PRÁCTICA 1

OPTIMIZACIÓN DE GRANDES VOLÚMENES DE DATOS

Álvaro Fraile, Esteban Martínez, Jaime Álvarez, Alejandro Mendoza

Dispositivo hardware y configuración de los experimentos.

El hardware utilizado ha sido el mismo a lo largo de todos los experimentos. Se ha utilizado un procesador con AMD Ryzen 5 7600X con 6 cores y 12 hilos. Además, los experimentos se han realizado en un contenedor docker, corriendo el mínimo número de procesos en el dispositivo, para que los resultados sean comparables. Además, se ha establecido un máximo de 10 iteraciones de descenso del gradiente. Cabe destacar que el docker utilizaba todos los hilos por lo cual en algunas pruebas el procesador puede que esté corriendo tareas del sistema operativo y programas a la vez, aunque durante las pruebas se han cerrado todas las aplicaciones, dejando solo el entorno de Spark.

Ejercicio 1

Ejercicio 1a: Exploración de learning rate

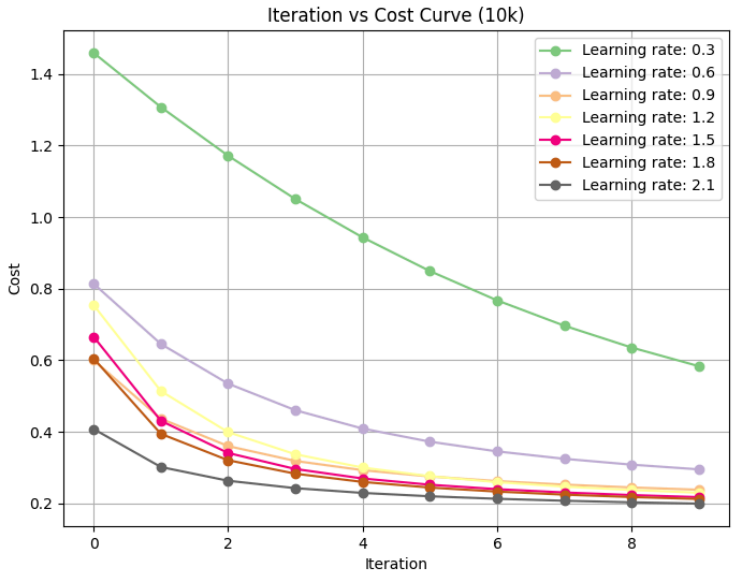


Figura 1a. Comparación de distintos learning rates para el dataset de 10k

El *learning rate* con valor 2.1 es el que mejor pérdida obtiene tras las 10 iteraciones del descenso del gradiente (Figura 1). Se debe a que los saltos de los cambios de los pesos son más grandes, y como se estableció un máximo de 10 iteraciones del descenso del gradiente, converge antes a una pérdida menor. Los learning rates con valores 0.3 y 0.6 tienen una pérdida demasiado grande en comparación con los otros valores. El resto de learning rates parece que convergen a un mismo valor de pérdida (las diferencias son muy pequeñas).

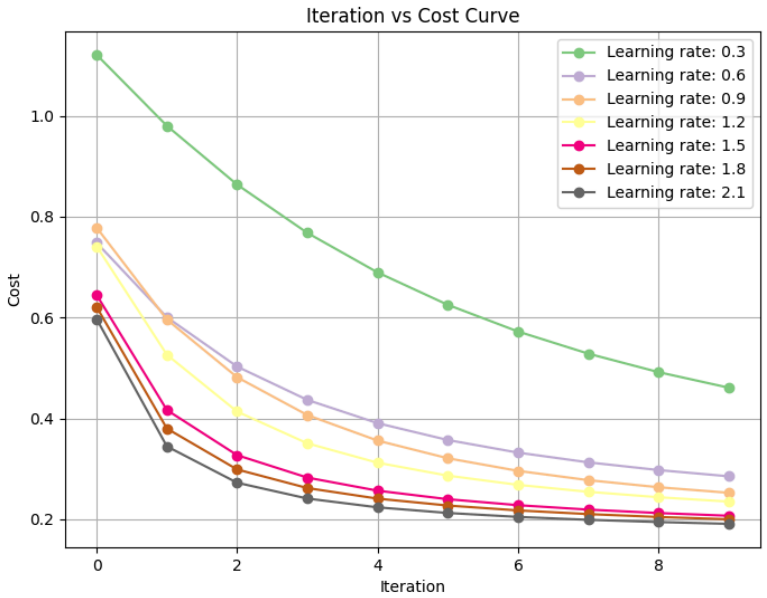


Figura 2. Comparación de distintos learning rates para el dataset de 1M.

El *learning rate* con valor 2.1 vuelve a ser el mejor en rendimiento (Figura 2). De nuevo, se debe a que los saltos de los gradientes son más grandes, y la pérdida converge más rápido. No obstante, otros valores hasta 1.5 obtienen pérdidas similares.

