UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

APLICACIÓN DE TÉCNICAS DE BIG DATA PARA EL PROCESAMIENTO DE DATOS DE LA OPERACIÓN DEL SECTOR ELÉCTRICO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO

JORGE ANDRÉS RECABAL ÁVILA

PROFESOR GUÍA:
CARLOS BENAVIDES FARÍAS

PROFESOR CO-GUÍA:
SEBASTIÁN GWINNER SILVA

COMISIÓN:
PABLO ESTÉVEZ VALENCIA

SANTIAGO DE CHILE
2024

Resumen

Este trabajo tiene como objetivo aplicar técnicas de *Big Data* para el procesamiento eficiente de datos en el sector eléctrico, con el fin de mejorar la capacidad de análisis y toma de decisiones. Las técnicas implementadas se aplicaron a datos de generación eléctrica, flujos por las líneas de transmisión y costos marginales del Sistema Eléctrico Nacional. El procesamiento de esta información permite calcular indicadores como generación por tipo de tecnología, factores de planta de centrales, percentiles de los flujos por las líneas de transmisión y costos marginales. Los datos con resolución horaria fueron construidos a partir de simulaciones realizadas con modelo proyección a largo plazo (modelo PLP). En específico, se aplicaron herramientas de *Big Data* que permiten la optimización de las rutinas de procesamiento de salidas y manejo de archivos con alto volumen de información.

En una primera etapa, las rutinas preexistentes fueron optimizadas de manera que la comparación con respecto al uso de técnicas de *Big Data* fuera más justa. Para ello, inicialmente se centró en la optimización de la lectura y escritura de archivos mediante el uso eficiente de la biblioteca *Pandas*. Esto se tradujo en un procesamiento más rápido y en la reducción del uso de memoria RAM al crear y almacenar *Dataframes* de manera directa.

Para ahorrar recursos en memoria y disminuir el volumen de información, se exploraron alternativas como *Parquet* y *Feather*, demostrando reducciones significativas en el tamaño de archivos y mejoras en la eficiencia de lectura y escritura, tanto para los archivos originales en formato CSV como para los datos con resolución horaria. Las salidas del modelo PLP originales pesaban decenas de *gigabytes*, lo cual se reduce a solo unos pocos o inclusive al orden de *megabytes*. Mientras que para los archivos transformados a resolución horaria, el tamaño de estos disminuyó de cientos de *gigabytes* a tamaños menores a 20 *gigabytes*.

Para el desarrollo de las rutinas de procesamiento, se exploraron herramientas como *Dask* y *Pyspark*, destacando la ejecución perezosa y la computación distribuida como enfoques clave para mejorar la eficiencia en grandes volúmenes de datos. Las rutinas implementadas se aplicaron para procesar las salidas de 4 casos de estudio del modelo PLP que representaban casos de diferentes tamaños. Se consiguió mejorar los tiempos de ejecución de las rutinas en hasta seis veces con los respecto a los tiempos originales. Para el caso más grande, el tiempo de procesamiento pasó de una hora a demorar entre diez a quince minutos aproximadamente.

De esta forma, se concluye que gracias a las herramientas de *Big Data* implementadas, es posible lograr resultados positivos en cuanto al manejo eficiente del volumen de información y la velocidad de procesamiento de los datos.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Identificación del Problema

El sector eléctrico ha experimentado una creciente generación de datos debido al avance de las tecnologías de medición, control y monitoreo. Estos datos provienen de diversas fuentes, como plantas de generación, sistemas de distribución, medidores inteligentes y dispositivos de monitoreo en tiempo real. Sin embargo, el volumen y la diversidad de estos datos estan paulatinamente superado las capacidades de las herramientas y técnicas tradicionales utilizadas para su procesamiento y análisis.

El problema radica en la dificultad para gestionar, analizar y comprender eficientemente estos grandes volúmenes de datos eléctricos. Las herramientas y técnicas tradicionales no están diseñadas para lidiar con la complejidad y la escala de estos datos, lo que limita la capacidad de los profesionales del sector eléctrico para obtener información valiosa y tomar decisiones fundamentadas en tiempo real.

El procesamiento de datos en el sector eléctrico es fundamental para garantizar una operación eficiente y confiable. Sin embargo, la falta de herramientas adecuadas para gestionar la creciente cantidad y diversidad de datos eléctricos puede llegar a obstaculizar el progreso y el rendimiento óptimo de este sector. Por lo tanto, es crucial abordar este problema mediante nuevas herramientas como técnicas de *Big Data.*

El uso de técnicas de *Big Data* permite gestionar y analizar eficientemente grandes volúmenes de datos, lo que brindará una comprensión más profunda de la operación y el rendimiento de los diferentes actores del sistema eléctrico,ayudando a identificar patrones, tendencias y anomalías clave, facilitando la toma de decisiones informadas y la implementación de estrategias de mejora.

De esta forma, el procesamiento de datos del sector eléctrico presenta desafíos significati-

vos que dificultan la eficiencia y la toma de decisiones fundamentadas en tiempo real. Estos desafíos se pueden resumir en dos partes principales: la cantidad o volumen de los datos y la velocidad de procesamiento.

- Volumen de datos: el sector eléctrico genera una gran cantidad de datos en tiempo real, incluyendo mediciones de consumo y generación de electricidad. Esta avalancha de datos ha superado las capacidades de las herramientas y técnicas tradicionales de procesamiento de datos, lo que dificulta la capacidad de analizar y comprender eficientemente la vasta cantidad de información generada. La gestión de grandes volúmenes de datos eléctricos se vuelve crucial para extraer conocimientos valiosos y tomar decisiones informadas.
- Velocidad de procesamiento: las bases de datos tradicionales y los sistemas de análisis pueden enfrentar dificultades en términos de escalabilidad y rendimiento al procesar grandes volúmenes de datos eléctricos en tiempo hábil. La necesidad de respuestas rápidas y en tiempo real frente a los eventos y cambios en la operación del sistema eléctrico requiere soluciones que puedan procesar los datos con eficiencia y agilidad.

Al abordar este problema, se espera mejorar la eficiencia, la confiabilidad y la sostenibilidad en el manejo de datos del sector eléctrico. Además, con este trabajo de título se espera contribuir a la aplicación de técnicas de *Big Data* en el campo de la energía, fomentando la adopción de soluciones innovadoras y el desarrollo de sistemas más inteligentes y adaptativos.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de este trabajo de título consiste en aplicar técnicas de *Big Data* para el procesamiento eficiente de datos en el sector eléctrico, con el fin de mejorar la capacidad de análisis y toma de decisiones fundamentadas. De este modo, se investigará y explorará nuevas formas de procesamiento de datos eléctricos, particularmente en el contexto de los archivos de salida del modelo PLP, utilizando técnicas de *Big Data* con la idea de desarrollar una implementación computacional que reduzca significativamente los tiempos de procesamiento de las rutinas ya existentes. Considerando de estas su eficiencia en términos de tiempo y espacio. Para, de este modo, comparar herramientas o soluciones que permitan gestionar y comprender de manera más eficiente los datos, facilitando la toma de decisiones informadas y estratégicas para optimizar la operación y el rendimiento del sistema eléctrico.

De este modo el trabajo se centra en la optimización de las rutinas del modelo PLP, abordando el desafío de procesar eficientemente grandes conjuntos de datos. A través de la implementación de herramientas de procesamiento distribuido y la adopción de formatos de archivo eficientes, con ello se busca mejorar la eficiencia general del procesamiento. El objetivo principal es explorar estrategias que permitan reducir los tiempos de ejecución y mejorar la gestión de los recursos disponibles.

1.2.2. Objetivos Específicos

A continuación, se explorarán los objetivos específicos de este trabajo de título en base a lo anteriormente mencionado.

- Realizar una revisión y análisis de las técnicas de procesamiento de datos en el sector eléctrico: se llevará a cabo una investigación detallada sobre las técnicas actuales de procesamiento de datos en el sector eléctrico, identificando las limitaciones y desafíos existentes.
- Comparar y evaluar herramientas de *Big Data*: se realizará un análisis comparativo de las herramientas y aplicaciones de *Big Data* más relevantes como por ejemplo *Spark*, *Dask*, *Hadoop* o Vaex en el contexto del sector eléctrico, considerando su capacidad para procesar eficientemente grandes volúmenes de datos eléctricos y su adaptabilidad ante los archivos y rutinas de este trabajo.
- Implementar una solución computacional: se desarrollará una solución computacional que permita el procesamiento eficiente y escalable de grandes volúmenes de datos utilizando las herramientas y técnicas de *Big Data* seleccionadas, con el objetivo de procesar los datos eléctricos de manera eficiente, optimizando los tiempos de procesamiento.
- Evaluar el rendimiento y la eficacia de la solución implementada: se realizarán pruebas y comparaciones entre la solución implementada y las técnicas tradicionales de procesamiento de datos eléctricos, analizando métricas como el tiempo de procesamiento, la escalabilidad y la calidad de los resultados obtenidos.

Capítulo 2

Marco Teórico

Para analizar el marco teórico necesario para realizar este trabajo, se debe tener en consideración ver ambas aristas que conforman el desarrollo de este proyecto, es decir, ver los datos más importantes en el sector eléctrico y los fundamentos clave de *Big Data* y análisis de datos. En primer lugar, se analizan los conceptos clave para entender el funcionamiento del sector eléctrico para entender las aplicaciones que tendrán el uso de las técnicas de procesamiento de datos.

2.1. Sector Eléctrico

2.1.1. Mercado Eléctrico

El sector eléctrico se compone de tres componentes principales: generación, transmisión y distribución. La generación se refiere a la producción de energía eléctrica a través de diversas fuentes como hidroeléctrica, solar, eólica, etc. La transmisión se encarga de transportar la energía eléctrica generada desde las centrales hasta los puntos de consumo a través de una red de líneas de transmisión y transformadores. La distribución se encarga de entregar la energía eléctrica a los usuarios finales, como hogares, industrias y comercios, a través de redes de distribución.

2.1.2. Costos Marginales

En el mercado eléctrico chileno, se utiliza el concepto de costos marginales para la valorización de las transferencias de energía entre las empresas generadoras del Sistema Eléctrico Nacional. Los costos marginales representan el valor incremental de generar una unidad adicional de energía en una central específica en un determinado intervalo de tiempo, generalmente con periodicidad horaria. Estos costos marginales son calculados teniendo en cuenta la oferta y la demanda de energía eléctrica en el mercado, y son utilizados para determinar los precios de la energía en el mercado mayorista.

2.1.3. Flujos por línea de transmisión

Los flujos por línea de transmisión son los flujos netos de potencia históricos transitados por el sistema de transmisión. Estos flujos representan la cantidad de energía eléctrica que se transfiere a través de las líneas de transmisión y los transformadores. El monitoreo y registro de los flujos por línea de transmisión es esencial para garantizar un correcto funcionamiento del sistema eléctrico, identificar posibles congestiones en la red y evaluar la capacidad de transmisión disponible. Estos datos históricos de flujos por línea de transmisión son una fuente de información importante para el análisis y la planificación del sistema eléctrico.

2.1.4. Generación por central

El reporte de generación por central proporciona información temporal, anual, mensual, diaria y horaria sobre la energía generada por cada central en un periodo de tiempo determinado. Este reporte permite tener un seguimiento detallado de la producción de energía eléctrica por parte de cada central, identificar patrones de generación, evaluar la eficiencia operativa de las centrales y realizar análisis de disponibilidad y confiabilidad del sistema eléctrico. Estos datos son fundamentales para la gestión y planificación del sector eléctrico, así como para la toma de decisiones en cuanto a la expansión y optimización de la capacidad de generación en el país.

2.2. Big Data

Big Data es el término con el que se refiere a la gestión y análisis de grandes volúmenes de datos, con el objetivo de extraer información valiosa y conocimiento a partir de ellos. Este enfoque, es particularmente relevante en el contexto de la operación del sector eléctrico, donde la creciente cantidad de datos generados, ya sea por los pequeños y medianos generadores, el futuro de las redes distribuidas o las mejoras en la toma de datos en distribución con medidores inteligentes harán que el procesamiento de datos masivos desempeñe un papel crucial para mejorar la eficiencia, la seguridad y la toma de decisiones en el futuro.

El término *Big Data* se caracteriza por las denominadas "4V": Volumen, Variedad, Velocidad y Veracidad, aunque a menudo se han añadido más dimensiones, como Valor o la Visualización de los datos. A continuación, se analizan cada una de estas dimensiones en detalle:

- **Volumen:** esta dimensión se refiere a la cantidad masiva de datos generados y almacenados continuamente. En el contexto de la operación del sector eléctrico, el volumen de datos proviene de múltiples fuentes, como medidores inteligentes, sensores en subestaciones, sistemas de control y monitoreo, entre otros. Estos datos pueden ser series temporales, datos geoespaciales y registros transaccionales que suman *terabytes* o incluso *petabytes*.

- **Variedad:** la variedad se refiere a la diversidad de tipos de datos que se manejan. Los tipos de datos se pueden clasificar en datos estructurados (por ejemplo, datos tabulares de operación o cualquier dato relacional), datos semi-estructurados (como registros, texto, datos XML o JSON) y datos no estructurados (como informes de fallos, imágenes, videos o datos de texto de redes sociales).
- **Velocidad:** la velocidad hace referencia a la rapidez con la que se generan, se transmiten y se deben analizar los datos. En entornos de *Big Data*, los datos pueden llegar en tiempo real o con una alta frecuencia. Esto exige sistemas y herramientas que puedan procesar y analizar datos en tiempo casi real para respaldar la toma de decisiones oportunas.
- **Veracidad:** la veracidad se relaciona con la precisión y la confiabilidad de los datos. Es esencial asegurarse de que los datos sean correctos y precisos, ya que decisiones basadas en datos inexactos pueden llevar a resultados erróneos.
- **Visualización:** la visualización se ha convertido en una dimensión esencial en el contexto del *Big Data* y en este trabajo en particular, puesto que implica representar datos complejos de una manera comprensible y significativa mediante técnicas como gráficos, diagramas u otras representaciones visuales. La visualización facilita la identificación de patrones, tendencias y relaciones en los datos, lo que a su vez ayuda en la toma de decisiones.

2.3. Formato de archivos

2.3.1. CSV

CSV *(Comma-Separated Values)* es un formato de archivo ampliamente utilizado para almacenar datos tabulares en forma de texto plano. En archivos CSV, los datos se organizan en filas y columnas, con columnas separadas por un carácter delimitador, típicamente una coma. Esto facilita la creación y la lectura de archivos CSV, lo que lo convierte en una opción simple y legible tanto para seres humanos como para aplicaciones informáticas.

Las ventajas de CSV radican en su simplicidad y legibilidad, lo que facilita la compartición de datos, así como en su amplia compatibilidad con una variedad de programas y lenguajes de programación. Además, los archivos CSV tienden a ser pequeños en tamaño debido a la falta de información adicional relacionada con el formato o la estructura, lo que reduce el espacio de almacenamiento necesario.

Sin embargo, al trabajar con grandes volúmenes de datos, CSV presenta desafíos. El rendimiento puede degradarse a medida que se procesan archivos CSV de gran tamaño debido a la necesidad de analizar texto completo, lo que puede ser ineficiente. Además, la falta de información sobre tipos de datos en los archivos CSV puede dar lugar a problemas de interpretación de datos, lo que puede requerir un esfuerzo adicional para garantizar la calidad de los datos. La integridad de los datos puede ser un problema, ya que no existen mecanismos integrados para garantizar la coherencia y la integridad de los datos. Finalmente, en entornos de procesamiento de *Big Data*, los archivos CSV pueden no ser la opción más eficiente en términos de rendimiento y escalabilidad.

2.3.2. JSON

JSON, cuyo acrónimo significa *"JavaScript Object Notation"*, es un formato de intercambio de datos ampliamente empleado en la programación y la comunicación entre sistemas. Se caracteriza por su estructura basada en pares clave-valor y objetos anidados, lo que lo convierte en una forma efectiva de representar información estructurada en un formato de texto. Los objetos JSON se delimitan mediante llaves , y los pares clave-valor se separan por comas. Las claves son cadenas de texto que definen los atributos, y los valores pueden ser números, booleanos, cadenas de texto, objetos JSON anidados o el valor especial null. JSON es lenguaje-agnóstico, lo que significa que es independiente del lenguaje de programación y se utiliza en una amplia variedad de aplicaciones y sistemas para el intercambio de datos. Su simplicidad y facilidad de lectura, tanto para humanos como para máquinas, han contribuido a su popularidad en el desarrollo de software y la comunicación entre componentes de aplicaciones.

2.3.3. Parquet

Parquet es un formato de archivo de almacenamiento de datos que se ha convertido en una elección destacada en el mundo del *Big Data*. Su principal característica distintiva es su diseño de almacenamiento por columnas. En lugar de guardar los datos en filas completas, este almacena los datos de una columna en secuencia. Esto se traduce en una serie de ventajas significativas, incluida una compresión más eficiente y una lectura optimizada, ya que solo se accede a las columnas relevantes en lugar de todo el conjunto de datos.

Como se menciona previamente, la compresión es otro aspecto destacado de *Parquet*. esto ya que utiliza técnicas de compresión avanzadas que reducen significativamente el espacio en disco necesario y mejoran el rendimiento, especialmente en columnas con datos repetitivos o patrones. Esta compresión no solo ahorra espacio de almacenamiento, sino que también agiliza la velocidad de lectura.

Parquet además es compatible con la evolución de esquema, lo que significa que es posible modificar la estructura de los datos sin necesidad de reescribir todo el conjunto de datos. Esto resulta útil en situaciones donde los esquemas de datos evolucionan con el tiempo y permite una mayor flexibilidad en el manejo de datos.

Además, Parquet es altamente compatible con una variedad de lenguajes de programación y herramientas de procesamiento de datos, lo que facilita la interoperabilidad y el intercambio de datos entre diferentes componentes de un ecosistema de *Big Data*. Los archivos *Parquet* se almacenan en un formato binario altamente eficiente, lo que se traduce en un rendimiento rápido y una ocupación mínima de espacio en disco. Estas características hacen que *Parquet* sea una de las elecciones más populares para el almacenamiento y análisis de datos a gran escala.

2.3.4. Feather

Feather es un formato de almacenamiento de datos binarios que se utiliza en el campo de la ciencia de datos. *Feather* fue desarrollado para abordar la necesidad de un formato de archivo eficiente y fácil de usar que permitiera el intercambio de datos entre diferentes lenguajes de programación, especialmente *Python* y *R*. Este formato se ha convertido en una herramienta valiosa con una variedad de aplicaciones, desde la ingestión y el almacenamiento de datos hasta el análisis y la visualización.

Feather es altamente eficiente en términos de rendimiento y tamaño de archivos, lo que lo hace adecuado para trabajar con grandes conjuntos de datos. Utiliza compresión y un diseño de almacenamiento columnar, lo que resulta en una lectura y escritura rápida de datos. Además, *Feather* preserva los tipos de datos y las estructuras de datos, lo que garantiza que los datos se mantengan íntegros y legibles en diferentes entornos.

2.4. Herramientas de procesamiento de *Big Data*

2.4.1. Hadoop

Hadoop es una plataforma de código abierto desarrollada desde 2005 para el procesamiento de *Big Data* y desde entonces ha sido una de las plataformas o técnicas más usadas en la industria. Esta se basa en el concepto de procesamiento distribuido, el cual, permite gestionar y analizar grandes volúmenes de datos en un entorno escalable y tolerante a fallos.

Una de las piezas fundamentales de Hadoop es el *Hadoop Distributed File System (HDFS)*. Este es un sistema de archivos distribuido diseñado específicamente para almacenar grandes cantidades de datos de manera eficiente. Lo hace dividiendo los archivos en bloques y distribuyéndolos en clústeres de servidores llamados nodos. Cada bloque se replica en varios nodos, lo que garantiza la tolerancia a fallos y la disponibilidad de datos incluso si algunos nodos fallan.

Otro concepto clave de Hadoop es el modelo de programación *MapReduce*. *MapReduce* permite procesar grandes conjuntos de datos en paralelo mediante dos etapas principales: *Map* y *Reduce*. Los programas *MapReduce* se escribían originalmente en *Java*, aunque hoy en día existen APIs para utilizar esta herramienta en otras plataformas. En particular se busca utilizar *Pydoop*, la API de *Python*. De este modo *Hadoop* se encarga de distribuir tareas y gestionar datos entre los nodos del clúster. El funcionamiento de esta técnica radica en que en la etapa de *Map*, los datos se dividen en pares clave-valor, y se aplica una función de mapeo a cada par para generar un conjunto de pares clave-valor intermedios. En la etapa de Reduce, los datos intermedios se agrupan por clave y se procesan mediante una función de reducción para generar así los resultados finales.

