Performing Optimizations in Spark

Mohit Batra
Founder, Crystal Talks

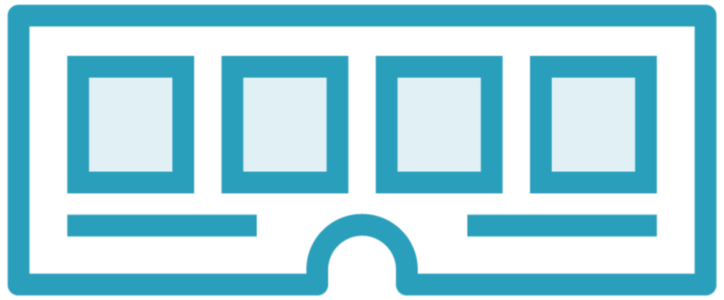
linkedin.com/in/mohitbatra

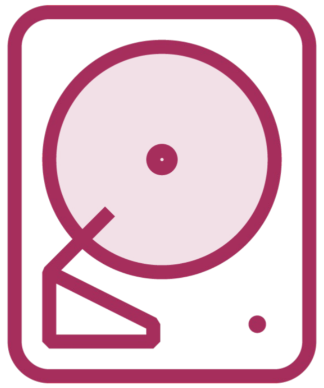
Overview

- Work with Spark partitions
- Change DataFrame partitions
- Memory management
- Persist data
- Spark join strategies and broadcast join
- Optimize join with bucketing
- Dynamic resource allocation
- Resource allocation using fair scheduling

Working with Spark Partitions

Types of Partitioning





In-Memory Partitioning

- Chunks of data read in memory
- All partitions together constitute RDD/DataFrame

Disk Partitioning

- Writing output to disk by physically partitioning data based on columns
- Done using partitionBy method

In-Memory Partition Settings

Settings for Reading Data

Settings for Data Shuffling

1. For Reading Data

spark.default.parallelism = 4 [default = no. of cores]

File 1

128 MB

P1 (32 MB)

P2 (33 MB)

P3 (31 MB)

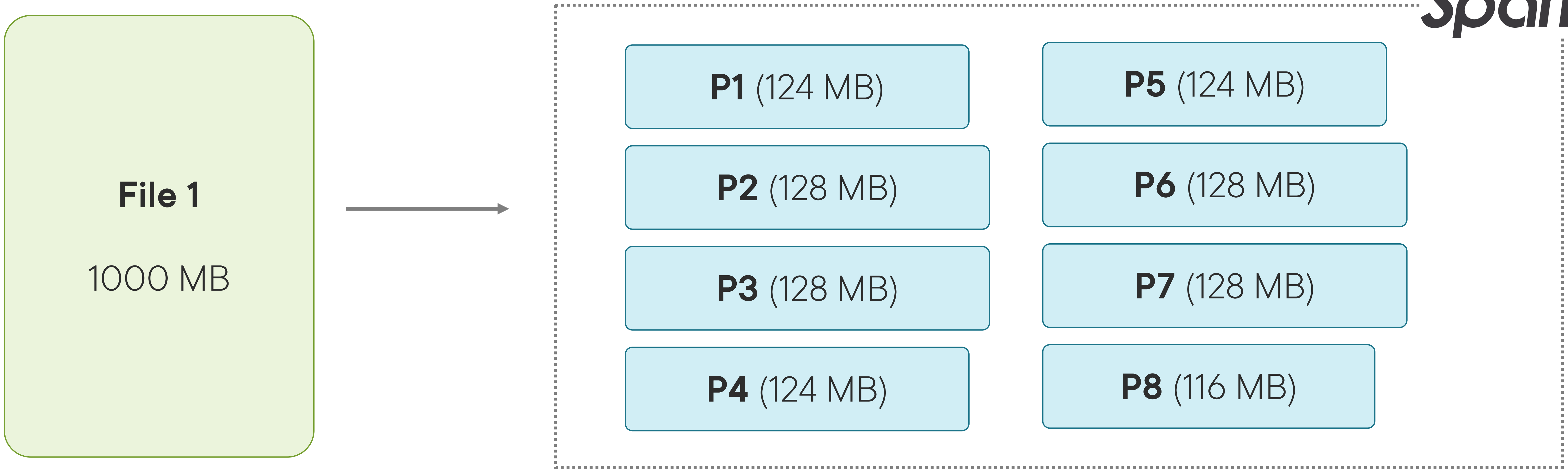
P4 (32 MB)

Partitions may be lesser than default parallelism setting if data size is small

1. For Reading Data

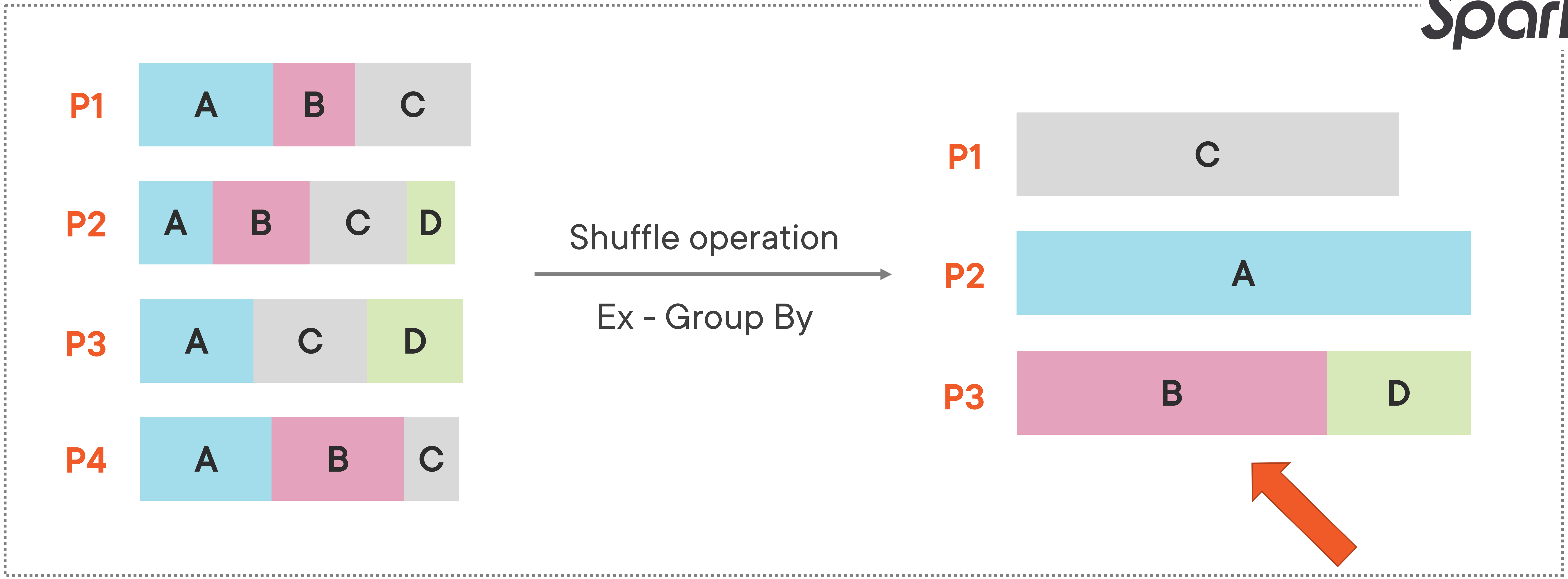
spark.default.parallelism = 4 [default = no. of cores]

spark.sql.files.maxPartitionBytes = 128 MB [default = 128 MB]

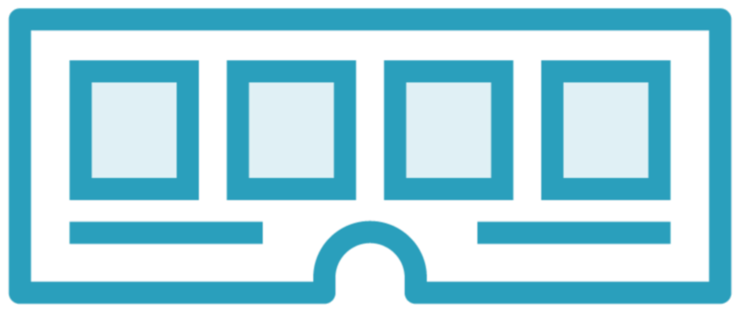


2. For Shuffling Data

spark.sql.shuffle.partitions = 3 [default = 200]



Impact of In-Memory Partitions



Partitions are processed in parallel

Partitions and Cores determine parallelism of a Job

Having very few/big partitions:

- Each task may need to process lot of data
- Cluster resources may be under-utilized

Having lot of/small partitions:

- Too many tasks are created
- Reduces parallelism since tasks go in waiting state

Changing DataFrame Partitions

Methods to Change DataFrame Partitions

Repartition Method

Coalesce Method

Repartition Method

Typically used to increase number of partitions

Wide transformation – performs shuffling

- Avoid using it to decrease partitions

Partitioning Options

- Round Robin – Creates equal sized partitions
- Hash – Co-locates data based on columns
- Range – Sorts & co-locates data based on columns

Use Cases

- Reduce skewness (some partitions have much more data than others)
- Reduce size of partitions (when they are too big)
- Co-locate data based on certain columns

Coalesce Method

Used for decreasing number of partitions of DataFrame

- Cannot increase partitions

Narrow transformation – no shuffling

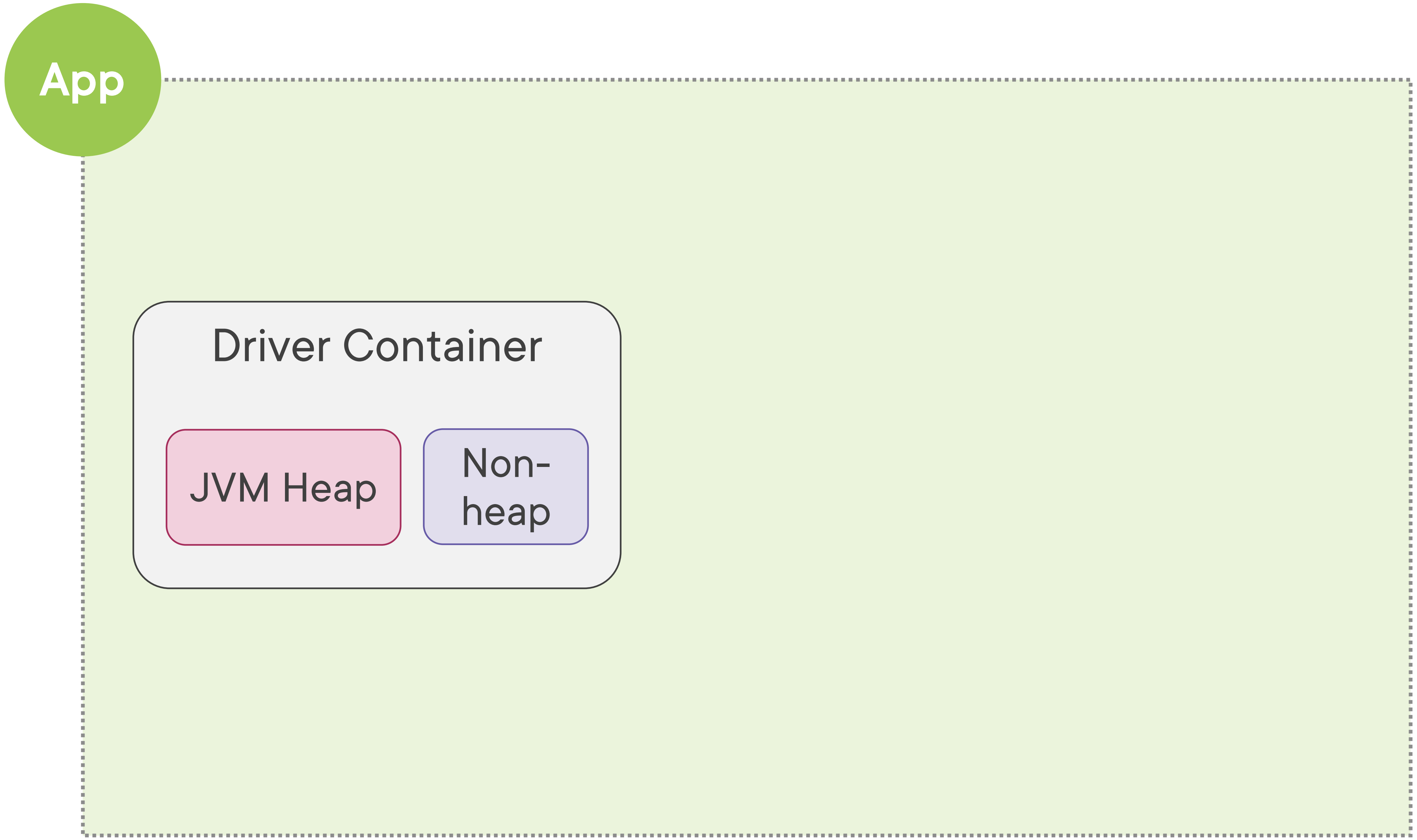
Can result in Out-of-Memory exceptions

- Eg: Existing DF = 100 partitions * 100 MB
- Coalesce(1) → New DF = 10,000 MB

Use Cases

- Partitions are very small
- Output is required in lesser number of files

Memory Management



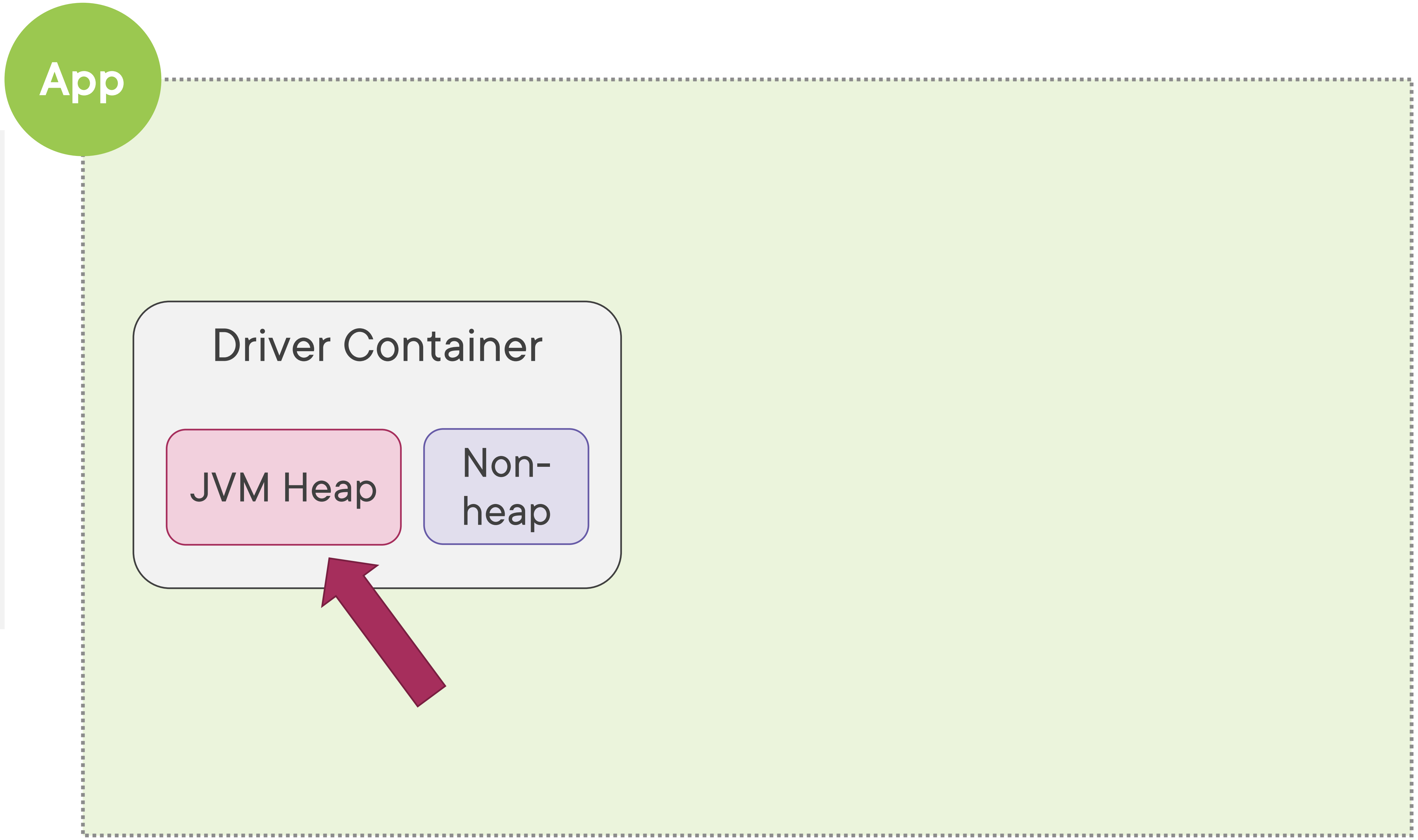
JVM Heap memory is used for Spark activity

Overhead memory (non-heap memory) is used by non-JVM processes like VM overheads, buffers etc.

