cliente que quiera hacer uso de *OSCAR* como *backend* de cómputo distribuido necesitará proporcionar el *endpoint* y las credenciales de acceso tanto de *OSCAR* como del sistema interno de almacenamiento, aunque en un futuro si se solventa el problema de los *wildcards* para las invocaciones de los servicios, solamente debería ser necesario proporcionar las credenciales de MINIO pues no tendríamos que crear los servicios para cada experimento. En el Código 3.5 se muestra un ejemplo de cómo se realizaría esta inicialización en un entorno en el que todas estas credenciales ya se encuentran guardadas como variables de entorno.

```
import ROOT
import os

RDataFrame = ROOT.RDF.Experimental.Distributed.OSCAR.RDataFrame

oscarclient = {
    "minio_endpoint": os.environ['minio_endpoint'],
    "minio_access":   os.environ['minio_access'],
    "minio_secret":   os.environ['minio_secret'],
    "oscar_endpoint": os.environ['oscar_endpoint'],
    "oscar_access":   os.environ['oscar_access'],
    "oscar_secret":   os.environ['oscar_secret']
}

df = RDataFrame(1024, oscarclient=oscarlient, npartitions=12)

# Continue as if 'df' was a normal RDataFrame object.
```

Código 3.5: Requisitos para el usuario.

Actualmente, otro de los requisitos de este *backend* es la necesidad, por parte del usuario, de definir el número de servicios Mappers a invocar para el análisis. En otros *backends* se puede obtener el número de *cores* del clúster pero en *OSCAR*, y otros entornos *FaaS* no es una información de la que podamos disponer en el momento de crear los servicios, pues se asume que podemos invocar tantas funciones como queramos. Una posibilidad de futuro sería intentar calcular este número de funciones a invocar base a alguna heurística definida utilizando el número de operaciones que se van a realizar y la estimación del número de entradas que tienen todas las fuentes.

CAPÍTULO 4
Análisis de Rendimiento

En este capítulo se va a a estudiar el rendimiento y la escalabilidad de la solución propuesta. Se comienza explicando cual ha sido la metodología desarrollada para analizar el rendimiento, seguido de una definición detallada del hardware sobre el que se han realizado las pruebas. Finalmente, se analiza el rendimiento del *backend* sacando conclusiones de los resultados obtenidos.

4.1 Metodología

Para medir el rendimiento de la implementación realizada se definen diversas métricas. La más importante de todas es el *Time to Plot*. Esta medida hace referencia al tiempo que pasa desde que un usuario termina un análisis incluyendo una operación que genera salida hasta que el resultado de este análisis está disponible para el usuario. Este tiempo se mide íntegramente desde el lado del cliente. Esta medida determinará la escalabilidad real de la solución. Para el *Time to Plot* cabe mencionar que no se tiene en cuenta el tiempo de creación de los servicios, pues se asume que en un futuro este problema podría solucionarse. Por esta razón consideramos que el tiempo empieza a contar una vez se ha comprobado que los servicios se han creado de forma correcta, línea 3 del Código 3.1, y el tiempo se para justo antes de que la función *ProcessAndMerge* devuelva el resultado final.

Medir solamente el tiempo global no proporciona información sobre lo que está sucediendo en *OSCAR*, por lo que además de este tiempo global también se mide el tiempo de las distintas secciones en las que se divide el proceso para determinar cuáles son las que lastran más al proceso.

En la Figura 4.1 se muestran las secciones en las que se divide el proceso y los tiempos que se han podido recopilar. En azul se muestran las secciones de tiempo correspondientes a *OSCAR*, la subida de ficheros, la creación de los trabajos y el arranque de los contenedores. Mientras que en rojo se contempla solamente los tiempos relativos a la ejecución del código que se ha desarrollado, desde el inicio del script de entrada al contenedor hasta su finalización. Los tamaños de los bloques de tiempo no son representativos.

Por último, y para disponer de información con un mayor nivel de precisión, se han realizado unas pequeñas modificaciones a los servicios desarrollados. Ahora se ejecuta en primera instancia un *wrapper* en Python que se encargará de invocar al script Python de map o reduce, dependiendo del servicio invocado. Este *wrapper* se encarga de monitorizar la utilización de los recursos del proceso Mapper o Reducer tales como CPU y memoria entre otros. Estas medidas se obtienen utilizando la librería *psutil* [38]. Se toma una muestra de ellas cada medio segundo, una frecuencia de muestreo suficiente dada la duración de los servicios.

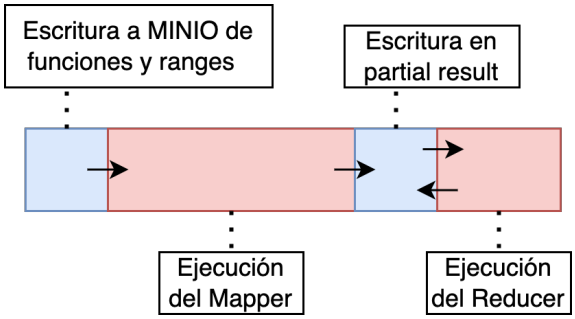


Figura 4.1: División de tiempos.

Adicionalmente, se mide el uso de la red que realizan los servicios. Este uso se mide solamente al principio y al final de la ejecución del proceso Mapper o Reducer. El Código 4.1 muestra un resumen del código del *wrapper* desarrollado que incluye las partes relevantes.

El Código 4.1 se corresponde concretamente con la versión para el servicio Mapper. En la línea 4 se definen los argumentos que se utilizarán para crear el proceso Python. En la línea 5 se configuran las métricas globales que queremos obtener, tanto al principio de la ejecución como al final de la misma. En la línea 8 se obtienen los datos de uso de red antes de llamar al proceso. En las líneas 11 y 12 se invoca al proceso. En la línea 14 se toman la primera muestra de datos globales y en la 15 se copia el valor al que se usará como final por si el servicio termina antes de poder volver a realizar un muestreo. En la línea 17 se crea una lista que irá almacenando los datos de utilización de la CPU y de la memoria. De las líneas 20 a la 30 se realiza un bucle que consulta si el estado del proceso sigue vivo, en caso de seguir vivo se muestrea la utilización y se sobrescribe el resultado del estado final del proceso. El proceso se duerme medio segundo antes de volver a intentar tomar datos. Una vez finalizado el bucle, en la línea 32 se vuelven a tomar los datos de uso de red del contenedor. En la línea 37 se obtiene el *id* del nodo del clúster. A partir de la línea 39 vendrían las líneas en las que se escriben los resultados en ficheros para que *OSCAR* los suba a *MINIO*.

