Aclaraciones sobre caché en nodos Spark y asignación dinámica

En la práctica, el comportamiento de Spark en cuanto a **creación y destrucción de nodos (executors)** depende en gran medida de cómo esté configurado el **cluster manager** (YARN, Kubernetes, Standalone, Mesos, etc.) y de la **duración de la aplicación Spark** (el *Spark Application*). A grandes rasgos:

1. **Mientras la aplicación Spark está "viva"** (es decir, mientras no se cierra la sesión Spark o el programa que inició el *SparkContext*), los **executors** continúan existiendo (salvo que se use *dynamic allocation* y se reduzca el número de nodos o se incremente según necesidad).
2. **Al terminar la aplicación Spark** (cuando el *SparkContext* se cierra o el cuaderno/driver finaliza), se liberan los recursos en el cluster manager.

Las **variables broadcast** y los **cachés/persistencias** viven mientras la aplicación Spark siga activa.

1. Variables *Broadcast*

- Cuando creas una variable *broadcast*, Spark **distribuye** (broadcast) el contenido a todos los *executors*.
- Cada *executor* la **almacena en caché** localmente para que las tareas (tasks) puedan acceder a ella sin necesidad de volver a traerla desde el *driver* o desde el almacenamiento externo.
- **Si la aplicación sigue corriendo**, y las tareas posteriores necesitan esa variable, se sigue usando la misma copia en los *executors*.
- **Si se destruyen los *executors*** (por ejemplo, la aplicación finaliza o Spark hace *decommissioning* de nodos con *dynamic allocation*), se pierde esa información. Si se crean nuevos *executors* más adelante (porque Spark necesita más recursos), Spark volverá a hacer el broadcast de esa variable a los nuevos nodos.

El *broadcast* está pensado para **compartir datos "invariables"** a lo largo de las *stages* de una **misma aplicación Spark**. No persiste fuera de esa aplicación.

2. Persist o Cache

- Cuando realizas un df.cache(), df.persist(), o cuando cacheas un RDD, Spark intentará **mantener esos datos en memoria** (o memoria + disco, según el nivel de persistencia) **dentro de los ejecutors**.
- Esa caché es **válida** mientras el *SparkContext* siga activo y **mientras esos ejecutors no se liberen**.

- Si en un punto posterior (dentro de la misma aplicación) se vuelve a necesitar ese *DataFrame* o RDD, Spark lo recupera desde la caché en lugar de recalcularlo desde la fuente original.
- **Si finaliza la aplicación** o si Spark decide liberar ciertos ejecutors por inactividad (con *dynamic allocation*), se pierden los datos en caché en esos nodos.
- En caso de que vuelvas a solicitar un cálculo sobre ese *DataFrame* o RDD y no existan los datos en memoria (por haberse liberado los nodos), Spark recalculará o volverá a leer los datos desde el origen.

persist() y **cache**() **mejoran el rendimiento** de operaciones **dentro de la misma ejecución** (aplicación). No garantizan persistencia más allá de la vida de la aplicación o de la presencia de los *executors*.

¿Se mantienen los nodos "levantados" o se recrean?

- **En un cluster "fijo"** (por ejemplo, modo *Standalone* o un cluster YARN con un número fijo de contenedores asignados), los *executors* permanecen hasta que acaba la aplicación. Durante ese tiempo, tus datos *cacheados* y *broadcasts* están disponibles.
- **En un cluster con *dynamic allocation***, Spark puede **aumentar o reducir** el número de *executors* según la carga. Si algunos *executors* se liberan, se pierde la información cacheada en ellos.
- **Si terminas la aplicación** (o cierras el cuaderno Spark) y luego lanzas otra, se **crean de nuevo** los *executors* y se **retransmiten** (broadcast) las variables y datos necesarios. No hay persistencia automática entre aplicaciones diferentes a menos que tú mismo la gestiones (por ejemplo, guardando en HDFS, S3 o un data lake).

Uso típico de Broadcast y Cache/Persist

- **Broadcast**: Para tablas pequeñas o datos de referencia que se usan de forma repetida en *joins* o transformaciones. Evita que cada *task* tenga que leer esos datos desde la fuente externa.
- **Cache/Persist**: Para reutilizar resultados intermedios de cómputo intensivo o lecturas costosas. Esto **acelera** acciones posteriores que requieren esos mismos datos.

Las instrucciones **(broadcast, persist, cache) no tienen efecto más allá de la vida de la aplicación Spark o de la disponibilidad de los ejecutors**. Son optimizaciones para **reutilizar datos dentro del mismo contexto de ejecución** y

evitar recálculos o lecturas reiteradas desde fuentes externas mientras la aplicación siga corriendo.