Ejercicio 1b: Exploración de regularización

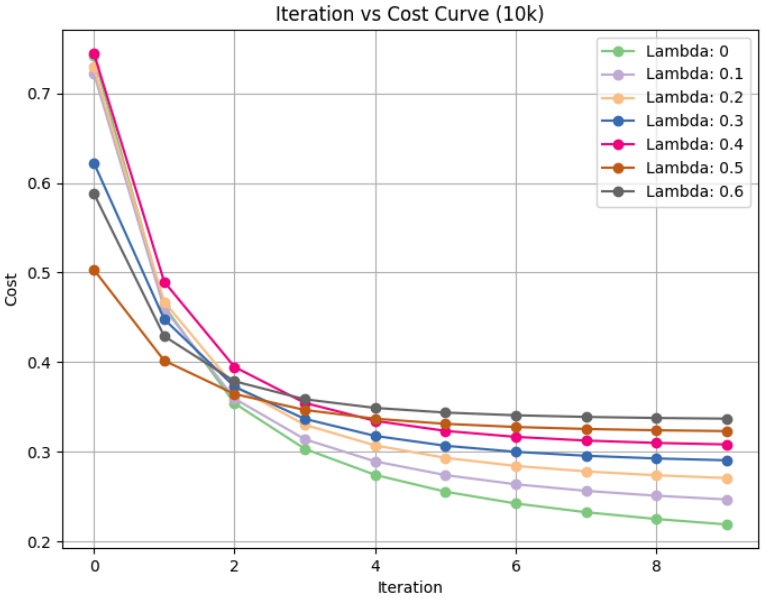


Figura 3. Comparación de distintas regularizaciones para el dataset de 10k.

La regularización consiste en penalizar los pesos muy grandes en valor absoluto. Lambda es el factor por el que se multiplica a ese valor, y según la Figura 3, el mejor lambda es el de valor 0. Esto significa que para este problema y para esta arquitectura no es importante el tamaño de los pesos. De hecho, si el valor de lambda es muy grande (0.6) se obliga a que los pesos sean cercanos a 0, y por lo tanto, es posible que se esté siendo demasiado

restrictivo, obteniendo peores métricas. La regularización L2 se ha demostrado que no es la técnica más efectiva, siendo sustituida en muchos casos por el dropout. Valores altos de λ se estabilizan en una pérdida mayor, lo que indica que la regularización evita que el modelo se ajuste a los datos de entrenamiento.

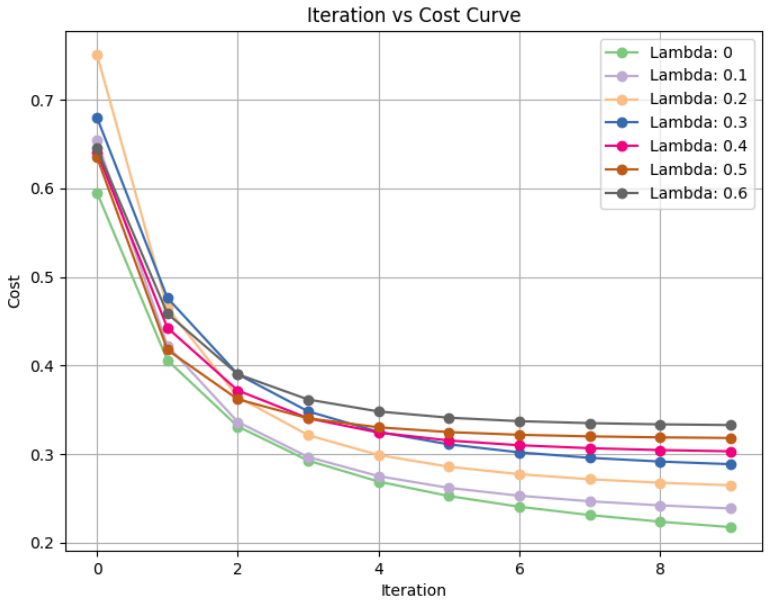


Figura 4. Comparación de distintas regularizaciones para el dataset de 1M.

De nuevo se obtienen los mismos resultados para el dataset de 1M, el mejor lambda es 0, eliminando la regularización L2 de la función de pérdida. Que los resultados para ambos conjuntos de datos sean iguales, nos hace pensar que el subconjunto de 10k sigue la misma distribución estadística que el dataset de 1M.

Ejercicio 2: Rendimiento y speed-up

Para la gráfica de rendimiento, las líneas verticales indican el número de **cores físicos (6)** e **hilos lógicos (12)**.

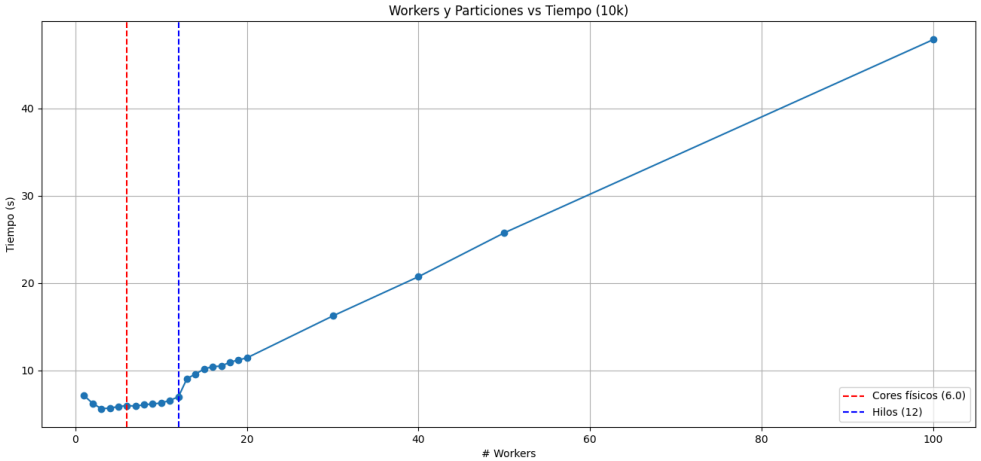


Figura 5. Rendimiento dependendiendo dell número de workers y particiones, dataset 10k.