4.2 Especificaciones del hardware

Los resultados que se muestran en este capítulo han sido realizados en dos máquinas distintas. Por un lado, tenemos un clúster facilitado por el GRyCAP, donde se han realizado la mayoría de las pruebas. Por otro lado, también se ha utilizado un ordenador personal para algunas pruebas. Las características de estos sistemas son:

- Clúster GRyCAP:
  - 6 nodos, cada uno de ellos con:
    - CPU: 16 cores. Arquitectura Skylake. Sin HyperThreading. Cada core dispone de 64 KB de L1, 4 MB de L2 y 16 MB de L3.
    - Memoria: 62.5GB de RAM.
  - Red privada del clúster: 10GbE.
  - Red exterior del clúster: 10GbE.
  - MINIO: disco de máquina virtual con 3500 MB/s de lectura secuencial y 3000 MB/s de escritura secuencial.
- PC local:

- CPU: 8 cores, 16 hilos. Arquitectura Vermeer. Cada core consta de 64 KB de caché L1 y 512 KB de caché L2. La caché L3 es compartida y son 32 MB.
- Memoria: 32 GB de RAM.
- Disco: 1 TB SSD NVMe. 3400MB/s de lectura secuencial y 3000MB/s de escritura secuencial.
- Red: 100MbE.
- MINIO: alojado en el disco descrito.

```python
#!/usr/bin/env python3
# Library imports

mapper_args = ['python3', '/opt/python-runner.py'] + sys.argv[1:]
p_attrs = ['create_time', 'cpu_times', 'io_counters',
           'num_ctx_switches', 'memory_full_info']

net_start = psutil.net_io_counters()

# Call to the actual mapper / reducer.
mapper_p = psutil.Popen(mapper_args)
mapper = psutil.Process(mapper_p.pid)

bench_start = mapper.as_dict(attrs=p_attrs)
bench_end = bench_start

cpu_usage = []

# While the process is running.
while mapper_p.poll() is None:
    cpu_usage.append([mapper.cpu_percent(), mapper.memory_percent()])

    tmp = mapper.as_dict(attrs=p_attrs[1:])

    # Check the process has not ended while reading the process metrics.
    if tmp['memory_full_info'] is not None:
        bench_end = tmp

    # Sleep half a second
    sleep(0.5)

net_end = psutil.net_io_counters()

print(f'Writing benchmarks {datetime.datetime.now()}')

# Get the ID of the node that is executing this script.
node = os.environ['RESOURCE_ID']

# From this point onwards the results would be written to files in the local
    file system for OSCAR to upload them to MINIO.
```

Código 4.1: Código Python para la toma de métricas de los procesos mapper y reducer.

4.3 Descripción de Experimentos

Para estudiar las prestaciones del backend desarrollado se va a replicar un experimento de análisis de Dimuones[1] junto con el conjunto de datos físicos aportados en el mismo tutorial del análisis[2].

Uno de los límites más comunes en estos experimentos es la ubicación de los datos, pues la distancia entre la ubicación de los datos y los nodos que los están procesando debería suponer un límite a la escalabilidad de la solución. Por ello, se van a realizar tres variaciones del mismo experimento cambiando la ubicación de los datos. Las fuentes de datos estudiadas son:

- **CERN Data**: los datos estarán alojados en los servidores del CERN, en Suiza.
- **MINIO Data**: el dataset estará almacenado en el sistema de objetos del clúster, proporcionando una localidad mayor que los datos alojados en el CERN.
- **Simulated Data**: Para este experimento se genera una carga de cómputo similar a la realizada por el resto de experimentos. Su funcionalidad es la de tener una referencia de cuál debería ser el resultado óptimo al no realizar ninguna conexión de datos a través de la red, por lo que el rendimiento de este experimento depende exclusivamente de las prestaciones de la CPU, de la memoria y de *OSCAR*.

Todos los experimentos se realizan utilizando el mismo conjunto de datos de aproximadamente 2GB de tamaño y replicándolo cien veces obteniendo una carga de ejecución aproximada de 200GB de eventos físicos.

Además del rendimiento en base a la localidad de los datos, se estudia la escalabilidad de la solución y cómo evoluciona el rendimiento a medida que aumentamos el número de servicios Mapper concurrentes. Se estudian los siguientes números de servicios Mapper: 1, 2, 4, 8, 16, 32, 48, 64 y 80. Inicialmente se planteaba lanzar hasta 96 Mappers de forma simultánea pero Kubernetes utiliza un pequeño porcentaje de la CPU en cada uno de los nodos del clúster, por lo que en cada nodo se quedaba un trabajo pendiente de que algún Mapper terminase para liberar los recursos necesarios. Esto dificultaba la obtención de un *Time to plot* realista por lo que se decidió reducir el número de Mappers a 80.

En la definición del servicio, a cada Mapper se le asigna 1 CPU, y la memoria RAM asignada a cada uno de ellos es de 3 GB. A los servicios Reducer se les asigna la misma cantidad de recursos.

Otra de las pruebas que se realizan consiste en variar el sistema de reducción de los resultados parciales. De esta manera se puede determinar qué enfoque de reducción resulta más efectivo en la práctica.

Para todos los experimentos se realiza una prueba previa antes de tomar tiempos con objeto de asegurar que las imágenes Docker que se utilizarán para arrancar los contenedores en los que se ejecutarán los servicios están disponibles en los seis nodos del clúster. Con esto se persigue evitar las diferencias en los tiempos de ejecución que puedan presentarse entre dos experimentos debido al tiempo de descarga de las imágenes.

[1] https://root.cern/doc/master/df102__NanoAODDimuonAnalysis_8C.html.

[2] root://eospublic.cern.ch//eos/opendata/cms/derived-data/AOD2NanoAODOutreachTool/Run2012BC_DoubleMuParked_Muons.root. Para poder descargar el fichero es necesario utilizar la herramienta *XRootD* [44].

4.4 Resultados experimentales

En las Figuras 4.2 y 4.3 se muestran, respectivamente, la evolución del *Time to Plot* y el *speedup* a medida que aumentamos el número de servicios Mapper, combinado con la reducción no coordinada. Este tipo de reducción será el que se muestre en los gráficos que contengan tiempos globales o relacionados con los servicios Reducer hasta que se indique lo contrario.

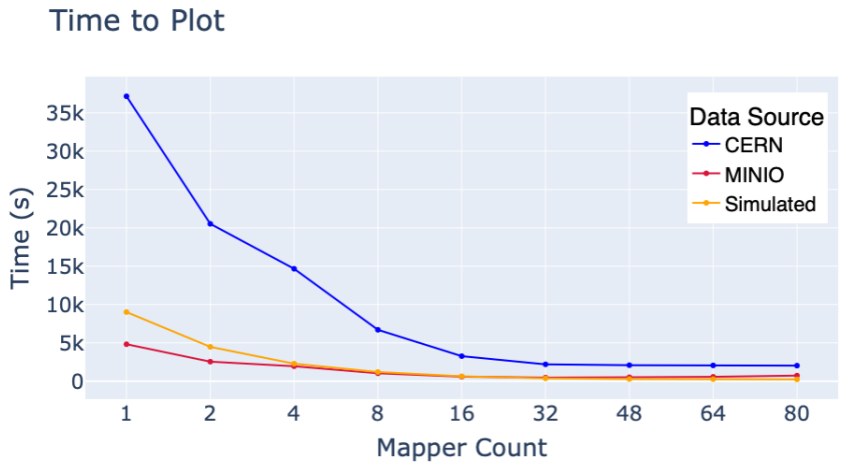


Figura 4.2: Evolución del *Time to Plot* en *OSCAR* para análisis Dimuon.

