

L U M B R E



Spark: RDDs y DataFrames

Grado: 6

Materia: Big Data y Procesamiento Distribuido · Ingeniería de Datos

11/10/2026

Antes de empezar

Spark heredó el paralelismo de MapReduce pero lo hizo en memoria y perezoso: no calcula hasta que necesitas un resultado, y planifica el camino. Sus RDDs dan control fino; sus DataFrames, velocidad optimizada.

QUÉ APRENDERÁS

- Transformar datos distribuidos con Spark aprovechando la ejecución perezosa, distinguiendo el nivel bajo de RDD de la abstracción optimizada de DataFrame.

Spark: RDDs y DataFrames

No se calcula hasta que se pide

La ejecución perezosa es el corazón de Spark: al encadenar transformaciones no se calcula nada, solo se dibuja un plano. Hasta que una acción no pide el resultado, el motor puede optimizar el todo antes de tocar los datos.

VIDEO

Spark: RDD frente a DataFrame

Vídeo en español que compara las dos abstracciones y cuándo conviene cada una.



Escanea para verlo en Lumbre

<https://www.youtube.com/watch?v=dFMazU09PQQ>

Resilient Distributed Dataset

Un RDD es una colección de partículas repartida entre nodos, tolerante a fallos por linaje. Se transforma con map, filter y reduce, y si un nodo cae, Spark reconstruye la partición perdida repitiendo los pasos del linaje.

Pereza y planificación

Las transformaciones son perezosas: construyen un grafo de instrucciones. Las acciones, como `count` o `collect`, disparan el cálculo. Esa separación deja al planificador fusionar y reordenar etapas.

Del RDD al DataFrame

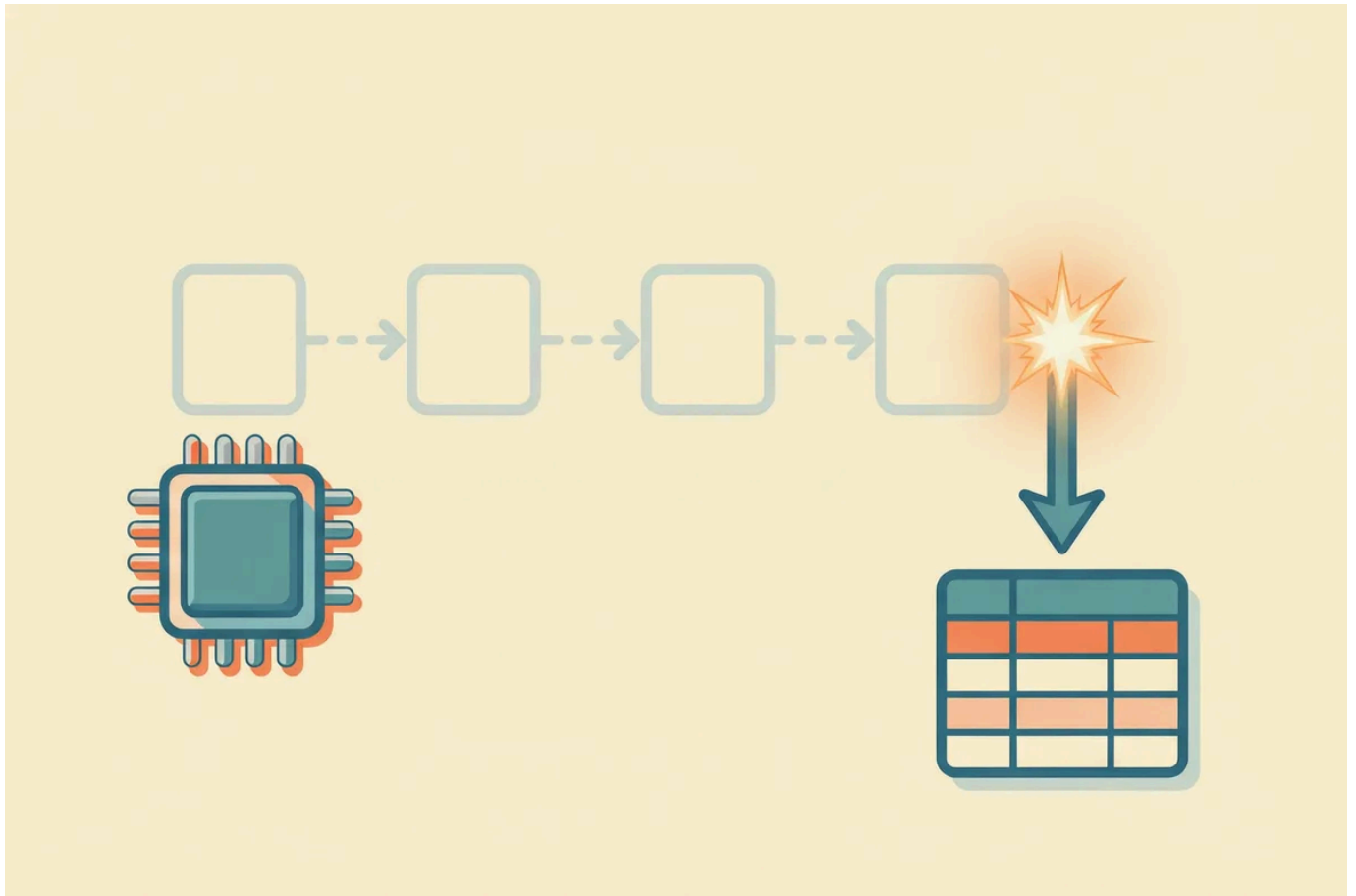
El RDD obliga a pensar en partículas y escribir la lógica de datos a mano. `DataFrame` añade esquema y una capa que optimiza la consulta, como un motor SQL, quitando al programador el trabajo pesado.