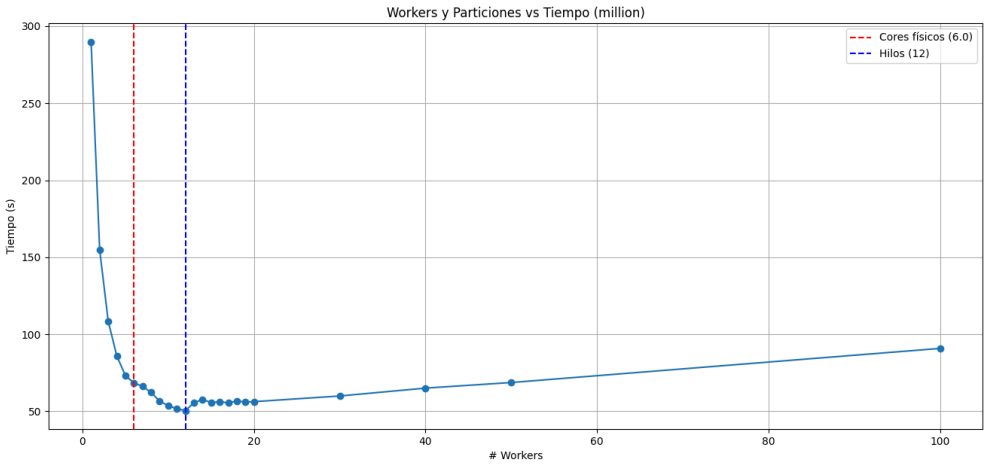


Figura 6. Rendimiento dependendiendo del número de workers y particiones, dataset 1M.

En ambas figuras se puede ver que para valores bajos(1-6), el tiempo que se tarda en ejecutar decae de forma considerable. Esto se debe a que Spark es capaz de distribuir la carga entre más cores funcionando de forma más eficiente.
La zona más óptima se encuentre entre el número físico de cores y el número de hilos, esto es donde el sistema maximiza el rendimiento.
Cuando se aumenta el número de workers a un número mayor que el número de hilos el sistema deja de mejorar, incluso empeorando significativamente los tiempos. Entonces a partir de cierto número de workers y particiones el rendimiento decrece considerablemente.

Para concluir, para obtener un rendimiento óptimo el número de workers y particiones debe estar comprendido entre número de cores y el de hilos. Aumentar más allá de estos límites el número de workers empeora el rendimiento.

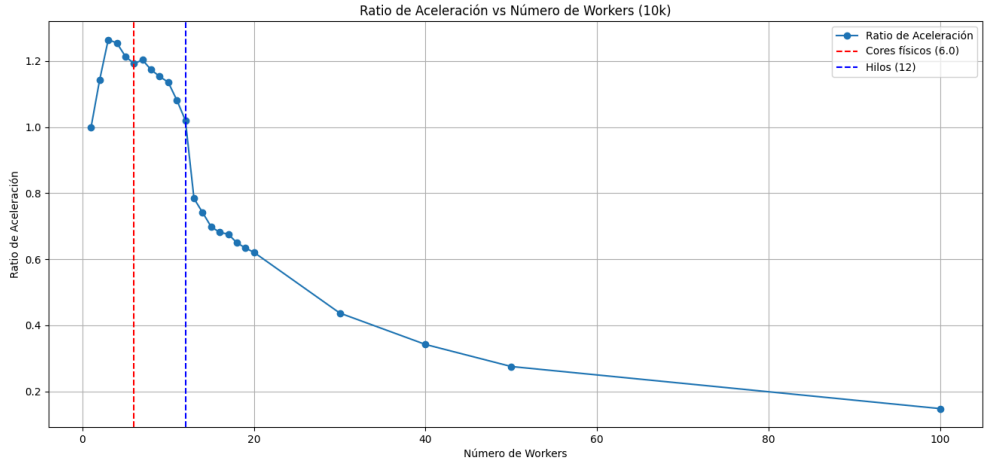


Figura 7. Ratio de aceleración dependiendo del número de workers y particiones, dataset 10k.

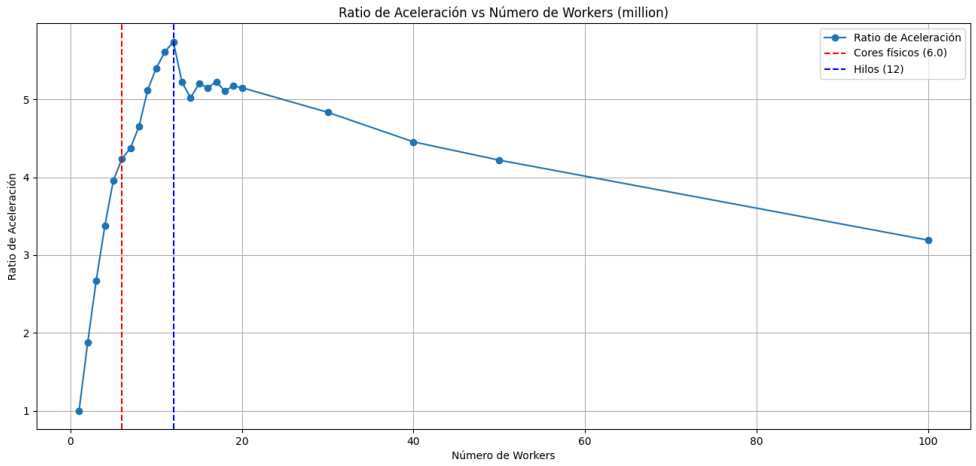


Figura 8. Ratio de aceleración dependiendo del número de workers y particiones, dataset 1M.