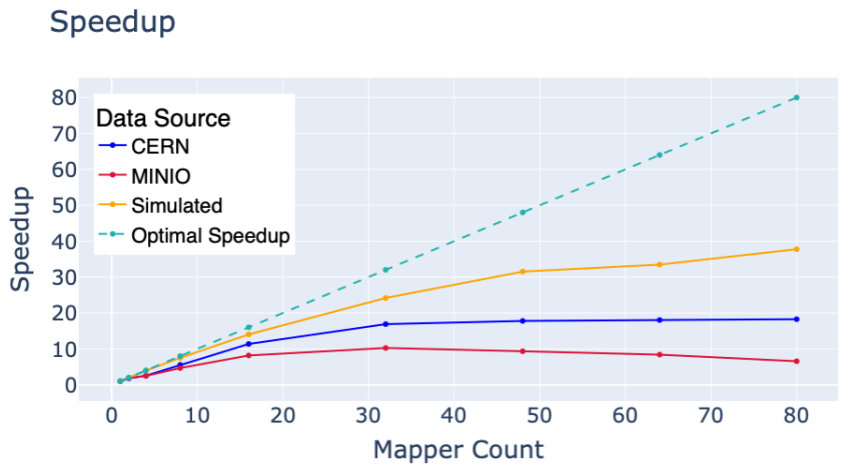


Figura 4.3: Evolución del *Speedup* en *OSCAR* para análisis Dimuon.

Como se puede observar, para el *Time to Plot* todos los experimentos mejoran en mayor o menor medida, aunque la gráfica se ve desvirtuada por los tiempos iniciales de conexión de datos al CERN. Esto sucede especialmente cuando se utilizan pocos servicios Mapper dado que tienen un tiempo de ejecución considerable, aunque no deja de ser un dato que entra dentro de lo esperado.

La carga simulada inicialmente se encuentra en un punto intermedio entre los otros dos experimentos, por lo que podemos aceptarla como una buena aproximación para el caso a pesar de tener más carga real que la del resto de experimentos. Esta discrepancia de la carga se deberá tener en cuenta en caso de ser necesario a la hora de sacar conclusiones.

La carga simulada, a pesar de comenzar con un tiempo peor a *MINIO* escala mucho mejor siendo el tiempo total del análisis, al alcanzar los 32 servicios Mapper, inferior

al tiempo de *MINIO*. Probablemente, esto es debido a que no se requiere de conexión, combinado con algún tipo de saturación relacionada con los accesos al sistema de objetos *MINIO* ubicado en el clúster. En cualquier caso todavía es pronto para sacar conclusiones acerca de esto.

Para observar de forma más precisa la evolución se proporciona el incremento de velocidad o *speedup* obtenido con respecto al incremento en el número de nodos utilizados. El experimento que utiliza los datos en el CERN va escalando hasta encontrarse en una meseta a partir de los 48 servicios Mapper. Este comportamiento lo podríamos atribuir, preliminarmente, a una limitación de la red de interconexión.

Algo muy preocupante de la gráfica del *speedup* es la escalabilidad para el experimento con datos simulados, que, aunque ofrece una mejor escalabilidad que para el resto de implementaciones, no debería quedar tan alejada del *speedup* ideal. Hasta que se encuentren las causas que generan estos malos resultados para el experimento simulado carece de sentido centrarse en los otros dos experimentos.

Para intentar determinar la causa de este comportamiento anómalo se muestra a continuación el uso de CPU y memoria de los Mappers para los tres experimentos. En las Figuras 4.4 y 4.5 se muestran el porcentaje de utilización de CPU y el uso de memoria en MB. Más concretamente, la métrica de uso de memoria es el *Resident Set Size* o el número de páginas alojadas en memoria principal por el proceso. Cada uno de los puntos que forman los diagramas de cajas de estos gráficos se corresponde con el valor medio de la utilización de todos los Mappers invocados para cada experimento con un determinado número de Mappers.

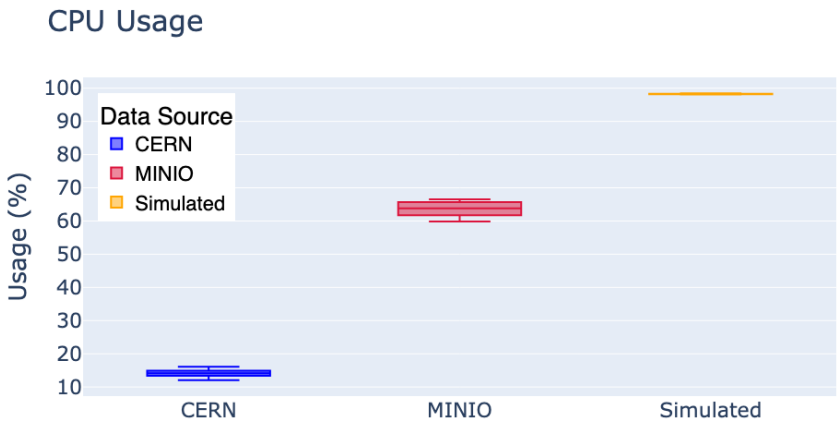


Figura 4.4: Utilización de CPU por parte de los Mappers.

Como es de esperar los datos simulados utilizan la CPU cerca del 100 % durante toda la ejecución, seguido por *MINIO*, con una utilización de entorno al 65 % y, finalmente, el uso de CPU por parte del experimento con los datos en el CERN realiza una utilización muy pobre de la CPU, encontrándose por debajo del 20 %. En cuanto al uso de memoria, ninguno de ellos se encuentra cerca del límite que se le proporciona al servicio, 3 GB, por lo que tampoco se puede atribuir la degradación de rendimiento a que los procesos se encuentren paginando.

Estos datos siguen sin proporcionar información al respecto de cuál puede ser la causa del mal rendimiento. Para intentar encontrar la causa se procede a estudiar el experimento de datos simulados con 80 Mappers, desgranando el tiempo total por secciones hasta conseguir determinar qué está sucediendo.