Driver Container

spark.driver.cores = 2

spark.driver.memory = 8 GB



Driver Container

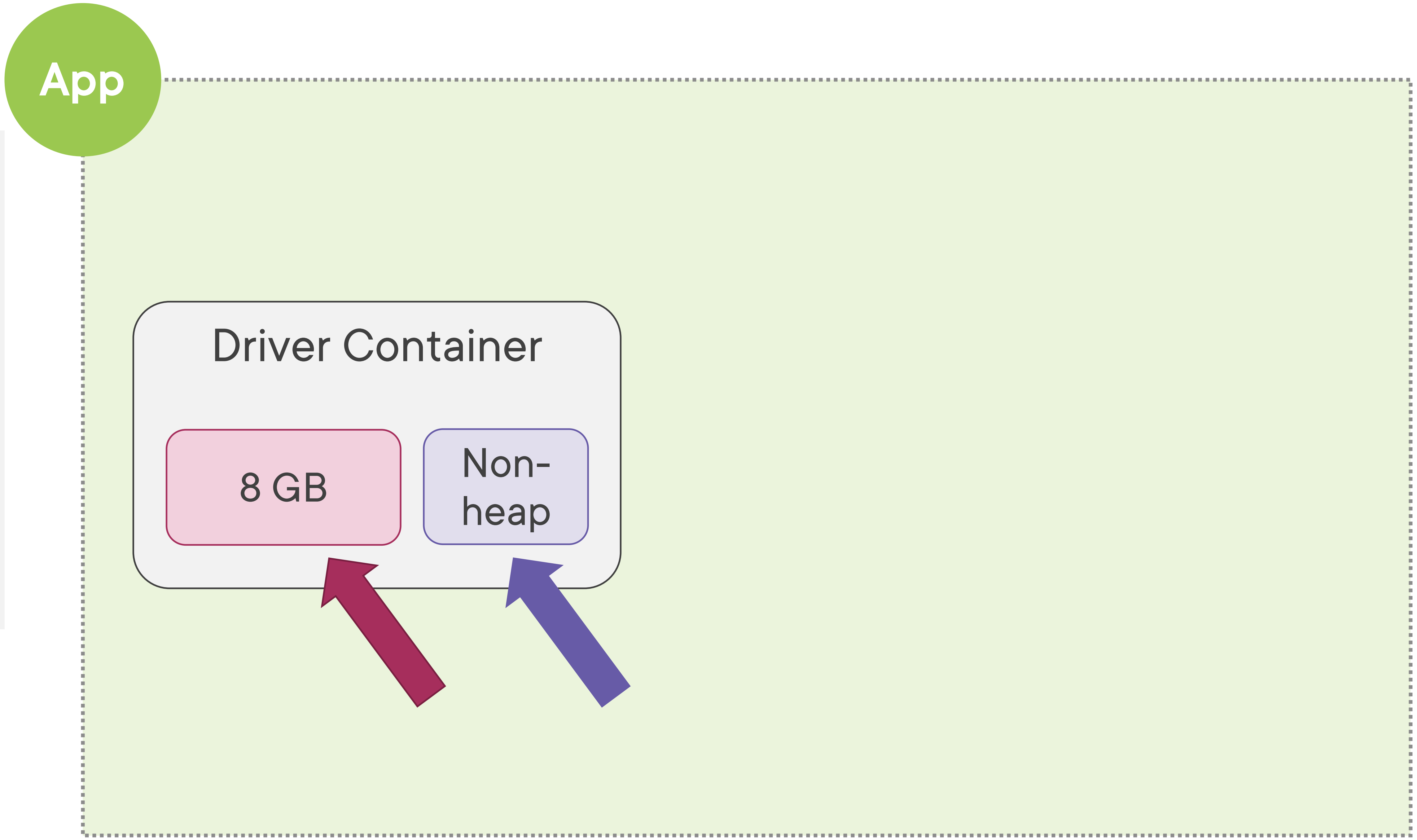
spark.driver.cores = 2

spark.driver.memory = 8 GB

spark.driver.memoryOverhead

= max (10% of memory or 384 MB)

= 800MB



Driver Container

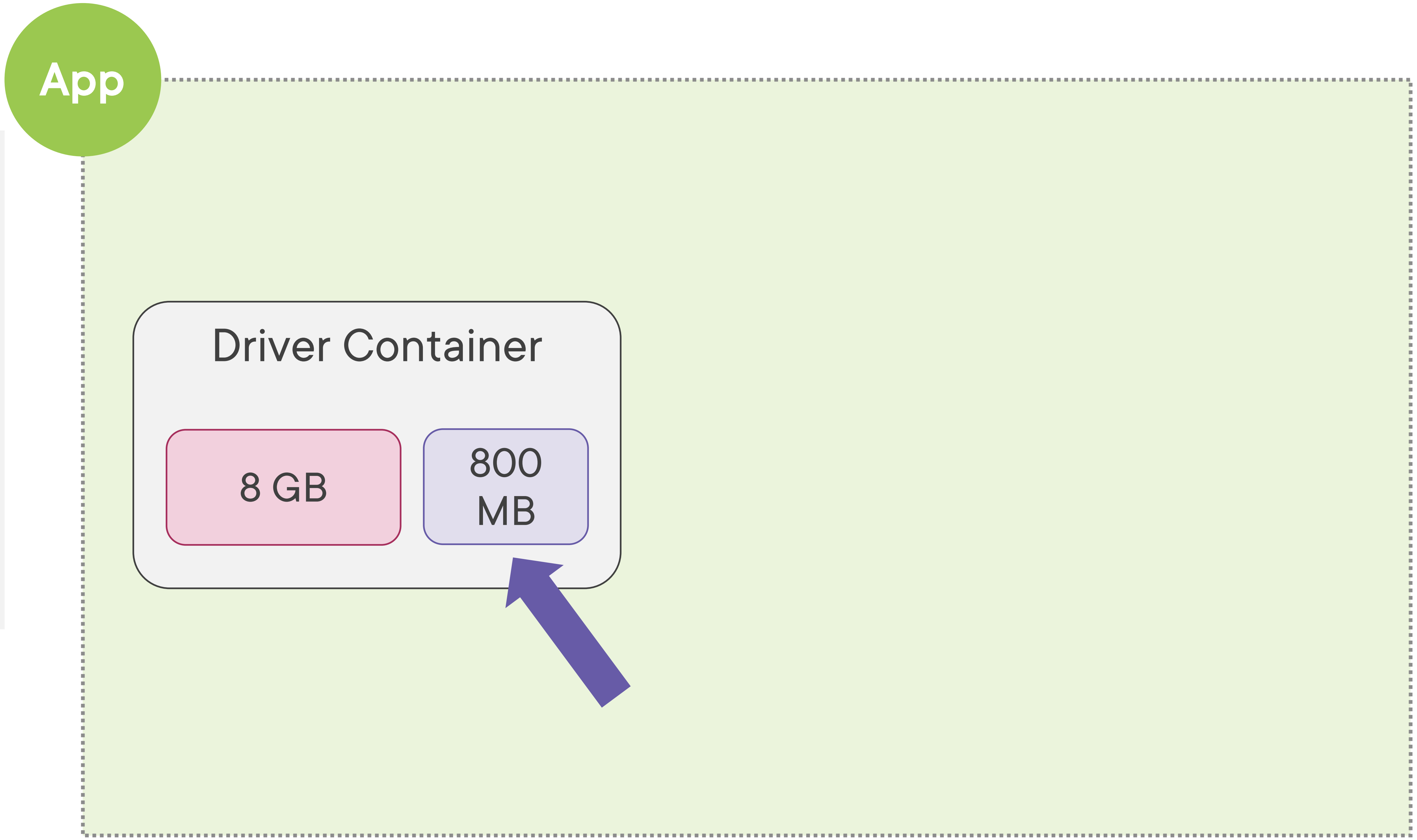
spark.driver.cores = 2

spark.driver.memory = 8 GB

spark.driver.memoryOverhead

= max (10% of memory or 384 MB)

= 800MB



Driver Container

spark.driver.cores = 2

spark.driver.memory = 8 GB

spark.driver.memoryOverhead

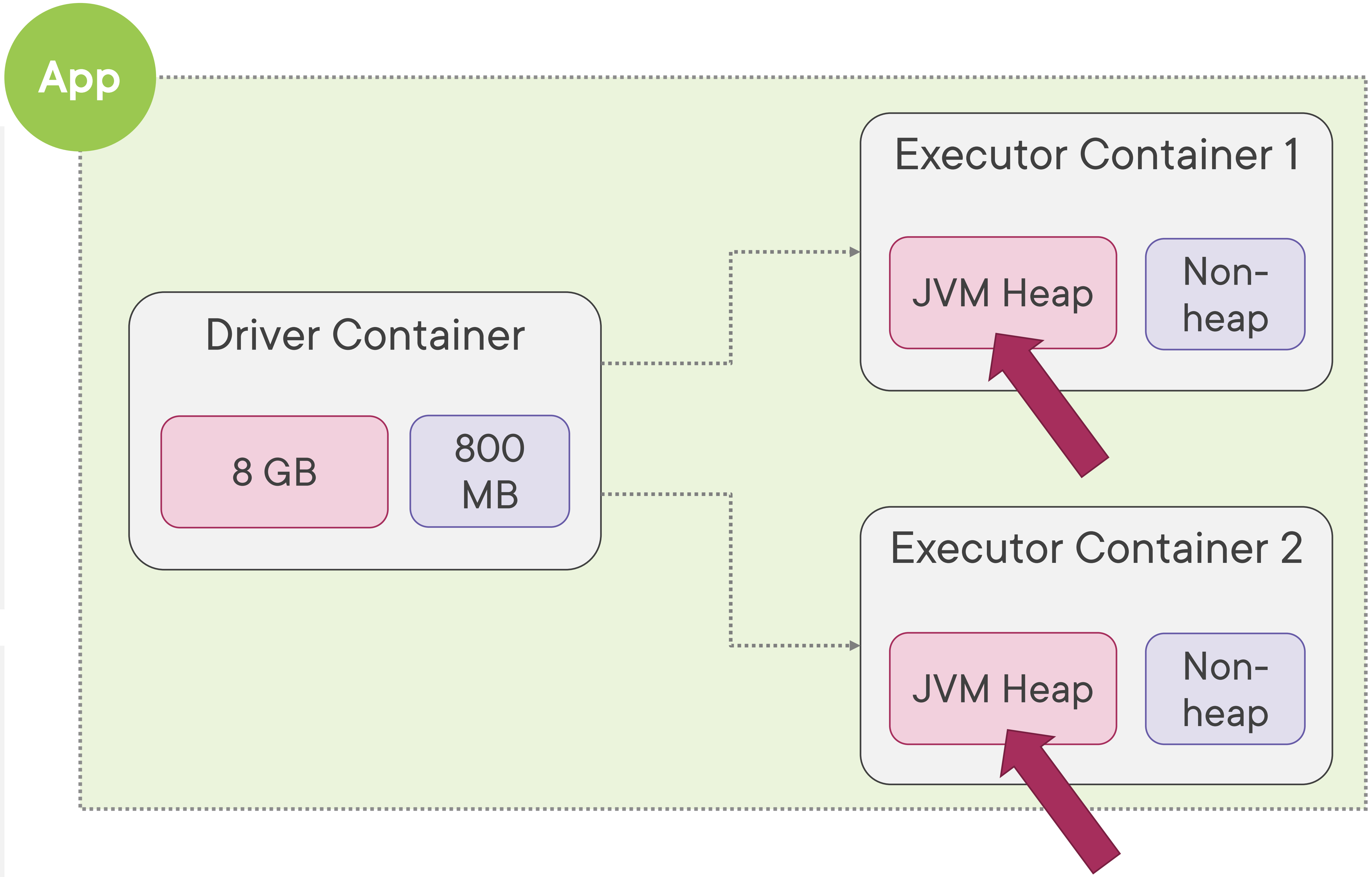
= max (10% of memory or 384 MB)

= 800MB

Executor Container

spark.executor.cores = 4

spark.executor.memory = 14 GB



Driver Container

spark.driver.cores = 2

spark.driver.memory = 8 GB

spark.driver.memoryOverhead

= max (10% of memory or 384 MB)

= 800MB

Executor Container

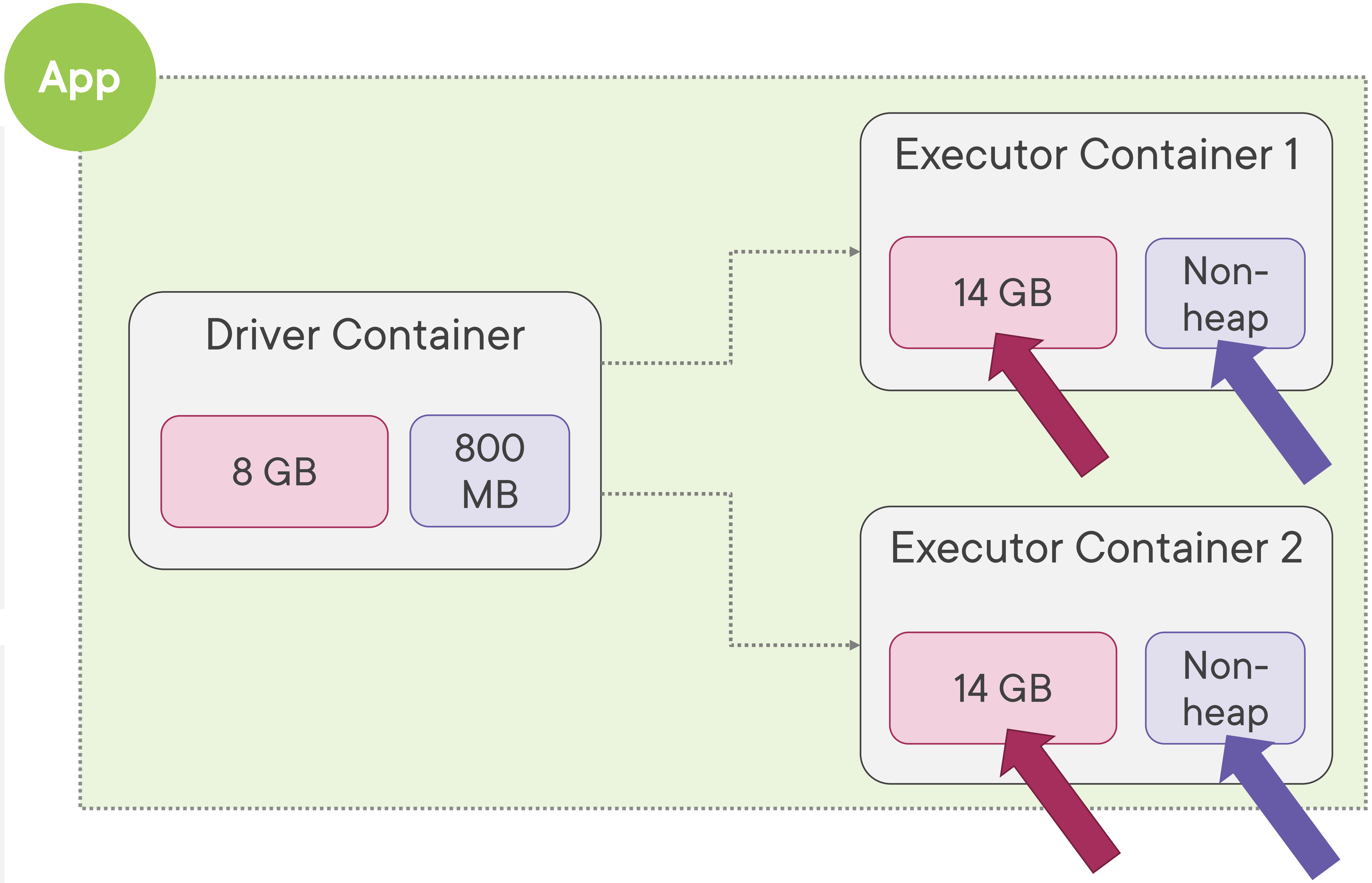
spark.executor.cores = 4

spark.executor.memory = 14 GB

spark.executor.memoryOverhead

= max (10% of memory or 384 MB)

= 1.4 GB



Driver Container

spark.driver.cores = 2

spark.driver.memory = 8 GB

spark.driver.memoryOverhead

= max (10% of memory or 384 MB)

= 800MB

Executor Container

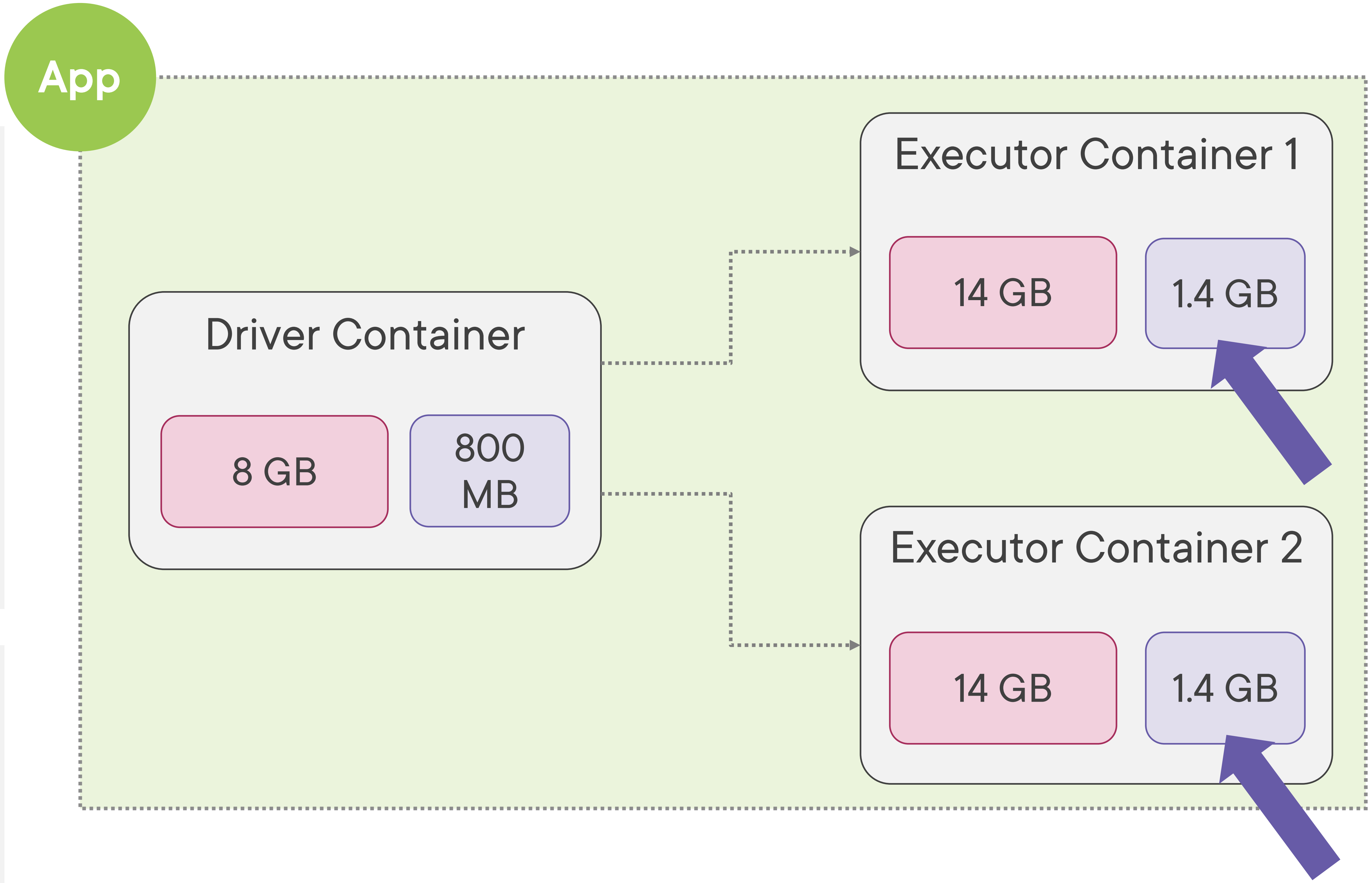
spark.executor.cores = 4

spark.executor.memory = 14 GB

spark.executor.memoryOverhead

= max (10% of memory or 384 MB)