En ambas figuras se aprecia la relación con las gráficas de rendimiento previas, en la gráfica de un millón de ejemplos se observa como el ratio de aceleración mejora hasta llegar al número de hilos del sistema, a partir de ese punto el speed up se ve afectado degradando de manera significante debido a la sobrecarga de los workers.
El comportamiento con 10k ejemplos es el mismo, lo único que al trabajar con menos ejemplos y la escala es inferior se ve que como afecta porcentualmente al speed up es mayor. Además en este caso vemos un mínimo local al llegar al número de workers igual al número de núcleos físicos, pero esto puede ser causado por procesos de fondo del sistema operativo.

Ejercicio 3

Ejercicio 3a: Estrategias de “caching”

Se prueban diferentes estrategias de *caching* por separado, así como la combinación entre ellas. Se distinguen 3 tipos de RDD cacheados:

1. El dataset normalizado (antes de entrar al *train*)
2. Los gradientes del *train,* necesarios para calcular los pesos del modelo.
3. Las predicciones del modelo en el train, necesarias para calcular los gradientes.

Para ambos casos se establece el número de workers y particiones a 12, pues es la combinación más óptima en el caso de un millón y de las mejores en el caso de 10k.

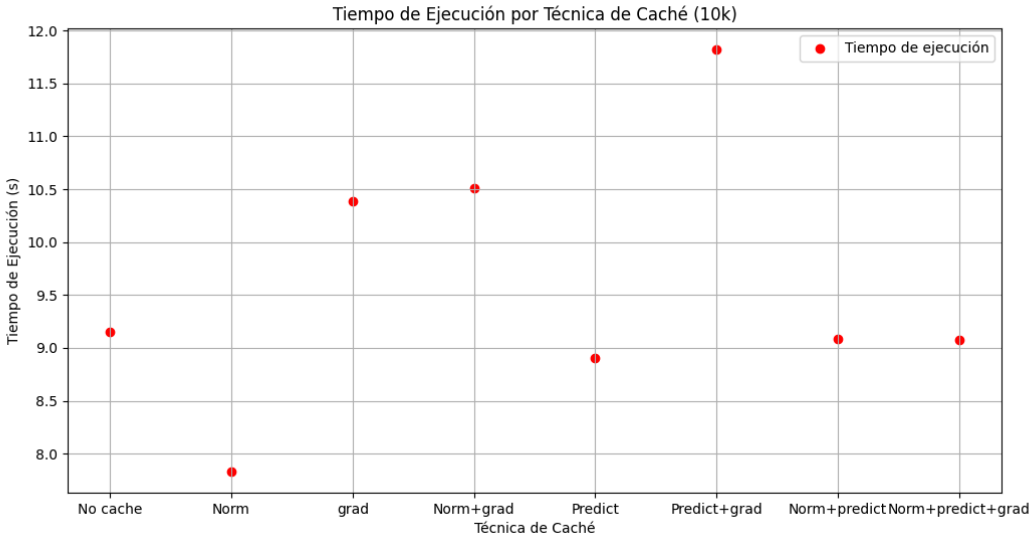


Figura 9. Comparación de tiempos según la técnica de *caching*, dataset 10k.

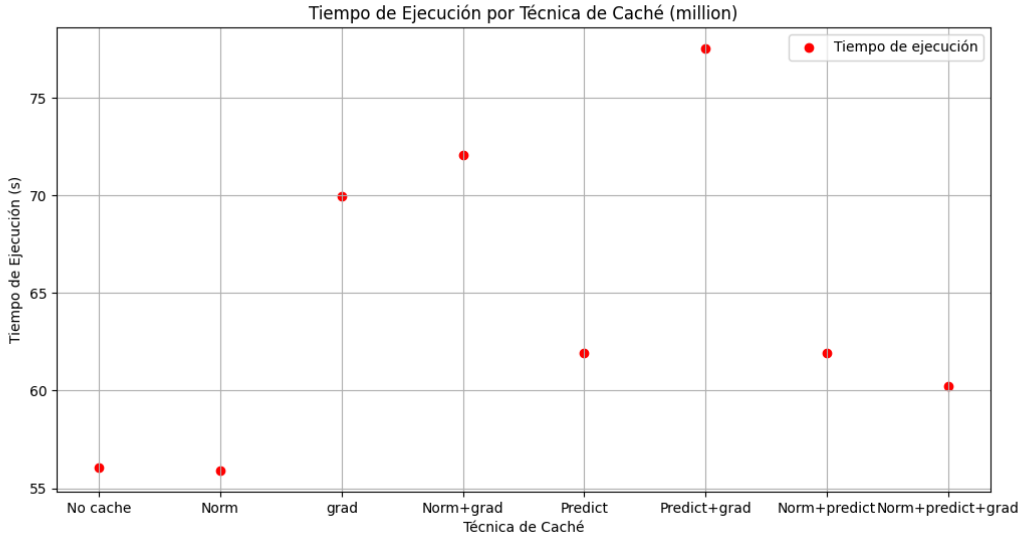


Figura 10. Comparación de tiempos según la técnica de *caching*, dataset 1M.