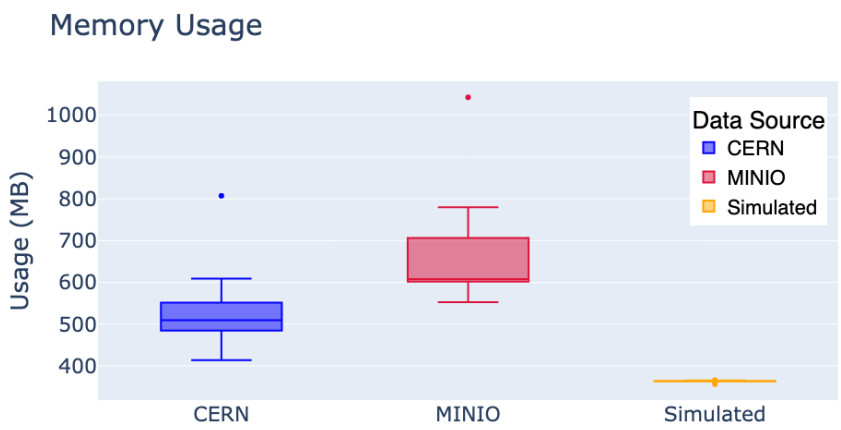


Figura 4.5: Utilización de memoria por parte de los Mappers.

El tiempo total desde que se inicia el experimento hasta que obtenemos los resultados en el caso que nos interesa es de 238,95 segundos. En primera instancia, lo que más limita es el servicio Mapper que más tiempo emplee en realizar su trabajo, aunque, en este caso, deberían tardar todos aproximadamente lo mismo. El tiempo máximo reportado es de 179,5 segundos. Dado que solamente tomamos medidas cada medio segundo y que cabe la posibilidad de que sea un tiempo más cercano a 180 segundos, se utilizará este último valor. Ahora, si restamos el tiempo del script del servicio Mapper al tiempo total quedan 58,95 segundos por asignar al resto del proceso.

El siguiente paso a considerar es el proceso de reducción y cuánto tiempo está consumiendo este. Para los servicios Reducer, el tiempo de muestreo ha de ser tenido en cuenta, pues las mediciones de tiempo se toman cada 0.5 segundos, siendo el proceso de reducción una operación rápida.

En la Figura 4.6 se muestra un diagrama de caja con la muestra de los puntos donde se puede ver que los servicios Reducer tardan más de 0.5 segundos y menos de 2.5 segundos. En esta figura se recogen los tiempos de ejecución de los script Reducer para todos los experimentos realizados con los datos simulados.

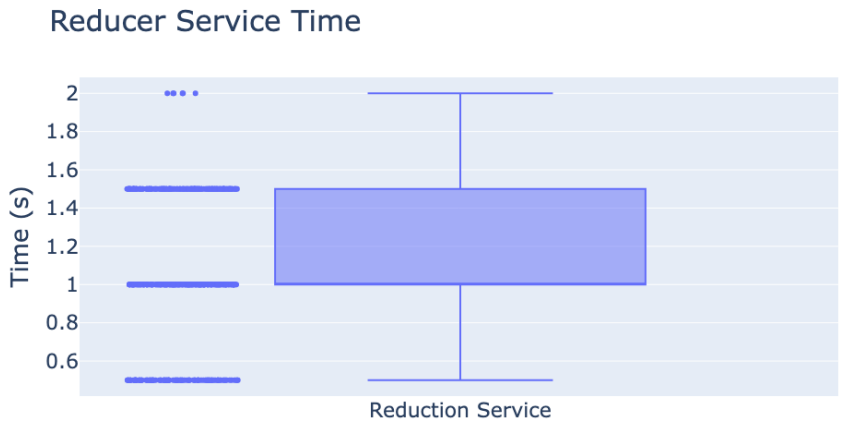


Figura 4.6: Tiempo utilizado por los Reducer de forma individual.

Los tiempos superiores a 2 segundos no guardan correlación con la agregación de resultados parciales grandes como puedan ser los Reducers para un proceso de dos Map-

pers o la última reducción de cualquiera de los experimentos. Concretamente, los scripts del servicio Reducer que tardan más de 2 segundos se corresponden con las siguientes reducciones: “24_24-25_25”, “46_47-48_49”, “8_9-10_11”, “20_21-22_23” y “34_34-35_35”.

Con estos datos podemos establecer una cota superior de 2 segundos para el tiempo de ejecución del script de reducción. Como nos encontramos en el escenario de reducción binaria, cuando termine el último Mapper este tendrá que recorrer todo el camino de ascenso hasta llegar a la raíz del árbol generando el resultado final. Con 80 Mappers la profundidad del árbol será de 7, asumiendo el peor caso en la ejecución de la reducción de cada nivel, es decir, 2 segundos. Desde que termina dicho Mapper hasta que se genera el resultado final tardaríamos 16 segundos, algo muy alejado de los 58,95 segundos restantes tras el Mapper. Si descontamos este nuevo tiempo, quedan 42,95 segundos de los que desconocemos su origen.

Si eliminamos los tiempos de ejecución de los scripts de mapeo y reducción solo nos queda por ver a qué etapas de *OSCAR* se deben estos 43 segundos extra. No se dispone de una forma directa de recopilar estos tiempos pero son accesibles a través de la interfaz de *OSCAR*, que nos proporciona información respecto a los instantes en los que se crea el evento, se arranca el contenedor asociado y se termina el contenedor. Por un lado, cuando se crea un servicio en *OSCAR* y se recibe la notificación se tardará un tiempo en arrancar el contenedor de la imagen correspondiente. En la Tabla 4.1 se ejemplifica este transcurso de tiempo descrito mostrando cinco ejemplos extraídos de los experimentos realizados. La columna *CREATION TIME* se corresponde con el tiempo en el que *OSCAR* ha recibido la notificación de que un archivo ha sido subido. Mientras que la columna *START TIME* hace referencia al momento en que se arranca el contenedor.

Tabla 4.1: Tiempos de arranque de contenedores.

CREATION TIME	START TIME	Tiempo de Arranque (s)
2022-07-21 21:17:33	2022-07-21 21:17:35	2
2022-07-21 21:17:31	2022-07-21 21:17:36	5
2022-07-21 21:17:40	2022-07-21 21:17:47	7
2022-07-21 21:17:32	2022-07-21 21:17:35	3
2022-07-21 21:17:38	2022-07-21 21:17:41	3

De los datos recopilados en algunos experimentos, de media, el tiempo de arranque suele estar al rededor de los 3 segundos. Aunque en algunos casos puede llegar a ser un tiempo muy superior, en la tercera entrada de la figura descrita se puede apreciar como tarda 7 segundos en arrancar uno de los contenedores. Asumiendo unos 3 segundos para el Mapper que más tarda y otros 3 segundos en cada salto del recorrido de reducción obtendríamos aproximadamente unos 27 segundos adicionales, que restados a los 43 segundos pendientes de clasificar, aún quedarían 16 segundos.

