Why

Spark applications have jobs

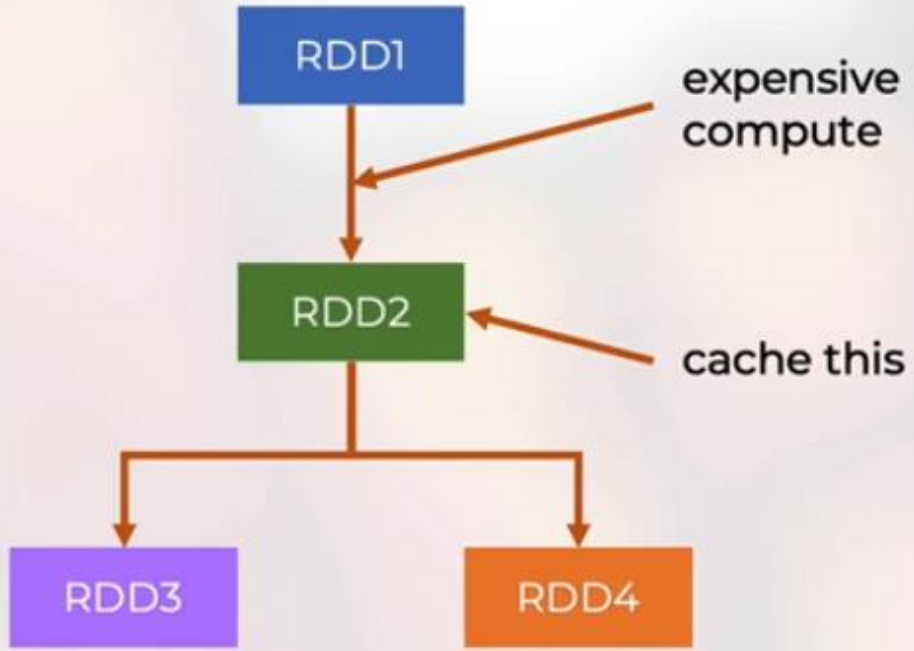
- each action triggers a job
- a job executes the query plan
- all dependencies in the query plan are evaluated

Save expensive computations

- the same RDD can be evaluated multiple times
- if RDD is expensive – job is expensive multiple times

Cache expensive RDDs

- the RDD lineage is kept
- the RDD data is kept in memory
- subsequent dependencies will fetch the cached data



Caching Mechanics

Can cache into

- memory in heap
- memory off heap (with Tungsten)
- disk
- memory + disk

Caching is done by executors on worker nodes

Beware of JVM limits

- min JVM memory 4-8GB
- max JVM memory 40GB
- the more JVM memory, the more time needed for GC
- large JVM heap may lead to decrease in perf

```scala
// simulate an "expensive" operation
val orderedFlightsDF = flightsDF.orderBy("dist")

// scenario: use this DF multiple times

orderedFlightsDF.persist(
  // no argument = MEMORY_AND_DISK
  // StorageLevel.MEMORY_ONLY // cache the DF in memory EXACTLY - CPU efficient, memory expensive
  // StorageLevel.DISK_ONLY // cache the DF to DISK - CPU efficient and mem efficient, but slower
  // StorageLevel.MEMORY_AND_DISK // cache this DF to both the heap AND the disk - first caches to memory, but if the DF is EV

  /* modifiers: */
  // StorageLevel.MEMORY_ONLY_SER // memory only, serialized - more CPU intensive, memory saving - more impactful for RDDs
  // StorageLevel.MEMORY_ONLY_2 // memory only, replicated twice - for resiliency, 2x memory usage
  // StorageLevel.MEMORY_ONLY_SER_2 // memory only, serialized, replicated 2x

  /* off-heap */
  StorageLevel.OFF_HEAP // cache outside the JVM, still stored on the machine RAM, needs to be configured, CPU efficient and m
)

orderedFlightsDF.count()
orderedFlightsDF.count()

/*
  Without cache: sorted count ~0.1s
```

Caching

Run: Caching

```
20/05/19 12:43:56 INFO BlockManagerInfo: Added broadcast_8_piece0 in memory on ciocrlanul-pro:57204 (size: 14.1 KiB, free: 4.1 G
20/05/19 12:43:56 INFO SparkContext: Created broadcast 8 from broadcast at DAGScheduler.scala:1206
20/05/19 12:43:56 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 3 (MapPartitionsRDD[19] at count at Caching.sca
20/05/19 12:43:56 INFO TaskSchedulerImpl: Adding task set 3.0 with 1 tasks
20/05/19 12:43:56 INFO TaskSetManager: Starting task 0.0 in stage 3.0 (TID 3, ciocrlanul-pro, executor driver, partition 0, PROC
20/05/19 12:43:56 INFO Executor: Running task 0.0 in stage 3.0 (TID 3)
20/05/19 12:43:56 INFO BlockManager: Found block rdd_11_0 locally
20/05/19 12:43:56 INFO Executor: Finished task 0.0 in stage 3.0 (TID 3). 2575 bytes result sent to driver
20/05/19 12:43:56 INFO TaskSetManager: Finished task 0.0 in stage 3.0 (TID 3) in 22 ms on ciocrlanul-pro (executor driver) (1/1)
20/05/19 12:43:56 INFO TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have all completed, from pool
20/05/19 12:43:56 INFO DAGScheduler: ResultStage 3 (count at Caching.scala:42) finished in 0.035 s
20/05/19 12:43:56 INFO DAGScheduler: Job 3 is finished. Cancelling potential speculative or zombie tasks for this job
20/05/19 12:43:56 INFO TaskSchedulerImpl: Killing all running tasks in stage 3: Stage finished
20/05/19 12:43:56 INFO DAGScheduler: Job 3 finished: count at Caching.scala:42, took 0.041680 s
```

Caching Recap

Memory-only storage

- very CPU efficient
- can increase the risk of memory failures

```
mySuperRDD.cache(StorageLevel.MEMORY_ONLY)
```

Disk storage

- memory efficient
- slow to access

```
mySuperRDD.cache(StorageLevel.DISK_ONLY)
```

Serialization

- more CPU intensize
- 3x – 5x memory saving

```
mySuperRDD.cache(StorageLevel.MEMORY_ONLY_SER)
```

Replication

- 2x memory/disk usage
- fault tolerance

```
mySuperRDD.cache(StorageLevel.MEMORY_ONLY_2)
```

Off-heap

- free executor memory
- needs to be configured

```
mySuperRDD.cache(StorageLevel.OFF_HEAP)
```

```
spark.memory.offHeap.enabled = true
spark.memory.offHeap.size = 10485760
```

Caching Tradeoffs

Raw objects

- consume 3x-5x more memory (either RAM or disk)
- take 20x less time to process in RAM
- take more time to read from disk

Serialized objects

- max memory efficiency
- CPU intensive
- take less time to read from disk

Fault tolerance

- failed nodes will lose cached partitions
- cached partitions will be recomputed by other nodes (unless replicated)

Caching Recommendations

Only cache what's being reused a lot

- don't cache too much or you risk OOMing the executors
- the LRU data will be evicted

If data fits in memory, use MEMORY_ONLY (default)

- most CPU efficient

If data is larger, use MEMORY_ONLY_SER

- more CPU intensive, but still faster than anything else

Use disk caching only for really expensive computations

- simple filters take just as much (or even less) to recompute than reread from disk