Como se puede ver en ambas figuras el normalizado es la técnica de cacheado más efectiva que se ha encontrado esto se debe a que cada vez que se calculan los gradientes las predicciones etc, se tienen que calcular los gradientes de todas las filas por el número de iteraciones. Además, la operación de almacenamiento en memoria sólo ocurriría una vez justo antes de ejecutarse el train, frente a las demás estrategias, pues los rdds que se cachén al estar dentro del bucle de iteraciones del modelo se escribirán varias veces, por este hecho concluimos que es más efectivo cachear el rdd normalizado, pues se escribe una única vez en memoría y se utiliza en todas las iteraciones, frente a las demás técnicas que se usan como máximo 2 veces en el caso del predict y 2 el gradiente por iteración haciéndolo menos óptimo. Además como se puede ver para la máquina que ha ejecutado el jupyter la diferencia de tiempo no es significativa y no se distancia mucho de no cachear. La razón por la cual el resto de estrategia no mejoran al no cachear se debe al tiempo de

escritura que debe hacer spark por cada rdd que se cachea que como se ha comentado en el caso de predict y gradientes es una vez por iteración.

Ejercicio 3b: Comparación entre *cache* y no *cache*

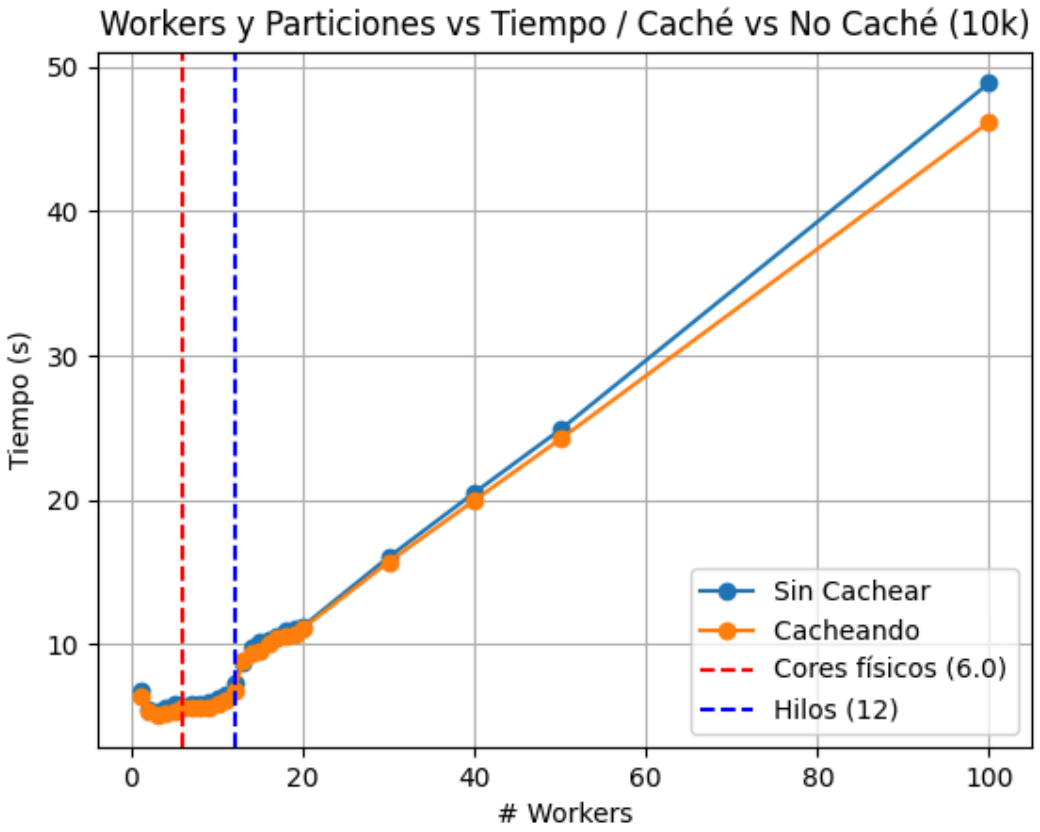


Figura 11. Comparación entre caché y no caché, dataset 10k.