Durante la búsqueda de explicaciones para estos *overheads* también se diseñó una prueba en la que el script que se le pasaba a *OSCAR* simplemente dormía durante 1 segundo y escribía en un fichero que invocaba otro servicio, encadenando este proceso tres veces. La imagen utilizada era una imagen muy ligera, al contrario que la de *ROOT*.

Los resultados de este experimento mostraron que *OSCAR* añadía entre 1 y 2 segundos extra a ese segundo que dormía el script. Restando este overhead al tiempo restante, quedaría desglosado todo el tiempo de ejecución del análisis. Es importante tener en cuenta que este overhead puede no ser fijo e ir vinculado a la duración del servicio.

El desglose resumido de los tiempos quedaría de la siguiente forma:

238,95 = 179,5	Mapper
+ 16	Reducer
+ 27	Arranque de contenedores
+ 17	Overhead de ejecución

Todos estos overheads explican, en parte, los malos resultados obtenidos, pero todavía queda por descubrir qué está generando la mala escalabilidad, pues los tiempos deberían ser más estables en la escalabilidad y no degradarse de forma tan repentina.

Habiendo analizado todas las partes relativas al *backend* desarrollado y los overheads introducidos solo queda estudiar si el problema reside en *OSCAR* y si este está interfiriendo de alguna forma con la ejecución de los servicios Mapper y/o los servicios Reducer durante la ejecución de los mismos. En las Tablas 4.2, 4.3, 4.4 se muestra la diferencia de tiempo entre los scripts de los servicios Mappers que más tardan y los que menos, es decir, exclusivamente de la ejecución del script desarrollado sin tener en cuenta el tiempo que tarde *OSCAR* en subir los ficheros ni en el arranque del contenedor. Se muestra la diferencia de tiempo, tanto en valores absolutos como en porcentaje de tiempo.

Tabla 4.2: Variabilidad del tiempo de ejecución de los Mappers con carga simulada.

Mappers	Diferencia de Tiempo Absoluta (s)	Diferencia de Tiempo relativa (%)
1	0.0	0.0
2	119.0	2.79
4	154.0	7.41
8	105.5	10.03
16	63.5	11.72
32	40.5	13.64
48	32.5	15.44
64	48.0	29.45
80	39.0	27.76

Tabla 4.3: Variabilidad del tiempo de ejecución de los Mappers con datos en el CERN.

Mappers	Diferencia de Tiempo Absoluta (s)	Diferencia de Tiempo relativa (%)
1	0.0	0.0
2	1505.5	8.28
4	1504.0	11.93
8	784.0	13.75
16	844.5	36.53
32	518.0	32.41
48	427.0	27.14
64	430.0	27.99
80	363.5	23.06

Tabla 4.4: Variabilidad del tiempo de ejecución de los Mappers con datos en MINIO.

Mappers	Diferencia de Tiempo Absoluta (s)	Diferencia de Tiempo relativa (%)
1	0.0	0.0
2	22.0	0.92
4	220.0	13.44
8	94.0	10.64
16	61.0	12.43
32	39.5	10.25
48	48.5	11.95
64	57.0	12.49
80	60.5	10.46

Como se puede observar en las tablas existe una diferencia de tiempo considerable entre el script del Mapper que más tarda con el que menos tarda. Para los casos de los datos en el CERN o *MINIO* podríamos seguir investigando motivos de este malo rendimiento, como saturación del disco o de limitaciones de la red. Pero este comportamiento también se produce para la carga simulada, Tabla 4.2, en la que a medida que aumentamos el número de servicios Mapper el overhead incurrido es mayor, llegando a observar un pico de 28,45 puntos de diferencia con 64 servicios Mapper.

Estas pruebas se volvieron a replicar en el PC local que se describe en la Sección 4.2, con 4 servicios Mapper, desactivando la opción multihilo y utilizando una frecuencia fija para minimizar la variabilidad. En este caso se obtenía una diferencia del 32,59 % ejecutando la misma prueba pero sin *OSCAR*. Utilizando 4 procesos la diferencia de tiempo se reducía a un 2,8 %. Con esto podemos descartar que el problema sea del hardware del clúster y atribuirlo de forma directa al funcionamiento de *OSCAR*.

A pesar de esta variabilidad, una información interesante a visualizar consiste en ver el cómo se distribuyen los tiempos de los diferentes servicios Mappers. En la Figura 4.7 se muestra un ejemplo de distribución de los tiempos de servicios Mappers para los experimentos de carga simulada con 48, 64 y 80 servicios Mapper.

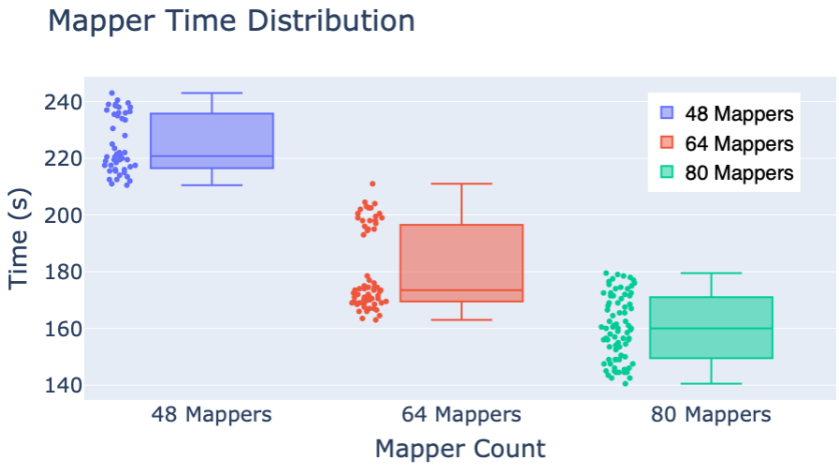


Figura 4.7: Distribución de los tiempos de los servicios Mapers.

En la Figura 4.8 se muestra el uso de la red por parte de los servicios Mapper para el experimento con 80 invocaciones. Cada punto del diagrama de cajas representa el tráfico recibido por cada uno de los nodos del clúster. Como se puede apreciar, el uso de la red es prácticamente uniforme por parte de los Mappers. Al contar con 6 nodos en el clúster

y no ser el reparto de los servicios uniforme no todos los nodos tendrán el mismo tráfico, tal como se aprecia en la figura.