Por otro lado, *YARN (Yet Another Resource Negotiator)* es otra parte primordial de *Hadoop*. Esta es el forma de adminitrar recursos en *Hadoop*, siendo responsable de asignar los recursos como CPU y memoria para cada clúster. Esto permite una administración eficiente

y un uso compartido de recursos entre aplicaciones.

Actualmente, *Hadoop* no se limita al núcleo de *HDFS* y *MapReduce*, sino que se ha expandido para incluir un ecosistema de proyectos y herramientas adicionales que se integran con el núcleo de *Hadoop*. Algunos ejemplos son herramientas como *Hive*, *HBase* o *Spark*, siendo este último un framework de procesamiento de datos en memoria.

2.4.2. Spark

Apache Spark es un *framework* de procesamiento de datos de código abierto utilizado para el análisis de datos a gran escala. Es conocido por su capacidad para manejar tareas de procesamiento de datos complejas y por su rendimiento superior en comparación con enfoques más antiguos como *MapReduce*.

Una de las características más destacadas de *Apache Spark* es su capacidad para procesar datos en memoria de manera eficiente, lo que se traduce en tiempos de ejecución más rápidos. A diferencia de MapReduce, que requiere lectura y escritura frecuentes en disco, Spark puede mantener los datos en memoria, lo que agiliza significativamente el procesamiento.

Además, *Apache Spark* es compatible con diversas fuentes de datos, incluyendo HDFS *(Hadoop Distributed File System)*, bases de datos SQL y demás formatos. Esto facilita la ingestión y el procesamiento de datos desde una gran variedad de fuentes.

El *framework* ofrece una API de alto nivel disponible en varios lenguajes, como *Scala, Java, Python y R*. En este caso se utilizará Spark con la API de Python *Pyspark*.

2.4.3. Dask

Dask es un *framework* de computación paralela y distribuida de código abierto diseñado para el análisis y procesamiento de datos a gran escala. Su objetivo principal es permitir el procesamiento eficiente de conjuntos de datos que superan la capacidad de memoria de una sola máquina. Dask se basa en el lenguaje de programación Python y proporciona una abstracción de alto nivel que permite a los desarrolladores realizar operaciones paralelas y distribuidas en datos, lo que resulta en una mayor escalabilidad y rendimiento.

Dask utiliza una estructura de gráfico dirigido acíclico (DAG) para representar las tareas y sus dependencias. Esto le permite realizar cómputos de manera eficiente y gestionar la paralelización de tareas de manera automática. El *framework* se integra con diversas bibliotecas de *Python*, incluyendo *NumPy, pandas y scikit-learn,* lo que permite a los usuarios aprovechar estas herramientas familiares mientras se benefician de la escalabilidad y la paralelización de Dask.

Una de las ventajas clave de *Dask* es su capacidad para escalarse en clústeres de máquinas, lo que lo convierte en una herramienta útil para el procesamiento distribuido en entornos de *big data*. *Dask* ofrece funcionalidades para cargar, transformar y analizar grandes conjuntos de datos en paralelo, y también es adecuado para aplicaciones de machine learning distribuido

y computación a gran escala. Su flexibilidad y facilidad de uso lo convierten en una elección popular para aquellos que buscan una solución de cómputo distribuido en el ecosistema de *Python*.

2.4.4. Vaex

Vaex es una biblioteca de Python diseñada para trabajar con conjuntos de *Big Data* de manera eficiente y escalable. Su nombre proviene de la combinación de *"Vanishingly Fast"* (velocidad desvanecientemente rápida) y *"Expression"* (expresión), lo que refleja su enfoque en la velocidad y en la manipulación de datos mediante expresiones. Lo que destaca a *Vaex* es, de igual manera que *Dask*, su capacidad para trabajar con datos que superan la capacidad de la memoria RAM de una computadora, gracias a su enfoque perezoso o *"lazy"* que realiza operaciones solo cuando son necesarias.

Esta biblioteca proporciona una sintaxis de expresión similar a *NumPy* que permite realizar operaciones de filtrado, transformación y análisis de datos de una manera intuitiva. Además, Vaex optimiza el rendimiento al aprovechar el procesamiento paralelo y, en algunos casos, la distribución en *clsteres* de computadoras para acelerar las operaciones en datos de gran tamaño.

Vaex se destaca por su eficiencia en la lectura de datos desde diversas fuentes, desde archivos *CSV* hasta formatos Hadoop como *HDF5*, con un rendimiento rápido. También ofrece operaciones de agregación rápidas, lo que lo hace ideal para tareas de resumen y análisis de datos.

Además, *Vaex* se integra sin problemas con otras bibliotecas de *Python*, como *Pandas* y *NumPy*, lo que nos permite combinar estas herramientas para abordar tareas de procesamiento de manera más eficiente.

2.5. Modelos de simulación del sector eléctrico

2.5.1. Modelo PLP

El Modelo de Programación de Largo Plazo (PLP) es un software empleado por el Coordinador Eléctrico Nacional para planificar y gestionar la operación diaria del Sistema Eléctrico Nacional (SEN). Además, este modelo se utiliza para diseñar la expansión de la red de transmisión y llevar a cabo análisis a medio y largo plazo. También, diversas empresas del sector eléctrico emplean el PLP para realizar proyecciones de la operación eléctrica a futuro. A través de este modelo, es posible estimar diversos resultados de operación, como por ejemplo, la generación de energía de cada planta, prever los flujos eléctricos en las líneas de transmisión, calcular los costos marginales, y pronosticar la evolución de los niveles de agua en los embalses, entre otros resultados relevantes.

Este emplea programación dinámica dual estocástica para abordar la incertidumbre hi-

drológica. Recibiendo información relacionada con escenarios hidrológicos y generación de aperturas a través de archivos de entrada externos. El sistema de transmisión se representa mediante un flujo de potencia de corriente continua, considerando pérdidas lineales por tramos y cumpliendo con las leyes de Kirchhoff de la electricidad.

La estructura del ambiente de trabajo del Modelo PLP incluye dos interfaces, una para la creación de archivos de entrada y otra para la lectura de resultados. Recibiendo así datos para archivos de entrada y resultados de optimización.

Los archivos de entrada contienen información detallada sobre etapas, demanda, barras, líneas, centrales, curvas de costos, caudales afluentes, simulaciones, aperturas y mantenimientos. Con ellos, el modelo PLP realiza simulaciones iniciales y aperturas para representar la incertidumbre hidrológica, sorteando eventos del registro histórico.

2.5.2. Modelo PLEXOS

PLEXOS es un programa computacional desarrollado por Energy Exemplar y representado por Consultora On Energy en Latinoamérica, diseñado como una herramienta de simulación basada en la optimización. Este software se centra en la coordinación hidrotérmica y cuenta con un módulo de análisis estocástico para examinar hidrologías.

Una característica distintiva de PLEXOS es que está montado sobre el motor de MS Access, lo que facilita el acceso a las bases de datos sin necesidad de poseer una licencia del software. El programa se destaca por su capacidad ilimitada de aplicación en optimización y simulación, proporcionando amplias capacidades de modelación. Su diseño orientado a objetos simplifica el aprendizaje y la utilización de sus características, permitiendo la expansión del conjunto de funciones según sea necesario.

El modelo de objetos de PLEXOS se basa en tres elementos fundamentales: objetos (clases como generadores, líneas, embalses, etc.), asociaciones (también llamadas relaciones que definen la conexión entre objetos) y propiedades (datos de los objetos). Todas las funciones del software son accesibles a través de este modelo de objetos.

PLEXOS presenta diversas características significativas, como la determinación del tamaño y la fecha óptima de nuevas inversiones, análisis de mercado para transmisión y generación, cálculo de estrategias comerciales óptimas, valoración de sistemas de generación térmica y transmisión, proyección de precios del pool en diferentes escenarios, simulación AC de flujos de energía, cálculo de precios de nudo, congestión de líneas, y factores de penalización, entre otros.

Un aspecto relevante es su capacidad para la optimización del tiempo y la duración de los mantenimientos, así como la evaluación del impacto de los límites de emisiones y la optimización de la operación hidráulica frente a la incertidumbre. PLEXOS también se destaca por su capacidad de resolución mediante distintos optimizadores matemáticos como Mosek, Cplex, Xpress, Gurobi, entre otros.

Es esencial destacar que PLEXOS es un software en constante desarrollo, con versiones

actualizadas que se lanzan continuamente para adaptarse a las necesidades de los modelos creados.

2.6. Caracterización de rutinas y modelo PLP

Las proyecciones a largo plazo generadas por este modelo se dividen en cuatro categorías: centrales, barras, líneas y fallas, proporcionando proyecciones en diversos dominios según cada una de estas categorías. El horizonte de evaluación se divide en etapas (semanales o mensuales) y cada etapa se divide en bloques. Por ejemplo, cada etapa puede tener entre 5 a 10 bloques. A partir del archivo llamado "IndHor" los datos de proyección por bloque se pueden transformar a los datos con resolución horaria. Además, hay un archivo denominado "Centrales Info" que proporciona información adicional sobre las centrales (por ejemplo, tipo de tecnología: solar, eólica, hidráulica, etc.) y puede ser empleado para mejorar los datos a partir de los resultados de las proyecciones relacionadas con las centrales. En la figura 2.1 se puede apreciar una visualización de las rutinas realizadas con las salidas del modelo PLP.

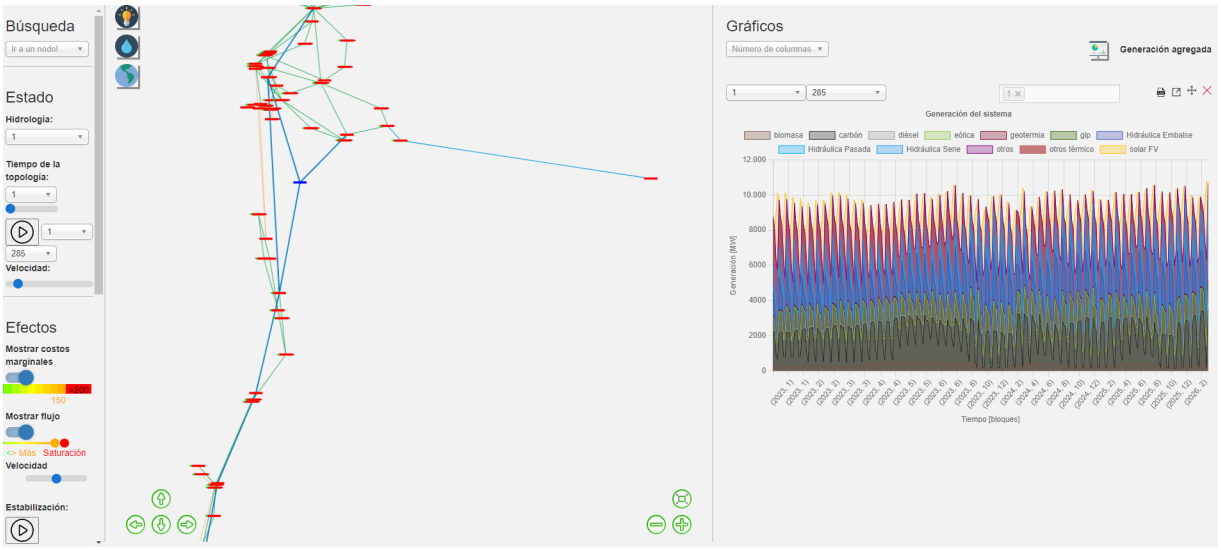

Figura 2.1: Ejemplo de Visualización Modelo PLP [3]

El formato de los archivos entregados es estructurado y estos vienen ordenados de manera tabular originalmente en formato CSV. Los datos que entrega el modelo PLP, conocidos como datos de salida son los siguientes:

- plpcen.csv: entrega la información de la generación de todas las centrales que conforman el sistema.
- plpfal.csv: entrega la información de la simulación de la operación de centrales de falla.
- plpbar.csv: entrega la información de las distintas barras del sistema, como por ejemplo, los costos marginales, la demanda o pérdidas.

- plplin.csv: contiene la información relacionada a las distintas líneas del sistema, como por ejemplo, los flujos de potencia o pérdidas.
- indhor.csv: entrega la información que permite relacionar los bloques o etapas con el tiempo en que estas ocurren, hasta en resolución horaria.
- centrales.csv: archivo auxiliar que entrega información adicional de las centrales, como por ejemplo, tipo de tecnología.

En base a estos archivos se examinarán diversos casos de estudio que involucran simulaciones del modelo PLP. Estos casos se categorizan como pequeño, mediano, grande y muy grande, según el tamaño de los archivos y la fecha de la proyección futura. A continuación, se dará un contexto y análisis exploratorio de datos de los diferentes archivos que componen la salida o resultado del modelo PLP con los cuales se construyen las diferentes rutinas de procesamiento útiles para los diferentes análisis.

2.6.1. PlpBar

Este archivo entrega la información de las proyecciones de las barras y está compuesto por 12 columnas, las cuales son las siguientes:

- Hidro: simulación de hidrología
- Bloque: etapa
- TipoEtapa: clasificación de etapa
- BarNum: número de la barra
- BarNom: nombre de la barra
- CMgBar: costo Marginal [US$/MWh]
- DemBarP: carga de la barra [MW]
- DemBarE: carga de la barra [GWh]
- PerBarP: pérdidas asociadas a la barra [MW]
- PerBarE: pérdidas asociadas a la barra [GWh]
- BarRetP: retiro [US$/h]
- BarRetE: retiro [kUS$]

Entregando este archivo, de manera resumida, una simulación para cada hidrología de la columna Hidro, separada por bloques, entregando el nombre y número de la respectiva barra, dando así para cada una de estas las proyecciones de sus costos marginales, cargas, pérdidas y retiros.

Sabiendo esto, se realiza un pequeño análisis exploratorio de datos de este archivo para cada uno de los modelos. Obteniendo así que, para el modelo pequeño, se tiene que este consta de 15 diferentes simulaciones de hidrología separadas en 260 diferentes bloques de tiempo, contando con un total de 229 barras, lo que genera un total de 893.000 filas en el archivo.

Con respecto al modelo mediano, se tiene que este cuenta con 55 diferentes hidrologías, está separado en 840 bloques y posee un total de 227 barras, generando así un archivo con un total de 10.487.400 filas.

Para el modelo grande, de manera análoga, este posee 26 diferentes hidrologías, contando con 2.760 bloques y un total de 296 barras, generando así un archivo de 21.240.960 filas.

Por último, el modelo muy grande posee 27 hidrologías y 2.670 bloques, con un total de 315 barras, obteniendo así un total de 22.708.350 filas.

	Hidro	Bloque	TipoEtapa	BarNum	BarNom	CMgBar	DemBarP	DemBarE	PerBarP	PerBarE	BarRetP	BarRetE
0	Sim 1	1	10 Bloques	1	Paposo220	37.94	2.08	0.383	1.748	0.322	78.925	14.522
1	Sim 1	2	10 Bloques	1	Paposo220	39.47	2.08	0.112	1.626	0.088	82.092	4.433
2	Sim 1	3	10 Bloques	1	Paposo220	41.93	2.12	0.136	1.008	0.065	88.895	5.689
3	Sim 1	4	10 Bloques	1	Paposo220	42.68	2.12	0.059	0.848	0.024	90.485	2.534
4	Sim 1	5	10 Bloques	1	Paposo220	50.12	2.16	0.112	0.121	0.006	108.263	5.630

Figura 2.2: Muestra de archivo Plpbar

De esta manera, en la figura 2.2, se puede apreciar una muestra del archivo de salida de Plpbar mostrando cinco filas, donde se pueden apreciar las diversas columnas mencionadas anteriormente, como el nombre de la respectiva barra, los bloques, etc.

2.6.2. PlpCen

Este archivo entrega la proyección de las diferentes centrales, contando con 12 columnas, las cuales son las siguientes:

- Hidro: simulación de hidrología
- Bloque: etapa
- TipoEtapa: clasificación de etapa
- CenNum: número de la central
- CenNom: nombre de la central
- CenTip: tipo de central
- CenBar: barra de la central (número)
- BarNom: nombre de la barra de la central

- CenQgen: caudal turbinado [m^3/s]
- CenPgen: potencia generada [MW]
- CenEgen: energía generada [GWh]
- CenInyP: inyección [US$/h]
- CenInyE: inyección [kUS$]
- CenCVar: costos variables
- CenCostOp: costos operacionales
- CenPMax: potencia máxima

Entregando así una simulación según hidrología en columna Hidro, separada por bloques. Se proporciona el nombre, número, tipo y barra de la respectiva central, entregando así su caudal turbinado, potencia generada, energía generada, inyecciones, costos y potencia máxima.

A continuación, se detalla el análisis exploratorio de los diferentes modelos. Iniciando por el pequeño, el cual cuenta con 15 hidrologías, 260 bloques y 1.678 centrales en total, generando así un archivo de 6.544.200 filas.

Por otra parte, el modelo mediano posee un total de simulaciones de hidrología de 55, mientras que su número de bloques llega a los 840 y posee 349 centrales en total, dando un número de filas de 16.123.800.

Luego, el modelo grande posee un total de 26 hidrologías, contando además con 2.760 bloques y un total de 1.014 centrales 72.764.640 filas en el archivo de salida.

Finalmente, el modelo muy grande posee un total de 27 hidrologías y además 2.670 bloques, contando también con 1575 centrales, completando así al archivo con 113.541.750 filas.

	Hidro	Bloque	TipoEtapa	CenNum	CenNom	CenTip	CenBar	BarNom	CenQgen	CenPgen	CenEgen	CenInyP	CenInyE	CenRen	CenCVar	CenCostOp	CenPMax
0	Sim 1	1	5 Bloques	2	CIPRESES	Emb	123	Cipreses154	35.56	100.57	2.11	14848.28	311.81	2.83	0.0	0.0	100.57
1	Sim 1	2	5 Bloques	2	CIPRESES	Emb	123	Cipreses154	35.56	100.57	2.11	14795.63	310.71	2.83	0.0	0.0	100.57
2	Sim 1	3	5 Bloques	2	CIPRESES	Emb	123	Cipreses154	35.56	100.57	3.52	14666.29	513.32	2.83	0.0	0.0	100.57
3	Sim 1	4	5 Bloques	2	CIPRESES	Emb	123	Cipreses154	8.37	23.68	1.49	2944.52	185.50	2.83	0.0	0.0	100.57
4	Sim 1	5	5 Bloques	2	CIPRESES	Emb	123	Cipreses154	35.56	100.57	2.82	15107.95	423.02	2.83	0.0	0.0	100.57

Figura 2.3: Muestra de archivo Plpcen

De este modo, en la figura 2.3, se puede apreciar una muestra con las primeras cinco filas de uno de los archivos de Plpcen, donde se pueden ver las diferentes columnas descritas previamente.

2.6.3. PlpLin

Este archivo tiene la información de las diferentes líneas y está compuesto por 17 columnas, las cuales son las siguientes:

- Hidro: simulación
- Bloque: etapa
- TipoEtapa: clasificación de etapa
- LinNum: número de la línea
- LinNom: nombre de la línea
- BarA: barra extremo A
- BarB: barra extremo B
- LinFluP: flujo de potencia [MW]
- LinFluE: flujo de energía [GWh]
- LinPerP: aproximación lineal de pérdidas [MW]
- LinPerE: aproximación lineal de pérdidas [GWh]
- LinPer2P: estimación cuadrática de las pérdidas [MW]
- LinPer2E: estimación cuadrática de las pérdidas [GWh]
- LinITP: ingresos por tramos [US$/h]
- LinITE: ingresos por tramos [kUS$]

Entregando así una simulación según hidrología en columna Hidro, separada por bloques, entregando el nombre y número, de la respectiva línea, entregando así que barras conecta en sus extremos, su flujo de energía, aproximaciones lineales y cuadráticas de pérdidas, ingresos por tramo, inyecciones, costos y potencia máxima.

Para el análisis exploratorio de datos, el primer modelo, es decir, el modelo pequeño posee 15 diferentes simulaciones de hidrología con 260 bloques y 325 líneas, con un total de 1.267.500 filas.