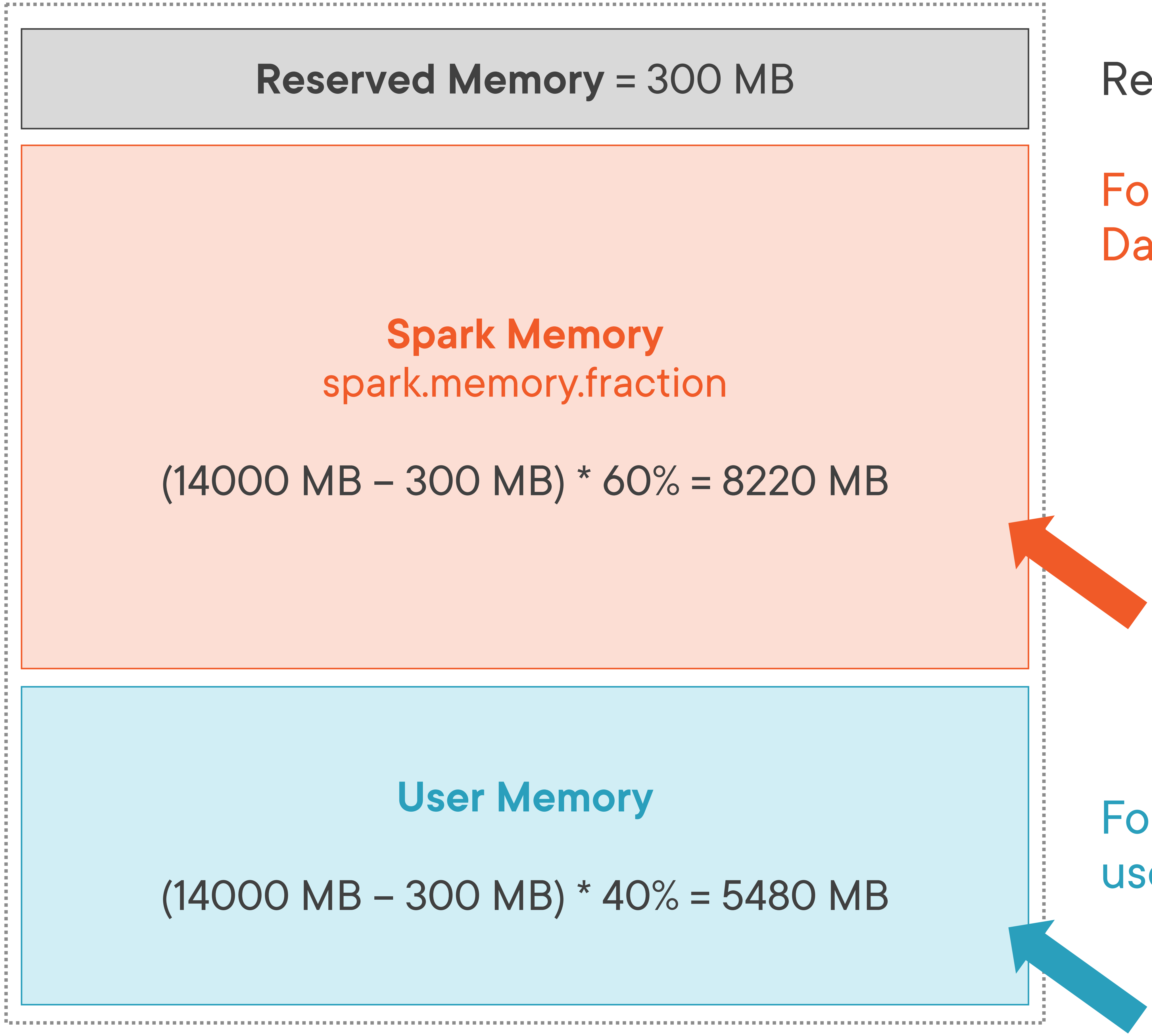
= 1.4 GB



Executor JVM Heap

spark.executor.cores	= 4
spark.executor.memory	= 14 GB

Executor JVM Heap



Reserved by Spark for internal purposes

For task execution, caching, shuffling, DataFrame operations etc.

spark.executor.cores	= 4
spark.executor.memory	= 14 GB
spark.memory.fraction	= 0.6

For storing data structures created by user, metadata, RDD operations, UDFs etc.

Executor JVM Heap

Reserved Memory = 300 MB

Spark Memory = 8220 MB

Execution Memory (50%) = 4110 MB

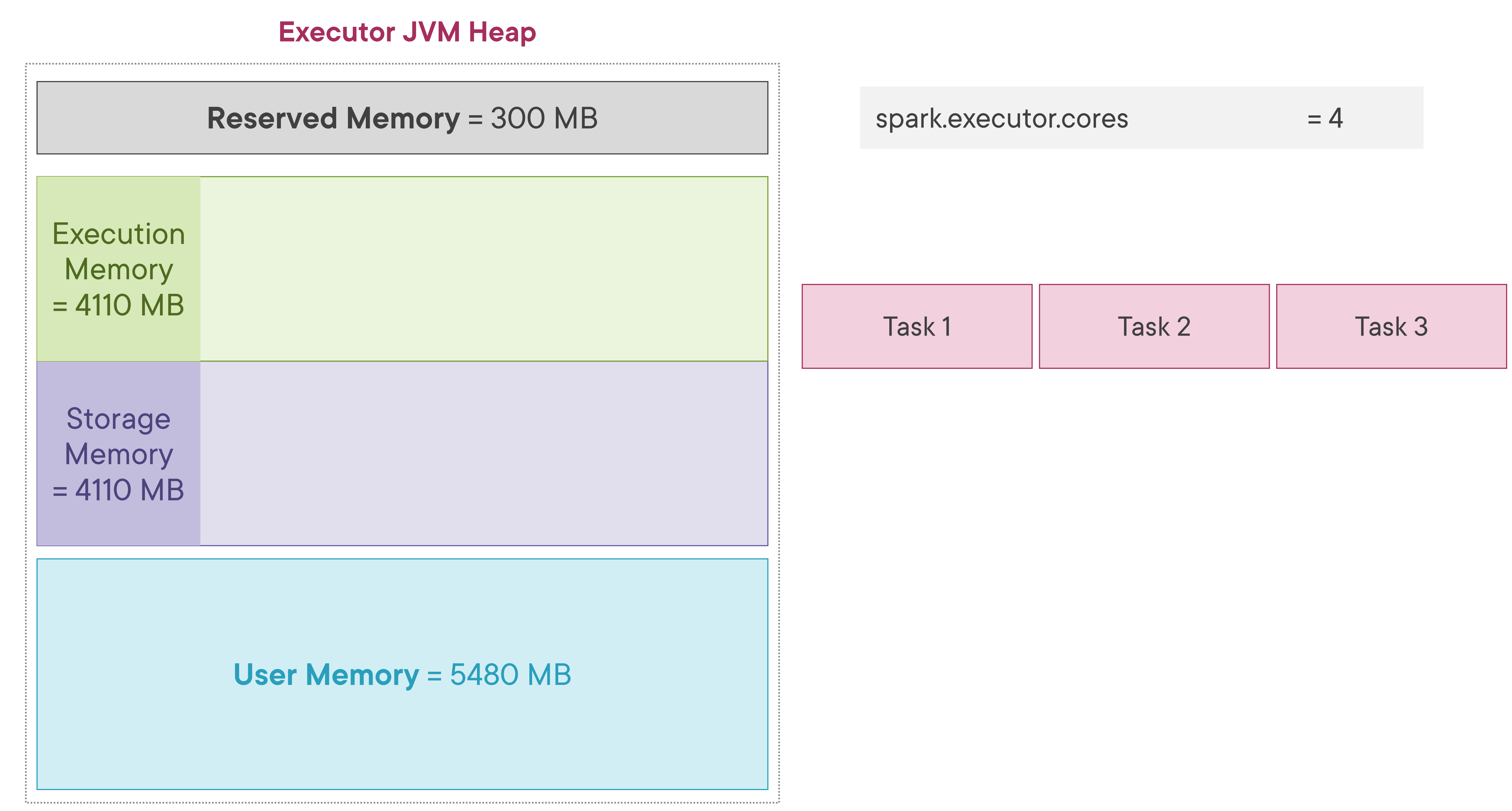
Storage Memory (50%) = 4110 MB

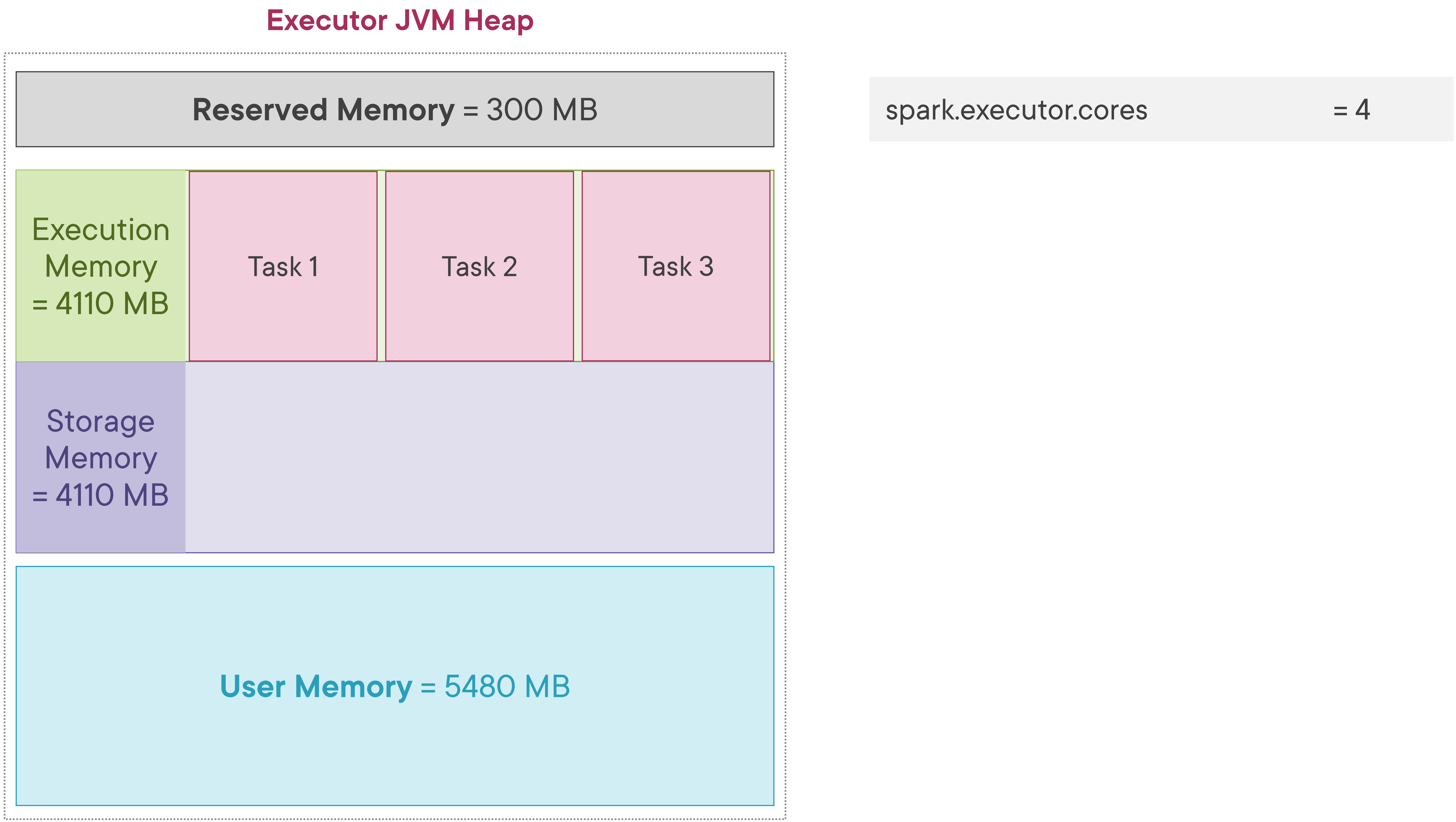
User Memory = 5480 MB

For DataFrame operations, task execution, shuffling etc.

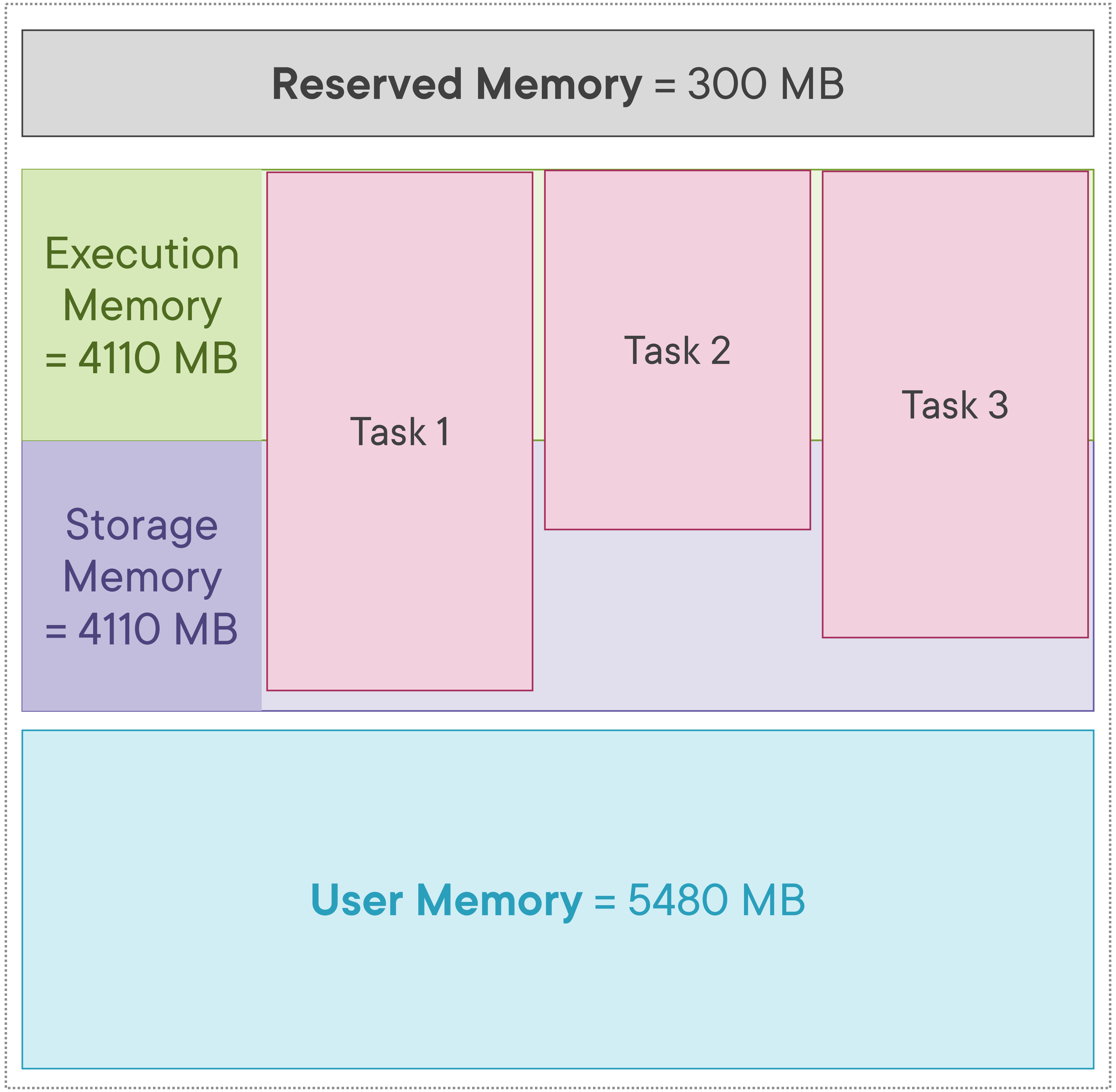
For storing cached data

spark.executor.cores	= 4
spark.executor.memory	= 14 GB
spark.memory.fraction	= 0.6
spark.memory.storageFraction	= 0.5


Executor JVM Heap
Reserved Memory = 300 MB
Execution Memory = 4110 MB
Storage Memory = 4110 MB
User Memory = 5480 MB
spark.executor.cores = 4
Task 1
Task 2
Task 3