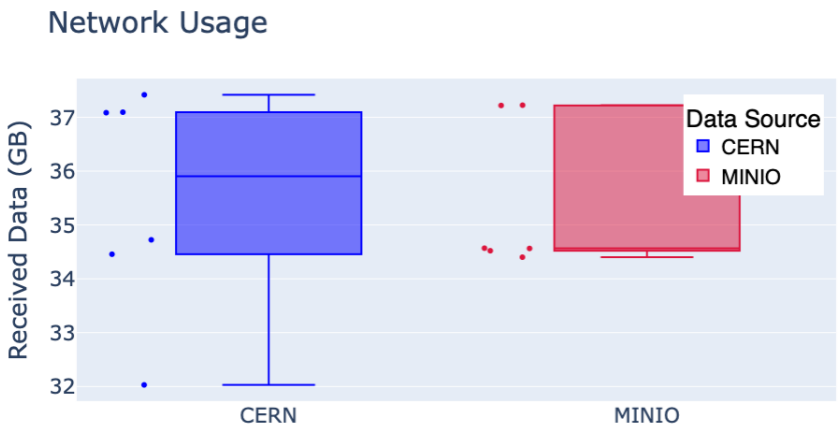


Figura 4.8: Utilización de la red por parte de 80 Mappers.

4.5 Pruebas con el backend de Dask

Para determinar, finalmente, que el problema generado sea de *OSCAR* y no esté relacionado con el hardware subyacente se han repetido los experimentos de carga simulada desplegando un clúster Dask en el hardware facilitado por el GRyCAP y en un clúster del CERN y se han comparado los resultados obtenidos.

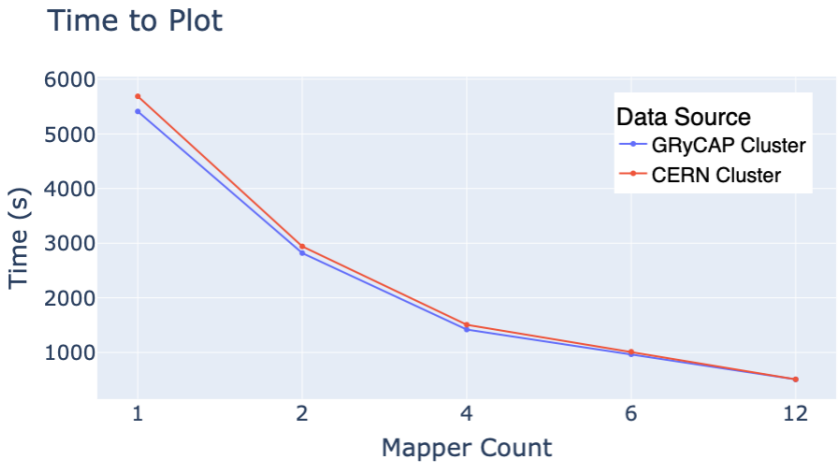


Figura 4.9: Evolución del *Time to Plot* con Dask. I.

Los resultados de estas pruebas se muestran en las Figuras 4.9 y 4.10, ambas figuras representan los resultados del mismo experimento, la Figura 4.9 representa los datos para el rango de 1 a 12 Mappers, mientras que la Figura 4.10 para el rango de 12 a 24 Mappers. Como se puede observar en la Figura 4.9 el clúster del GRyCAP comienza siendo ligeramente superior en rendimiento, probablemente debido a las diferencias de rendimiento mono-núcleo de ambos sistemas. Pero, a medida que vamos aumentando el número de Mappers a ejecutar en paralelo, el rendimiento del clúster de GRyCAP va disminuyendo, empezando a ofrecer tiempos peores al clúster del CERN a partir de los 12 Mappers y llegando a un punto en el que incluso empeoramos el tiempo al aumentar el número de

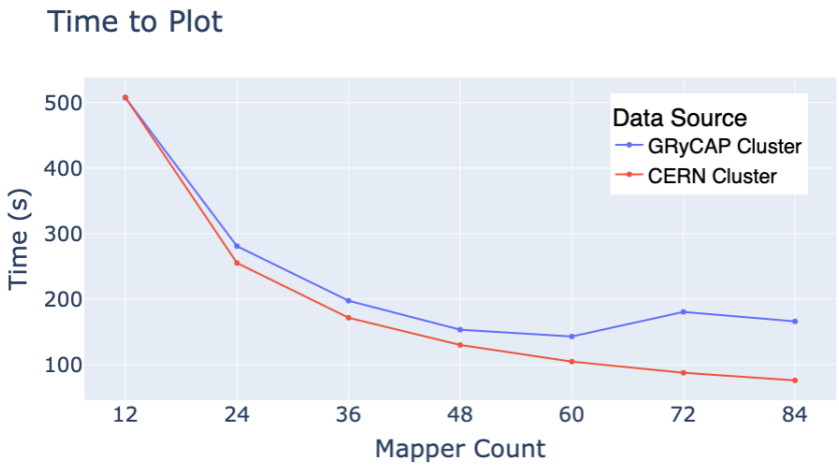


Figura 4.10: Evolución del *Time to Plot* con Dask. II.

Mappers utilizados en el propio clúster, en el salto de 60 a 72 Mappers. De las métricas obtenidas, este salto del tiempo no parece tener una causa aparente directa pues la utilización de todos los cores del clúster de GRyCAP para estos últimos casos está entorno al 100 % igual que para los casos con un menor número de Mappers.

Con estos datos podemos concluir que los resultados obtenidos en los experimentos realizados en las secciones anteriores también se han visto influenciados por problemas del clúster.

4.6 Comparación de los sistemas de reducción

En esta sección se procede a analizar los datos obtenidos para ambos tipos de reducción. Para el modelo de reducción sin coordinador, el grado del árbol se quedará fijado en dos.

El modelo con coordinador ofrece una mayor flexibilidad. Tomando como punto de partida los overheads existentes de invocar servicios en *OSCAR* se ha optado por estudiar solamente la realización de dos fases de reducción, una primera en un servicio Reducer invocado por el coordinador y la segunda en el propio coordinador. Para este estudio se ha utilizando solamente el experimento que emplea 80 servicios Mapper, pues es el que más reducciones ha de realizar. Se han estudiado tres configuraciones para las dos fases de reducción: 0 %, donde se realizan todas las reducciones en el coordinador, 50 % donde el servicio que se crea procesa un 50 % de las reducciones y del otro 50 % se encarga el coordinador y, por último, 87,5 % en la que el servicio reducirá 70 resultados parciales y las reducciones restantes las hará el coordinador.

Tabla 4.5: Time to plot para distintas configuraciones del coordinador.

Divisón de Trabajo	Time To Plot
0 %	205
50 %	208
87,5 %	202

Como se puede observar en la Tabla 4.5 los tiempos son muy similares entre sí y probablemente las diferencias entre ellos esté generada por la variabilidad de los servicios Mapper. Para la comparación con la reducción binaria se utilizará el caso de 87,5 %. Este

valor se toma en base a los resultados que se muestran para 48 y 64 servicios Mappers en la Figura 4.7, en la que más de la mitad de los servicios terminan en un tiempo considerablemente inferior al resto.