Catalyst

- Un optimizador de consultas reescribe el plan lógico antes de ejecutarlo.
- Se aprovecha el esquema para podar columnas y empujar filtros al origen.
- Programar en `DataFrame` suele ser más rápido que en RDD puro por esa optimización.

Persistir en memoria

Cuando un resultado intermedio se reutiliza, conviene guardarlo en memoria o disco con `persist`. Es la herida que curaba Spark frente a MapReduce: evita recalcular y reescribir en cada etapa.



Nada se calcula hasta que una acción lo pide; entonces el plano optimizado aterriza sobre la memoria.

Shuffle aún pesa

Aunque sea rápido, el shuffle sigue siendo caro en Spark. Particionar bien y evitar reagrupaciones innecesarias marca la diferencia entre un job ágil y uno que se arrastra por mover datos entre nodos.

Cuándo cuál

DataFrame para casi todo: datos tabulares, optimización y claridad. RDD cuando hace falta control fino sobre partículas o estructuras no tabulares que el optimizador no entendería.

Errores frecuentes

- Encadenar acciones por error y provocar varios cálculos completos del mismo plano.
- Programar en RDD todo lo que es tabular y perder la optimización de Catalyst.
- Reparticionar sin necesidad y pagar shuffles que el job no pedía.

Ejemplo resuelto: acelerar un job que repite el mismo volumen

Un entrenamiento lee cada vuelta los mismos cien gigabytes y tarda una hora por iteración. La corrección:

1. Medir primero dónde se va el tiempo, mirando el plan y las etapas con shuffle.
2. Persistir en memoria el conjunto base que todas las iteraciones reutilizan.
3. Preferir DataFrame para que el optimizador pode columnas y empuje filtros al origen.
4. Reparticionar solo por la clave con la que se agrega, evitando reagrupaciones extra.
5. Comprobar con el historial de etapas que la segunda iteración ya no relea el disco.

¿Para qué sirve en la realidad?

Todo trabajo iterativo, del machine learning a los gráficos de recomendación, relea el mismo dato vuelta y vuelta. Entender la pereza y la persistencia decide si el experimento tarda minutos u horas.

1. Encadenas varias transformaciones en un DataFrame y no ves que se calcule nada hasta llamar a una acción. El comportamiento se debe a que Spark es:

- ☐ estricto, calcula en cada paso
- ☐ perezoso, difiere el cálculo hasta la acción
- ☐ single-threaded
- ☐ puramente relacional

2. Un DataFrame suele rendir mejor que un RDD equivalente porque Spark puede optimizar la consulta.

- ☐ Verdadero
- ☐ Falso

ACTIVIDAD INTERACTIVA DISPONIBLE EN LUMBRE

3. Spark



4. Ordena el ciclo de vida de un job Spark perezoso:

Elementos:

- encadenar transformaciones sin calcular
- construir el grafo o plano de líneas
- reescribir y podar el plan
- disparar el cálculo con una acción

Transformar	. . .
Planificar	. . .
Optimizar	. . .
Acción	. . .

Guía de RDDs de Spark <https://spark.apache.org/docs/latest/rdd-programming-guide.html>
Programación con RDDs, transformaciones perezosas y linaje.

5. Une cada pieza de Spark con su papel.

5.	colección distribuida con control fino	-----
5.	abstracción con esquema que el optimizador acelera	-----
5.	diferir el cálculo hasta que una acción lo pide	-----
5.	guardar un intermedio para no recalcular	-----

Opciones: RDD · DataFrame · pereza · persistir

No se calcula hasta que se pide

Spark

RDD

control fino

linaje reconstruye

optimización a mano

DataFrame

esquema visible

Catalyst poda y empuja

Pereza

planos sin calcular

la acción dispara

Memoria

persist evita recalcular

shuffle sigue caro

El mapa mental interactivo es interactivo en la versión online.



Guía de DataFrames de Spark <https://spark.apache.org/docs/latest/sql-programming-guide.html> Documentación oficial del API de DataFrame y del optimizador Catalyst.

6. Tu job iterativo reprocesa cada vuelta el mismo gran volumen. ¿Dónde aplicarías persist y por qué no basta con confiar en la pereza? Razona por reuso y memoria frente a recalcular.

Referencia curricular

Grado:	6
Materia:	Big Data y Procesamiento Distribuido · Ingeniería de Datos

Versión interactiva completa (con actividades, videos y autoevaluación):



<https://univ.lumbre.lat/object/cdia-bigd-002-spark-rdds-y-dataframes>

Material educativo generado desde Lumbre · lumbre.lat