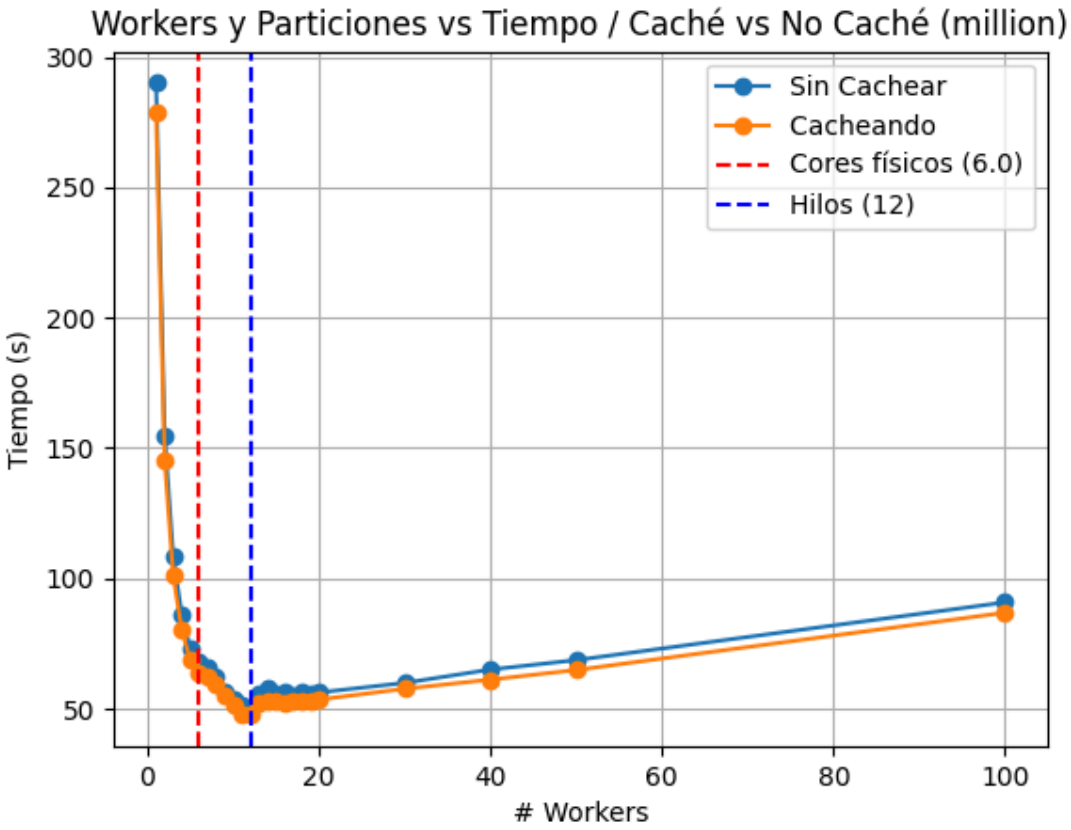


Figura 12. Comparación entre caché y no caché, dataset 1M.

Como hemos visto antes la estrategia de caché que generaba una mayor mejora era cachear el RDD de normalizado, por lo que ahora compararemos esta estrategia contra no cachear nada con diferentes números de workers y particiones.
Ambas variantes de ejecución producen curvas prácticamente iguales, pero en estos casos se observa que el tiempo de ejecución con caché es mejor. Aún así en la mayor parte de la curva ambos puntos están prácticamente solapados con diferencias pequeñas tanto en el dataset de un millón como en el dataset de 10k

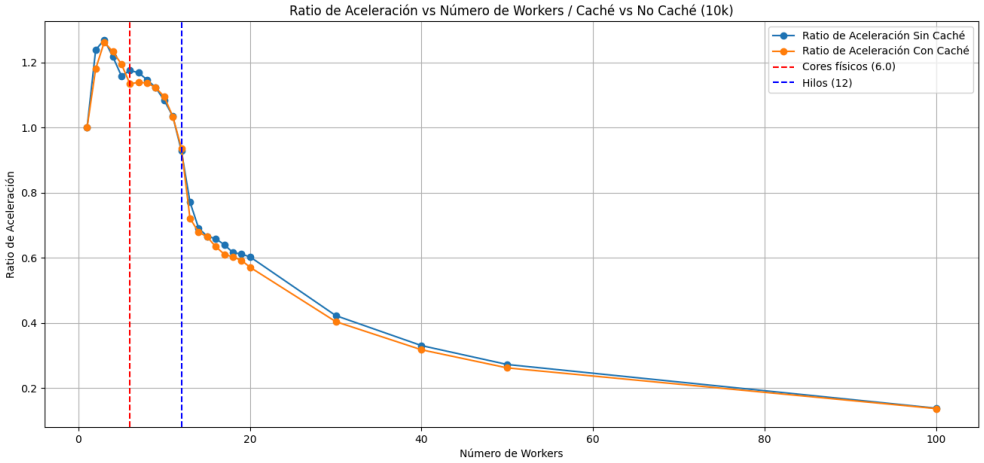


Figura 13. Aceleración entre caché y no caché, dataset 10k.

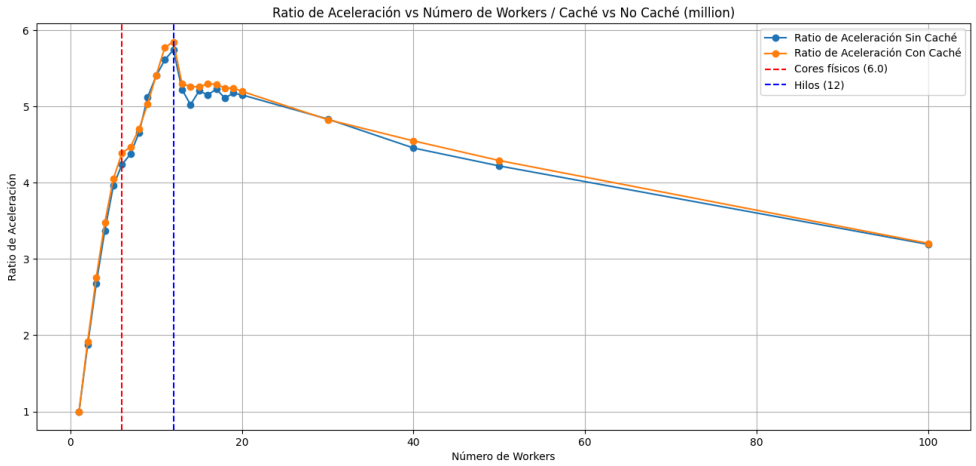


Figura 14. Comparación entre caché y no caché, dataset 1M.

En las gráficas de aceleración podemos ver que para el dataset reducido cachear reduce el ratio de aceleración obtenido al aumentar los workers y particiones, mientras que en el dataset completo vemos que ambos ratios de aceleración son muy similares. Estas diferencias son causadas porque la ganancia que ofrece cachear el dataset reducido es menor que el coste de cachear esos datos, pero a más datos tengamos, es más beneficioso cachear.
A causa de la gráfica anterior, y por ser un dataset de mayor tamaño, vemos cómo se recrudece la aceleración al sobrepasar el límite de workers.