En la Figura 4.11 se puede ver cómo, a pesar de la variabilidad en la que puedan incurrir los servicios Mappers, la reducción coordinada se mantiene por delante de la reducción binaria. A medida que aumentamos el número de servicios Mappers a utilizar esta diferencia se va incrementando. Esto se debe a que el árbol es cada vez más profundo y el último servicio Mapper en terminar ha de recorrer, salto a salto, el árbol hasta llegar a la raíz. Probablemente, si se dispusiese de suficientes recursos podríamos ver cómo los tiempos empiezan a empeorar, pues tardaríamos más en realizar el proceso de reducción que en procesar los datos.

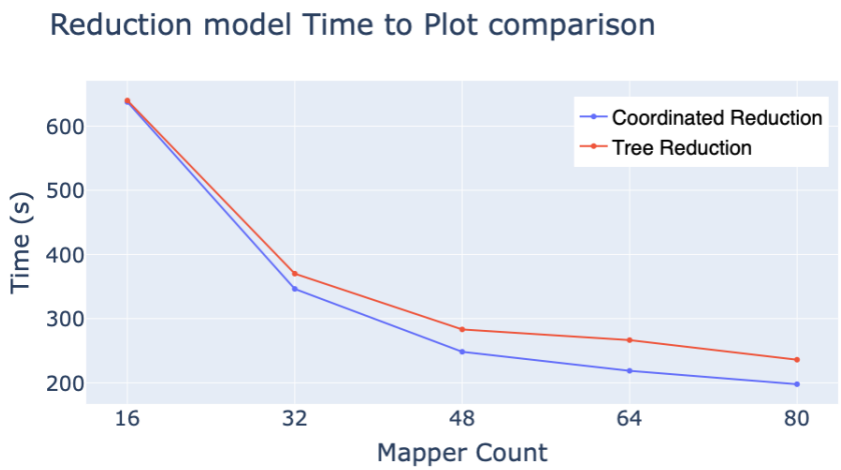


Figura 4.11: Comparación de tiempos de reducción.

El modelo no coordinado podría mejorarse si cada vez que se invoca una función de reducción, en vez de consultar solamente el resultado parcial con el que se ha de reducir de forma directa, también consultase todo el recorrido que ha de realizar en el árbol. Con esto podríamos reducir el número de funciones invocadas y el número de escrituras a *MINIO*, reduciendo algunos de los overheads de *OSCAR*.

En la Figura 4.12 se puede ver un ejemplo de esta modificación. Los nodos verdes se corresponden con los trabajos que ya se han reducido y están disponibles. En naranja se encuentra el último servicio Mapper pendiente de generar sus resultados parciales. Una vez este servicio Mapper termina, todos los resultados parciales necesarios para alcanzar la raíz ya están disponibles y no debería ser necesario realizar múltiples invocaciones del servicio Reducer.

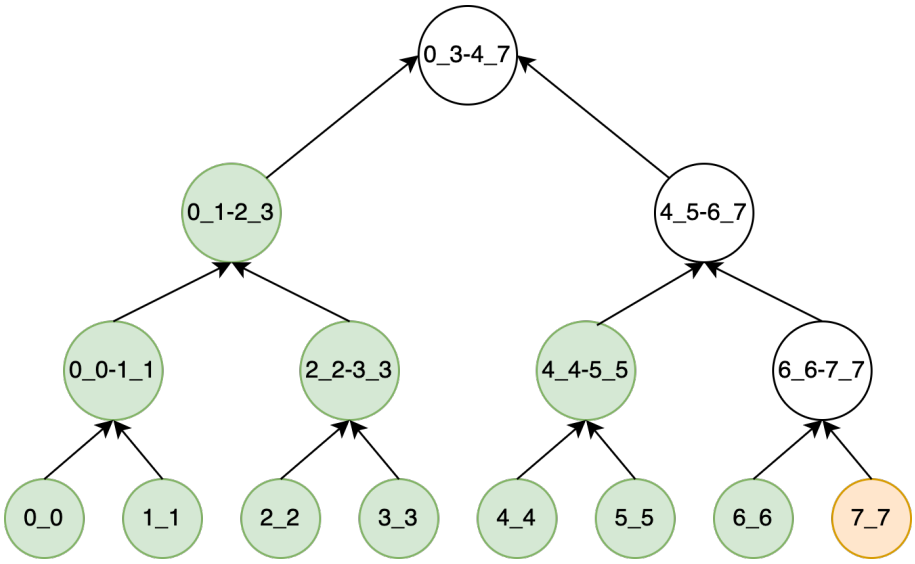


Figura 4.12: Mejora en reducción binaria.

CAPÍTULO 5
Conclusiones y Trabajo Futuro

5.1 Conclusiones

En este trabajo de fin de máster se ha diseñado e implementado un *backend serverless* para *Distributed RDataFrame* mediante una interfaz Python, la cual no está altamente acoplada a *OSCAR* y que podría ser generalizable para ser utilizada por otros *frameworks serverless* sin grandes problemas. Esta implementación también ofrece ciertas mejores sobre algunos aspectos de otra implementación ya realizada en *AWS*.

Una de las principales mejoras es la migración del proceso de reducción desde el lado cliente al *backend serverless*, lo que permitiría evitar dejar bloqueado al cliente durante la realización del análisis si se implementase una forma de obtener el resultado de la última reducción.

El rendimiento real de la implementación no ha podido ser estudiado y comparado en igualdad de condiciones con otros *backends* debido a los problemas detectados en *OSCAR*.

En cuanto a los objetivos planteados en el primer capítulo se han cumplido todos:

- Se ha analizado la arquitectura de *OSCAR*, junto con todos sus componentes y su flujo de trabajo. También se ha realizado un estudio del estado del arte de otros *frameworks serverless* para la computación científica.
- Se ha realizado un estudio de la herramienta *ROOT*, centrado en la parte de análisis de datos, comprendiendo tanto el modelo de programación anterior centrado en los *TTree*, cómo en las novedades introducidas por *RDataFrame* y su extensión por parte de *Distributed RDataFrame* para la computación distribuida.
- Se ha proporcionado una integración del modelo *event-driven* de *OSCAR* con el modelo de computación distribuida empleado por *Distributed RDataFrame*, explotando al máximo las capacidades del primero.
- Se ha implementado el *backend* de forma satisfactoria y se han proporcionando dos formas distintas de aplicar la fase de reducción.
- Se ha realizado un análisis del rendimiento en profundidad dando una explicación de todos los datos obtenidos. Tanto para las partes desarrolladas en el trabajo como para el entorno en el que estas se ejecutaban.