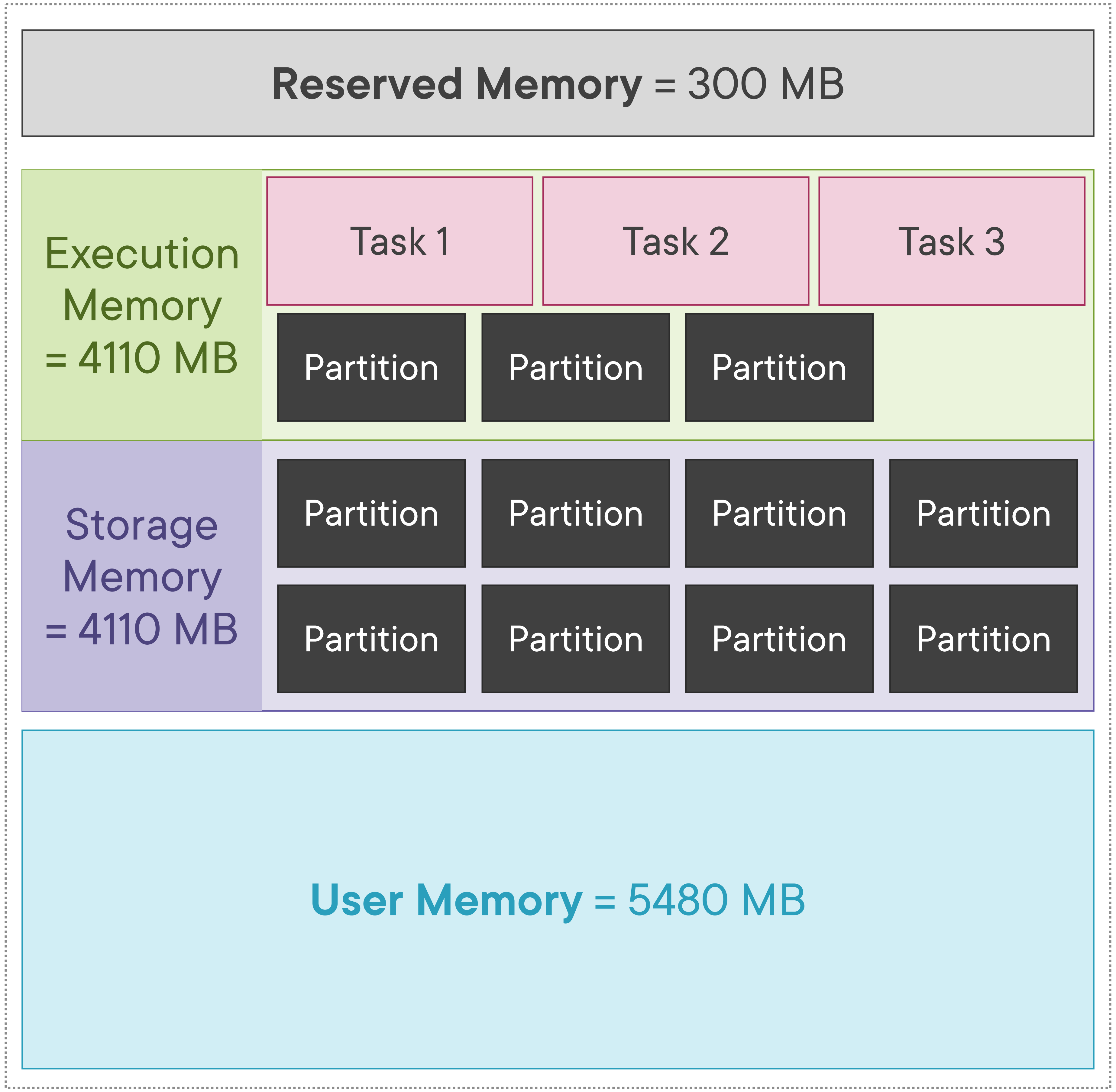

Executor JVM Heap
Reserved Memory = 300 MB
Execution Memory = 4110 MB
Task 1
Task 2
Task 3
Storage Memory = 4110 MB
User Memory = 5480 MB
spark.executor.cores = 4

Executor JVM Heap

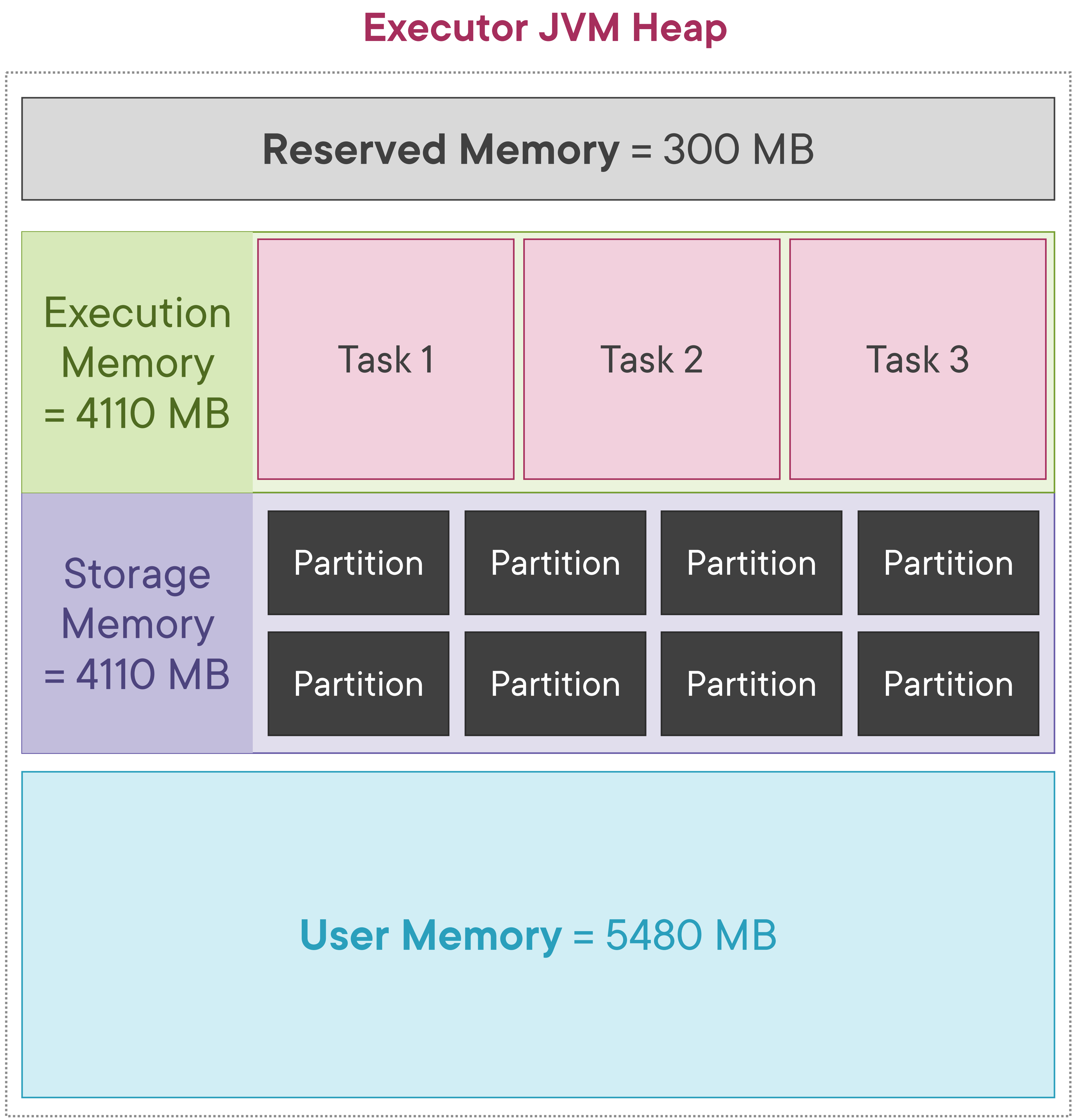


spark.executor.cores = 4

Executor JVM Heap



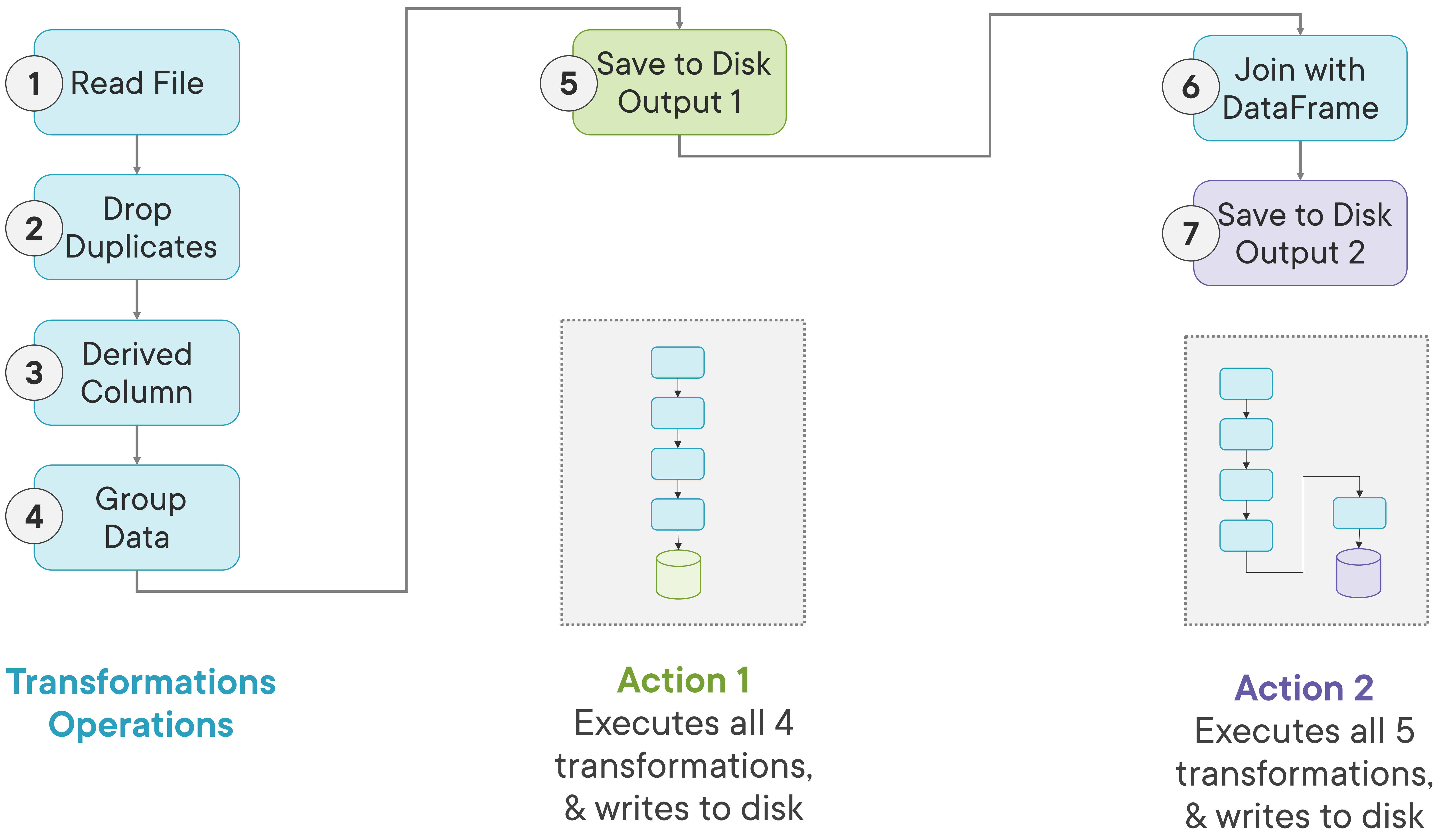
spark.executor.cores = 4



spark.executor.cores = 4

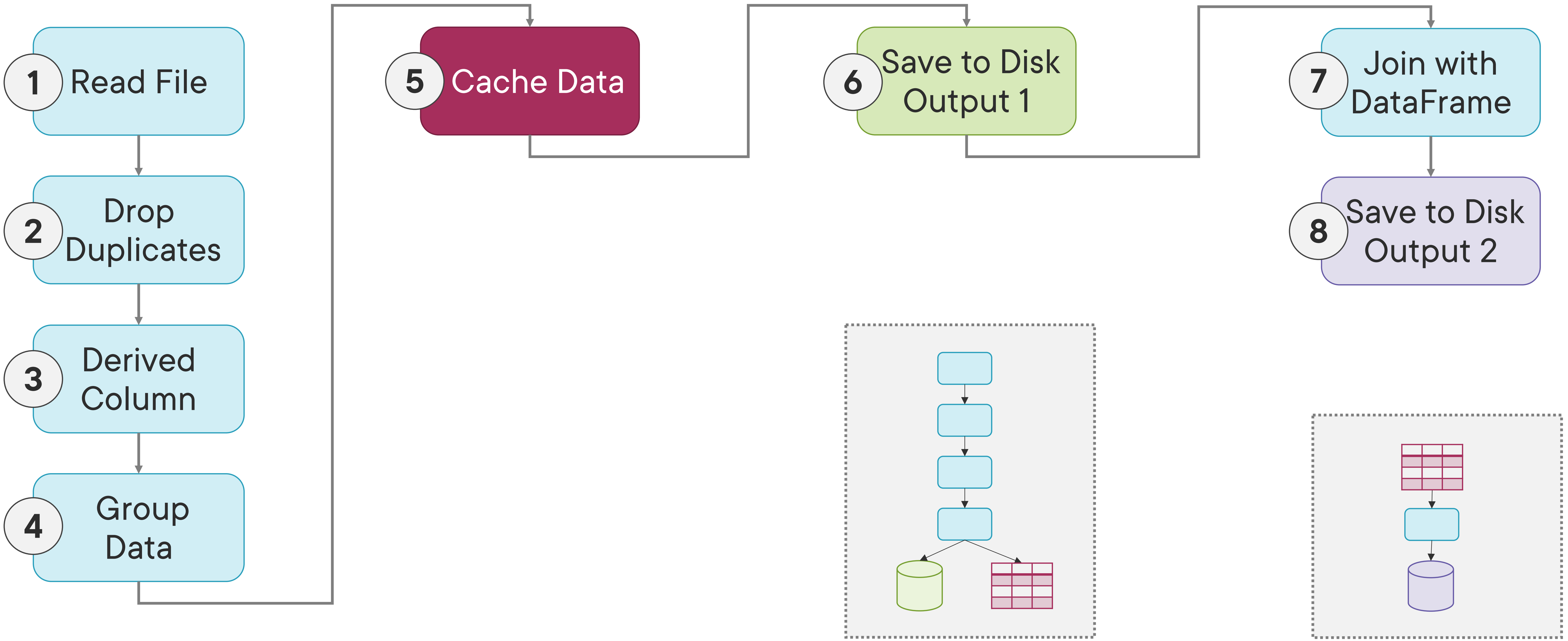
If Tasks need more memory, it may result in Out-of-Memory exceptions

Persisting Data


1 Read File
2 Drop Duplicates
3 Derived Column
4 Group Data
5 Save to Disk Output 1
6 Join with DataFrame
7 Save to Disk Output 2
Transformations Operations
Action 1
Executes all 4 transformations, & writes to disk
Action 2
Executes all 5 transformations, & writes to disk

Persist data to avoid re-computation of complex transformations with every Action operation