Por otro lado, el modelo mediano posee un total de 55 simulaciones de hidrología con 840 bloques y 365 líneas, dando así un archivo con 16.863.000 filas.

Luego, el modelo grande posee un total de 26 hidrologías, además cuenta con 2760 bloques y 409 líneas. De este modo el archivo de salida cuenta con 29.349.840 filas.

Finalmente, el modelo muy grande posee 27 hidrologías diferentes, además de 2760 bloques, junto con ello este cuenta con 440 líneas diferentes, para hacer un total de 31.719.600 filas.

	Hidro	Bloque	TipoEtapa	LinNum	LinNom	BarA	BarB	LinFluP	LinFluE	LinFluMax	LinUso	LinPerP	LinPerE	LinPer2P	LinPer2E	LinITP	LinITE
0	Sim 1	1	5 Bloques	1	Andes220->Oeste220	2	68	-1.39	-0.03	290.0	0.48	0.01	0.00	0.00	0.00	-1.39	-0.03
1	Sim 1	2	5 Bloques	1	Andes220->Oeste220	2	68	-1.40	-0.03	290.0	0.48	0.01	0.00	0.00	0.00	-1.33	-0.03
2	Sim 1	3	5 Bloques	1	Andes220->Oeste220	2	68	25.11	0.88	290.0	8.66	0.17	0.01	0.04	0.00	-4.88	-0.17
3	Sim 1	4	5 Bloques	1	Andes220->Oeste220	2	68	55.78	3.51	290.0	19.24	0.38	0.02	0.22	0.01	-5.8	-0.37
4	Sim 1	5	5 Bloques	1	Andes220->Oeste220	2	68	-0.40	-0.01	290.0	0.14	0.00	0.00	0.00	0.00	-0.42	-0.01

Figura 2.4: Muestra de archivo Plplin

Así, en la figura 2.4 se puede ver la muestra del archivo de salida de Plplin con sus diferentes columnas, teniendo los primeros cinco bloques para la línea "Andes220− >Oeste220".

2.6.4. PlpFal

Este archivo contiene las salidas e información de las centrales de falla y está compuesto por 11 columnas, las cuales son las siguientes:

- Hidro: simulación de hidrología
- Bloque: etapa
- TipoEtapa: clasificación de etapa
- CenNum: número de la central
- CenNom: nombre de la central
- CenTip: tipo de central
- CenBar: barra de la central (número)
- BarNom: nombre de la barra de la central
- CenQgen: caudal turbinado $[m^3/s]$
- CenPgen: potencia generada [MW]
- CenEgen: energía generada [GWh]

Entregando, de manera análoga al archivo PlpCen, una simulación según hidrología en columna Hidro, separada por bloques, dando el nombre, número, tipo y barra de la respectiva central de falla, entregando así su caudal turbinado, potencia generada y energía generada.

En los casos de modelos pequeño, mediano y muy grande como se pudo ver en la Tabla 4.1 estos archivos pesan solo 1 [KB] puesto que están vacíos y no contienen ninguna central de falla.

	Hidro	Bloque	TipoEtapa	CenNum	CenNom	CenTip	CenBar	BarNom	CenPgen	CenEgen
0	Sim 1	2181	10 Bloques	1298	FALLA_240	Fal	240	Melipulli220	0.91	0.16
1	Sim 1	2191	10 Bloques	1298	FALLA_240	Fal	240	Melipulli220	4.98	0.73
2	Sim 1	2197	10 Bloques	1298	FALLA_240	Fal	240	Melipulli220	5.48	0.17
3	Sim 1	2206	10 Bloques	1298	FALLA_240	Fal	240	Melipulli220	4.63	0.11
4	Sim 1	2225	10 Bloques	1298	FALLA_240	Fal	240	Melipulli220	5.46	0.34

Figura 2.5: Muestra de archivo Plpfal

De esta manera, se puede apreciar en la imágen 2.5 las once columnas de las centrales de falla descritas anteriormente con una muestra de cinco filas del modelo grande.

2.6.5. Centrales

Este archivo entrega información adicional con respecto a cada una de las centrales del modelo, este cuenta con 12 columnas, las cuales son las siguientes:

- cen_name: nombre de la central
- cen_type: tipo de la central
- emission_factor: factor de emisión
- zone: zona de la central
- is_ernc: indica si la central es de energías renovables
- cenbar: barra de la central
- censis: sistema eléctrico al que pertenece la central
- commitement
- pmax: potencia máxima de la central
- pmin: potencia mínima de la central

Cada uno de los archivos posee una misma cantidad de filas que dependerá y es equivalente a la cantidad de centrales de cada modelo.

	cen_name	cen_type	emission_factor	zone	is_ernc	cenbar	censis	commitement	pmax	pmin
0	Sum_Isla_Mina	otros_hidro	0	SEN	0	0	SEN	0	9999.00	0
1	ALTOPOLC	otros_hidro	0	SEN	0	0	SEN	0	9999.00	0
2	LMAULE	hidraulica_embalse	0	SEN	0	0	SEN	0	100.00	0
3	LOS_CONDORES	hidraulica_serie	0	SEN	0	54	SEN	0	150.00	0
4	B_LaMina	hidraulica_serie	0	SEN	0	0	SEN	0	6.96	0

Figura 2.6: Muestra de archivo Centrales

Así, de manera análoga a los demás archivos, se enseña en la figura 2.6 una muestra de cinco filas del archivo centrales, mostrando de manera visual lo señalado anteriormente.

2.6.6. Indhor

Este archivo tiene la información de en qué momentos o fechas sucede cada uno de los bloques, teniendo así este una resolución horaria, mostrando el año, mes, día y hora en las que se están ejecutando cada uno de los bloques.

- Bloque: etapa
- Año: año del respectivo bloque
- Mes: mes del respectivo bloque
- Día: día del respectivo bloque
- Hora: hora del respectivo bloque

Este archivo es el utilizado para realizar la unión o cruce de cada uno de los archivos de salida del modelo para poder transformarlos a un formato horario, donde para cada uno de estos se posee una fila por cada hora del rango de fechas mencionado previamente y se indica que bloque está actuando durante esa hora en específico.

	Año	Mes	Dia	Hora	Bloque
0	2023	3	25	1	1
1	2023	3	25	2	1
2	2023	3	25	3	1
3	2023	3	25	4	2
4	2023	3	25	5	2

Figura 2.7: Muestra de archivo Indhor

De la figura 2.7, se puede apreciar una muestra de cinco filas del archivo Inhdor, donde se puede apreciar la unión que existe entre los bloques y cada hora, donde para cada hora dentro del rango de la simulación, esta se asocia a uno de los bloques.

2.6.7. Rutinas

Como se mencionó previamente, gracias al uso de los archivos de salida del modelo PLP, se puede extraer información relevante con respecto a la operación del sector eléctrico, específicamente para el desarrollo de este trabajo. Se facilita un modelo base de propuestas de rutinas que hacen posible extraer los siguientes resultados:

Costos marginales por barra

Esta rutina, de manera simplificada, permite calcular y graficar los costos marginales por cada barra, calculando los percentiles mensuales: 0, 20, 80 y 100, además del costo marginal promedio mensual de cada barra.

Listing 2.1: Calcular costos marginales por barra

```
def Costos_Marginales():
    datos_bar = read_file(plpbar_path,
            columnas=['Hidro', 'Bloque', 'BarNom', 'CMgBar'])
    barras = obtener_valores_unicos(datos_bar['BarNom'])
    indhor = read_file(indhor_path)

    for barra in barras:
        data_barraTx = filtrar(datos_bar, 'BarNom' == barra)
        data_barraTx = merge(indhor, data_barraTx, 'Bloque')
        data_barraTx = filtrar(data_barraTx,
                data_barraTx['Hidro'] != 'MEDIA')

        agrupado = agrupar_datos(data_barraTx, ['Anho', 'Mes'])

        percentiles = calcular_percentiles(agrupado['CMgBar'],
                [0, 0.2, 0.8, 1])
        promedio = calcular_promedio(agrupado['CMgBar'])

        resultado = unir(percentiles, promedio)

        plot(resultado)
```

Para realizar aquello se debe diseñar una rutina como la del pseudocódigo anterior, donde los pasos a seguir son, en primer lugar, leer los archivos Plpbar e Indhor, filtrar la columna "BarNom" para obtener una lista con todas las barras del archivo, luego, para cada una de las barras se debe hacer la transformación a resolución horaria, eliminar la simulación de hidrología "Media" puesto que esta última es el promedio de las demás simulaciones, agrupar el resultado por cada mes de la simulación y graficar, cabe destacar la gran cantidad de barras y tamaño de los cruces al hacer la transformación horaria, lo que genera bastante trabajo computacional.

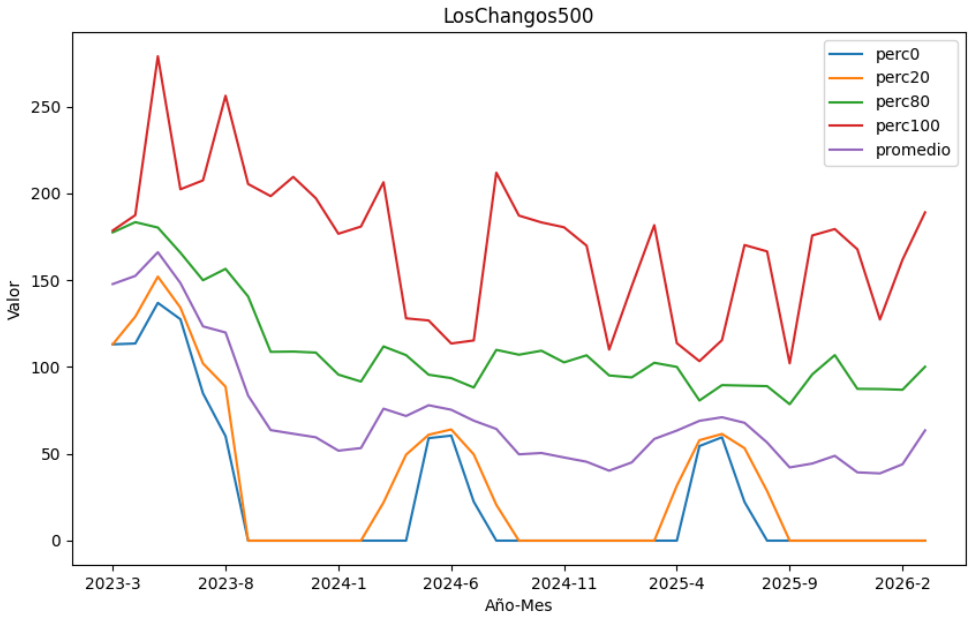

Figura 2.8: Ejemplo de resultado para rutina de gráfico de costos marginales por barra

Con el procesamiento mencionado anteriormente, se puede obtener un resultado como el que se ve en la figura 2.8 donde se observan los diferentes percentiles proyectados para la barra "LosChangos500" del modelo pequeño, teniendo un simulación hasta 2026. Obteniendo así un gráfico análogo para cada una de las barras de cada modelo.

Costos marginales día y noche

Esta rutina se encarga de calcular y almacenar los costos marginales promedio anuales separados entre día y noche, considerando el día entre las 6:00 AM hasta las 19:00 PM.

Listing 2.2: Calcular costos marginales día y noche

```
def costos_marginales_promedio_dia_noche():
    datos_bar = read_file(plpbar_path,
            columnas=['Hidro', 'Bloque', 'BarNom', 'CMgBar'])

    barras = obtener_valores_unicos(datos_bar['BarNom'])

    indhor = read_file(indhor_path)

    for barra in barras:
        datos_bar_tx = filtrar(datos_bar, 'BarNom' == barra)
        datos_bar_tx = merge(indhor, datos_bar_tx, by='Bloque')
        datos_bar_tx = filtrar(datos_bar_tx,
                datos_bar_tx['Hidro'] != 'MEDIA')
```

```
        datos_noche = filtrar((datos_bar_tx['Hora'] >= 19)
                or (datos_bar_tx['Hora'] <= 6))
        datos_dia = filtrar((datos_bar_tx['Hora'] < 19)
                & (datos_bar_tx['Hora'] > 6))

        promedio_noche = datos_noche[['Anho', 'CMgBar']]
                .groupby(['Anho']).mean()
        promedio_dia = datos_dia[['Anho', 'CMgBar']]
                .groupby(['Anho']).mean()

    write_csv(promedio_dia)
    write_csv(datos_noche)
```

Para obtener los resultados de esta rutina se deben cargar los archivos Plpbar e Indhor y de manera análoga a la rutina de graficar los costos marginales por barra, se deben encontrar cada una de las barras para iterar sobre ellas. Luego, para cada bara se debe hacer la unión entre el archivo de salida e Indhor para transformar este a resolución horaria, a continuación se debe filtrar según la hora para obtener los datos de día y los datos de noche, separando por las horas dichas anteriormente. Finalmente, se deben agrupar los resultados por año y calcular el promedio para cada crupo y así poder obtener el resultado deseado.

	BarNom	2023.0	2024.0	2025.0	2026.0	2027.0	2028.0	2029.0	2030.0	2031.0	...	2036.0	2037.0	2038.0	2039
0	Paposo220	26.411599	16.260701	7.533164	9.000602	7.787021	8.512081	14.224840	16.072721	13.596940	...	15.401776	13.388534	14.430430	16.1483
1	DAlmagro220	26.943040	16.544306	7.659480	9.153430	7.917192	8.654324	14.470842	16.351616	13.832380	...	15.671063	13.621380	14.682152	16.4305
2	Illapa220	27.238788	16.616044	7.667792	9.168499	7.934854	8.665522	14.555613	16.461047	13.911941	...	15.779128	13.707553	14.781141	16.5455
3	CarreraPinto220	26.919572	16.549520	7.644536	9.108939	7.877776	8.612534	14.453218	16.341187	13.827355	...	15.841134	13.865267	14.965472	16.7337
4	Cardones220	28.285223	16.814411	7.743906	9.231225	7.967307	8.714715	14.689721	16.618010	14.079909	...	16.286935	14.457714	15.648882	17.4784

Figura 2.9: Ejemplo de resultado para rutina de costos marginales promedio en el día o noche (Ejemplo día)

De esta manera, con esta rutina se logra obtener para cada barra los costos promedios anuales, separados según el día o noche, tal como se puede ver en la figura 2.9 donde se puede ver un ejemplo de la muestra del dataframe para los resultados de día en el modelo grande.

Factor de planta anual y mensual

Esta rutina permite crear un archivo con los factores de planta anual o mensual de cada central.

Listing 2.3: Calcular factor de planta anual

```
def factor_planta_anual():
    indhor_anual = read_file(indhor_path,
        columns=["Bloque","Anho"])

    centrales = read_file(centrales_path,
        columns=["cen_name","cen_type","emission_factor","pmax"])
```

```
plpcen = read_file(plpcen_path,
    columns=["Hidro","Bloque","CenNom","CenEgen"])

resultado = merge(plpcen="CenNom", centrales'cen_name')

resultado = merge(resultado, indhor_anual, by="Bloque")

# Calcular emisiones
resultado["emision"] =
    resultado["emission_factor"] * resultado["CenEgen"]

# Calcular el factor de planta
resultado_factor_planta = resultado.
    groupby(['Anho', 'Hidro', 'cen_name', 'cen_type',
    'pmax']).sum(['CenEgen', 'emision'])

resultado_factor_planta["Factor_Planta"] =
    (resultado_factor_planta["CenEgen"] * 1000)
    / (resultado_factor_planta["pmax"] * 8760)

write_csv(resultado_factor_planta)
```

Para el desarrollo de esta rutina se necesitan leer los archivos IndHor, Plpcen y Centrales, y realizar el respectivo cruce entre los tres archivos. Esta rutina no tiene realmente, resolución horaria, sino que calcula el factor de planta y las emisiones de manera anual (o mensual) por lo que los cruces son un poco más pequeños. Con los cruces realizados se obtiene el factor de emisión y potencia máxima de cada central, con lo que gracias a estos y los valores propios de plpcen se logra con las funciones señaladas en el pseudocódigo obtener los factores de planta y emisiones respectivas de cada planta. Esta rutina es completamente análoga para el factor de planta mensual, solo variando que, a la hora de calcular el factor de planta, este debe ser considerar en la ecuación la cantidad de horas de cada mes y no el total anual, teniendo que reemplazar el 8760 de la ecuación por la cantidad de horas del mes que corresponda.

	year	hidro	central	energy	emission	factor_planta
0	2023	MEDIA	ABANICO	275.97	0.0	0.123543
1	2023	MEDIA	ABASTIBLE_CONCON_FV	0.20	0.0	0.000023
2	2023	MEDIA	AEROPUERTO_FV	7.84	0.0	0.001398
3	2023	MEDIA	AGUAS_BLANCAS	0.00	0.0	0.000000
4	2023	MEDIA	AGUAS_CLARAS_FV	0.00	0.0	0.000000

Figura 2.10: Ejemplo de resultado para rutina de factor de planta anual por central

De esta manera, se logra crear las rutinas de factor de planta entregando un archivo csv como el de la figura 2.10 mostrando esta los resultados de muestra del cálculo del factor de

planta anual.

Flujos por línea

Esta rutina calcula y grafica los flujos de energía por cada línea, calculando los percentiles mensuales: 0, 20, 80 y 100 además del flujo promedio mensual de cada línea.

Listing 2.4: Calcular flujos por linea

```
def flujos_linea():
    datos_lineas = read_file(plplin_path,
        columns=['Hidro', 'Bloque', 'LinNom',
        'LinFluP', 'LinFluMax'])

    indhor = read_file(indhor_path)

    lista_lineas = obtener_valores_unicos(datos_lineas['LinNom'])

    for linea in lista_lineas:
        # Filtrar datos para la linea actual
        datos_lineaTx = filtrar(datos_lineas, 'LinNom' == linea)
        datos_lineaTx = merge(indhor, datos_lineaTx, en='Bloque')
        datos_lineaTx = filtrar(datos_lineaTx,
            datos_lineaTx['Hidro'] != 'MEDIA')

        # Calcular flujo maximo
        fluMax = seleccionar_columnas(datos_lineaTx,
            ['Anho', 'Mes', 'LinFluMax'])

        # Agrupar por anho y mes
        agrupado = agrupar_datos(datos_lineaTx, ['Anho', 'Mes'])

        # Calcular el promedio y los cuantiles del flujo
        percentliles = calcular_percentiles(agrupado, 'LinFluP',
            [0, 0.2, 0.8, 1]))

        promedio = calcular_promedio(agrupado['LinFluP'])

        resultado = unir(percentiles, promedio)

        plot(resultado)
```

Esta rutina está diseñada de manera totalmente análoga al cálculo de costos marginales por barra, necesitando del archivo Plplin para obtener los datos de cada línea y estos pasarlos a resolución horaria, para de este modo calcular los percentiles y promedio del flujo de cada línea además del flujo máximo, para luego terminar graficando estos en el resultado final.

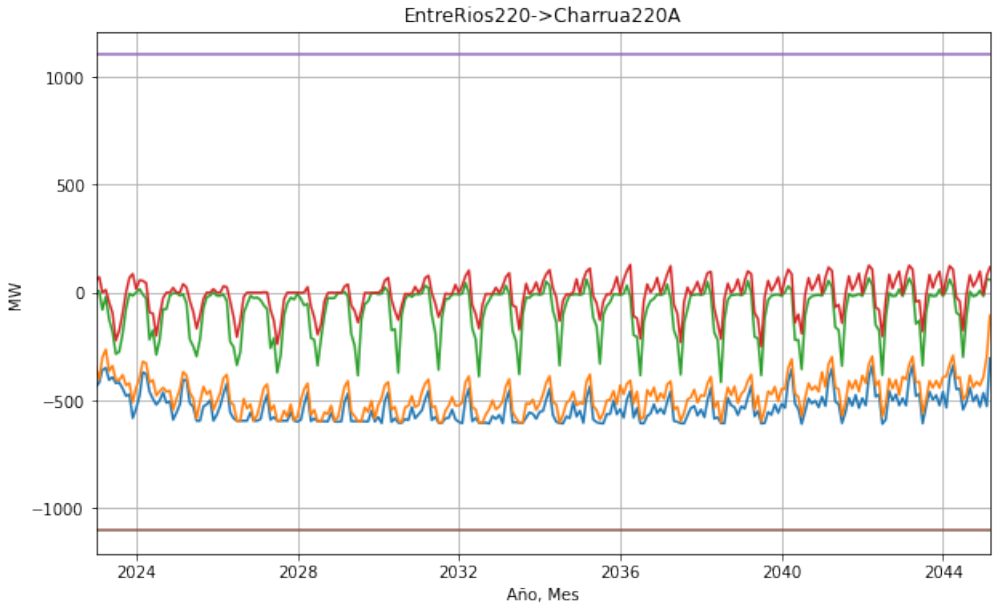

Figura 2.11: Ejemplo de resultado para rutina de flujos por línea

Obteniendo un resultado como el que se aprecia en la figura 2.11 donde se aprecian los diferentes percentiles par la línea "EntreRios220− >Charrua220A", siendo este el resultado particular de esta barra, entregando un gráfico para cada línea de cada modelo.