Ejercicio 4: Efecto del nº de particiones

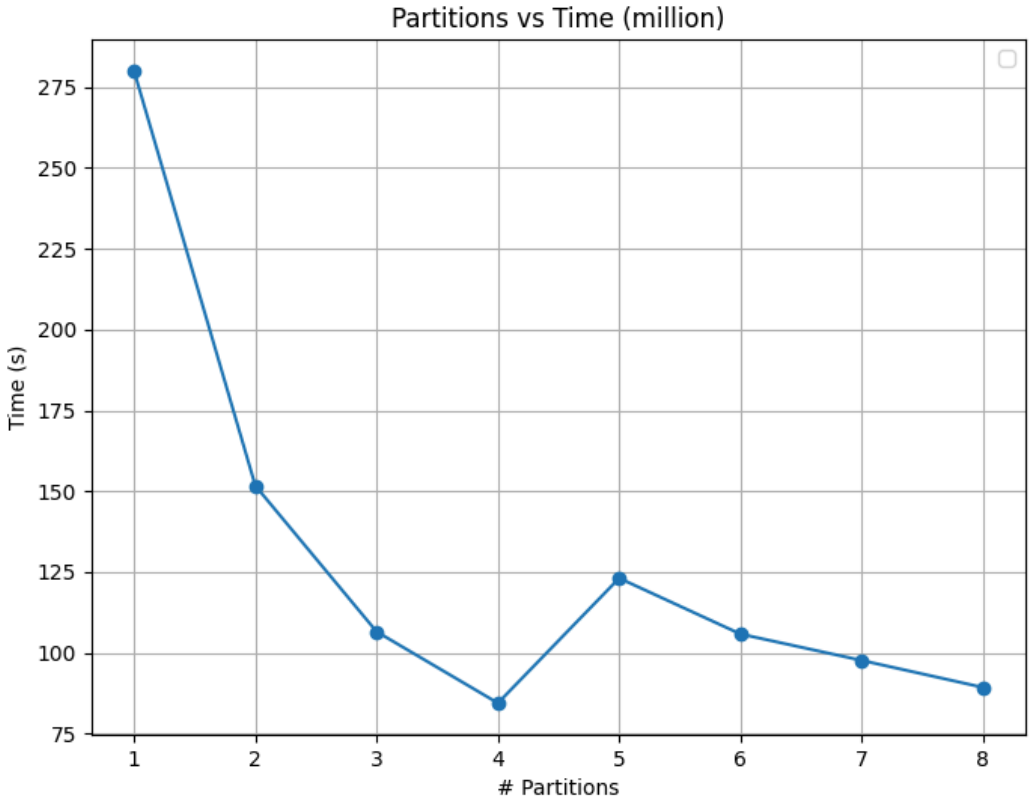


Figura 15. Comparación de número de particiones para 4 workers. Dataset 1M.

Hay estáticamente 4 workers. El rendimiento mejora cuando más cerca estamos de que todos los workers tengan el mismo número de particiones, es este caso, con 4 workers y 4 particiones cada worker tendría una única partición, y con 4 workers y 8 particiones cada worker tendría 2 particiones asignadas. Ambos de estos puntos son los que menor tiempo de computación tienen. Esto se debe a los tiempos de sincronización y transmisión de información entre workers.