Persisting data is a Lazy operation

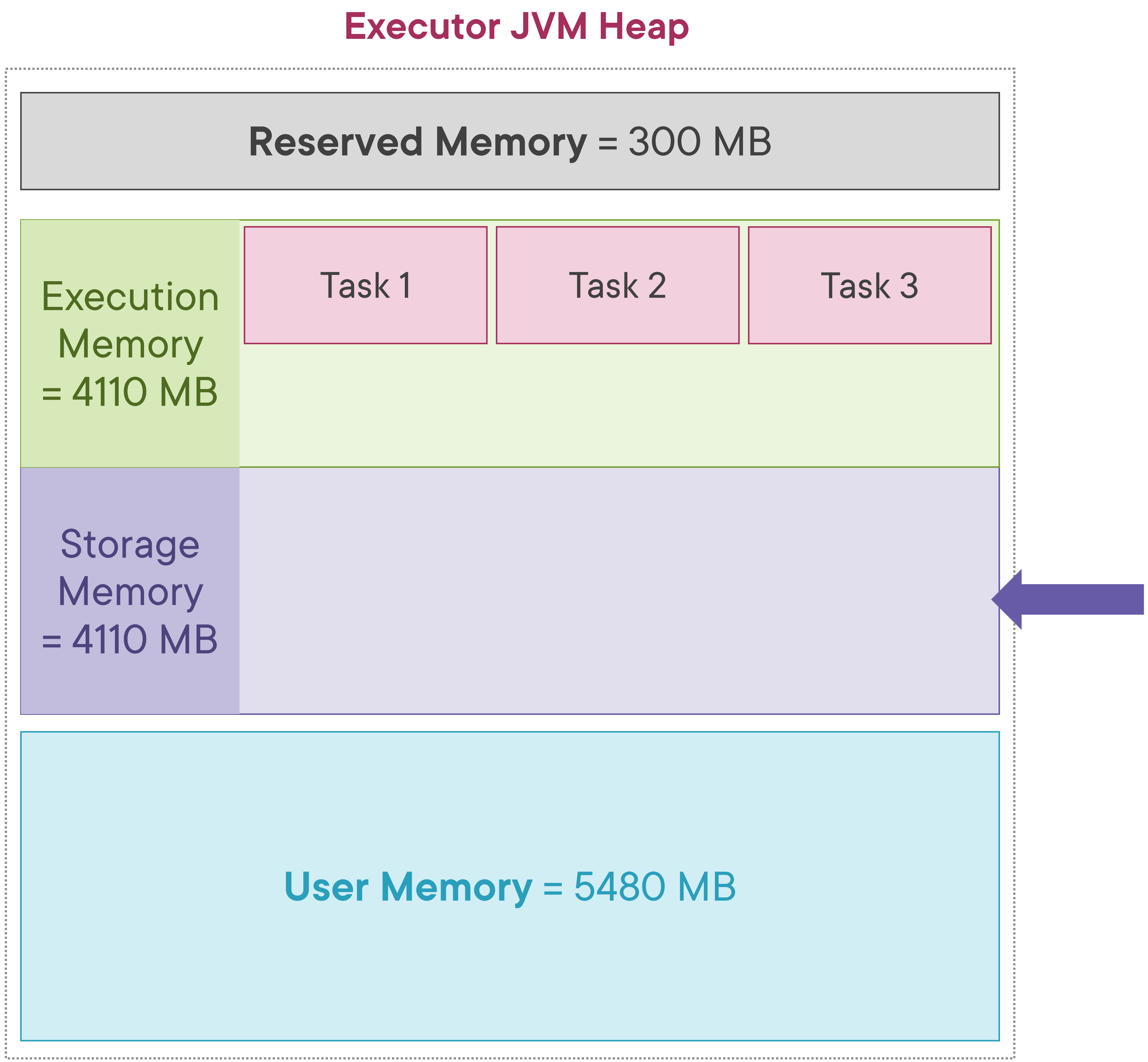


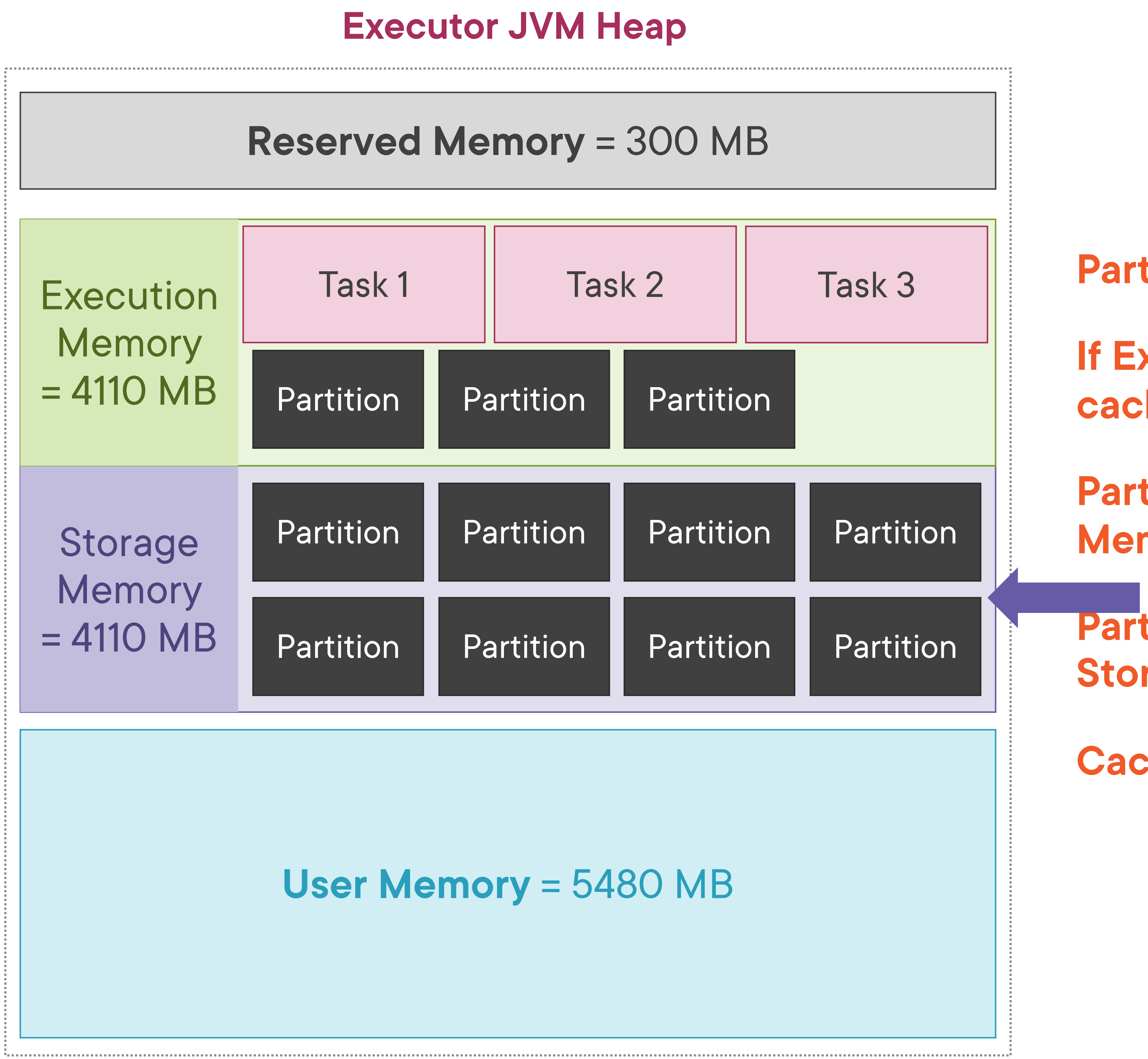
Transformations Operations

No data is cached
Lazy operation

Action 1
Executes all 4 transformations, caches data, & writes to disk

Action 2
Uses cached data, applies join, & writes to disk


Executor JVM Heap
Reserved Memory = 300 MB
Execution Memory = 4110 MB
Task 1
Task 2
Task 3
Storage Memory = 4110 MB
User Memory = 5480 MB



Partitions can be cached in Storage Memory

If Execution Memory is free, partitions can be cached there too

Partitions will be evicted from Execution Memory if tasks need them

Partitions can be spilled over to disk if Storage Memory does not have enough space

Cache can be manually evicted

Partitions are evicted from cache in LRU (Least-Recently-Used) fashion

Caching Methods

Two methods – cache() and persist()

Applying cache() and persist() without arguments

- On RDD: MEMORY_ONLY
- On DataFrame: MEMORY_AND_DISK

cache() has no arguments

persist() supports Storage Level as argument

- MEMORY_ONLY: Partitions that can fit in memory are cached; others are recomputed each time
- MEMORY_AND_DISK: Partitions that can fit in memory are cached; others are spilled to local disk of Worker
- DISK_ONLY: Only stored on disk; pulled when required
- *more…*

Recommended to unpersist data if not required

Spark Join Strategies and Broadcast Joins

Spark Join Strategy determines how to move, shuffle, sort, group & merge data across executors during join operation