Generación según tecnología

Esta rutina permite en términos simples graficar la generación de energía anual por cada tipo de tecnología.

Listing 2.5: Calcular generación por tecnología

```
def tecnologia():
    indhor_anual = read_file(indhor_path,
        columns=["Bloque","Anho","Mes"])
    indhor_anual = eliminar_duplicados(indhor_anual, on="Bloque")

    centrales = read_file(centrales_path,
        columns=["cen_name","cen_type","emission_factor","pmax"])

    plpcen = read_file(plpcen_path,
        columns=["Hidro","Bloque","CenNom","CenEgen"])

    result = merge(plpcen="CenNom", centrales='cen_name')

    result = merge(result, indhor_anual, by="Bloque")

    result["emision"] = multiplicar(
```

```
    result["emission_factor"] * result["CenEgen"])

# Calcular energia y emisiones generadas segun la agrupacion
resultado_cent = result.
    gropuby(['Anho', 'Hidro', 'cen_type']).
    sum(['CenEgen', 'emision'])

plpfal = read_file(plpfal_path,
    columns=["Hidro","Bloque","CenEgen","CenNom"])

plpfal = merge(plpfal, indhor_anual, by="Bloque")

# Calcular energia y emisiones para fallas
resultado_fallas = plpfal
    .gropuby(['Anho', 'Hidro', 'CenNom'])
    .sum(['CenEgen'])

resultado_final = unir(resultado_cent, resultado_fallas)

plot(resultado_final)
```

De esta forma, de manera más detallada, la rutina consiste en cargar los archivos Centrales, Indhor, PlpCen y PlpFal para de estos últimos dos, al ser transformados a resolución horaria realizar la suma de su energía generada agrupando los datos según el tipo de tecnología que ocupa cada central, finalmente se unen los datos obtenidos de Plpcen con los de Plpfal graficando el resultado final.

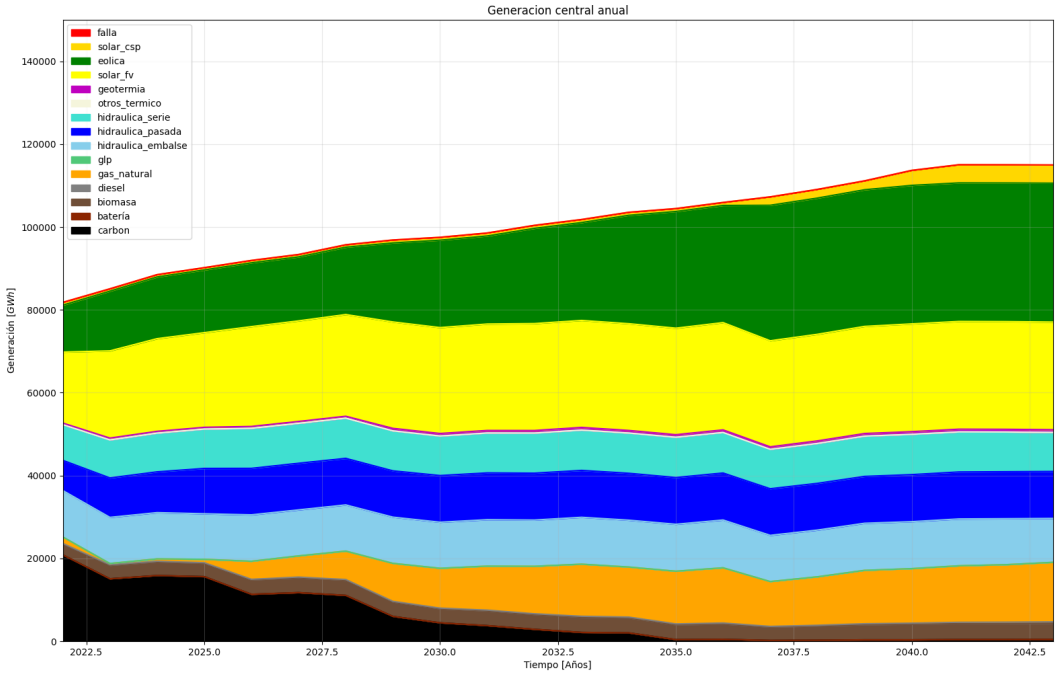

Figura 2.12: Ejemplo de resultado para rutina de factor de generación por tecnología

El resultado entregado por esta rutina consiste en lo anteriormente mencionado, como se puede ver en la figura 2.12 se puede ver la proyección del aporte que realizará cada tipo de central en el total de la generación, viéndose tecnologías como el carbón, eólica, solar fotovoltaica, etc. Con una proyección hasta 2044.

Pérdidas

Esta rutina es la encargada de graficar las pérdidas anuales totales en las barras.

Listing 2.6: Calcular generación por tecnología

```
def perdidas():
    indhor_anual = read_file(indhor_path
        columns=["Bloque","Anho","Mes"])
    indhor_anual = eliminar_duplicados(indhor_anual, on="Bloque")

    plpbar = read_file(plpbar_path,
        columns=["Hidro","Bloque","BarNom","PerBarE"])

    result = merge(plpbar, indhor_anual, by="Bloque")

    # Calcular perdidas
    perdida = result.groupby(['Anho', 'Hidro']).sum(['PerBarE'])
```

```
write_csv(perdida)
```

Para lograr generar el gráfico se realiza un merge entre el archivo Plpbar e indhor, pero con resolución anual, para luego calcular la sumatoria total de las pérdidas según cada simulación de hidrología.

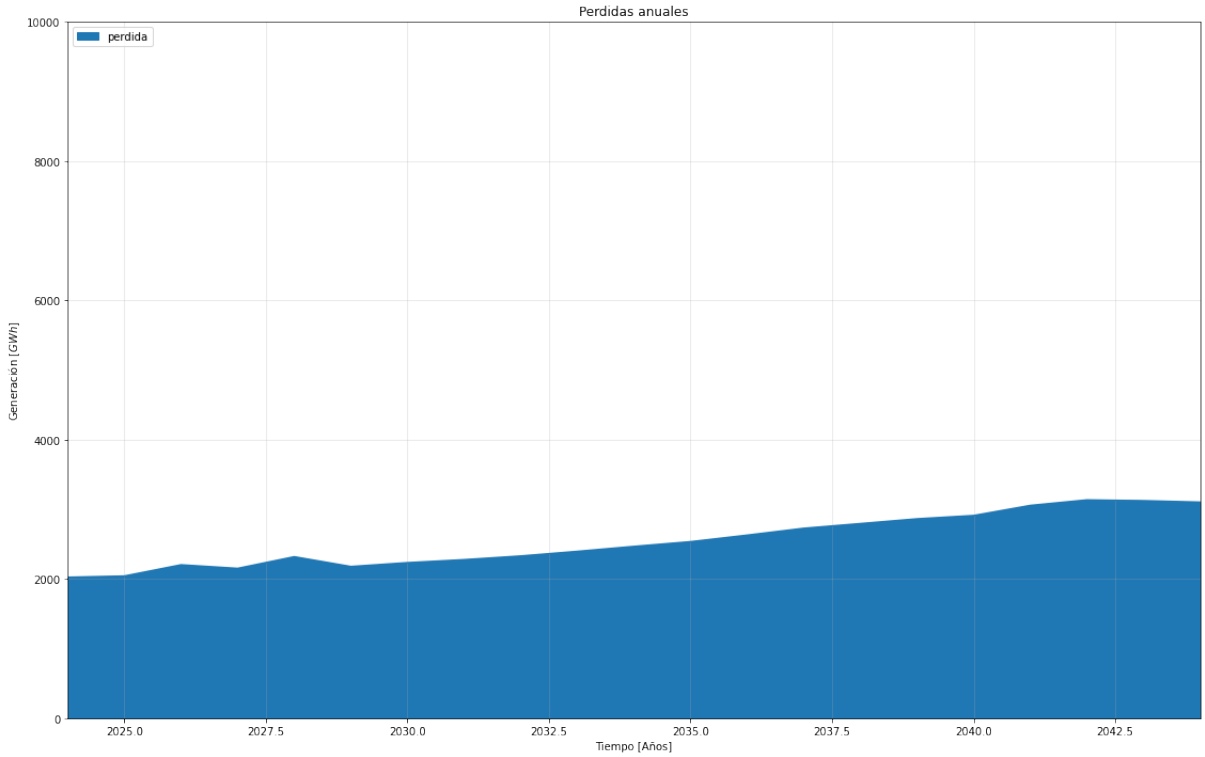

Figura 2.13: Ejemplo de resultado para rutina de pérdidas anuales

Dando como resultado el gráfico de la figura 2.13 donde, tal como se indica en la rutina, se muestran la sumatoria de pérdidas anuales.

Tiempo de ejecución rutinas

Como primera iteración, se intenta analizar el problema en su estado actual. Para lograr aquello, estas rutinas están diseñadas con técnicas tradicionales, utilizando la librería Pandas y para los archivos más grandes leyendo estos línea por línea y almacenando su información en forma de diccionario en Python. De este modo, para ver el tiempo que se demora en ejecutar cada una de estas rutinas y así tener un primer acercamiento del problema y entender de primera mano este, se ejecuta cada una de estas rutinas por separado. Midiendo su respectivo tiempo de ejecución.

Tabla 2.1: Tiempo de ejecución rutinas originales modelo PLP

Rutina / Modelo	Pequeño	Mediano	Grande	Muy Grande
Graficar Costos Marginales por barra	128 [s]	1005 [s]	1301 [s]	1398 [s]
Graficar Flujos de energía por linea	196 [s]	1641 [s]	1712 [s]	2117 [s]
Costos Marginales Dia y Noche	46 [s]	494 [s]	620 [s]	782 [s]
Factor Planta Anual	21 [s]	50 [s]	216 [s]	342 [s]
Factor Planta Mensual	22 [s]	56 [s]	232 [s]	362 [s]
Grafico generacion anual	16 [s]	42 [s]	193 [s]	304 [s]
Grafico pérdidas	18 [s]	62 [s]	203 [s]	350 [s]

De este modo, la primera prueba a realizar es probar las rutinas originales con el modelo pequeño, observando de la Tabla 2.1 que los resultados demoran entre 16 y 196 segundos según cada rutina. Las rutinas que más tiempo demoran son las que se pide graficar costos marginales por barra y flujos por líneas, superando ambas rutinas los 100 sgundos, mientras que todas las demás solo rondaron los 20 segundos, notando lo eficiente que son estas para rutinas con proyecciones más cortas o pocos datos en general, tardando solo 447 segundos en la ejecución completa de todas las rutinas.

A continuación, se realiza la segunda prueba con el modelo mediano, donde los tiempos escalan rápidamente al realizar la rutina con 10 años más de proyección, los tiempos en general escalan cerca de 10 veces en comparación con el modelo pequeño, tardando los gráficos de flujos por línea más de 25 minutos y la prueba completa con todas las rutinas en aproximadamente 55 minutos, notando cómo al escarlar el tamaño de los modelos, los tiempos se hacen cada vez más elevados.

Luego, se obtienen los resultados del modelo grande, donde se observa cómo el modelo tarda más de 20 minutos en las rutinas más costosas computacionalmente, tardando cerca de 30 minutos en la rutina más pesada. Demorando aproximadamente 75 minutos en la ejecución de todas las rutinas, si bien el aumento no es tan significativo como del caso pequeño al mediano, el crecimiento de los archivos tampoco y se comienza a notar cómo las variaciones de tamaño, a pesar de ser más pequeñas, pueden ir afectando el tiempo de procesamiento mientras más escalan los datos.

Finalmente, se procesan los archivos del modelo muy grande en las rutinas originales, donde se ve cómo para un aumento mucho menor de datos y dos proyecciones de prácticamente el mismo tiempo los aumentos siguen siendo significativos, superando el tiempo de la rutina de graficar flujos por línea los 30 minutos y teniendo un aumento constante en todas las demás rutinas. El tiempo en este caso total fue de 89 minutos, cercano a la hora y media, siendo esto ya desde el caso mediano tiempos demasiado elevados para un procesamiento eficiente, constante y escalable de los datos.

De este modo, la primera prueba consistió en mejorar los tiempos de ejecución de estas rutinas, evitando el uso de ciclos for anidados e intentando realizar todo el procesamiento en una única secuencia de funciones propuestas en Pandas de manera optimizada y de esta manera tener un mejor caso base con el que comparar las herramientas de *Big Data*.

2.7. Estado del Arte

En los últimos años, se ha observado un creciente interés en el campo de las plataformas de *Big Data* y su aplicación en el sector eléctrico. A continuación, se presentan algunos trabajos relevantes que abordan la gestión y el procesamiento de datos masivos en el contexto de la operación del sector eléctrico, así como la importancia del *Big Data* en el desarrollo de las *Smart Grids*, las cuales si bien no son el foco de este proyecto, si son parte del futuro cercano del sector eléctrico y su uso hará un escalamiento importante en la cantidad de datos que maneje el sistema. A continuación se detallarán los libros y papers investigados que servirán de base en el desarrollo de este trabajo:

Evolution of knowledge mining from data in power systems: The Big Data Analytics breakthrough (Javier Domínguez , Álvaro Pradob, Pablo Arboleya, Vladimir Terzijac. 2023): este estudio examina la evolución de la extracción de datos en sistemas de energía desde 1980 hasta la fecha, y destaca las tendencias actuales en *Big Data Analytics* en el área. Se exploran las transformaciones y los avances tecnológicos que han llevado al surgimiento de técnicas avanzadas de análisis de datos másivos en el contexto de los sistemas de energía.

Role of Big Data Analytics in Power System Application (Ravi V Angadi, P. S Venkataramu, Suresh Babu Daram. 2020): este trabajo investiga el papel del análisis de Big Data en diversas industrias y explora cómo estas aplicaciones se extienden al sistema de energía. Se examinan los beneficios y las oportunidades que brinda el uso de técnicas de *Big Data Analytics* en el sector eléctrico, como la optimización de la operación, la gestión de la demanda y la detección de fallas en tiempo real.

Big Data Application in Power Systems (Reza Arghandeh, Yuxun Zhou. 2018): en este libro, expertos en el campo de la energía se reúnen para compartir su comprensión y discutir las aplicaciones de *Big Data* en el diagnóstico, la operación y el control de los sistemas de energía. Este se enfoca en el manejo de *Big Data* y los enfoques de aprendizaje automático para procesar datos de alta dimensión, heterogéneos y espaciotemporales.

Big Data Platforms and Applications (Florin Pop, Gabriel Neagu. 2021): este libro ofrece una revisión exhaustiva de los últimos desarrollos en plataformas de *Big Data* y propone soluciones tecnológicas de vanguardia para abordar problemas importantes en el procesamiento de datos eléctricos. El enfoque principal es mejorar en el procesamiento de gestión de recursos y datos, tolerancia a falla, monitoreo y control.

Big Data Management in Smart Grids: Technologies and Challenges (Ameema Zainab, Ali Ghrayeb, et al. 2021): este trabajo destaca las limitaciones de las soluciones existentes en el procesamiento de datos y resalta la importancia del *Big Data* en el futuro de las *Smart Grids*. Se exploran tecnologías y desafíos en la gestión de datos masivos en el contexto de las redes eléctricas inteligentes, destacando además las limitaciones actuales.

Big data analytics in smart grids: a review (Yang Zhang, Tao Huang, Ettore Francesco Bompard. 2018): este estudio resalta la importancia del procesamiento de datos y el análisis de *Big Data* en la transmisión y distribución de electricidad, especialmente con el crecimiento del uso de medidores inteligentes y el desarrollo de *Smart Grids*. Se revisan diferentes técnicas y enfoques de análisis de datos masivos aplicados al sector eléctrico.

Capítulo 3

Metodología

En este trabajo se realizaron optimizaciones tanto a los archivos de salida del modelo del modelo PLP como de las rutinas de procesamiento. Los pasos de la metodología son los siguientes:

- Optimización de rutinas existentes: Las rutinas preexistentes fueron optimizadas de manera que la comparación con respecto al uso de técnicas de *Big Data* fuera más justa. La idea es comparar la aplicación de técnicas de *Big Data* con respecto al uso de técnicas clásicas que estén implementadas de la manera más eficiente.
- Transformación de datos con resolución por bloques a resolución horaria: Los datos del modelo PLP fueron transformados a datos con resolución horaria de manera de tener archivos de gran tamaño y que cumplan con la condición de *Big Data*.
- Implementación de rutinas en *Dask*: Se modifican las rutinas preexistentes para ser procesadas mediante la biblioteca *Dask* y hacer de estas escalables para los archivos archivos horarios.
- Implementación de rutinas en *PySpark*: Análogamente a *Dask* se busca usar las capacidades de procesamiento distribuido de *Apache Spark* intentando usar su API de *Python*, *PySpark*, para mantener la igualdad de condiciones con las demás herramientas.

Cabe destacar que en este trabajo no se crean nuevas simulaciones, sino que los resultados de estas fueron entregadas por el Centro de Energía y se pretende con estas solo ser adaptadas para usar las herramientas previamente mencionadas.

3.1. Implementación computacional

Para procesar los archivos y realizar las rutinas, se trabaja con un servidor local del centro de energía, el cual trabaja con un sistema *Linux* con sistema operativo *Ubuntu*, el cuál cuenta con 540 [GB] de memoria RAM y aproximadamente 20 [TB] de espacio en disco total. Sin

embargo, dado que ya hay archivos ocupando espacio y este servidor está en constante uso, la memoria juega un factor limitante a la hora de procesar archivos demasiado grandes.

Con respecto a la conexión del servidor, esta se realizará mediante protocolos *SSH* y *FSTP* para la conexión remota y traspaso de archivos, respectivamente. Para realizar aquello, se trabajó de manera local en un computador con sistema operativo *Windows*, por lo que para realizar estas conexiones se utilizaron las aplicaciones *PuTTY* para conexión de cliente *SSH* y *WinSCP* para el cliente *FSTP*.

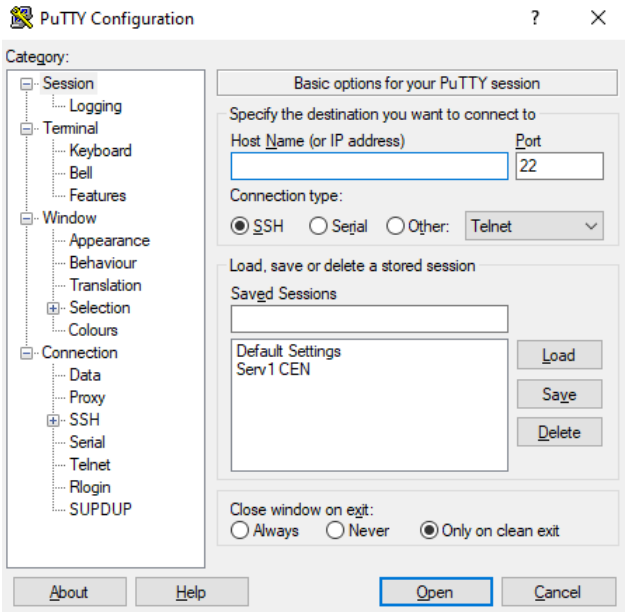

Figura 3.1: Interfaz PuTTY

Como se puede ver en la figura 3.1, se aprecia la interfaz principal del programa *PuTTY*, en la cual se muestra una ventana donde los usuarios pueden ingresar la dirección IP o el nombre del host del servidor al que desean conectarse.

Además, con respecto a este, es importante tener en mente que al estar conectado a un servidor mediante SSH, para realizar la conexión a los diferentes puertos del servidor, se debe hacer un tunelamiento al respectivo puerto donde se configure la interfaz.

