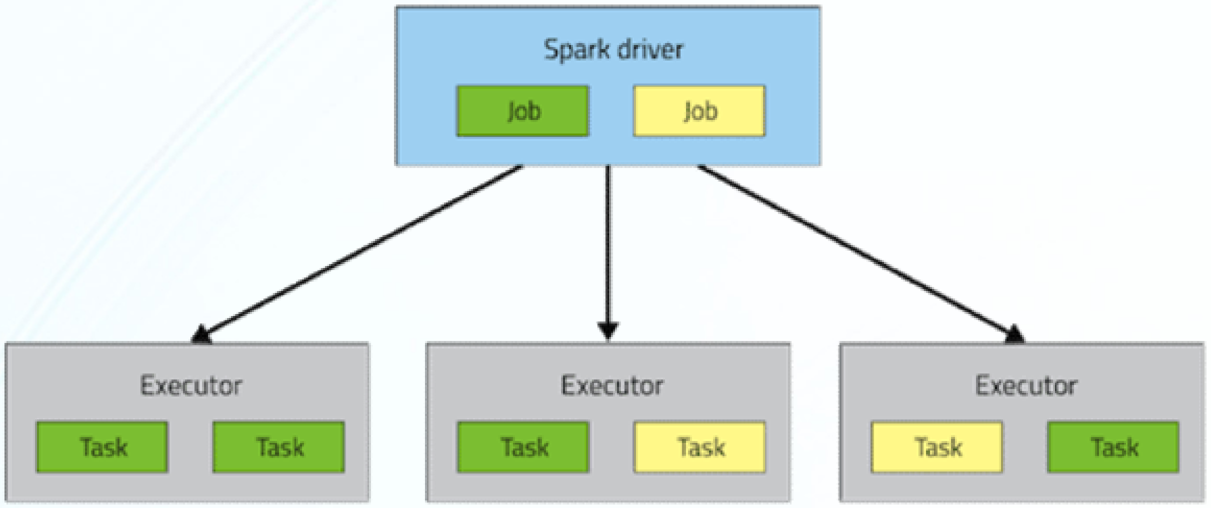
Spark Performance Tuning

Spark Performance Tuning

How Spark executes your program?

Spark execution process

- A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.



Spark execution process

- The driver is the process that is in charge of the high-level control flow of work that needs to be done.

- The executor processes are responsible for :

  - Executing this work, in the form of tasks.
  - Storing any data that the user chooses to cache.

- A single executor has a number of slots for running tasks, and will run many tasks concurrently throughout its lifetime.

- Deploying these processes on the cluster is up to the cluster manager in use, but the driver and executor themselves exist in every Spark application.

Spark execution plan

- At the top of the execution hierarchy are jobs.
- Invoking an action inside a Spark application triggers the launch of a Spark job to fulfill it.
- To decide what this job looks like, Spark examines the graph of RDDs on which that action depends and formulates an ***execution plan***.
- This plan starts with the farthest-back RDDs—that is, those that depend on no other RDDs or reference already-cached data–and culminates in the final RDD required to produce the action’s results.

Spark stages

- The execution plan consists of assembling the job’s transformations into ***stages***.
- A stage corresponds to a collection of tasks that all execute the same code, each on a different subset of the data.
- Each stage contains a sequence of transformations that can be completed without shuffling the full data.