Join Strategies

Broadcast Hash Join

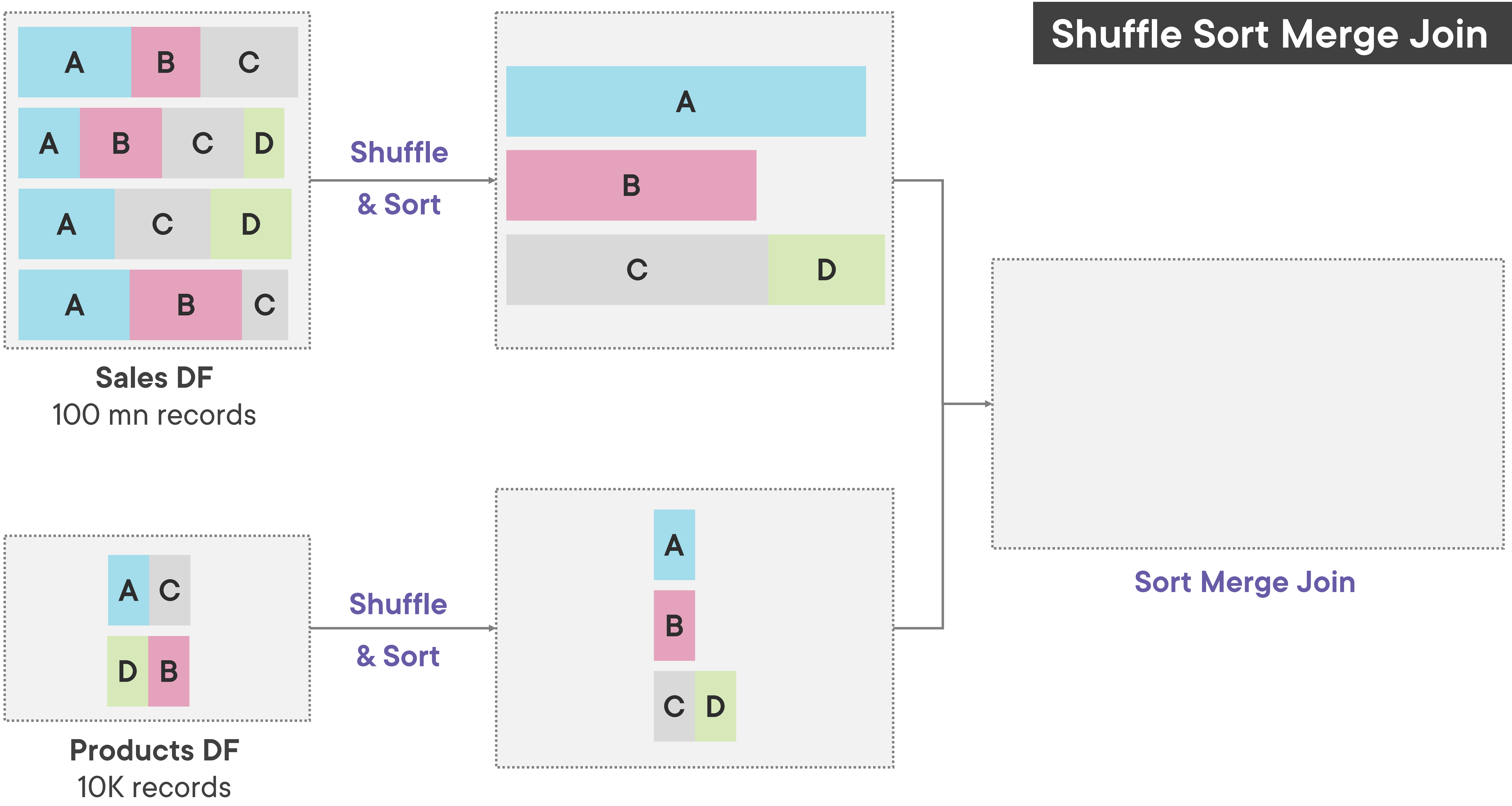
Shuffle Sort Merge Join

Shuffle Hash Join

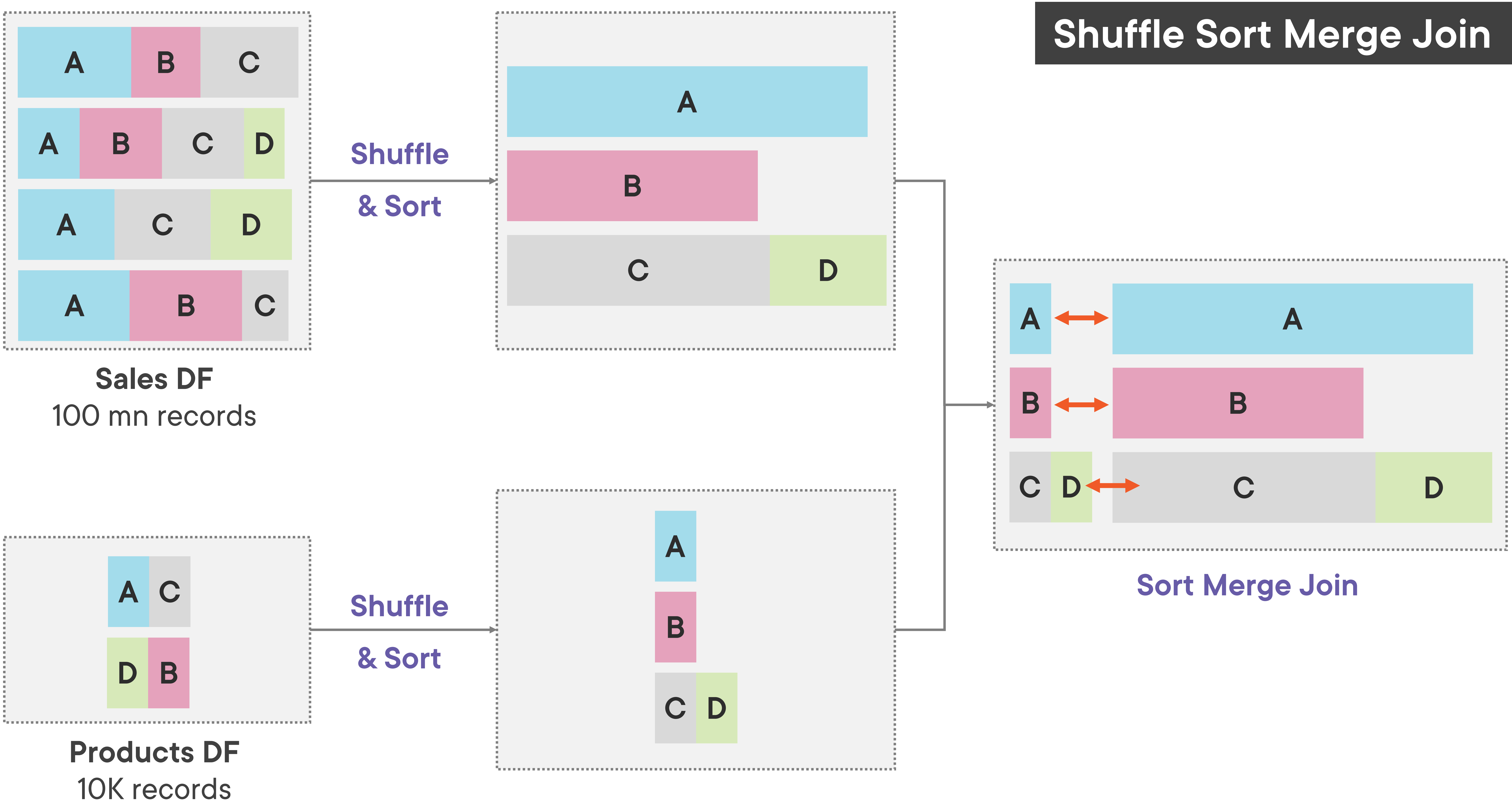
Cartesian Join

Broadcast Nested Loop Join

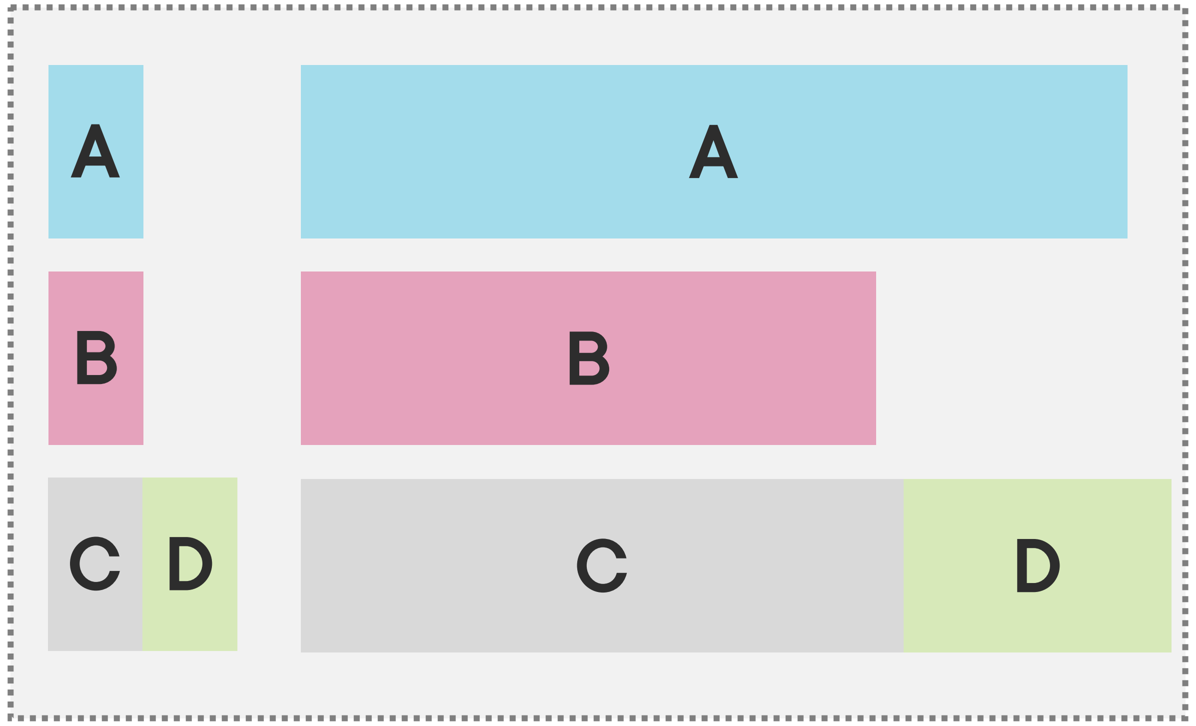
Shuffle Sort Merge Join



Shuffle Sort Merge Join



Shuffle Sort Merge Join



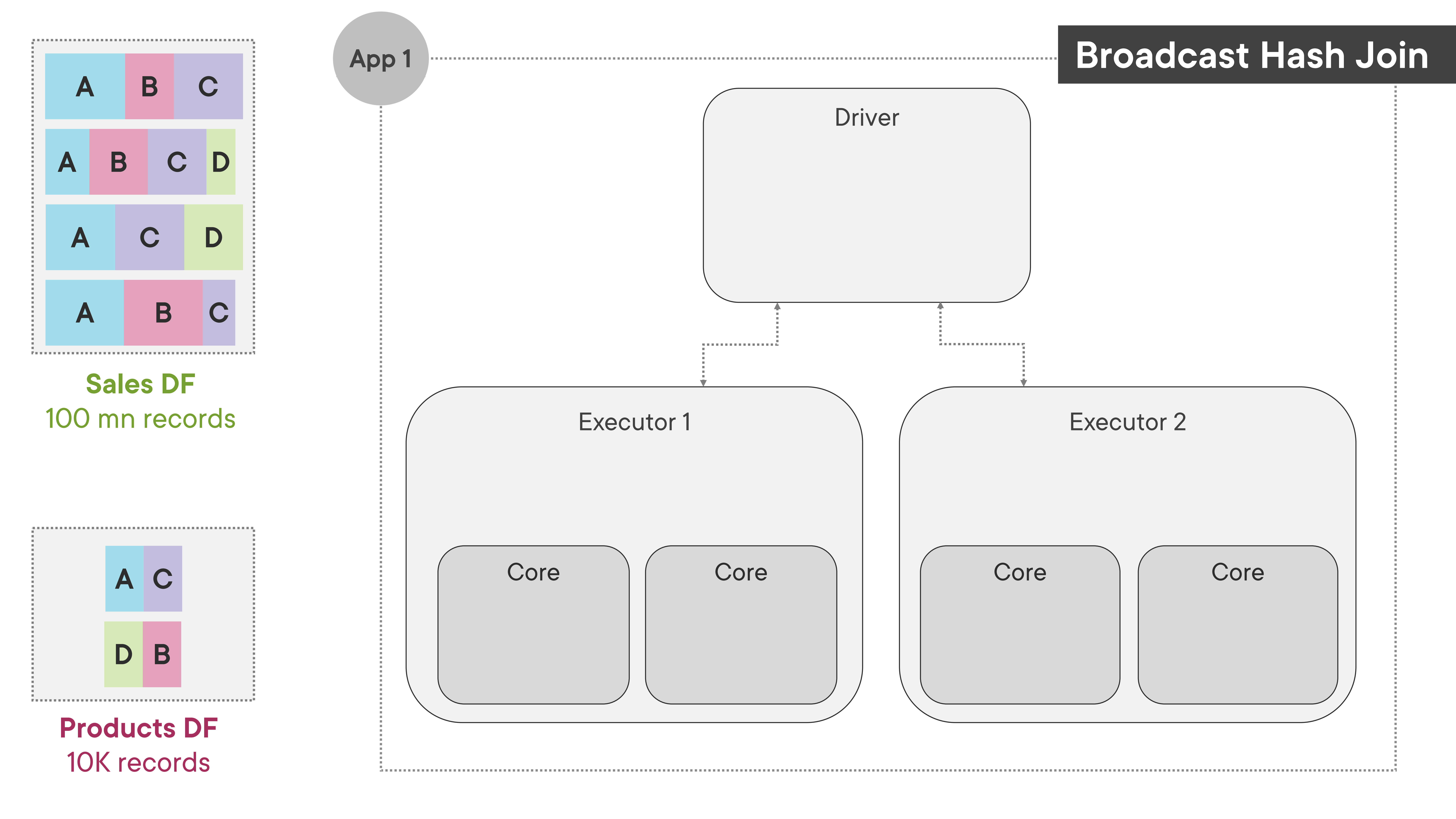
Involves shuffling & sorting of data for both datasets

Great for joining two large datasets

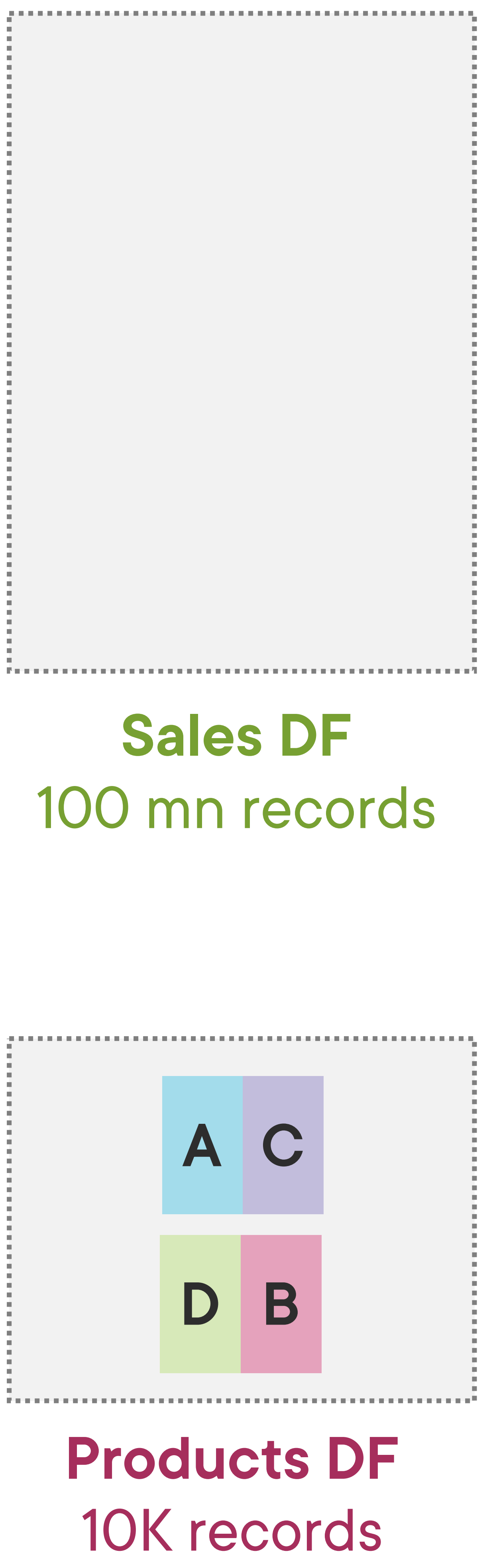
But even if one dataset is small, shuffling still happens

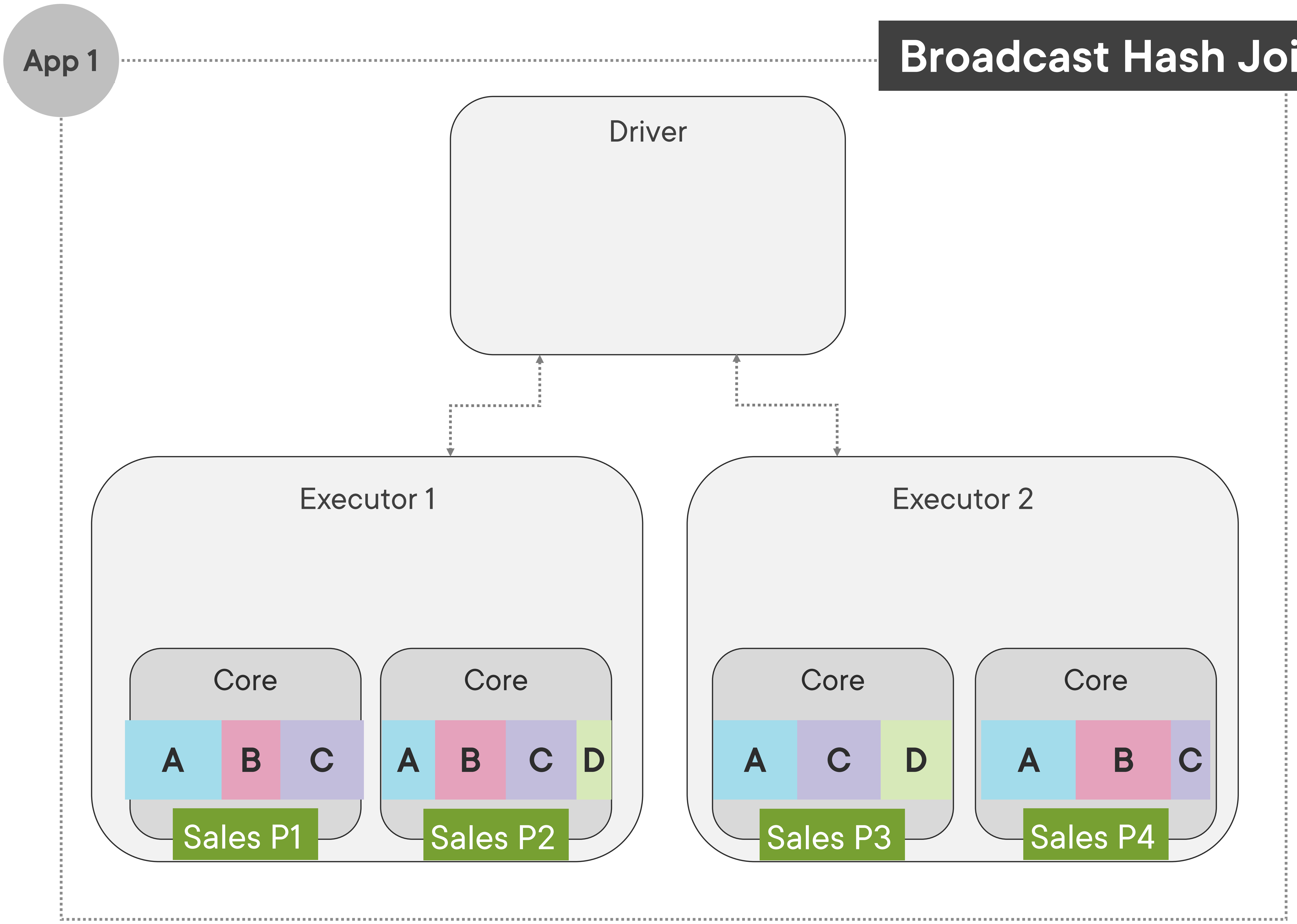
Expensive join strategy

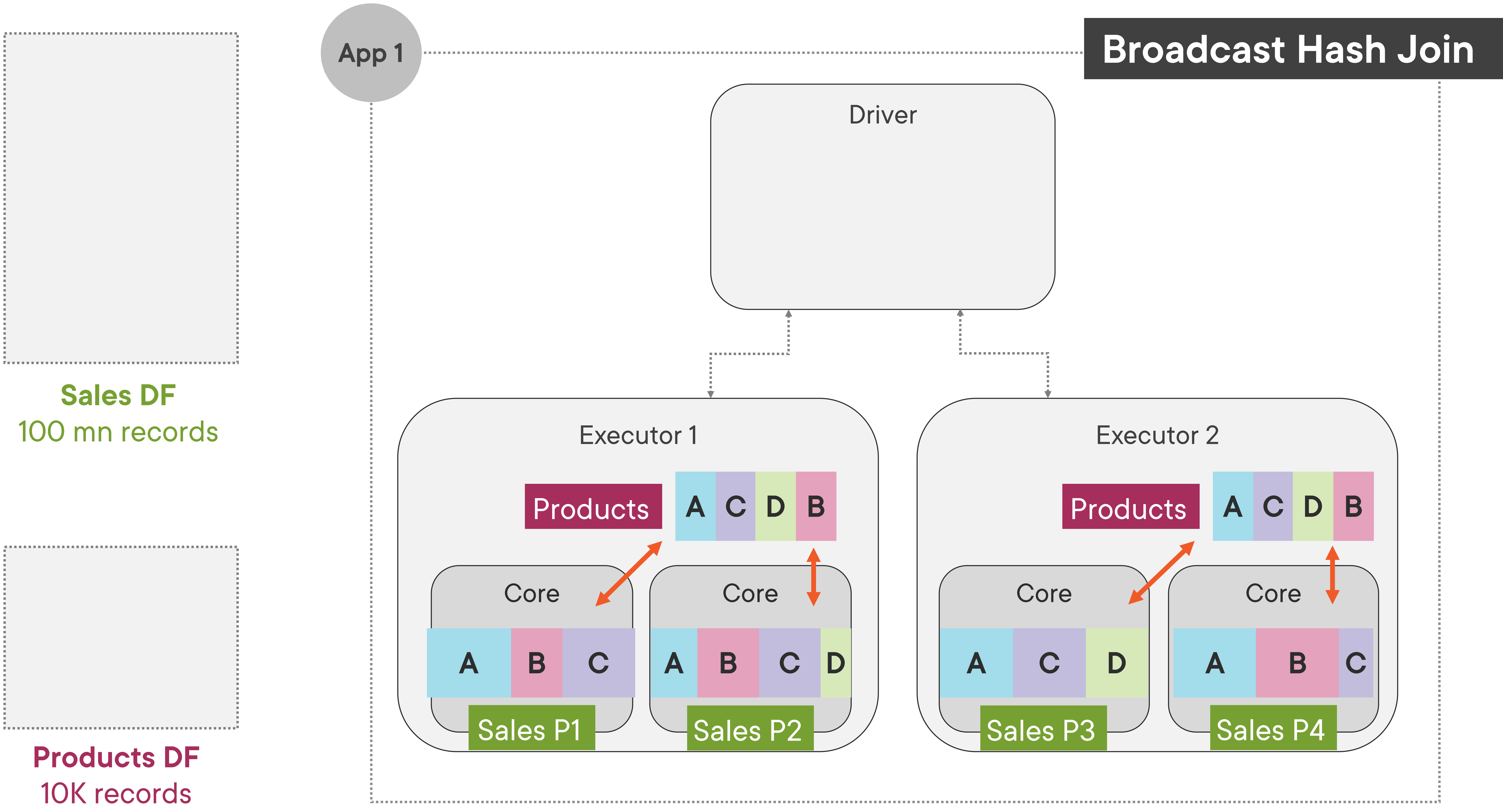
If one dataset in a join operation is small, avoid shuffling using Broadcast Hash Join


Broadcast Hash Join
App 1
Driver
Executor 1
Executor 2
Core
Core
Core
Core
A B C
A B C D
A C D
A B C
Sales DF
100 mn records
A C
D B
Products DF
10K records

Broadcast Hash Join







Broadcast Hash Join
App 1
Sales DF
100 mn records
Products DF
10K records
Driver
Executor 1
Executor 2
Products
A C D B
Products
A C D B
Core
Core
Core
Core
A B C
A B C D
A C D
A B C
Sales P1
Sales P2
Sales P3
Sales P4

Broadcast Hash Join

Useful when one of datasets is small

Copy of smaller dataset is sent to each executor once

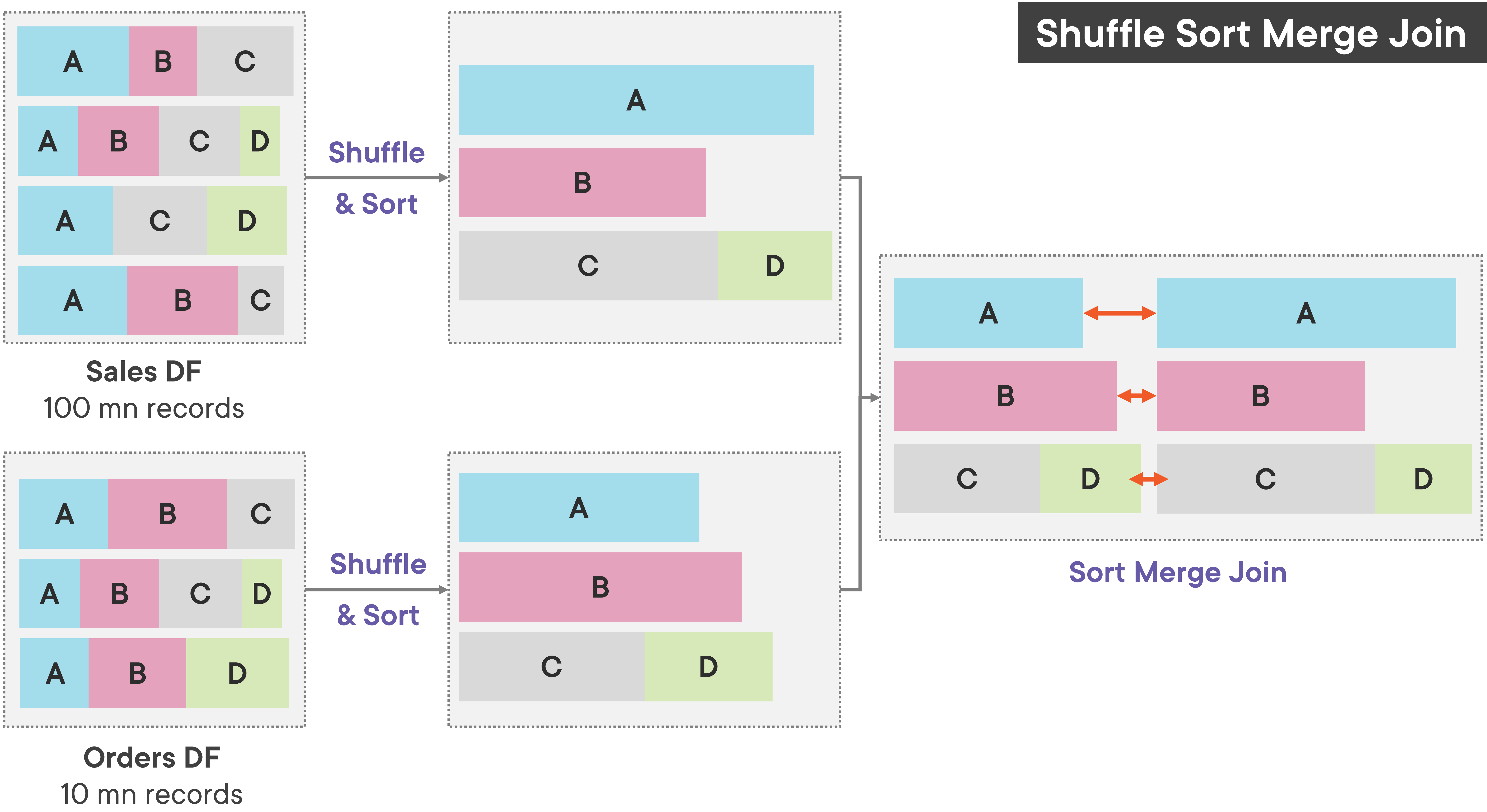
No data shuffling required

Non-expensive & fastest join operation

Spark can perform auto broadcast or can be forced

Optimizing Shuffle Sort Join with Bucketing

Shuffle Sort Merge Join



Bucketing is an optimization technique
to avoid shuffles during joins

What is Bucketing?