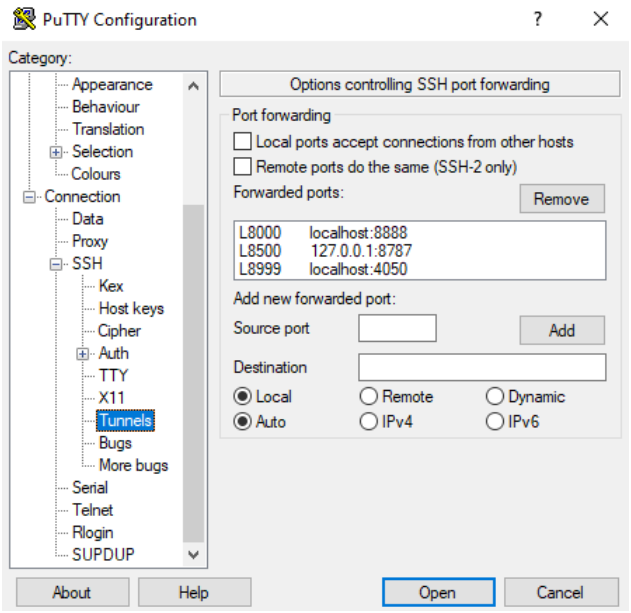

Figura 3.2: Conexión SSH con tunelamiento

De esta manera, en la figura 3.2 se pueden ver diferentes conexiones a algunos puertos del servidor por SSH, destacando el puerto 8888 que es para la conexión con *Jupyter Notebook* y los puertos 4050 y 8787, en donde el primero es utilizado de manera automática en *Dask* para su interfaz gráfica, vinculándolo al dispositivo local en el puerto 8999, y el último, de manera análoga para la interfaz de *Spark*, conectándose al puerto 8500.

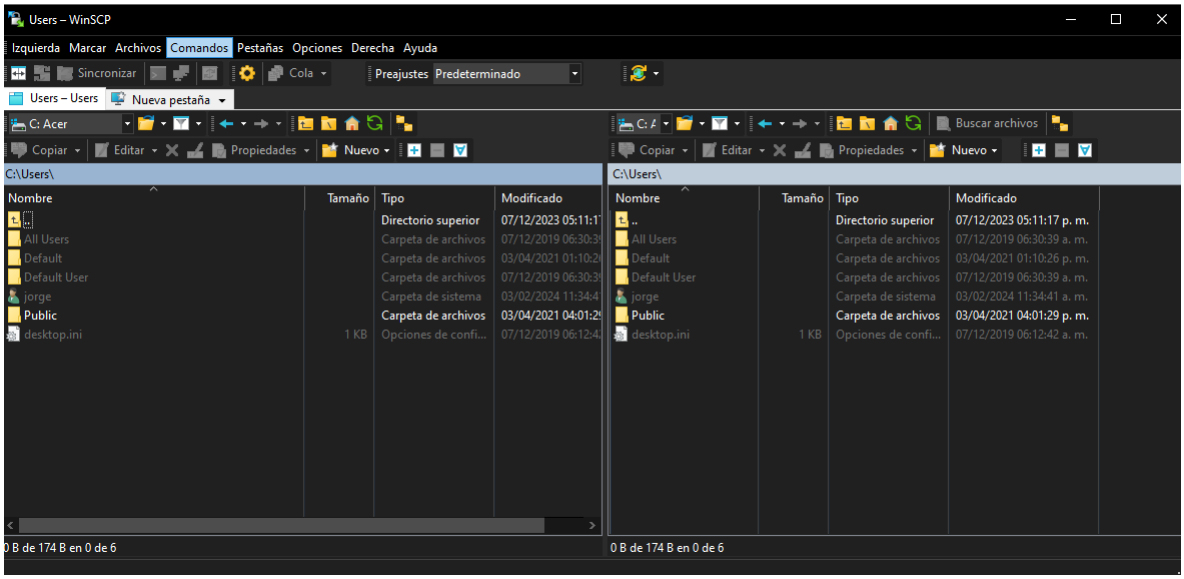

Figura 3.3: Interfaz WinSCP

Análogamente para *SFTP*, en la figura 3.3 se aprecia la interfaz de *WinSCP*, la cual muestra una interfaz de usuario intuitiva dividida en dos paneles: el panel izquierdo muestra los archivos locales, mientras que el panel derecho (cuando se está conectado) muestra los

archivos remotos del servidor. En este caso, análogamente a la imagen 3.1 aún no se ha realizado conexión, por lo que se necesita ingresar el puerto, dirección y demás permisos para el inicio de sesión.

De esta manera, se abordan desafíos como el consumo excesivo de memoria y la falta de escalabilidad. Como solución, se exploran formatos como *Parquet* y herramientas como *Dask* o *PySpark*, diseñadas para abordar eficazmente estos problemas. En las secciones siguientes se detallan las diferentes pruebas de carga y comparaciones entre estas herramientas, destacando conceptos como ejecución perezosa y computación distribuida.

3.2. Caracterización de rutinas y modelo PLP

Dado lo anterior, se comienzan a construir las diferentes etapas para la evaluación de procesamiento con *Big Data* donde, en primera instancia, se pretende entender y caracterizar los diferentes modelos y rutinas con las que se está trabajando.

De este modo, para el desarrollo de las rutinas, se trabajará con los datos de salida del modelo de proyección a largo plazo, o modelo PLP. Se abordarán cuatro ejemplos de proyecciones utilizando este modelo, los cuales abarcan los siguientes periodos de simulaciones:

El diseño de este modelo se basa en la programación por bloques de tiempo, lo que permite la intersección de datos para obtener resultados en resolución horaria. Las proyecciones a largo plazo generadas por este modelo se dividen en cuatro categorías: centrales, barras, líneas y fallas, proporcionando proyecciones en diversos dominios según cada una de estas categorías. Esto resulta en la creación de seis archivos distintos, cada uno destinado a una proyección específica, junto con un archivo llamado *"IndHor"* que se utiliza para transformar los datos a resolución horaria. Además, hay un archivo denominado "Centrales", que proporciona información adicional sobre las centrales y puede ser empleado para mejorar los datos a partir de los resultados de las proyecciones relacionadas con las centrales. En la figura 2.1, se puede apreciar una visualización de las rutinas realizadas con las salidas del modelo PLP.

3.3. Optimización de rutinas

De esta manera, para mejorar los tiempos de las rutinas originales, no se utilizaron nuevas herramientas, sino que, en principio, se optimizaron los procedimientos. Así, el primer cambio significativo se centró en la lectura y escritura de archivos. Anteriormente, en casos más pesados, como el archivo Plpcen, estos se leían línea por línea, almacenando las columnas necesarias en formato de diccionario de Python, lo cual se modificó para aprovechar eficazmente los recursos computacionales disponibles. Se utilizó íntegramente *Pandas* para leer y escribir todos los archivos, agilizando el proceso. Por otro lado, para escritura, antes se utilizaba una función auxiliar llamada *"DataWriter"* para crear archivos CSV línea por línea mediante la adición de cadenas de texto, lo que ralentizaba la eficiencia de escritura. Ahora, los *dataframes* hechos en Pandas se crean y almacenan de manera directa, aprovechando la memoria RAM.

Para realizar una optimización en la eficiencia de las rutinas iniciales, se debieron realizar preprocesamientos previos a los archivos originales, esto se realizó con el objetivo de reducir el tamaño de los archivos y con ello el tiempo de procesamiento a la hora de realizar las uniones de los diferentes archivos o para realizar de manera más óptima su transformación horaria.

Para realizar esto último, se utilizó el concepto de los bloques propios del modelo, esto porque, en la mayoría de las rutinas actuales, no es necesario saber la distribución horaria exacta de los diferentes sistemas, ya que, en general, los procedimientos realizados por las rutinas realizan sus cálculos en torno a proyecciones mensuales o incluso anuales. Es importante decir que evidentemente los bloques no son repartidos equitativamente cada mes, por lo que no se puede trabajar con los archivos en bruto, pero solo basta saber la cantidad de repeticiones por cada bloque y no las horas exactas en las que estos sucedieron. De esta manera se pretende realizar las uniones previamente dichas de manera más eficiente.

De este modo, para los ejemplos de las rutinas de generación anual por tecnología y pérdidas de transmisión en las barras se creó un archivo denominado *"Indhor Anual"* el cual establece en qué año sucede cada bloque. Esto se realizó filtrando en Pandas por valores únicos en las columnas de Año y Bloque juntas, de modo de saber que bloques suceden en el respectivo año. Esto permite que al hacer el cruce, ya sea con los archivos de centrales y fallas (en el caso de calcular la generación anual por tecnología) o el cruce con el archivo de barras (para el caso de calcular pérdidas), ser capaz de agrupar las diferentes filas según el año en que estas ocurren y de este modo poder sumarlas y calcular la rutina correspondiente.

Por otro lado, en las demás rutinas además se implementaron técnicas de optimización para agilizar el procesamiento de los datos. Por ejemplo, para calcular los costos marginales según el día y la noche, se creó una columna binaria en el archivo *"Indhor"* que indica el período. Luego, se fusionó este archivo con "Plpcen" y se dividió el *dataframe* resultante en dos según los valores de esta nueva columna. Luego, se agruparon los datos calculando el costo promedio anual de los costos marginales en ambos *dataframes* (día y noche) para obtener los resultados finales. En contraste, en el enfoque original, la función filtraba barra por barra, lo que requería 296 ciclos de filtrado en este ejemplo.

Es importante destacar que esta mejora en eficiencia es posible gracias a la capacidad computacional disponible. La primera forma de abordar el problema requiere una cantidad significativa de memoria RAM para los conjuntos de datos de día y noche, pero, dado que cuentan con el espacio necesario, la mejora en términos de tiempo es considerable.

Luego, para mejorar el procesamiento a la hora de calcular los costos marginales de cada barra y flujos por línea se aprovechó la idea de que tanto percentiles como el promedio dependen de la frecuencia que se repite cada valor, de este modo como cada bloque tiene un mismo valor para cada costo o flujo y no es necesario hacer la agrupación línea hora por hora sino que se puede contar durante cuantas horas estuvo activo cada bloque, de esta manera, es posible hacer una fusión mucho más pequeña que transformar los archivos a resolución horaria y de este modo se vuelve más eficiente hacer esta única unión y luego la respectiva agrupación calculando el promedio y los percentiles que aplicar la técnica tradicional que calculaba el promedio y percentiles con resolución horaria filtrando barra por barra.

3.4. Formato de archivos

Como se vio desde el marco teórico, el formato *CSV*, debido a su naturaleza de archivo de texto plano, tiende a generar archivos de mayor tamaño en comparación con formatos más eficientes. Esto puede ser problemático en términos de consumo de espacio en disco y ancho de banda al transportar datos. En contraste, otros formatos alternativos pueden aplicar mejores técnicas de compresión, lo que significa que los archivos resultantes pueden ser significativamente más pequeños. Permitiendo, como se menciona previamente, ahorrar espacio en disco y reducir los tiempos de transferencia de archivos.

En lo que respecta al procesamiento de datos, los archivos *CSV* son menos eficientes debido a la necesidad de convertir entre datos de texto y binarios. Esto genera una sobrecarga en el procesamiento que puede ralentizar la velocidad de acceso a los datos. Por otro lado, otros formatos tienen diferentes formas de procesamiento, como, por ejemplo, el cambio de almacenamiento por filas a columnas o el guardado de metadatos que permiten aligerar la carga de procesamiento, lo que se ve traducido en una velocidad de procesamiento más rápida. Esto, es particularmente valioso en aplicaciones que requieren un acceso rápido y eficiente a los datos, como análisis en tiempo real o sistemas con cargas de trabajo muy grandes.

La falta de índices y optimizaciones de consulta en archivos *CSV* también puede ser un inconveniente. La ejecución de consultas complejas en este formato suele requerir más tiempo y recursos. Por el contrario, otros formatos pueden ser más adecuados para consultas complejas, ya que pueden proporcionar estructuras de indexación y optimización que agilizan la recuperación y análisis de datos.

De este modo, elegir un formato de archivo más eficiente en lugar de *CSV* puede resultar en archivos más pequeños, velocidades de procesamiento más rápidas y una mayor eficiencia en la gestión y consulta de datos. A continuación se explorarán otras formas de almacenamiento de archivos comparando su tamaño y desempeño a la hora de ser procesados tanto en lectura como escritura versus los archivos originales.

Los nuevos formatos a probar son *Parquet* y *Feather*, y la primera prueba a realizar es transformar los datos de archivos originales a este formato para realizar una comparativa del espacio en disco que utilizan estos distintos tipos de archivo.

3.5. Herramientas de procesamiento

Utilizar la biblioteca Pandas en entornos de *Big Data* plantea una serie de desafíos técnicos y operativos a la hora de trabajar con estos grandes volúmenes de datos. Aunque Pandas es una herramienta robusta y ampliamente utilizada para el análisis de datos en Python, su eficiencia y escalabilidad se ven comprometidas cuando se enfrenta a conjuntos de datos significativamente grandes.

Uno de los problemas más notables al utilizar Pandas en *Big Data*, es su consumo de memoria. La biblioteca carga todo el conjunto de datos en la memoria principal, lo que puede agotar rápidamente la memoria RAM del sistema, resultando en una degradación del

rendimiento o incluso la interrupción de las operaciones. Es así que la escalabilidad de Pandas se ve limitada directamente por la capacidad de la memoria RAM disponible. Esto puede dificultar la manipulación de conjuntos de datos que superen la capacidad de memoria.

Además, Pandas no está diseñado para el procesamiento paralelo y/o distribuido, lo que conlleva a un rendimiento deficiente en el procesamiento de grandes volúmenes de datos. Esto puede aumentar significativamente el tiempo necesario para ejecutar análisis o manipulaciones de datos en comparación con herramientas específicas para *Big Data*.

Otro problema radica en la ineficiencia en la lectura y escritura de datos. Pandas no está optimizado para el almacenamiento y recuperación eficiente de grandes volúmenes de datos, lo que puede generar cuellos de botella en las operaciones de entrada y salida (E/S).

Para superar los problemas de memoria y rendimiento, una de las técnicas más utilizadas es fragmentar manualmente los datos en segmentos más pequeños denominados *chunks* para procesarlos por separado. Esto agrega una complejidad adicional al flujo de trabajo y puede resultar en una gestión más complicada de los resultados.

En consecuencia, si se desea escalar el volumen de datos de un sistema, será necesario explorar nuevas herramientas como *Dask* o *Apache Spark*, que actúen de manera complementaria o que reemplacen las técnicas tradicionales para abordar los problemas de escalabilidad y rendimiento en entornos de *Big Data*. A continuación, se realizarán una serie de pruebas de carga, procesamiento y escritura trabajando con los archivos horarios para evaluar y comparar estas nuevas herramientas.

De este modo, se procede a explorar cómo estas herramientas se comportan en el procesamiento de archivos. Por lo que, análogamente a lo realizado en la sección anterior, se vuelven a diseñar las rutinas del modelo de proyección a largo plazo, pero esta vez utilizando las técnicas que proveen estas herramientas, como lo son ejecución perezosa o computación distribuida.

La ejecución perezosa, también conocida como "Lazy Evaluation.º evaluación diferida, es un concepto que se aplica cuando la ejecución de cierto código se realiza únicamente cuando es necesario. Aunque esto pueda parecer evidente, el desarrollo de esta práctica no es tan simple. Cuando se aplica a herramientas de procesamiento de datos, como Pandas, la lectura, procesamiento y escritura de *dataframes* ocurren siempre que se solicitan o necesitan, aquella manera de trabajar. En contra de lo anteriormente mencionado, se le denomina ejecución ansiosa ("Eager Evaluation") y puede resultar ineficiente en términos de recursos y tiempo.

La ejecución perezosa aborda esta ineficiencia posponiendo la evaluación de las operaciones hasta el último momento posible, es decir, hasta que se requiera el resultado final. Este enfoque ofrece ventajas significativas en términos de optimización de recursos y mejora del rendimiento.

Para ello, las herramientas o bibliotecas como *Dask* o *Pyspark* antes de ejecutar o procesar cada archivo señalado crean un grafo de tareas, las cuales solo ejecutan al momento de ser necesarias, aportando este método un ahorro significativo de memoria RAM en el procesamiento. Junto con lo anterior, este enfoque permite organizar las tareas, de modo que estas puedan ser distribuidas en diferentes procesos, para de esta manera trabajar con

computación distribuida o paralela.

La computación distribuida es un método de trabajo en el cual las tareas se dividen en fragmentos más pequeños y se ejecutan simultáneamente en múltiples nodos de un sistema. Este enfoque contrasta con la dependencia de un solo servidor potente para la realización de tareas, promoviendo así un procesamiento más rápido y eficiente. La importancia de este modelo radica en su capacidad de escalabilidad, permitiendo la adición de más nodos para manejar mayores volúmenes de datos o cargas de trabajo más intensivas. Además, ofrece una mayor tolerancia a fallos al distribuir las tareas entre nodos, garantizando la continuidad de las operaciones incluso en caso de fallo en uno de los nodos.

3.5.1. Procesamiento con Dask

En primer lugar, se realizaron las pruebas utilizando la biblioteca *Dask*, para ello, como se mencionó previamente, se desarrolló un *cluster* local para trabajar en un entorno dividido con esta herramienta.

Figura 3.4: Cliente Dask

En este caso, como se puede ver en la figura 3.4 se crea un *cluster* local, utilizando todos los núcleos y memoria disponible del sistema, en este caso contando con 64 núcleos y 540 Gigabytes de memoria RAM. La cantidad de trabajadores o nodos del *cluster* en este caso son 8, siendo este el modo conectado por defecto en el sistema, sin embargo, para las diferentes pruebas se utilizarán diferentes configuraciones de trabajadores, las cuales serán vistas más adelante, para así realizar comparaciones entre la mejora en la distribución de tareas.

Por otro lado, *Dask* proporciona un *Dashboard* o interfaz gráfica (UI) que permite ver y analizar el desarrollo en el procesamiento de los datos, de este modo se puede entender de manera más intuitiva que acciones demoran más tiempo y en donde se pueden realizar optimizaciones a las rutinas.