5.2 Trabajos Futuros

Como trabajos futuros se plantean los siguientes:

- Generalización de la interfaz desarrollada para dar soporte tanto a cualquier *framework serverless* que ofrezcan características similares a *OSCAR*, como a diversos proveedores de *cloud* públicos.
- Seguir optimizando el proceso de reducción, ya sea replanteando este en su totalidad mediante el rediseño de las estructuras de datos utilizadas, o aplicando optimizaciones al diseño ya existente.
- Desarrollar funcionalidad para poder continuar la ejecución de un análisis que haya podido verse interrumpido por un particionado de la red entre o por una caída de los nodos del clúster mientras trabajaban.
- El código desarrollado aún se encuentra en una fase de pruebas y faltaría seguir trabajando sobre él para dar una implementación lista para producción y ser integrada en *ROOT*.

Bibliografía

[1] Slurm. https://slurm.schedmd.com/documentation.html, Accedido por última vez en julio de 2022.

[2] Johannes Albrecht. A roadmap for hep software and computing r&d for the 2020s. *Computing and Software for Big Science*, 3(7), 2019.

[3] James Beswick. Using amazon efs for aws lambda in your serverless applications. https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications/, Accedido por última vez en julio de 2022.

[4] I Bird, P Buncic, F Carminati, M Cattaneo, P Clarke, I Fisk, M Girone, J Harvey, B Kersevan, P Mato, R Mount, and B Panzer-Steindel. Update of the Computing Models of the WLCG and the LHC Experiments. Technical report, CERN, Apr 2014.

[5] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth. A*, 389:81–86, 1997.

[6] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, oct 2020.

[7] CERN. The high-luminosity lhc project.

[8] CERN. The high-luminosity lhc project planning, 2022.

[9] CERN. Tiers of worldwide lhc computing grid, 2022.

[10] CERN. Swan, interactive data analysis in the cloud. https://swan.web.cern.ch/swan/, Accedido por última vez en julio de 2022.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[12] Inc Docker. Docker. https://www.docker.com, Accedido por última vez en julio de 2022.

[13] Alex Ellis. Openfaas.

[14] V. Giménez-Alventosa, Germán Moltó, and Miguel Caballer. A framework and a performance assessment for serverless mapreduce on aws lambda. *Future Generation Computer Systems*, 97:259–274, 2019.

[15] Google. Kubernetes. https://kubernetes.io/es/, Accedido por última vez en julio de 2022.

[16] GRyCAP-I3M-UPV. Function definition language. https://scar.readthedocs.io/en/latest/prog_model.html, Accedido por última vez en julio de 2022.

[17] GRyCAP-I3M-UPV. Im: Infraestructure manager. https://www.grycap.upv.es/im/index.php, Accedido por última vez en julio de 2022.

[18] GRyCAP-I3M-UPV. Im: Infraestructure manager. https://servproject.i3m.upv.es/ec3/, Accedido por última vez en julio de 2022.

[19] GRyCAP-I3M-UPV. Im: Infraestructure manager. https://www.grycap.upv.es/clues/es/index.php, Accedido por última vez en julio de 2022.

[20] HashiCortp. Terraform. https://www.terraform.io, Accedido por última vez en julio de 2022.

[21] Jacqui Hayes. Happy 10th birthday, wlcg! https://sciencenode.org/feature/happy-10th-birthday-wlcg.php, Accedido por última vez en julio de 2022.

[22] HEPix. Hepix benchmarking working group, 2017.

[23] Intel. Intel threading building blocks. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html, Accedido por última vez en julio de 2022.

[24] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99

[25] Knative. Knative. https://knative.dev/docs/, Accedido por última vez en julio de 2022.

[26] Jacek Kuundefinednierz, Maciej Malawski, Vincenzo Eduardo Padulano, Enric Tejedor Saavedra, and Pedro Alonso-Jorda. Distributed parallel analysis engine for high energy physics using aws lambda. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, page 13–16, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Maarten Litmaath and Matthew Viljoen. *Collaboration Activities between EGI and WLCG*, April 2021.

[28] Wes McKinney. Pandas. https://pandas.pydata.org, Accedido por última vez en julio de 2022.

[29] MinIO. Minio. https://min.io, Accedido por última vez en julio de 2022.

[30] University of Wisconsin–Madison. Htcondor. https://htcondor.org, Accedido por última vez en julio de 2022.

[31] ONEDATA. Onedata. https://onedata.org, Accedido por última vez en julio de 2022.

[32] Padulano, Vincenzo Eduardo, Cervantes Villanueva, Javier, Guiraud, Enrico, and Tejedor Saavedra, Enric. Distributed data analysis with root rdataframe. *EPJ Web Conf.*, 245:03009, 2020.

[33] Piparo, Danilo, Canal, Philippe, Guiraud, Enrico, Pla, Xavier Valls, Ganis, Gerardo, Amadio, Guilherme, Naumann, Axel, and Tejedor, Enric. Rdataframe: Easy parallel root analysis at 100 threads. *EPJ Web Conf.*, 214:06029, 2019.

[34] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.

[35] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. On-premises serverless computing for event-driven data processing applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 414–421, 2019.

[36] Fons Rademakers, Philippe Canal, Axel Naumann, Olivier Couet, Lorenzo Moneta, Vassil Vassilev, Danilo Piparo, Gerardo GANIS, Bertrand Bellenot, Sergey Linev, wverkerke, Pere Mato, Enrico Guiraud, TimurP, Guilherme Amadio, Matevž Tadel, wlav, Andrei Gheata, Stefan Roiser, marsupial, Enric Tejedor, Anirudha Bose, CristinaCristescu, Raphael Isemann, Xavier Valls, Omar Zapata, Kim Albertsson, Anders Eie, Helge Voss, and Brian P Bockelman. root-project/root: First release of the v6-16 series., February 2019.

[37] Matthew Rocklin. Dask. https://www.dask.org/, Accedido por última vez en julio de 2022.

[38] Giampaolo Rodola. psutil. python system and process utilities. https://psutil.readthedocs.io/en/latest/, Accedido por última vez en julio de 2022.

[39] Amazon Web Services. https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-folders.html/, Accedido por última vez en julio de 2022.

[40] Amazon Web Services. Amazon s3. https://aws.amazon.com/es/s3/, Accedido por última vez en julio de 2022.

[41] Amazon Web Services. Aws lambda. https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/, Accedido por última vez en julio de 2022.

[42] E Sexton-Kennedy. HEP software development in the next decade; the views of the HSF community. *Journal of Physics: Conference Series*, 1085:022006, sep 2018.

[43] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra, 2018.

[44] SLAC and CERN. Xrootd, Accedido por última vez en juliode 2022.

[45] V Vasilev, Ph Canal, A Naumann, and P Russo. Cling – the new interactive interpreter for ROOT 6. *Journal of Physics: Conference Series*, 396(5):052071, dec 2012.

[46] Matei Zaharia. Apache spark. , Accedido por última vez en julio de 2022.