Id	City	Amount
1	Seattle	600
2	London	300
3	Seattle	700
4	Delhi	400
5	Paris	900
6	Seattle	900
7	Delhi	200

DataFrame

Hash function

What is Bucketing?

Id	City	Amount
1	Seattle	600
2	London	300
3	Seattle	700
4	Delhi	400
5	Paris	900
6	Seattle	900
7	Delhi	200

DataFrame

Hash function

1	Seattle	600

What is Bucketing?

Id	City	Amount
1	Seattle	600
2	London	300
3	Seattle	700
4	Delhi	400
5	Paris	900
6	Seattle	900
7	Delhi	200

DataFrame

Hash function

1	Seattle	600

2	London	300

What is Bucketing?

Id	City	Amount
1	Seattle	600
2	London	300
3	Seattle	700
4	Delhi	400
5	Paris	900
6	Seattle	900
7	Delhi	200

DataFrame

Hash function

1	Seattle	600
3	Seattle	700

2	London	300

What is Bucketing?

Id	City	Amount
1	Seattle	600
2	London	300
3	Seattle	700
4	Delhi	400
5	Paris	900
6	Seattle	900
7	Delhi	200

DataFrame

Hash function

1	Seattle	600
3	Seattle	700
6	Seattle	900

2	London	300
5	Paris	900

4	Delhi	400
7	Delhi	200

Bucketed data is written to disk

Bucketing

Pre-shuffle large datasets & write it to disk in buckets

Use bucketed data in subsequent queries

Use when dataset is frequently used in joins, aggregations or window operations

Conditions for joining bucketed datasets

- Bucketed datasets must be stored as tables
- Both datasets must be bucketed on join columns
- Number of buckets must be the same

Dynamic Resource Allocation

Resource Scheduling for a Job

Scheduling Across Applications

Scheduling Within an Application

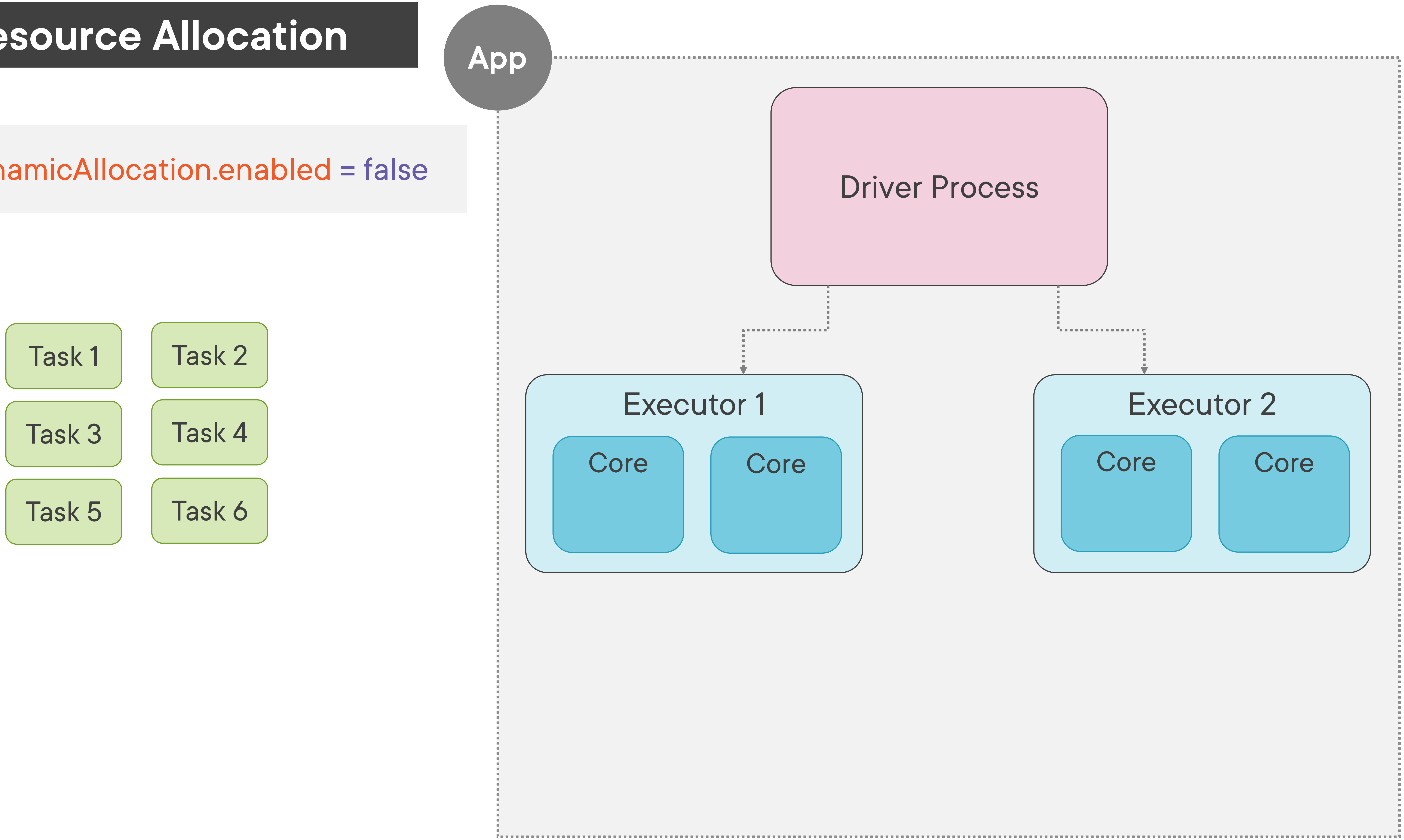
Resource Scheduling Across Applications

Static Resource Allocation

Dynamic Resource Allocation

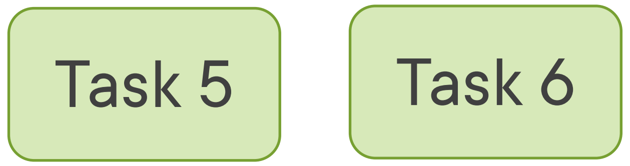
Static Resource Allocation

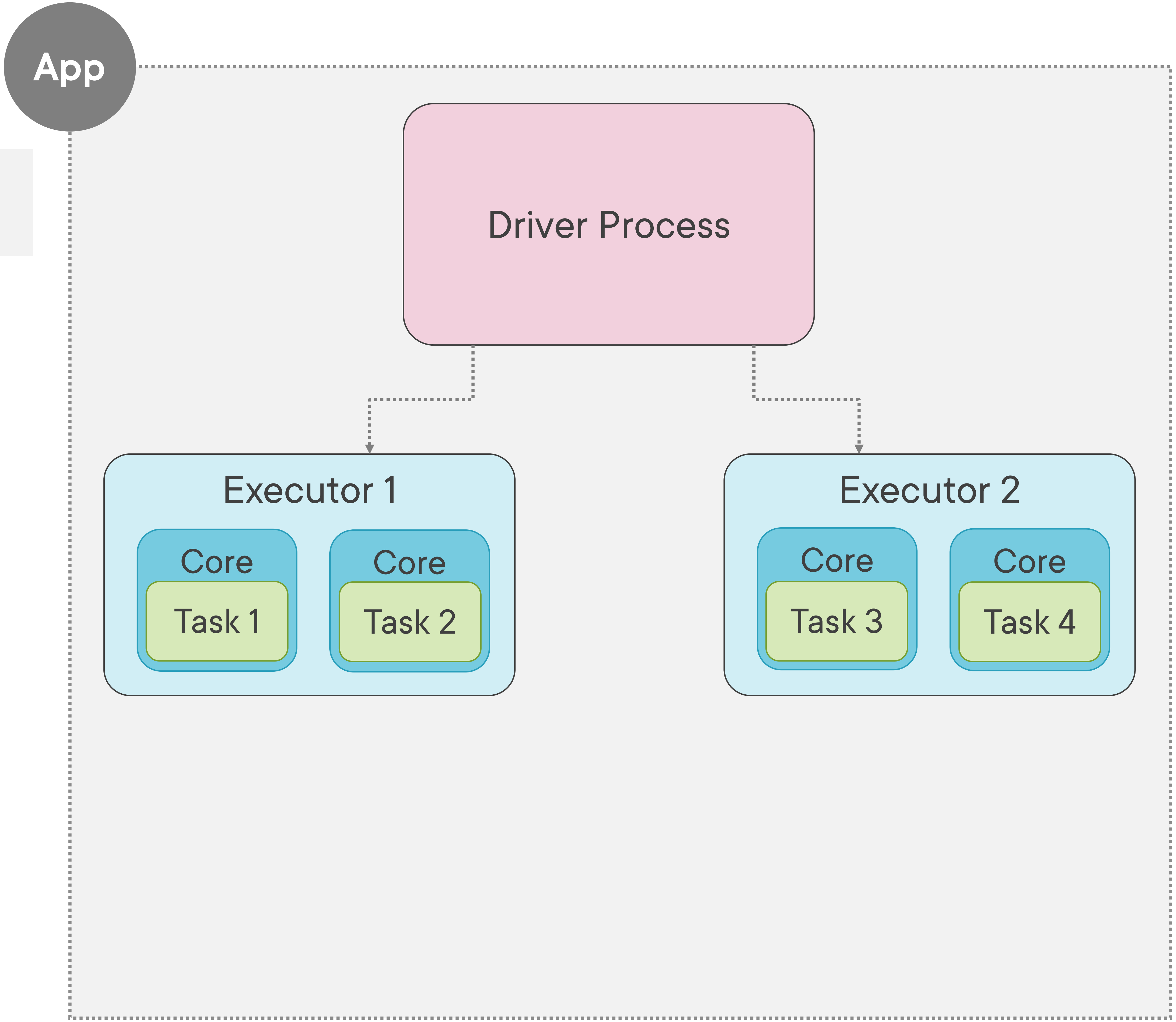
spark.dynamicAllocation.enabled = false



Static Resource Allocation

spark.dynamicAllocation.enabled = false

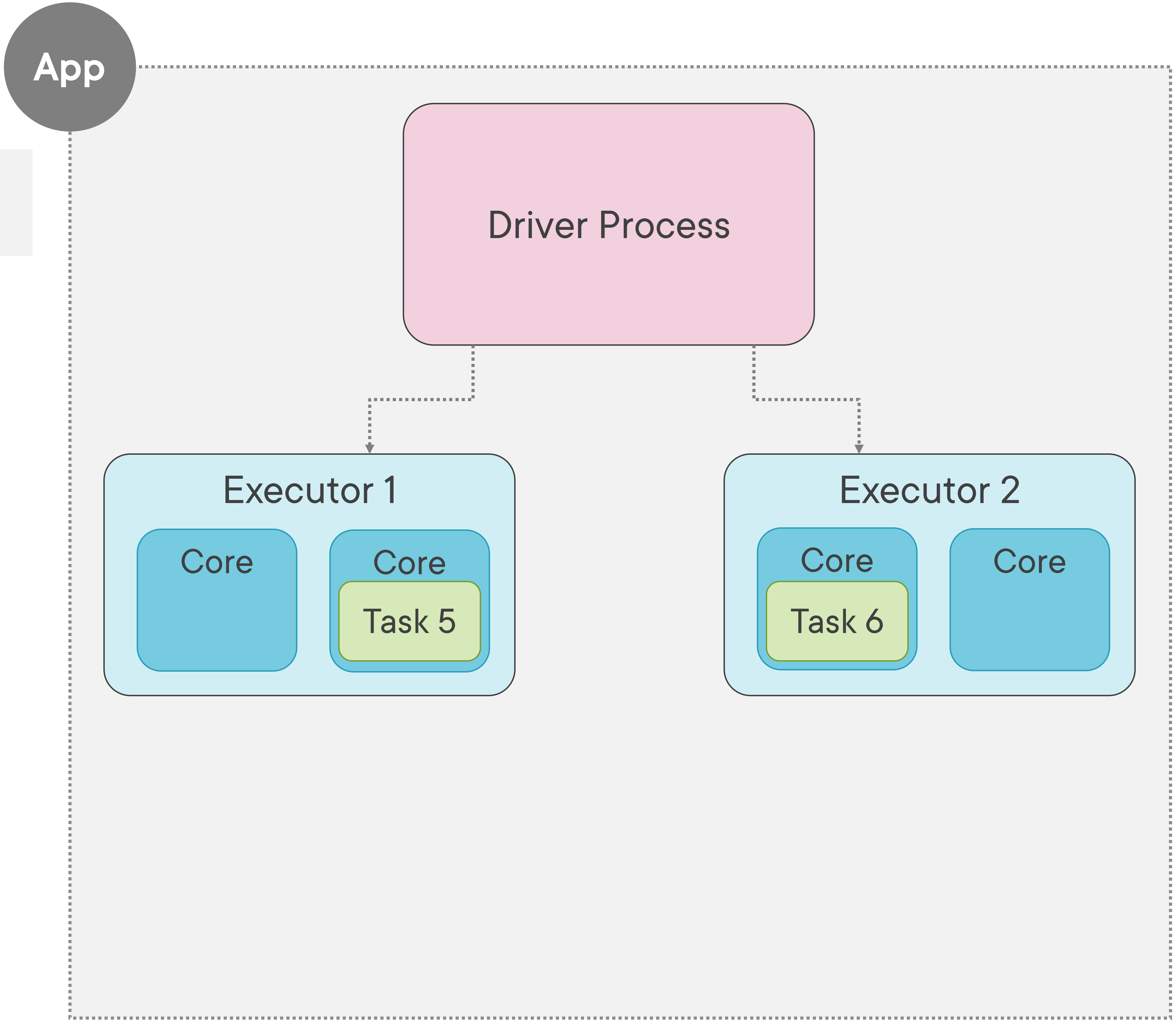




Static Resource Allocation

spark.dynamicAllocation.enabled = false

CPU Cores & Memory are pre-defined for an application



Dynamic Resource Allocation

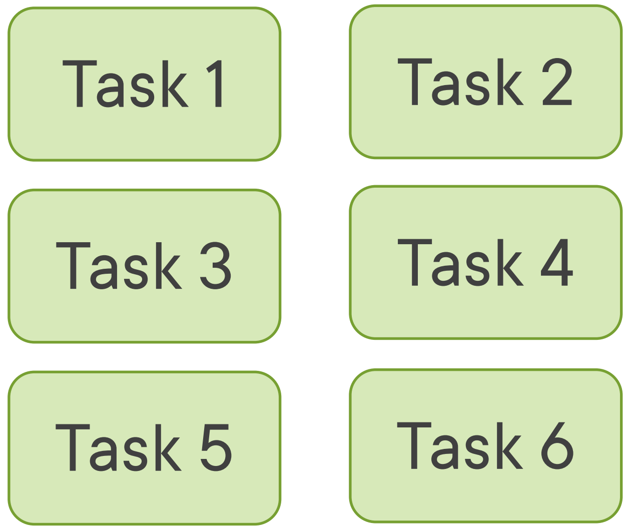
spark.dynamicAllocation :

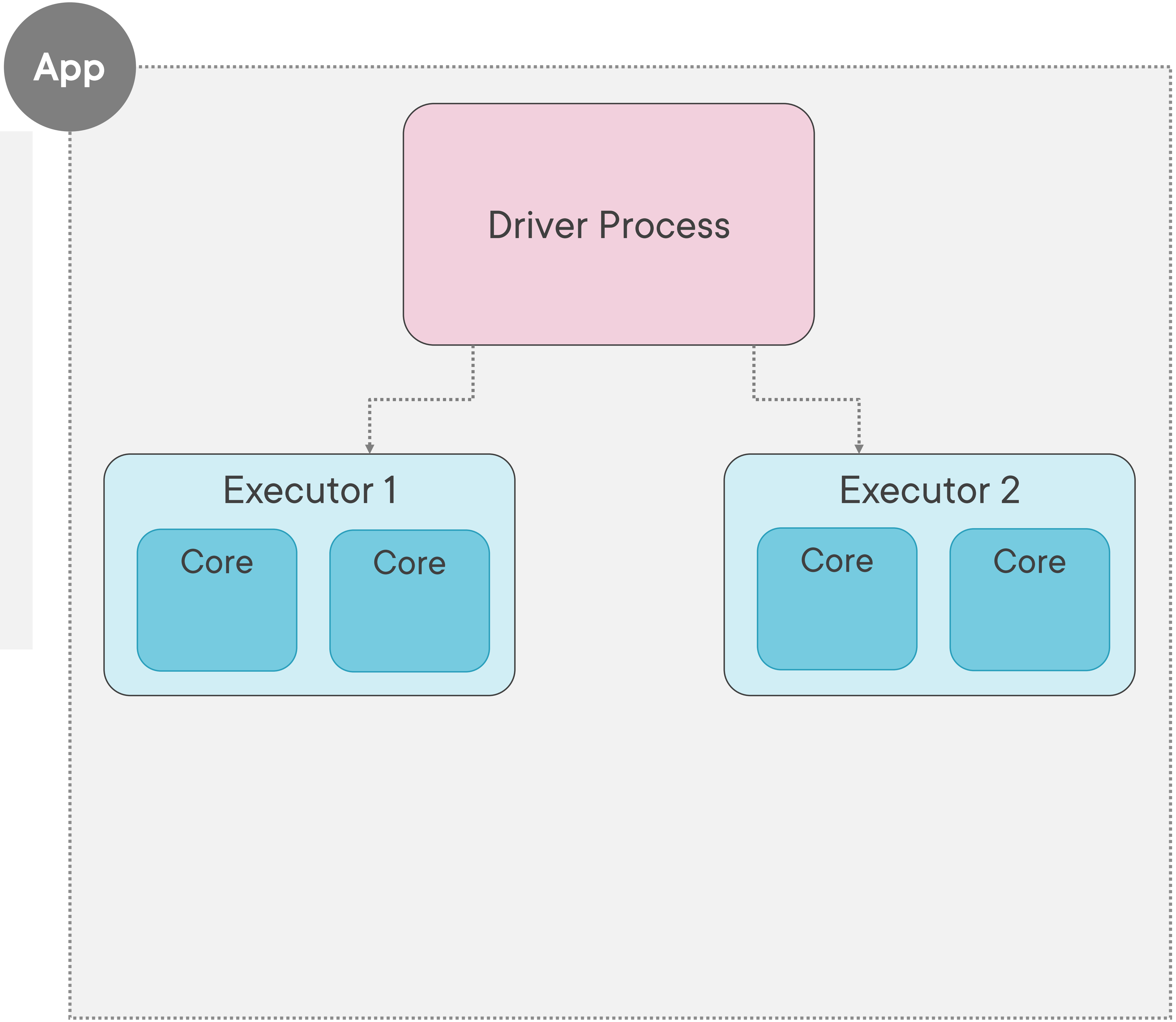
.enabled = true

.shuffleTracking. enabled = true

.minExecutors = 0

.maxExecutors = 5





Dynamic Resource Allocation

spark.dynamicAllocation :