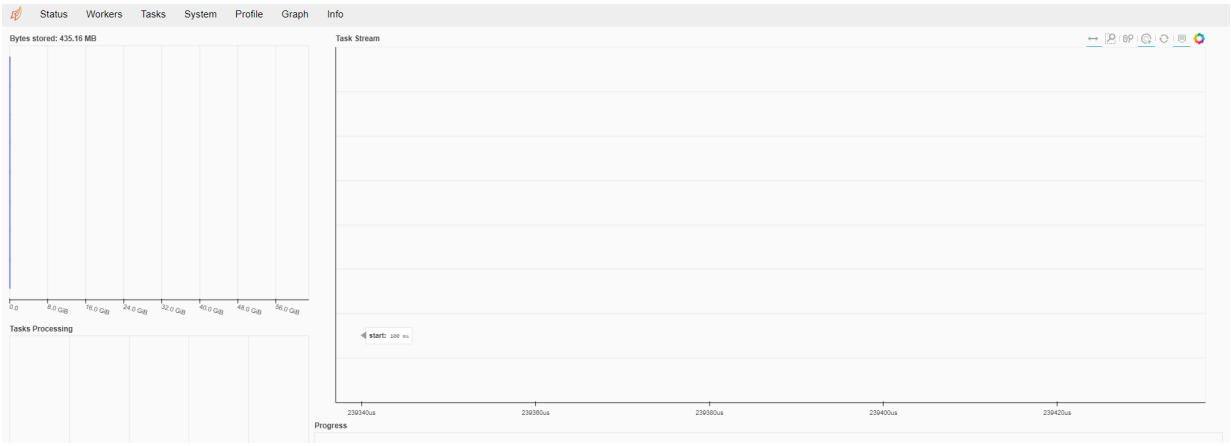

Figura 3.5: Interfaz gráfica de Dask

De este modo, en la figura 3.5, se puede ver la pestaña Status de la interfaz gráfica de *Dask*, en la que se destacan el despliegue visual al mostrar el procesamiento de cada núcleo en tiempo real (Task Stream), la memoria utilizada por cada trabajador (Bytes Stored), la cantidad de tareas por trabajador (Task Proccesing) y las tareas a procesar (Progress). A continuación se muestran ejemplos de cada una de las interfaces por separado:

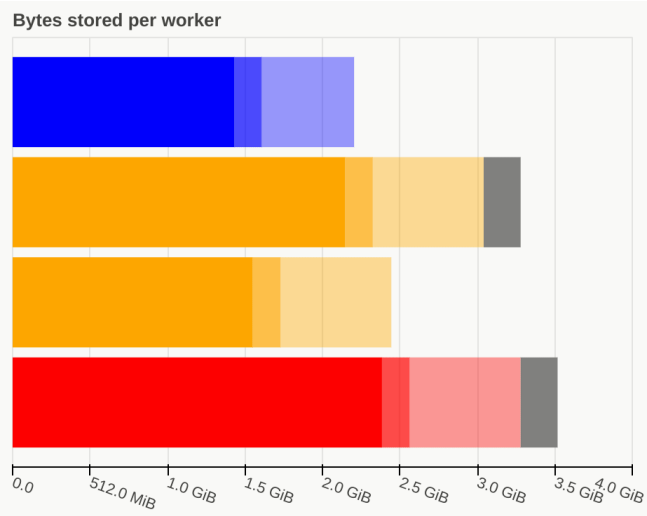

Figura 3.6: Interfaz Dask: Bytes por trabajador

Figura 3.7: Interfaz Dask: Tareas por trabajador

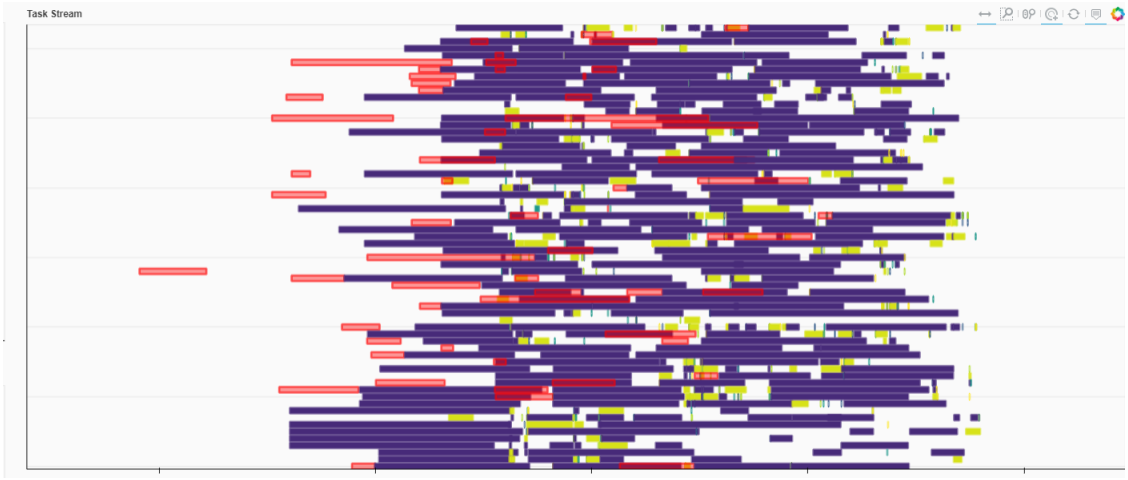

Figura 3.8: Interfaz Dask: Visualización tareas

En las figuras 3.6 y 3.7 se puede apreciar un ejemplo del estado de cada trabajador durante un procesamiento cualquiera. Es importante destacar el significado de cada color para ambas interfaces, donde el color azul indica una correcta carga de trabajo para el respectivo trabajador, ocupando menos del 70 % de memoria de este, mientras que el color naranjo (o verde en el caso de la cantidad de tareas) indica una saturación del sistema estando este entre un 70 % y 80 % de la memoria de este ocupada y el color rojo indica un trabajador sobresaturado que se pausa por seguridad y para que no existan errores por fugas de memoria, siendo este resultado al superar el 80 % de memoria por trabajador.

Por otra parte, en la figura 3.8 se puede apreciar la visualización de tareas ejecutándose en tiempo real, donde es importante destacar en el código de colores las barras de colores rojo o naranjo, donde el color rojo indica una tarea muy larga que tomó demasiado tiempo y una barra de color naranja que existió algún error, generalmente derrames en la memoria, lo que puede causar que la tarea respectiva se intente reiniciar cuando exista la memoria suficiente o en caso de superar la memoria total, el fallo de la ejecución.

Así, bajo el marco de lo anteriormente mencionado, se diseñan las mismas rutinas creadas

inicialmente, modificandolas para realizar los procesamientos utilizando esta herramienta. Donde, para construir las nuevas rutinas, se utilizan cuatro funciones clave para trabajar con procesamiento distribuido, las cuales son *persist()*, *n_partitions()*, *delayed()*, y *compute()*, cada una desempeñando un papel crucial en la optimización y ejecución eficiente de operaciones.

La función *persist()* se utiliza para almacenar en caché los resultados intermedios de las operaciones *Dask*, lo que permite reutilizarlos y evitar recalculos costosos, mejorando así el rendimiento. Al llamar a *persist()* en un objeto, este guarda el resultado en la memoria para su acceso rápido en operaciones futuras, reduciendo la necesidad de volver a calcularlo.

Por otro lado, *n_partitions()* se utiliza para modificar el número de particiones en las que se divide un objeto *Dask*. Estas particiones son unidades de trabajo que se distribuyen entre los distintos procesadores o nodos en cada clúster.

Por su parte, la función *delayed()* se emplea para posponer la ejecución de una función o una llamada a un objeto *Dask*, convirtiéndola en una tarea diferida. La cual solo se ejecuta posteriormente mediante *compute()*.

Finalmente, *compute()* es la función que desencadena la ejecución real de las tareas diferidas en *Dask*. Al llamar a *compute()* se inicia la computación distribuida y paralela de las operaciones.

De este modo, bajo los puntos anteriormente mencionados se realizaron las modificaciones respectivas para trabajar con *Dask* diseñando las nuevas rutinas.

3.5.2. Procesamiento con Pyspark

Luego, se buscó desarrollar un procedimiento completamente análogo con la herramienta *PySpark*, para realizar aquello, se instalaron las dependencias necesarias de *Apache Spark* y *PySpark*, la API para conectar esta a *Python*. Con las librerías ya instaladas, se creó una sesión de *Spark*, para procesar los datos de manera paralela.

```
# Configuración de Spark

conf = SparkConf()
conf.setAppName("Spark3")
conf.setMaster("local[8]")  # Ajusta el número de cores del driver

conf.set("spark.driver.memory", "128g")  # Ajusta la memoria del driver
conf.set("spark.executor.cores", "8")  # Ajusta el número de cores por executor
conf.set("spark.executor.memory", "32g")  # Ajusta la memoria por cada executor
conf.set("spark.submit.deployMode", "client")  #Ajusta unico o distribuido Puede cambiar a 'client' o 'cluster'

# Crear una sesión de Spark
spark = SparkSession.builder.config(conf=conf).getOrCreate()
```

Figura 3.9: Ejemplo de configuración PySpark

De este modo, tal como se muestra en la figura 3.9 se pueden realizar una serie de configuraciones, como ajustar la memoria o cantidad de núcleos para el driver o cada uno de los ejecutores. De este modo se busca distribuir los recursos de manera equitativa, repartiendo 8 núcleos para el driver y 8 para cada ejecutor.

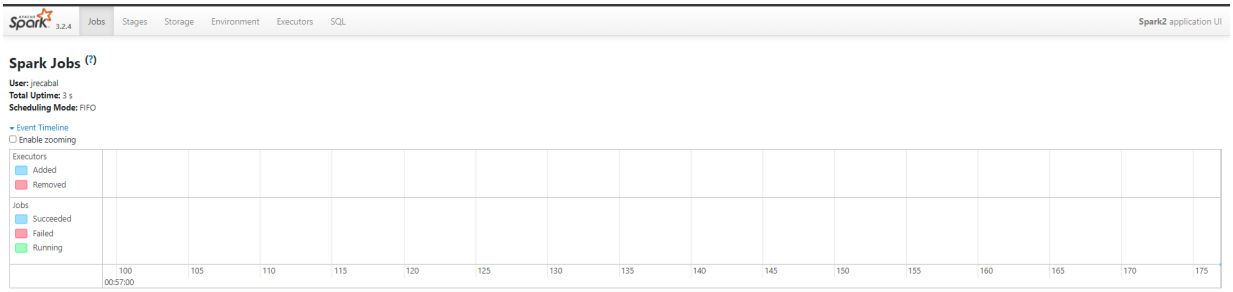

Figura 3.10: Interfaz gráfica Pyspark

Así, análogamente a *Dask*, se crea una sesión la cuál se puede visualizar mediante su interfaz gráfica en la cuál en su pestaña principal también se pueden apreciar los diferentes procesos ejecutandose en el tiempo. Además de contar con otras interfaces para vigilar demás parámetros de la herramienta o del sistena.

Una de ellas, es la pestaña de ejecutores, la cuál como se muestra en la figura 3.11 tiene un solo proceso activo, asociado al driver o nodo maestro, de esta manera, a pesar de intentar forzarlo, solo se crea el ejecutor respectivo del driver y no se crean nuevos workers o nodos esclavos. Por lo que de aún de manera local de este modo aún no se ha diseñado una división en la arquitectura que permita trabajar de manera correctamente distribuida.

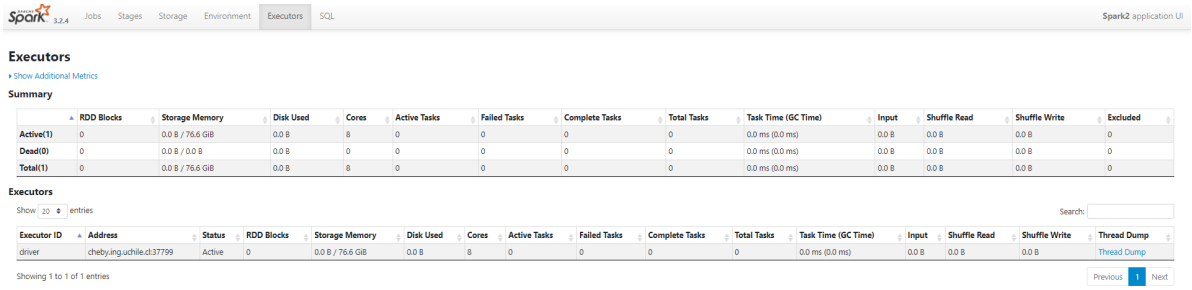

Figura 3.11: Ejecutores PySpark

Por esto último, se evaluaron nuevas maneras de poder trabajar con *Spark* de manera distribuida y local. Para ello, se utilizó la instalación de este mediante contenedores de *Docker*, para de esta manera generar varias máquinas virtuales diferentes que alojarán cada uno de los distintos trabajadores de *Spark*.

Capítulo 4

Resultados

4.1. Casos de estudio

Las técnicas de *Big Data* desarrolladas en esta memoria serán aplicadas a los datos de salida del modelo del modelo PLP. Los resultados del modelo PLP fueron obtenidos de 4 simulaciones previas desarrollados previamente por el Centro de Energía. Los periodos de simulación de los 4 ejemplos son los siguientes:

- Modelo pequeño: proyección desde 2023 hasta 2026
- Modelo mediano: proyección desde 2023 hasta 2035
- Modelo grande: proyección desde 2023 hasta 2044
- Modelo muy grande: proyección desde 2023 hasta 2045

En la Tabla 4.1, se pueden observar los diferentes tamaños de los archivos de los diferentes modelos. Donde se puede observar que en el modelo pequeño, los archivos de proyección de líneas y barras son del orden de solo cientos de *megabytes*, superando los *gigabytes* solo en la proyección de centrales. Mientras que en el caso contrario, los modelos grande y muy grande superan fácilmente esos tamaños en las tres proyecciones mencionadas previamente, mostrando así cómo estas, según la cantidad de elementos o el tamaño de la proyección, pueden generar archivos de muy grandes que escalan rápidamente.

Tabla 4.1: Tamaños archivos Modelo PLP

Archivo/Modelo	Pequeño	Mediano	Grande	Muy Grande
Centrales	82 [KB]	15 [KB]	52 [KB]	75 [KB]
Indhor	462 [KB]	2.14 [MB]	3.5 [MB]	3.45 [MB]
PlpBar	163 [MB]	1.92 [GB]	3.8 [GB]	4.15 [GB]
PlpCen	1.67 [GB]	4.13 [GB]	18.2 [GB]	29.1 [GB]
PlpFal	1 [KB]	1 [KB]	182 [KB]	1 [KB]
PlpLin	230 [MB]	3.07 [GB]	5.2 [GB]	5.77 [GB]

4.2. Resultados de rutina optimizada

Con las ideas mencionadas previamente en la sección de metodología, se construyen nuevamente todas las rutinas iniciales aplicando las modificaciones para optimizar las rutinas solo usando Pandas o técnicas tradicionales de ciencia de datos, de esta manera, se obtienen los siguientes tiempos de procesamiento de los archivos.

Los resultados muestran que los tiempos se reducen drásticamente puesto que la carga y procesamiento de estos archivos es ideal para el trabajo con este software, ejecutándose las rutinas en solo alrededor de 4 minutos en total.

La siguiente tabla resume el tiempo de procesamiento para los 4 casos de estudio.

Tabla 4.2: Tiempo de ejecución rutinas optimizadas en modelo PLP

Rutina / Modelos	Pequeño	Mediano	Grande	Muy Grande
Graficar Costos Marginales por barra	68 [s]	102 [s]	174 [s]	186 [s]
Graficar Flujos de energía por linea	95 [s]	166 [s]	239 [s]	256 [s]
Costos Marginales Dia y Noche	3 [s]	21 [s]	43 [s]	46 [s]
Factor Planta Anual	17 [s]	42 [s]	189 [s]	296 [s]
Factor Planta Mensual	30 [s]	82 [s]	296 [s]	464 [s]
Grafico generacion anual	19 [s]	39 [s]	177 [s]	292 [s]
Grafico pérdidas	2 [s]	16 [s]	41 [s]	34 [s]

En primer lugar, para el modelo pequeño, como se puede ver en la Tabla 4.2 los tiempos se reducen drásticamente, puesto que la carga y procesamiento de estos archivos es ideal para el trabajo con este software, ejecutándose las rutinas en solo alrededor de 4 minutos en total.

A continuación se ejecutan las rutinas optimizadas con el modelo mediano, en el cual los tiempos siguen siendo muy eficientes, demorando solo entre 15 segundos y 3 minutos aproximadamente dependiendo la rutina, con un tiempo total en promedio un poco menor a los 8 minutos de ejecución, mucho menor que en el caso inicial, el cual, como se pudo ver de la Tabla 2.1, tardó aproximadamente 55 minutos, representando una mejora de entre seis y siete veces los tiempos de ejecución, siendo este el caso en el que se obtiene proporcionalmente un mejor desempeño.

En el caso del modelo grande, los tiempos aumentan de manera más drástica en las rutinas que al inicio parecían muy eficientes. Esto se debe en gran parte a la demora en tiempos de lectura con *Pandas* a medida que los archivos crecen, especialmente reflejado en el caso del archivo Plpcen, tardando varias rutinas algunos minutos en su ejecución y dando un tiempo total de casi 20 minutos de ejecución, pero siendo aún bastante más eficiente que en el caso original, el cuál tardó sobre una hora de ejecución.

Luego, en el caso del modelo muy grande, los análisis son similares, tardando este sobre 26 minutos en ejecutarse, que continúa estando muy bien frente a las rutinas originales. Sin embargo, es notorio que estas optimizaciones funcionan aún mejor en modelos de pequeño y mediano tamaño, donde mejor desempeño posee en general trabajar con *Pandas* y se puede

hacer cada vez más pesado intentar procesar todo con solo fuerza bruta evitando dividir o hacer pequeños los cruces.

De este modo, como un primer paso, esto demuestra que aplicando buenas técnicas aún se es posible trabajar con los archivos y reducir los tiempos de procesamiento con las necesidades de esta rutina en particular, sin embargo, esto cambia en el momento en que ya no se pueda aprovechar el concepto de contar o sumar la cantidad de bloques o el procesar alguna columna en particular, ya que si estos datos vienen por defecto en resolución horaria o se buscan resultados con ese nivel de detalle o incluso aún mayor es claro que estas mejoras de procesamiento se vuelven inútiles y se recae en el mismo problema del inicio, hacer uniones o procesamientos de datos gigantescos que son lentas con las técnicas actuales, de ahí el valor de utilizar las nuevas herramientas que se verán a continuación.

4.3. Resultados formatos de archivos

4.3.1. Tamaño de archivos de salida de modelos originales con diferentes formatos

En primer lugar, como se venía hablando previamente, el primer problema a tener en consideración es el volumen de los datos y su capacidad de transformación al uso de técnicas más modernas en ese sentido, dado el análisis previo de por qué trabajar con CSV era un problema con estos datos que no son leídos directamente. Es por ello que el primer problema de este aspecto es transformar los archivos a los nuevos formatos y evaluar las comparativas de tamaños.

De este modo, como se puede ver en la Tabla 4.3, para el modelo pequeño donde los datos no representan más que 2[GB] de memoria, el cambio de formato no parece tan obligatorio, sin embargo, sigue siendo un gran aporte transformar un archivo mayor a 1[GB] en uno de tan solo 31[MB], el ahorro de espacio es demasiado notable, ocupando un espacio en disco para los archivos *Parquet* de solo 60[MB], más de 30 veces menos que en CSV. Por su parte *Feather* también muestra reducciones muy significativas, de unas 5 o 6 veces menos tamaño que en el caso original.

Tabla 4.3: Comparativa de tamaño de archivos según diferentes formatos en el modelo pequeño

Archivo	Tamaño Parquet	Tamaño Feather	Tamaño CSV
Centrales	30.2 [KB]	58.3 [KB]	82 [KB]
Indhor	9.44 [KB]	156 [KB]	462 [KB]
PlpBar	12.7 [MB]	33.9 [MB]	163 [MB]
PlpCen	31.8 [MB]	201 [MB]	1.67 [GB]
PlpFal	5.86 [KB]	3.99 [KB]	1 [KB]
PlpLin	16.5 [MB]	53.4[MB]	230 [MB]

En el modelo mediano, como se aprecia en la Tabla 4.4, al ser los datos más significativos, la transformación de formatos se vuelve aún más crucial, los archivos Parquet y Feather ofrecen reducciones sustanciales en términos de tamaño en comparación con el formato CSV. Por ejemplo, en el caso de Plpcen, el archivo Parquet ocupa solo 79[MB] en comparación con los impresionantes 4.1[GB] del archivo CSV original. Esto representa más de 50 veces de reducción en el espacio de almacenamiento. También es notorio que la reducción de tamaños no depende directamente del tamaño de cada CSV, sino que también afecta mucho para la compresión el tipo de formato de columnas que se poseen, ya que tablas con solo números pueden ser mejor comprimidas que otras con palabras u otros objetos o caracteres.

Tabla 4.4: Comparativa de tamaño de archivos según diferentes formatos en el modelo mediano

Archivo	Tamaño Parquet	Tamaño Feather	Tamaño CSV
Centrales	12.1 [KB]	16.3 [KB]	15 [KB]
Indhor	27.6 [KB]	579 [KB]	2.14 [MB]
PlpBar	148 [MB]	400 [MB]	1.92 [GB]
PlpCen	78.7 [MB]	489 [MB]	4.13 [GB]
PlpFal	5.86 [KB]	3.99 [KB]	1 [KB]
PlpLin	166 [MB]	596 [MB]	3.07 [GB]

En este nuevo caso para el modelo grande el análisis es muy similar, donde se aprecia en la Tabla 4.5 como el archivo Plpcen posee la disminución más alta en proporción, mientras que Plplin sufre una disminución menor, esto ya explicado por las diferencias en los tipos de datos de las diferentes columnas de los archivos. Por otro lado se hace constante que *Parquet* tiene mejores resultados que *Feather* en términos de volumen, siendo siempre dos o tres veces más pequeño en comparación

Tabla 4.5: Comparativa de tamaño de archivos según diferentes formatos en el modelo grande

Archivo	Tamaño Parquet	Tamaño Feather	Tamaño CSV
Centrales	25 KB	38.9 KB	52.7 KB
Indhor	62.5 KB	218 KB	3.67 MB
Plpbar	271 MB	686 MB	3.88 GB
Plpcen	415 MB	2.18 GB	18.6 GB
Plpfal	15.4 KB	38.1 KB	186 KB
Plplin	435 MB	1.11 GB	5.34 GB

Finalmente para el modelo muy grande, en la Tabla 4.6 el resultado es completamente análogo al modelo muy grande, escalando un poco más el tamaño de los archivos y llegando a pesar 37[GB] en el caso original, frente a solo un poco mas de 1[GB] para el caso de parquet y 2[GB] en Feaher, siendo una disminución y ahorro importante de espacio en disco.