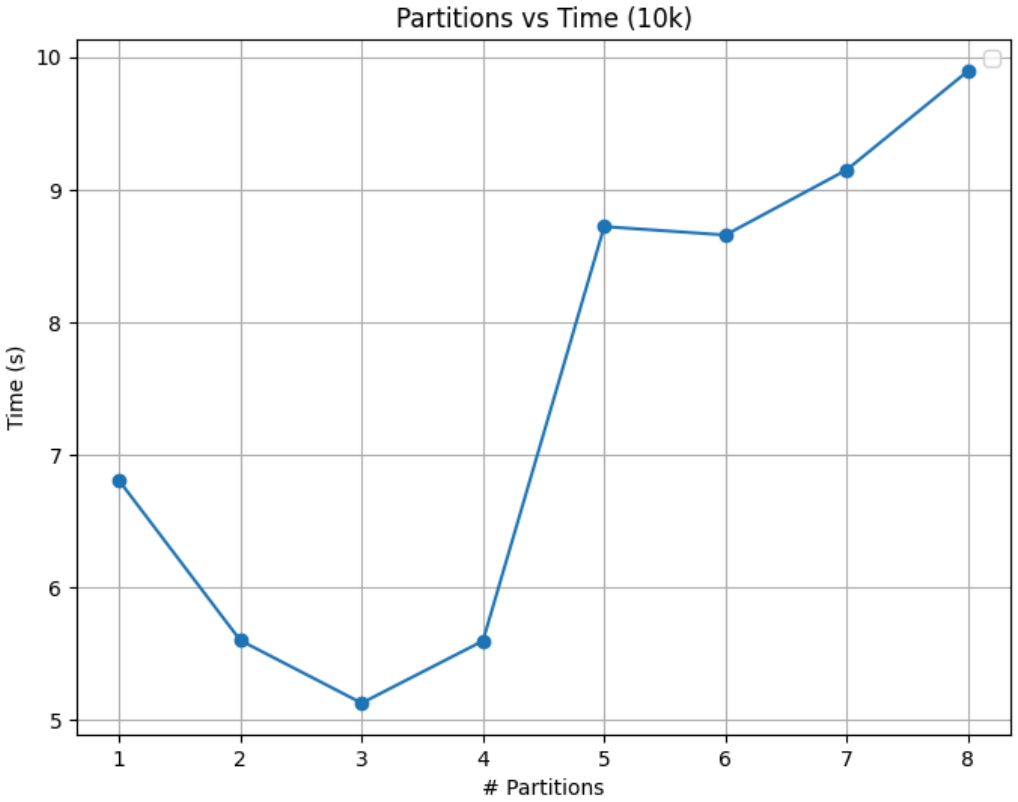
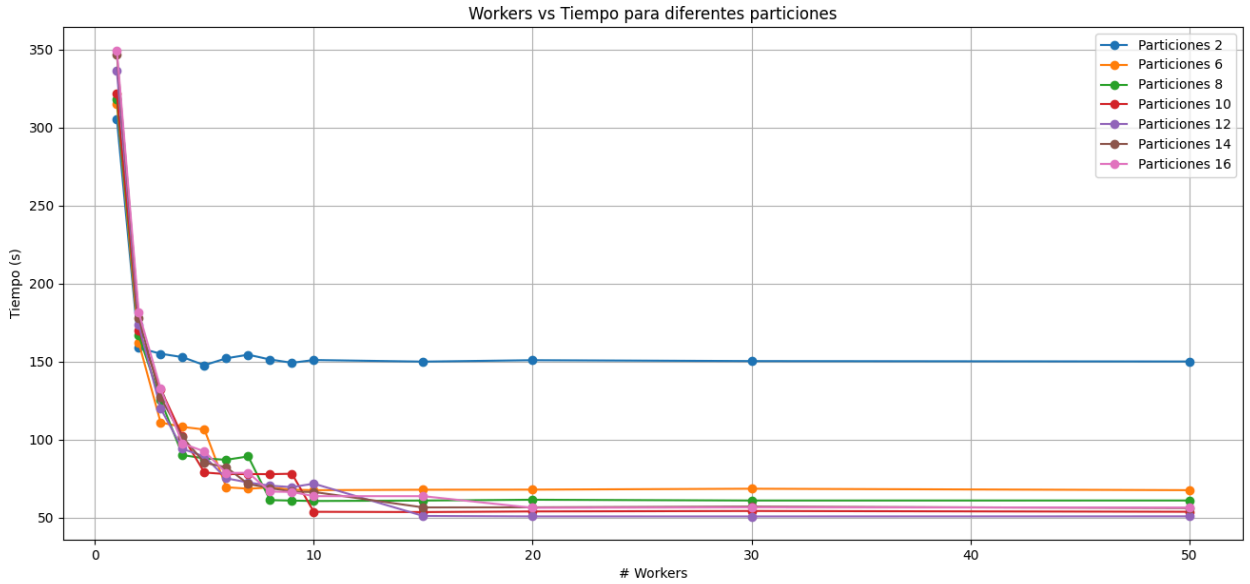


Figura 16. Comparación de número de particiones para 4 workers. Dataset 10k.

Esta gráfica tiene unos resultados muy distintos a los del experimento de un millón. El óptimo sigue siendo el caso de que el número de workers y particiones coincidan, pero a partir de ahí el comportamiento cambia considerablemente. Una explicación que se puede dar es el hecho de que al crear tantas particiones con un dataset tan pequeño el hecho de tener que repartir los datos aumente mucho el tiempo teniendo en cuenta que al tener tan pocos datos la ejecución es inmediata. Influenciando así extremadamente al tiempo de ejecución; en el caso de un millón el tiempo de ejecución vs el tiempo de carga de particiones tiene que ser muy grande, por ello en el caso de un millón 8 particiones tiene un rendimiento muy óptimo, pero en este caso el tiempo de carga tiene que ser bastante significativo frente al de ejecución causando que la gráfica se comporte de la forma que se aprecia por encima del 4.

Anexo



Está gráfica busca ilustrar el comportamiento de spark cuando modificas el número de particiones y el número de workers. En el eje Y se puede ver el tiempo que tarda cada combinación. El eje X es el número de workers por ejecución. Las líneas representan distinto número de particiones.

En la gráfica se puede apreciar que si aumentas el número de workers el tiempo de ejecución se ve disminuido de manera significativa en el comienzo de la curva y luego se estabiliza según aumenta el número de workers. Además con pocas particiones, por ejemplo el caso de 2 particiones, la disminución del tiempo es mucho inferior ya que los workers no trabajan efectivamente con tan pocas particiones. Con más particiones se aprecia que la velocidad es mayor, como se ha visto en el ejercicio 2 el ‘sweet spot’ es cuando coincide el número de workers con el número de hilos que tiene disponible spark.

Un resultado muy significativo es que cuando se llega a un número alto de workers la mejora es marginal o incluso nula, debido al límite de eficiencia de la paralelización que permite spark con el hardware disponible.

Podemos concluir que con menos de 10 workers, la diferencia entre configuraciones con diferentes números de particiones es más notoria. A partir de ese punto, casi todas convergen en tiempos similares, lo que sugiere que un mayor número de particiones ayuda a mejorar la eficiencia cuando hay suficientes workers para aprovecharlas. También, parece que un número de particiones mayor a 8 permite un mejor aprovechamiento de los recursos, ya que la ejecución es más rápida y estable a medida que aumentan los workers.