Understanding shuffling

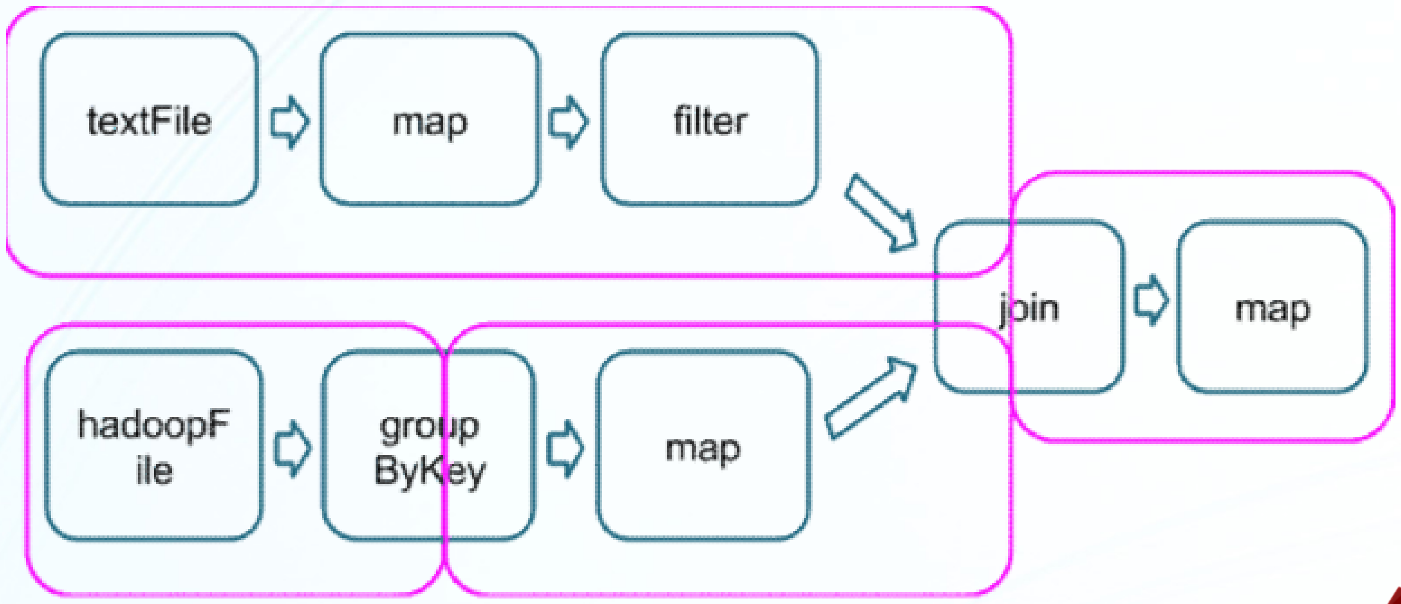
- The execution plan consists of assembling the job’s transformations into ***stages***.
- A stage corresponds to a collection of tasks that all execute the same code, each on a different subset of the data.
- Each stage contains a sequence of transformations that can be completed without shuffling the full data.

When do shuffles happen ?

- In the case of *wide* transformations, the data required to compute the records in a single partition may reside in many partitions of the parent RDD.
- All of the tuples with the same key must end up in the same partition, processed by the same task.
- To satisfy these operations, Spark must execute a shuffle, which transfers data around the cluster and results in a new stage with a new set of partitions.

When do shuffles happen ?

- Shown below is a transformation graph including a join transformation with multiple dependencies.
    - The pink boxes show the resulting stage graph used to execute it.



Stages & Shuffling

- At each stage boundary, data is written to disk by tasks in the *parent* stages and then fetched over the network by tasks in the *child* stage.

- Because they incur heavy disk and network I/O, stage boundaries can be expensive and should be avoided when possible.

- The number of data partitions in the parent stage may be different than the number of partitions in the child stage. Transformations that may trigger a stage boundary typically accept a *numPartitions* argument that determines how many partitions to split the data into in the child stage.

- Tuning the number of partitions at stage boundaries can often make or break an application’s performance.

Picking the Right Operators

- Avoid ***groupByKey*** when performing an associative reductive operation
  - For example, **`rdd.groupByKey().mapValues(_.sum)`** will produce the same results as **`rdd.reduceByKey(_ + _)`**.
  - However, the former will transfer the entire dataset across the network, while the latter will compute local sums for each key in each partition and combine those local sums into larger sums after shuffling.

Picking the Right Operators

- Avoid reduceByKey, when the input and output value types are different.
  - Consider the following two snippets to find all the unique strings corresponding to each key:

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2))
   .reduceByKey(_ ++ _)

val z = new collection.mutable.Set[String]()
rdd.aggregateByKey(z)( (s, v) => s += v, (s1, s2) => s1 ++= s2)
```

  - It's better to use ***aggregateByKey***, which performs the map-side aggregation more efficiently

Picking the Right Operators

- Avoid the **`flatMap-join-groupBy`** pattern
  - When two datasets are already grouped by key and you want to join them and keep them grouped, you can just use cogroup.
  - That avoids all the overhead associated with unpacking and repacking the groups.

When more shuffles are better ?

- There is an occasional exception to the rule of minimizing the number of shuffles.
- An extra shuffle can be advantageous to performance when it increases parallelism.
  - For example, if your data arrives in a few large unsplittable files, the partitioning dictated by the InputFormat might place large numbers of records in each partition, while not generating enough partitions to take advantage of all the available cores.
  - In this case, invoking repartition with a high number of partitions (which will trigger a shuffle) after loading the data will allow the operations that come after it to leverage more of the cluster’s CPU.

THANK YOU