Tabla 4.6: Comparativa de tamaño de archivos según diferentes formatos en el modelo muy grande

Archivo	Tamaño Parquet	Tamaño Feather	Tamaño CSV
Centrales	28.9 [KB]	55 [KB]	75 [KB]
Indhor	62.1 [KB]	1.01 [MB]	3.4 [MB]
PlpBar	263 [MB]	786 [MB]	4.15 [GB]
PlpCen	549 [MB]	3.7 [GB]	29.1 [GB]
PlpFal	5.86 [KB]	3.99 [KB]	1 [KB]
PlpLin	428 [MB]	1.26 [GB]	5.77 [GB]

Así, de los resultados en general, se pudo observar que el tamaño de los archivos se ve reducido drásticamente al trabajar con estos nuevos formatos. En particular, se destaca la compresión de *Parquet* con la cual se obtuvieron los mejores resultados en este aspecto.

4.3.2. Tamaño de archivos de salida en resolución horaria con diferentes formatos

A continuación, para comprender el problema real de espacio se procesan y crean los archivos en resolución horaria en formatos CSV y *Parquet* obteniendo los diferentes tamaños para cada uno de los archivos horarios. En los cuales se podrá observar el verdadero problema que se podría enfrentar en un corto o mediano plazo, recibiendo constantemente y siendo forzados a procesar archivos de estos tamaños.

En primer lugar, se aprecian los archivos del modelo pequeño en la Tabla 4.7, donde se ve como en contraste con los archivos originales, el tamaño de estos en formato CSV supera rápidamente los *Gigabytes* superando los 100[GB] en el caso de proyección de centrales con 132 [GB], haciéndose notar así, la importancia del cambio de formato de los archivos, donde estos disminuyendo drásticamente sus tamaños, donde en el caso de usar *Parquet* sigue siendo cómodo y fácil guardar estos archivos en disco o compartirlos.

Tabla 4.7: Comparativa de tamaño de archivos horarios según diferentes formatos en el modelo pequeño

Tipo de Archivo	Tamaño CSV	Tamaño Parquet
PlpBar Horario	13.7 [GB]	87.8 [MB]
PlpFal Horario	1 [KB]	1.44 [KB]
PlpLin Horario	20.7 [GB]	667 [MB]
PlpCen Horario	132 [GB]	1.2 [GB]

En el caso del modelo mediano, en la Tabla 4.8, se puede ber nuevamente como los archivos escalan rápidamente, entre 50 y 300 veces su tamaño, lo que hace súmamente dificil su creación y almacenamiento en formatos convencionles o de texto plano. Donde además, se aprecia por primera vez que no se crean todos los archivos, puesto que algunos de estos

exceden la memoria RAM del sistema y además en otras herramientas como *Spark* la escritura de un archivo tan grande se vuelve demasiado ineficiente, por lo que ya entendido el punto de las enormes diferencias entre formatos, se opta por no crear los archivos demasiado grandes, los cuales además serían costosos de mantener en el propio servidor por su gran tamaño.

Tabla 4.8: Comparativa de tamaño de archivos horarios según diferentes formatos en el modelo mediano

Tipo de Archivo	Tamaño CSV	Tamaño Parquet
PlpBar Horario	78.6 [GB]	1.48 [GB]
PlpFal Horario	1.5 [KB]	1.44 [KB]
PlpLin Horario	386 [GB]	1.59 [GB]
PlpCen Horario	>600 [GB]	2.4 [GB]

De la Tabla 4.9 se puede ver la gran disminución en el tamaño de los diferentes archivos según el tipo de formato para el modelo grande, para el que el archivo de salida Plpbar tuvo una disminución de cerca de un orden de magnitud menos en el caso original, pasando a dos órdenes de magnitud en el caso horario, siendo cerca de 100 veces más pequeños los archivos en este caso. Por la misma razón, no son realizados los archivos de lineas y centrales puesto que, al igual que en el caso mediano se estarían generando archivos de tamaños que pueden ser cercanos o superiores a los *Terabytes* lo que resulta complicado almacenar en disco por limitaciones computacionales. Sin embargo, queda clara la importancia de la compresión a medida que los archivos crecen.

Tabla 4.9: Comparativa de tamaño de archivos horarios según diferentes formatos en el modelo grande

Tipo de Archivo	Tamaño CSV	Tamaño Parquet
PlpBar Horario	213.6 [GB]	2.4 [GB]
PlpFal Horario	612 [KB]	48 [KB]
PlpLin Horario	> 600 [GB]	2.6 [GB]
PlpCen Horario	> 1 [TB]	5.7 [GB]

Finalmente en el caso muy grande, en la Tabla 4.10 todos los archivos tuvieron resultados similares al caso grande, resultado esperable, puesto que en el caso de Plpbar, al tener un tamaño de archivos similares y proyecciones de tiempo similares los tamaños son relativamente parecidos y en los casos de proyección de líneas y centrales la escala es demasiado grande y no se lograron crear los archivos.

Tabla 4.10: Comparativa de tamaño de archivos horarios según diferentes formatos en el modelo muy grande

Tipo de Archivo	Tamaño CSV	Tamaño Parquet
PlpBar Horario	312 [GB]	2.58 [GB]
PlpFal Horario	1.7 [KB]	1.44 [KB]
PlpLin Horario	>600 [GB]	3.9 [GB]
PlpCen Horario	>1 [TB]	6.1 [GB]

De esta forma, a partir de la creación de los archivos en resolución horaria, haciendo la unión de los archivos de salida completos con su respectivo *Indhor* en las columnas "Bloque", se puede notar como se hace necesaria la capacidad de compresión según los formatos, puesto que, manejar uno cuantos *Gigabytes* puede resultar sencillo en términos prácticos y económicos, pero, si se comienza a hablar de mayores ordenes de magnitud esto resulta en un problema costoso e ineficiente, demostrando así el beneficio e importancia de estas nuevas herramientas con respecto al volumen de los datos.

4.4. Resultados herramientas de procesamiento

La primera prueba a realizar, en términos de velocidad y herramientas de procesamiento fue la velocidad de escritura de archivos, creando en formato *Parquet* los archivos horarios mencionados en la sección anterior. Las herramientas a utilizar fueron *Pandas*, como caso base; *Dask* y *Apache Spark*, obteniendo de esta manera los tiempos para cada herramienta.

Así, en la Tabla 4.11, se pueden observar los tiempos de las diferentes herramientas para la escritura de archivos horarios del modelo pequeño, se puede observar como a medida que los archivos crecen, los tiempos se disparan, siendo la escritura de los archivos en general tan lenta como la propia transformación horaria y al escalar a archivos mas pesados esto se hace cada vez mas ineficiente.

Tabla 4.11: Comparativa de tiempo de escritura de archivos horarios según diferentes herramientas en el modelo pequeño

Archivo	Pandas	Dask	Spark
PlpBar Horario	65 [s]	104 [s]	186 [s]
PlpFal Horario	<1 [s]	<1 [s]	<1 [s]
PlpLin Horario	102 [s]	168 [s]	333 [s]
PlpCen Horario	984 [s]	1132 [s]	1980 [s]

En el caso mediano, en la Tabla 4.12 ya se observan como los tiempos escalan al punto de superar la hora de tiempo en el caso de *Spark*, donde la escritura resultó ser la mas ineficiente, sin embargo, esta opción es la única que permitió trabajar con archivos por sobre el tamaño del disco sin dividir el archivo en trozos o *chunks*, puesto que, para los casos de *Dask* y *Pandas* no se logró obtener el tiempo de salida en el caso del archivo horario de centrales, viéndose este en la tabla reflejados con una "X", esto debido a que los archivos superaban la capacidad en memoria del sistema y fueron imposibles de ejecutar.

Tabla 4.12: Comparativa de tiempo de escritura de archivos horarios según diferentes herramientas en el modelo mediano

Archivo	Pandas	Dask	Spark
PlpBar Horario	1168 [s]	1286 [s]	3197 [s]
PlpFal Horario	<1 [s]	<1 [s]	<1 [s]
PlpLin Horario	2457 [s]	2693 [s]	6398 [s]
PlpCen Horario	X	X	7031 [s]

El caso para el modelo grande, en la Tabla 4.13 es análogo al modelo mediano, con tiempos de espera aún más grandes y cercanos a las 5 horas en total para crear todos los archivos en *Spark*. En el caso de *Pandas* y *Dask* los tiempos son similares y menores que en *Spark*, pero dado que para escribir el archivo intentan ejecutar una tarea en el *dataframe* completo, estos superan el tamaño de la memoria RAM y el programa es incapaz de ejecutarse, resultando así, análogamente que en el caso anterior que la escritura de los archivos de líneas y centrales no pudieron ser ejecutados.

Tabla 4.13: Comparativa de tiempo de escritura de archivos horarios según diferentes herramientas en el modelo grande

Archivo	Pandas	Dask	Spark
Plpbar Horario	1214 [s]	1368 [s]	2767 [s]
Plpfal Horario	<1 [s]	<1 [s]	<1 [s]
PlpLin Horario	X	X	12248 [s]
PlpCen Horario	X	X	5587 [s]

Finalmente, para el caso muy grande, descrito en la Tabla 4.14 los tiempos se vieron acortados en el caso de lineas y aumentaron para el caso de centrales, sin embargo, como resultado general, los tiempos en general son muy similares, debido a la similitud en el tamaño de los archivos. Finalmente, cabe destacar que nuevamente por el gran tamaño de los archivos los casos de centrales y líneas no pudieron ser procesados en *Dask* ni *Pandas*.

Tabla 4.14: Comparativa de tiempo de escritura de archivos horarios según diferentes herramientas en el modelo muy grande

Archivo	Pandas	Dask	Spark
PlpBar Horario	1320 [s]	1487 [s]	3062 [s]
PlpFal Horario	<1 [s]	<1 [s]	<1 [s]
PlpLin Horario	X	X	6398 [s]
PlpCen Horario	X	X	9765 [s]

De esta forma, con estos últimos resultados, se puede observar como para *Pandas* y *Dask* es imposible crear los archivos horarios completos tanto en el caso de las centrales como de las lineas para los modelos más grandes, esto es debido a que se supera el tamaño de memoria RAM necesario para crear los *dataframes*. Este comportamiento es totalmente

esperado en los *dataframes* de *Pandas*, sin embargo, en *Dask* este problema ocurre aún cuando se realizan particiones y operaciones perezosas, esto, debido a que para la escritura del archivo *Dask* está intentando computar en memoria el archivo completo, transformando este a un *dataframe* de Pandas, por lo que el procedimiento es el mismo. De hecho, el tiempo extra de ejecución en el archivo de barras puede ser explicado con que ambas librerías realizan los mismos pasos agregando *Dask* las operaciones y transformaciones intermedias necesarias por su operación perezosa. Por otro lado, *PySpark* es la herramienta más lenta de escritura, tardando aproximadamente el doble en la escritura del archivo de barras, sin embargo dado a sus agrupaciones por clústers y trabajo paralelizado y secuencial este permite generar los nuevos archivos horarios los cuales fueron almacenados en formato *Parquet* con compresión *Snappy*.

4.4.1. Resultados Dask

Para optimizar las diferentes rutinas, estas fueron diseñadas de manera completamente análoga al procedimiento original, pero con las funcionalidades distribuidas mencionadas anteriormente en la metodología, con los usos de persistencia, reducción y número de particiones. Para lograr dichos resultados se realizaron cuatro experimentos en Dask por cada modelo, variando el número de trabajadores en cada cliente. Siempre se usó la máxima capacidad computacional disponible y los núcleos y memoria fueron distribuidos de manera totalmetne equitativa. Las diferentes pruebas realizadas fueron creadas a partir de tener 4, 8, 16 y 32 trabajadores.

Caso cuatro trabajadores:

En la primera prueba se utilizaron solo cuatro trabajadores, es decir, se dividió la capacidad computacional en cuatro partes, teniendo así cada una aproximadamente 140 gigabytes de memoria y con 12 núcleos de CPU cada uno. De esta manera se lograron obtener los siguientes tiempos de ejecución:

Tabla 4.15: Tiempo de ejecución rutinas Dask con cuatro trabajadores

Rutina / Modelo	Pequeño	Mediano	Grande	Muy Grande
Graficar Costos Marginales por barra	49 [s]	260 [s]	339 [s]	421 [s]
Graficar Flujos de energía por linea	84 [s]	681 [s]	815 [s]	453 [s]
Costos Marginales Dia y Noche	33 [s]	221 [s]	361 [s]	302 [s]
Factor Planta Anual	9 [s]	20 [s]	101 [s]	104 [s]
Factor Planta Mensual	31 [s]	72 [s]	218 [s]	293 [s]
Grafico generacion anual	2 [s]	22 [s]	97 [s]	90 [s]
Grafico pérdidas	2 [s]	13 [s]	56 [s]	48 [s]

De esta manera, en la Tabla 4.15, el primer modelo reduce sus tiempos de manera drástica, tardando solo 210 segundos en realizar la ejecución completa de todas las rutinas, es decir tan solo tres minutos y medio, prácticamente lo mismo que la ejecución de la rutina de costos

marginales en la rutina original. Además con tiempos muy similares a las rutinas optimizadas de la Tabla 4.2, la cual tardó un total de 232 segundos.

En el caso mediano para cuatro trabajadores, se aprecia como aumentan considerablemente los tiempos en general, dado el aumento de tamaño de los archivos y la cantidad de líneas, barras y centrales a procesar, tardando 1289 segundos en ejecutar todas sus rutinas, equvalente a más de 20 minutos en total, sin embargo, reduciendo el tiempo de ejecución a prácticamente un tercio de lo registrado en las rutinas originales, pero siendo la más lenta con respecto a las rutinas optimizadas, las cuales demoraron cerca de 10 minutos, siendo la rutina más pesada y que genera la demora el procesamiento para graficar los flujos de energía.

Por otro lado, para el caso grande, son 3 rutinas las que tardan mas de 5 minutos en su ejecución, estas asociadas a los archivos de barras y líneas (es decir rutinas de costos marginales y flujos por línea), tardando un total de 1987 segundos, es decir, más de 30 minutos. Sin embargo, se logra un ahorro de tiempo considerable, pensando en que la rutina original para el modelo grande tardaba en ejecutarse sobre una hora, demorando aproximadamente 75 minutos de ejecución, es decir, con una mejora de 2 veces y media la velocidad original.

Por último, en el caso muy grande, se puede observar como el tiempo total de ejecución queda en 1711 segunndos, un poco menor que en el caso grande, y prácticamente idéntico a la rutina optimizada de la Tabla 4.2 con un minuto menos de diferencia, mientras que en su caso original el tiempo de ejecución era de 1 hora y 34 minutos, es decir, disminuyendo los tiempos a un tercio. La explicación de obtener una mejora de tiempos con respecto al modelo grande es fácilmente explicable con el número de líneas o centrales de cada uno, donde se aprecia que la rutina de flujos por línea fue bastante más rápida que en el modelo anterior.

Caso ocho trabajadores:

A continuación se realizó la segunda prueba duplicando el número de trabajadores presentes en el trabajo, obteniendo los siguientes tiempos de ejecución:

Tabla 4.16: Tiempo de ejecución rutinas Dask con ocho trabajadores

Rutina / Modelo	Pequeño	Mediano	Grande	Muy Grande
Graficar Costos Marginales por barra	32 [s]	148 [s]	199 [s]	246 [s]
Graficar Flujos de energía por linea	58 [s]	312 [s]	437 [s]	211 [s]
Costos Marginales Dia y Noche	26 [s]	136 [s]	237 [s]	291 [s]
Factor Planta Anual	14 [s]	22 [s]	93 [s]	94 [s]
Factor Planta Mensual	23 [s]	71 [s]	213 [s]	287 [s]
Grafico generacion anual	8 [s]	18 [s]	89 [s]	94 [s]
Grafico pérdidas	5 [s]	13 [s]	78 [s]	30 [s]

En primer lugar, de la Tabla 4.16, para el caso pequeño los tiempos de ejecución de las rutinas disminuyen a los 166 segundos en total, mejorando así con respecto al caso con cuatro trabajadores.

Con respecto al modelo mediano, los tiempos aumentan a 720 segundos, cercano a la mitad que en el caso con cuatro trabajadores y prácticamente cinco veces menor que en el caso original.

Para el modelo grande, tambien se aprecia una disminución general en los tiempos de ejecución con respecto al caso de cuatro trabajadores, excepto en la rutina de graficar pérdidas, donde los tiempos de asignación de los diferentes nucleos son mas elevados que el propio procesamiento, por lo que es la única rutina donde el tiempo aumenta, que se entiende debido a que es la rutina más ligera por lo que realmente no requiere una elevada necesidad de procesamiento distribuido. Sin embargo, en términos generales, hay una disminución de prácticamente diez minutos, tardando esta solo poco mas de veinte minutos frente a los treinta de la rutina con cuatro trabajadores y mas de una hora en el caso original.

Finalmente, se puede ver como en el modelo muy grande, nuevamente el tiempo de ejecución al igual que en el caso de cuatro núcleos resultó levemente por debajo del modelo grande, cerca de cienn segundos menos que en el caso anterior, las explicaciones son totalmente análogas al igual que las mejoras conseguidas en tiempo.

De esta manera, en general se consiguió con ocho trabajadores disminuir casi a la mitad el tiempo que en el caso anterior con cuatro trabajadores, es decir, en promedio unas cinco o seis veces mejores tiempos que con la rutina original, demostrando esto sobre todo la importancia del procesamiento paralelo o distribuido.

Caso dieciséis trabajadores:

Luego, se diseñó el tercer experimento aumentando nuevamente el número de nodos de trabajadores a dieciséis, obteniendo los siguientes resultados.

Tabla 4.17: Tiempo de ejecución rutinas Dask con dieciseis trabajadores

Rutina / Modelo	Pequeño	Mediano	Grande	Muy Grande
Graficar Costos Marginales por barra	24 [s]	107 [s]	144 [s]	156 [s]
Graficar Flujos de energía por linea	46 [s]	233 [s]	274 [s]	153 [s]
Costos Marginales Dia y Noche	23 [s]	101 [s]	182 [s]	193 [s]
Factor Planta Anual	10 [s]	19 [s]	87 [s]	96 [s]
Factor Planta Mensual	56 [s]	72 [s]	208 [s]	275 [s]
Grafico generacion anual	45 [s]	19 [s]	86 [s]	91 [s]
Grafico pérdidas	46 [s]	12 [s]	108 [s]	29 [s]

En la Tabla 4.17 se pueden ver los tiempos asociados a las rutinas procesadas en el modelo pequeño, demorando cerca de 250 segundos, siendo en este caso peor incluso que en el caso de cuatro trabajadores, aquello puede ser explicado por una división muy grande de los datos en la cual existe un mayor tiempo asociado a la distribución de recursos que al propio procesamiento.

En el caso del procesamiento de las rutinas para el modelo mediano con dieciséis traba-

jadores, se logra apreciar que esta vez los tiempos si dismunyen con respecto a trabajar con solo ocho trabajadores. Tardando en este caso solo 563 segundos en total, casi tres minutos menos que en el caso anterior, reflejando como al incrementarse el tamaño de los archivos la división o paralelización de estos logra seguir mejorando los resultados.

Por otro lado, para el modelo grande, se pueden observar los resultados del tercer experimento con dieciséis trabajadores, donde nuevamente se observa una mejora de tiempos con respecto al experimento anterior, pero esta vez esta disminución es menos exagerada en proporción con respecto a los resultados entre cuatro a ocho núcleos, pero aun mejor que en el caso mediano, esta vez con 257 segundos de mejora, es decir mas de cuatro minutos.

A continuación, para el caso muy grande con dieciséis trabajadores, en el cual análogamente a los resultados anteriores los resultados son muy similares en tiempos al caso grande, con 993 segundos, nuevamente menos de dos minutos de diferencia entre ambos y con los dos mejorando sus tiempos con respecto al caso anterior con ocho trabajadores. Esta diferencia nuevamente se explica por la distribución y tamaño de datos en las líneas en Plplin y por la inexistencia de datos en el archivo de centrales de falla (PlpFal) para el modelo muy grande, además de los tamaños relativamente similares al convertir en casos horarios. Además, se destaca que al ser el archivo de centrales el mas desbalanceado en tamaño este logra la mayor diferencia de tiempos al calcular los factores de planta mensuales, siendo esta rutina en los demás casos un poco menos exigente, pero en esta última destaca por ser la de ejecución mas lenta.

Caso treinta y dos trabajadores:

Finalmente se intentó realizar un cuarto experimento dividiendo el cliente en 32 trabajadores, sin embargo, dada la cantidad de memoria por trabajador se obtuvieron errores y fugas de memoria por lo que resultó inviable trabajar con dicha cantidad de nodos.

```
create_file_factor_planta_anual(output_path)

distributed.nanny - WARNING - Worker exceeded 95% memory budget. Restarting
distributed.nanny - WARNING - Restarting worker
distributed.nanny - WARNING - Restarting worker
distributed.nanny - WARNING - Restarting worker
distributed.nanny - WARNING - Restarting worker
```

Figura 4.1: Error de memoria de worker en Dask

En la figura 4.1 se puede apreciar un ejemplo de como en el procesamiento de cálculo de la rutina de obtención del factor de planta anual, los diferentes wokers no son capaces de procesar toda la información y se obtiene una saturación en la memoria. Por ello, *Dask*, al percatarse de aquello reinicia el respectivo trabajador donde ocurre el error.

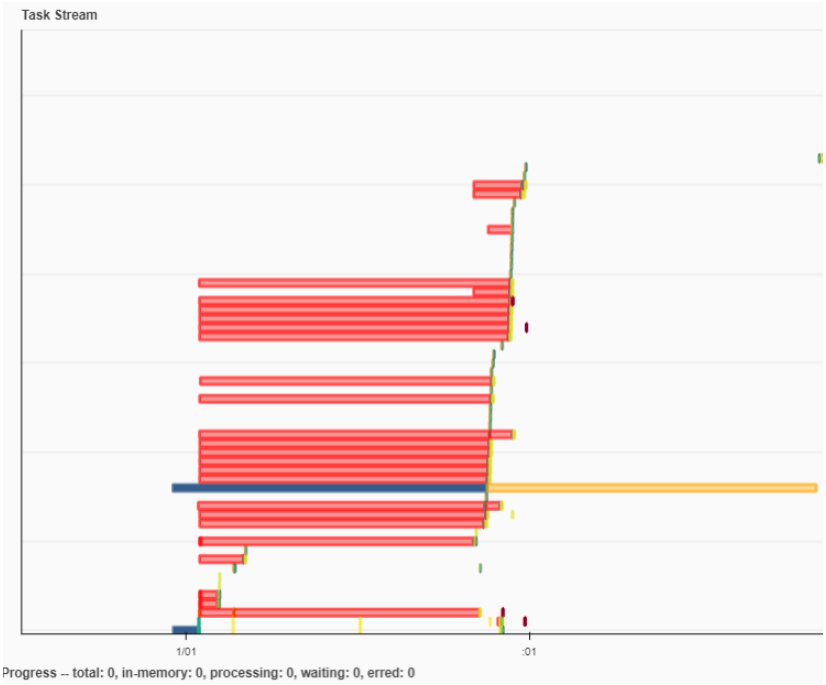

Figura 4.2: Ejemplo de visualización de procesamiento con 32 núcleos

Esto último se puede apreciar en la visualización en tiempo real de este proceso en la figura 4.2, donde se puede ver como prácticamente todas las tareas están demoradas y comenzando a reiniciarse puesto que el sistema no es capaz de procesar en memoria los datos.

De esta manera, se finalizan todos los casos de ejecución de rutinas con Dask y se destaca como el procesamiento distribuido y la ejecución perezosa se vuelven cada vez mas relevantes a medida que los casos de estudio crecen en tamaño, se hace notar además la diferencia entre el caso de cuatro núcleos y ocho contra el de dieciséis, donde entre los dos primeros ejemplos se logran grandes mejoras mas significativas con respecto a la velocidad de procesamiento, mientras que en el caso de dieciséis trabajadores la mejora no es tan considerable, claramente esto se debe a que aún cuatro trabajadores no eran suficientes para estos casos mientras que mas adelante el margen de mejora era mas limitado, acercandose mas al punto óptimo. Por último merece destacar el caso de treinta y dos trabajadores el cual debió ser descartado debido a que por limitaciones computacionales no era capaz de ejecutar las rutina en los modelos grandes y como se vio en un inicio estas mejoras no son tan relevantes en los casos pequeños.

4.4.2. Resultados PySpark

Con lo anteriormente mencionado, se elaboraron algunas de las rutinas del modelo PLP utilizando *PySpark* en su diseño, las rutinas creadas fueron las de costos marginales, generación y pérdidas.

Tabla 4.18: Tiempo de ejecución rutinas PySpark rutinas modelo PLP grande

Rutina	Tiempo (s)
Graficar Costos Marginales por barra	1500
Graficar Flujos de energía por linea	X
Costos Marginales Dia y Noche	X
Factor Planta Anual	X
Factor Planta Mensual	X
Grafico generacion anual	37
Grafico pérdidas	14

Así, se pudo notar en los resultados que al no tener un *cluster* distribuido los resultados no tenían mejoras con respcto a trabajar directamente con Pandas, puesto que al trabajar de manera cíclica sin paralelizar no merece la pena utilizar otras herramientas pensando en los robusta y funcional que es la primera. Por lo que mientras no se logre distribuir los nodos en el servidor de manera local se descarta realizar las rutinas en esta herramienta.

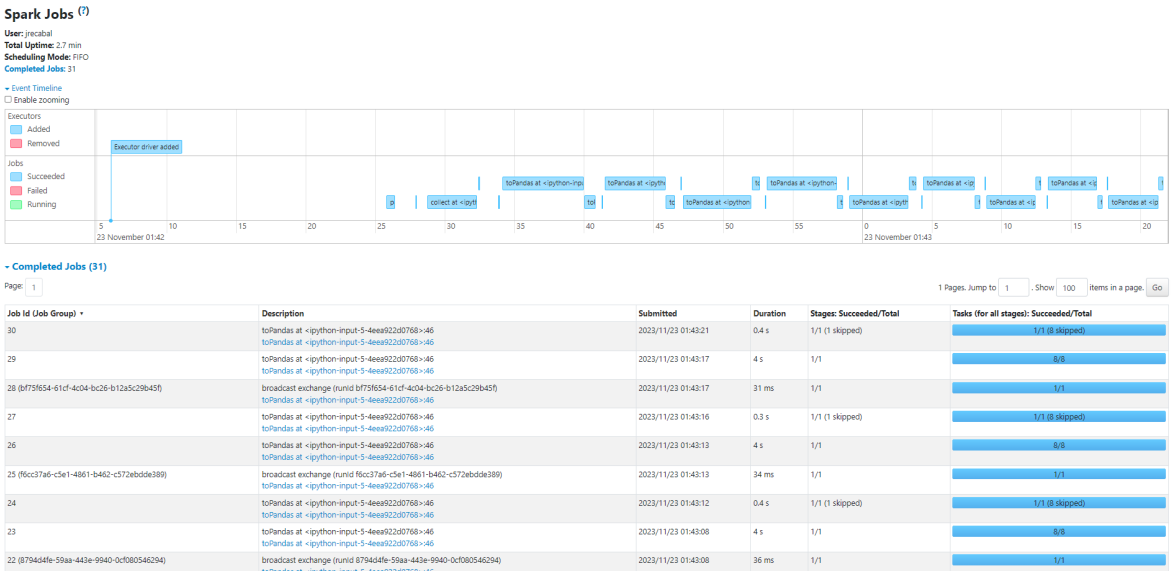

Figura 4.3: Ejecución de rutina en PySpark

En este ejemplo, en la figura 4.3, se puede apreciar la interfaz gráfica de *Spark* con la rutina de costos marginales, donde se puede apreciar el procesamiento lineal que esta posee y la necesidad de optimizar los recursos para mejorar sus resultados.

4.4.3. Problemas Pyspark

Como se mencionó en la sección anterior, se trabajó con *Pyspark* de manera local, creando un solo trabajador con toda la memoria y procesamiento del servidor. Esto debido a que no se logró crear un clúster local para el entorno de trabajo.

En primer lugar, se intentó realizar aquello directamente desde las configuraciones de *Spark*, con el parámetro *Spark_Worker_Instances*, donde, al igual que en *Dask* se buscaba que el propio sistema distribuyese los recursos del servidor. Además, como se vio en la metodología, en la figura 3.9, se buscó configurar además estos parámetros directamente desde *Pyspark* en *Python* al inicializar el cliente de la herramienta, se intentó modificando el número de núcleos por cada ejecutor o variando la memoria de cada uno de estos. Sin embargo, con ello la herramienta en local generaba un solo nodo con la memoria asignada, sin utilizar la memoria restante en dividir el trabajo en otros nodos. De este modo, se concluyó que no fue posible dividir los núcleos entre múltiples ejecutores en modo local debido a que existe un solo ejecutor y las configuraciones de instancias o núcleos por ejecutor se ignoran al configurar el programa en este modo.

Luego, se intentó crear un clúster independiente en *Docker*[14], creando contenedores separados para los diferentes nodos ya sea el maestro o los diferentes nodos esclavos. Para ello, se utilizó una imagen de *Docker* con las dependencias necesarias y la instalación de *Spark*, para de modo crear una imagen de *Docker* para inicializar los diferentes contenedores.

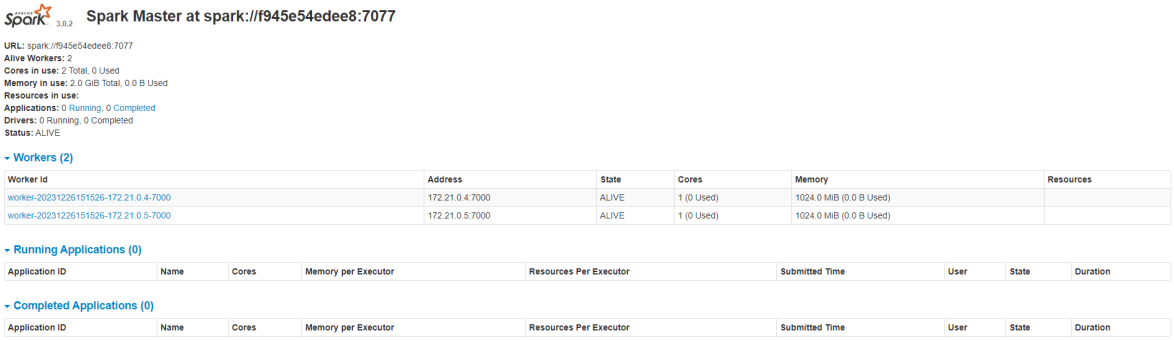

Figura 4.4: Interfaz nodo maestro Pyspark

De este modo, como se puede ver en la figura 4.4 se logra crear un nuevo nodo maestro además de dos nodos trabajadores o esclavos que permiten realizar los trabajos en paralelo, sin embargo, no se logró conectar estos a la API de *PySpark* y fijar este clúster como el sistema de procesamiento. No se logró identificar el error de este problema, pero se entiende que es un problema de comunicación entre el programa principal y el puerto del nodo maestro. Ante ello, solo se logró realizar las pruebas con el modo disponible local para *PySpark* otorgándole todos los recursos al *driver* o nodo maestro.

Finalmente, al tener que utilizar esta herramienta solo en modo local, no se realizan todas las pruebas y rutinas, ya que al intentar la ejecución de las rutinas en el modelo grande, esta fue demasiado lenta y no presenta ninguna mejora con respecto al caso base.

Capítulo 5

Conclusiones

5.1. Conclusiones generales

Para este trabajo se ha visto como la optimización de las rutinas del modelo PLP han resultado en una transformación significativa de la eficiencia de procesamiento, adoptando herramientas específicas para entornos de *Big Data*, como *Apache Spark* y *Dask*. Estas herramientas logran ofrecer un enfoque de procesamiento distribuido, dividiendo tareas entre nodos para aprovechar la capacidad de múltiples procesadores y superar las limitaciones de ejecución en memoria. La escalabilidad inherente a *Spark* y *Dask* se convierte en un elemento crucial, al permitir el manejo efectivo de conjuntos de datos voluminosos y cargas de trabajo muy pesadas, asegurando un rendimiento constante a medida que los requisitos de datos aumentan.

La elección del formato de archivo *Parquet* se presenta como un componente esencial de la optimización. La compresión eficiente de *Parquet* no solo reduce drásticamente los tamaños de archivos, mejorando la gestión de almacenamiento, sino que también contribuye a tiempos de transferencia más rápidos. Su estructura de almacenamiento por columnas y su capacidad para crear metadatos permiten un acceso rápido a los archivos, agilizando las operaciones, especialmente cuando el volumen de datos crece, obteniendo así un mucho mejor desempeño en comparación con formatos de texto plano como *CSV*. Los resultados obtenidos en este caso, muestran como en los archivos de salida originales, los cuales en formato *CSV* pesaban decenas de *gigabytes*, son reducidos a solo uno o inclusive reduciendo el tamaño a solo cientos de *megabytes*, mientras que en las transformaciones horarias de estos archivos es cuando estos formatos mejor funcionaron, donde archivos con cientos de *gigabytes* o *terabytes* inclusive, son reducidos a archivos de tamaños similares a los originales, reduciendo así alrededor de dos ordenes de magnitud el volumen de los archivos. Finalmente, se elige *Parquet* por sobre *Feather*, el otro formato analizado, principalmente por ser un formato mas robusto que consigue mejores métricas de compresión de tamaño y velocidad de lectura, además de mejor compatibilidad con las diferentes herramientas, ya que, en particular *Spark* no es capaz de leer archivos en formato *Feather*.

La gestión eficiente de recursos es otro aspecto destacado, donde *Spark* y *Dask* optimizan

el uso de nodos y procesadores, evitando en gran parte cuellos de botella en donde una sola tarea retrasa por completo la ejecución y mejorando así la velocidad de procesamiento. La ejecución perezosa, una característica clave, reduce la carga innecesaria en la memoria al postergar la evaluación de operaciones hasta el último momento posible. Específicamente la herramienta que obtuvo mejores resultados fue *Dask*, reduciendo los tiempos de ejecución de las rutinas entre dos a seis veces con respecto a los resultados iniciales, donde para esta herramienta resultó fundamental la modificación en el número de trabajadores o nodos para dividir el trabajo, logrando así trabajar en paralelo con las diferentes centrales, barras o lineas para calcular los respectivos datos necesarios para cada rutina. Por otro lado con *Spark* no se logró integrar la aplicación de manera distribuida con un grupo de nodos en local, por lo que la mejora en tiempos de procesamiento no se logró apreciar, sin embargo, se destaca de esta última su robustez para trabajar con archivos muy grandes fuera del tamaño de la memoria, donde si bien la aplicación al estar trabajando con un único nodo era lenta, la forma de realizar las particiones permite de manera general manejar archivos demasiado voluminosos de manera sencilla y práctica.

En conjunto, la integración de Apache *Spark* y *Dask* con el formato *Parquet* permite crear un ecosistema más optimizado para el procesamiento eficiente y escalable de grandes volúmenes de datos. La ejecución perezosa y distribuida se combinan para utilizar de manera efectiva los recursos disponibles, mejorando la eficiencia general de las rutinas. Estas soluciones proporcionan una solución que logra mejorar las limitaciones de herramientas convencionales, ofreciendo un marco más robusto para enfrentar los desafíos específicos asociados con el procesamiento de grandes conjuntos de datos.

5.2. Trabajo futuro

El avance de este trabajo sienta las bases para posibles mejoras y exploraciones futuras que podrían potenciar aún más el rendimiento y la eficiencia del procesamiento de las rutinas desarrolladas a partir de las salidas del modelo PLP. En particular, se identifican áreas clave que podrían constituir líneas de investigación valiosas.

En primer lugar, se sugiere la continuación del desarrollo del *cluster* local en el servidor para Apache *Spark* y así evaluar la capacidad de escalabilidad del modelo en este entorno. Esto permitiría aprovechar las capacidades latentes de procesamiento paralelo que ofrece *Spark*, reduciendo, tal vez, los tiempos de ejecución y permitiendo el manejo eficiente de los datos.

Además, la exploración de servicios en la nube, como AWS o Google Cloud, ofrece la posibilidad de evaluar el rendimiento del modelo en entornos remotos. Esta exploración no solo proporcionaría información sobre la eficiencia del modelo, ya que, aún el servidor local es lo suficientemente poderoso computacionalmente, sino que también permitiría analizar los costos asociados y determinar la viabilidad de implementaciones a una mayor escala.

En el ámbito de herramientas alternativas, se destaca la importancia de investigar otras herramientas como Vaex para una posible alternativa a las actuales. La eficiencia de estas en el manejo de grandes conjuntos de datos podría ofrecer mejoras sustanciales en términos de

velocidad y eficiencia que aún no han sido exploradas. La evaluación comparativa de estas herramientas con las soluciones actuales sería crucial para determinar su idoneidad.

Finalmente, también puede seguir siendo relevante la búsqueda o actualización de nuevos formatos de archivo eficientes, más allá de *Parquet* y *Feather*. Explorar opciones alternativas podría revelar soluciones que ofrezcan beneficios adicionales en términos de tamaño y velocidad de procesamiento.

Bibliografía

[1] Plexos, la herramienta para desvelar los secretos de los mercados de la energía, 2017. [Online]. Available: https://www.energias-renovables.com/panorama/plexos-la-herramienta-para-desvelar-los-secretos-20170301.

[2] Dask, 2023. [Online]. Available: https://www.dask.org/.

[3] E-viewer 2.0, 2023. [Online]. Available: https://e-viewer.centroenergia.cl/.

[4] Spark, 2023. [Online]. Available: https://spark.apache.org/.

[5] Spark web ui – understanding spark execution, 2023. [Online]. Available: https://sparkbyexamples.com/spark/spark-web-ui-understanding/.

[6] What is dag in spark or pyspark, 2023. [Online]. Available: https://sparkbyexamples.com/spark/what-is-dag-in-spark/#h-2-importance-of-dag-in-spark.

[7] Venkataramu P. S. Daram S. B. Angadi, R. V. Role of big data analytics in power system application. e3s web of conferences., 2020. [Online]. Available: https://www.e3s-conferences.org/articles/e3sconf/pdf/2020/44/e3sconf_icmed2020_01017.pd.

[8] Luis Bolvarán C. Análisis operacional del proyecto hidroaysén en contraste con una alta entrada de generación en base a ernc en el sic, 2010. [Online]. Available: https://repositorio.uchile.cl/bitstream/handle/2250/103914/cf-bolvaran_lc.pdf?sequence=3&isAllowed=y.

[9] Prado A. Arboleya P. Terzija V. Dominguez, X. Evolution of knowledge mining from data in power systems: The big data analytics breakthrough. electric power systems research., 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378779623000822.

[10] Elsevier. Big data application in power systems - 1st edition., 2017. [Online]. Available: https://www.elsevier.com/books/big-data-application-in-power-systems/arghandeh/978-0-12-811968-6.

[11] Gabriel Neagu Florin Pop. Big data platforms and applications in computer communications and networks., 2021. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-38836-2.

[12] Stephanie Kirmer. Lazy evaluation with dask, 2021. [Online]. Available: https://saturncloud.io/blog/a-data-scientist-s-guide-to-lazy-evaluation-with-dask/.

[13] Renato Valenzuela V. Modelo de coordinación hidrotérmica plp: Diseño de aplicaciones para futura implementación en laboratorio de simulación del departamento de ingeniería eléctrica, 2018. [Online]. Available: https://bibliotecadigital.ufro.cl/?a=view&item=2006.

[14] Marco Villarreal. Creating a spark standalone cluster with docker and docker-compose, 2021. [Online]. Available: https://github.com/mvillarrealb/docker-spark-cluster.

[15] Huang T. Bompard E. F. Zhang, Y. Big data analytics in smart grids: a review. energy informatics., 2018. [Online]. Available: https://energyinformatics.springeropen.com/articles/10.1186/s42162-018-0007-5.