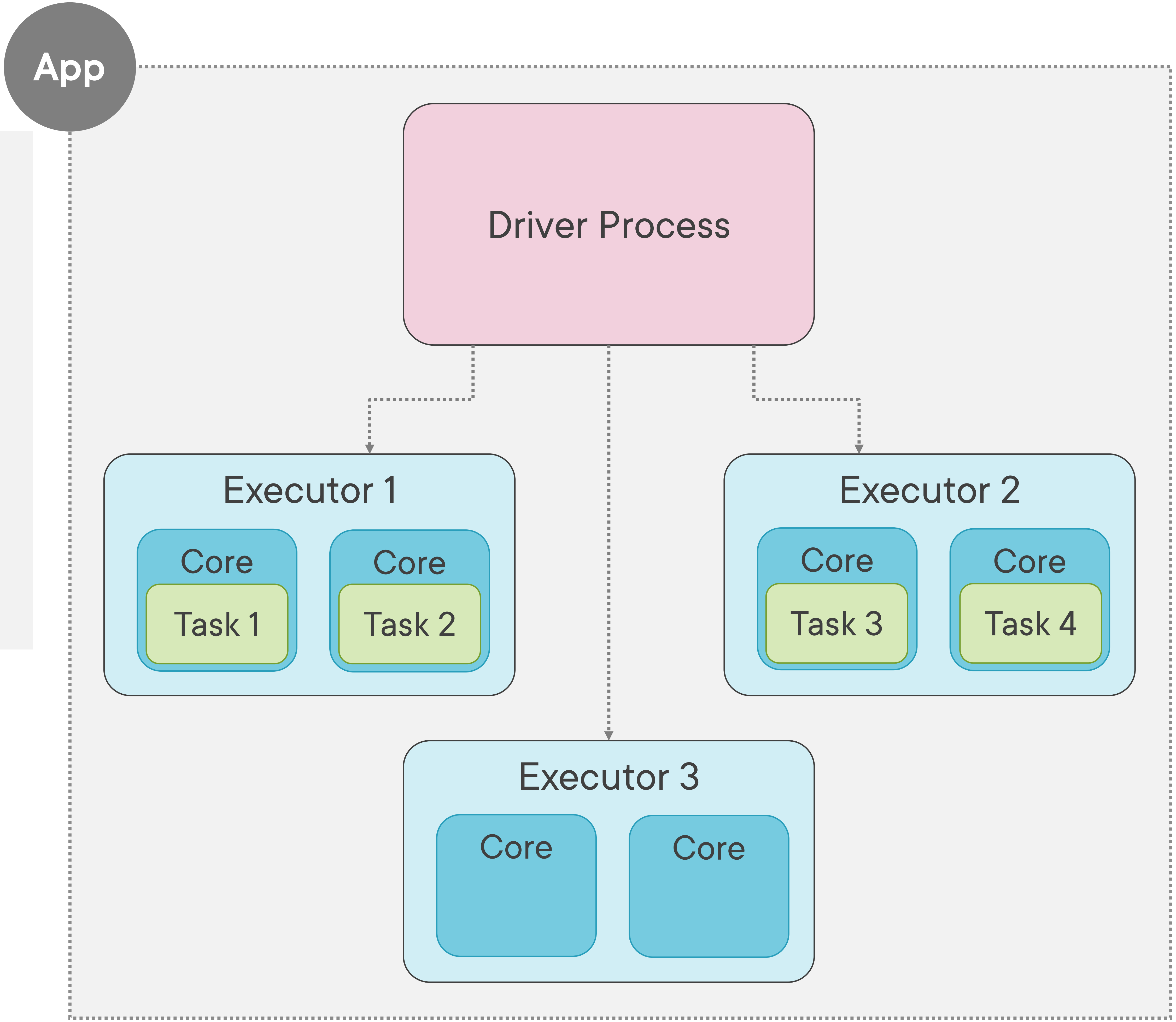
.enabled = true

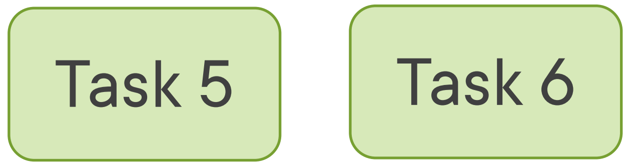
.shuffleTracking. enabled = true

.minExecutors = 0

.maxExecutors = 5

.schedulerBacklogTimeout = 1s





Dynamic Resource Allocation

spark.dynamicAllocation :

.enabled = true

.shuffleTracking. enabled = true

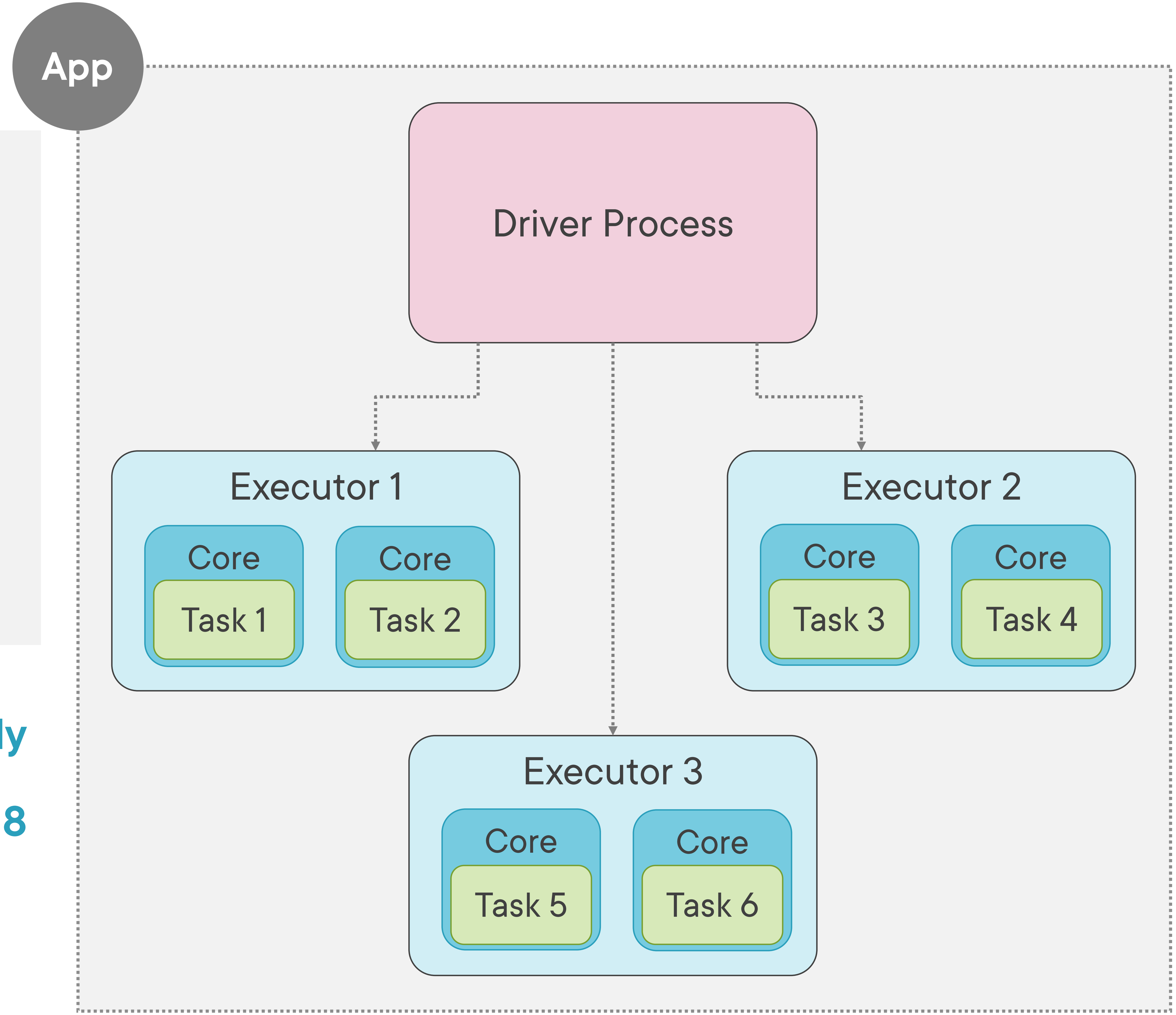
.minExecutors = 0

.maxExecutors = 5

.schedulerBacklogTimeout = 1s

Executors will increase exponentially

Starting with 1 executor – then 2, 4, 8



Dynamic Resource Allocation

spark.dynamicAllocation :

.enabled = true

.shuffleTracking. enabled = true

.minExecutors = 0

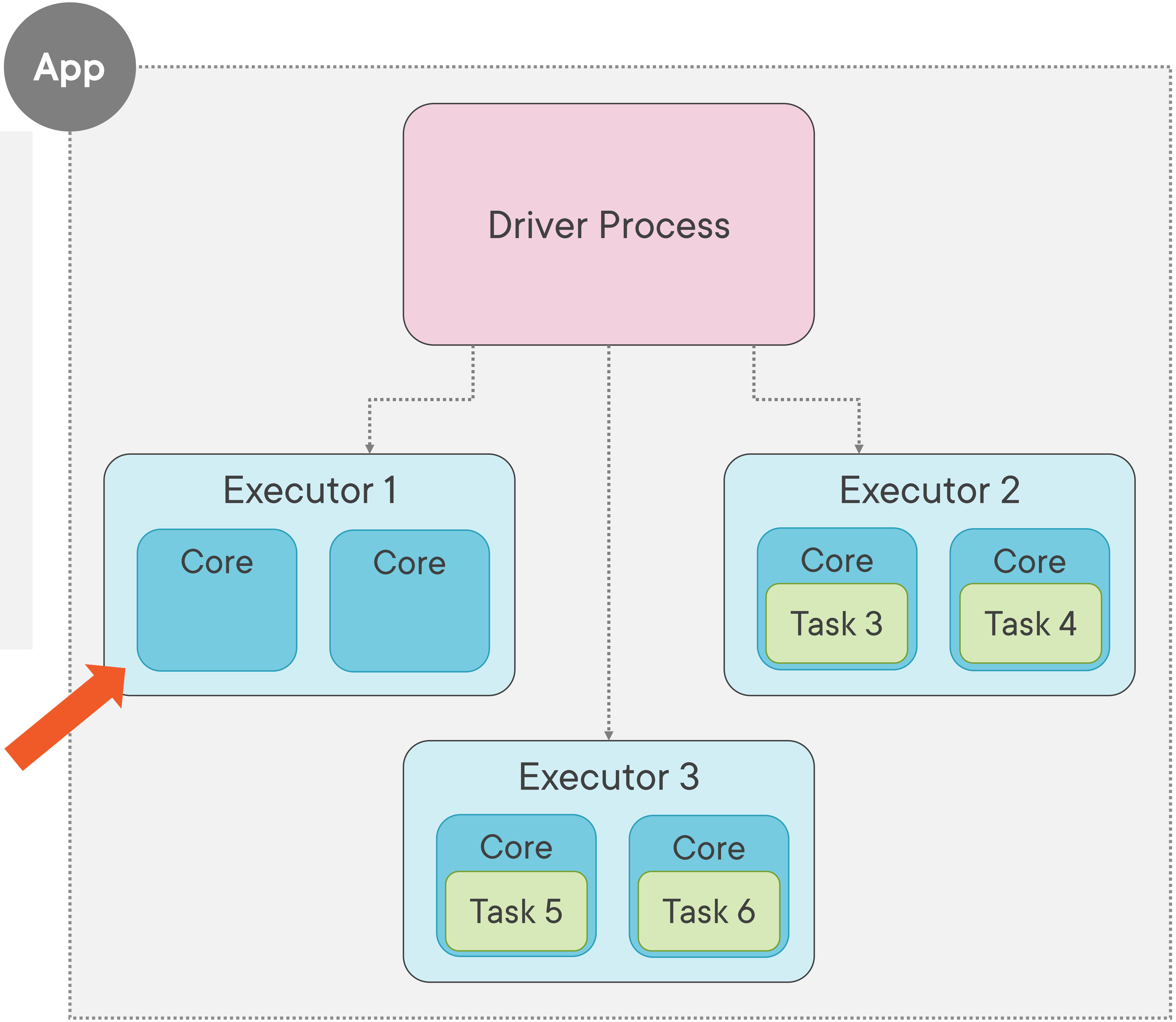
.maxExecutors = 5

.schedulerBacklogTimeout = 1s

.executorIdleTimeout = 60s

Any cached data on executor will also be removed

No shuffle data is lost



Request more executors
to complete pending tasks!

Remove executors
when they are idle!

Use Cases

Running ad-hoc interactive applications

Long running ETL jobs to process multiple entities

Jobs with large shuffle operations

Resource Allocation Using Fair Scheduling

Resource Scheduling for a Job

Scheduling Across Applications

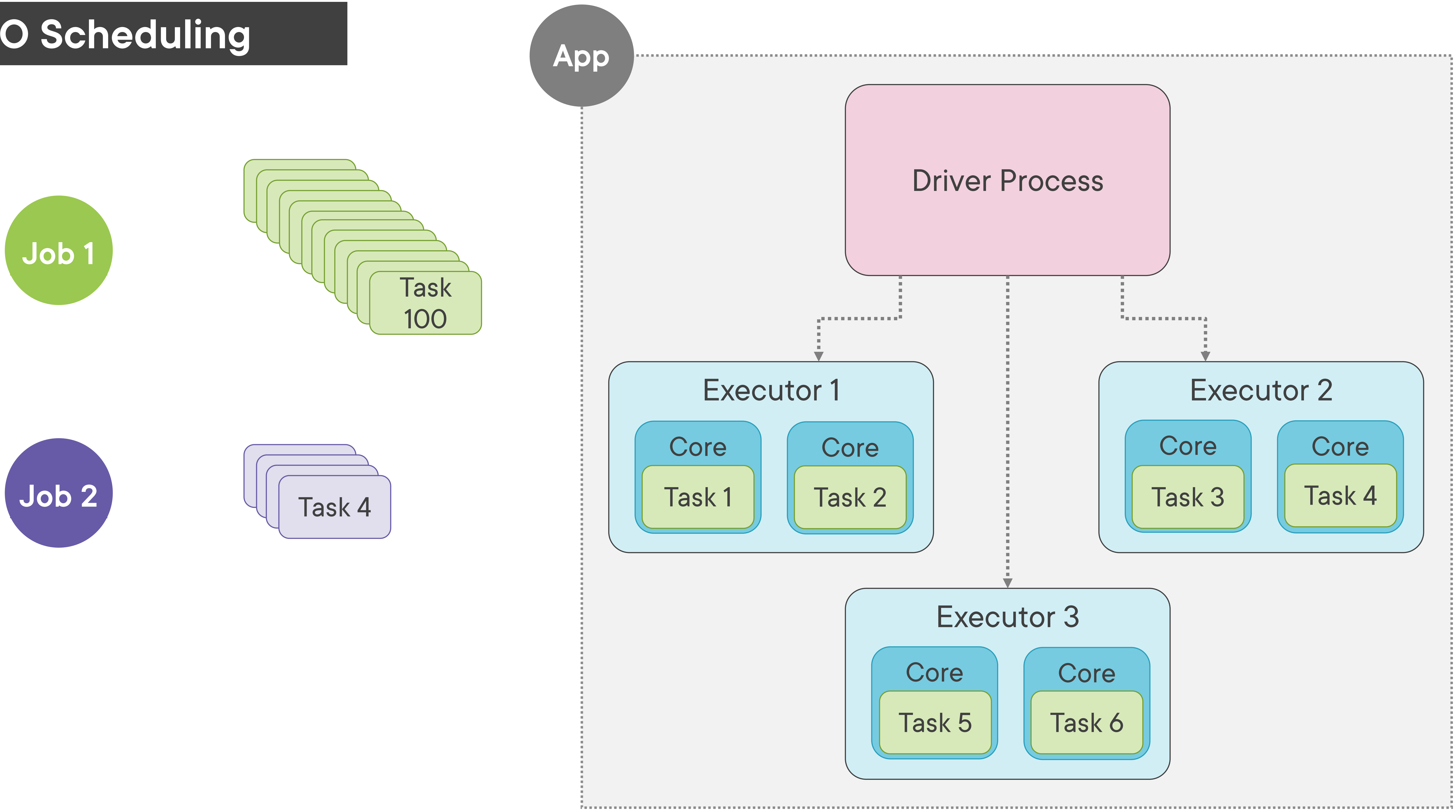
Scheduling Within an Application

Resource Scheduling Within Application

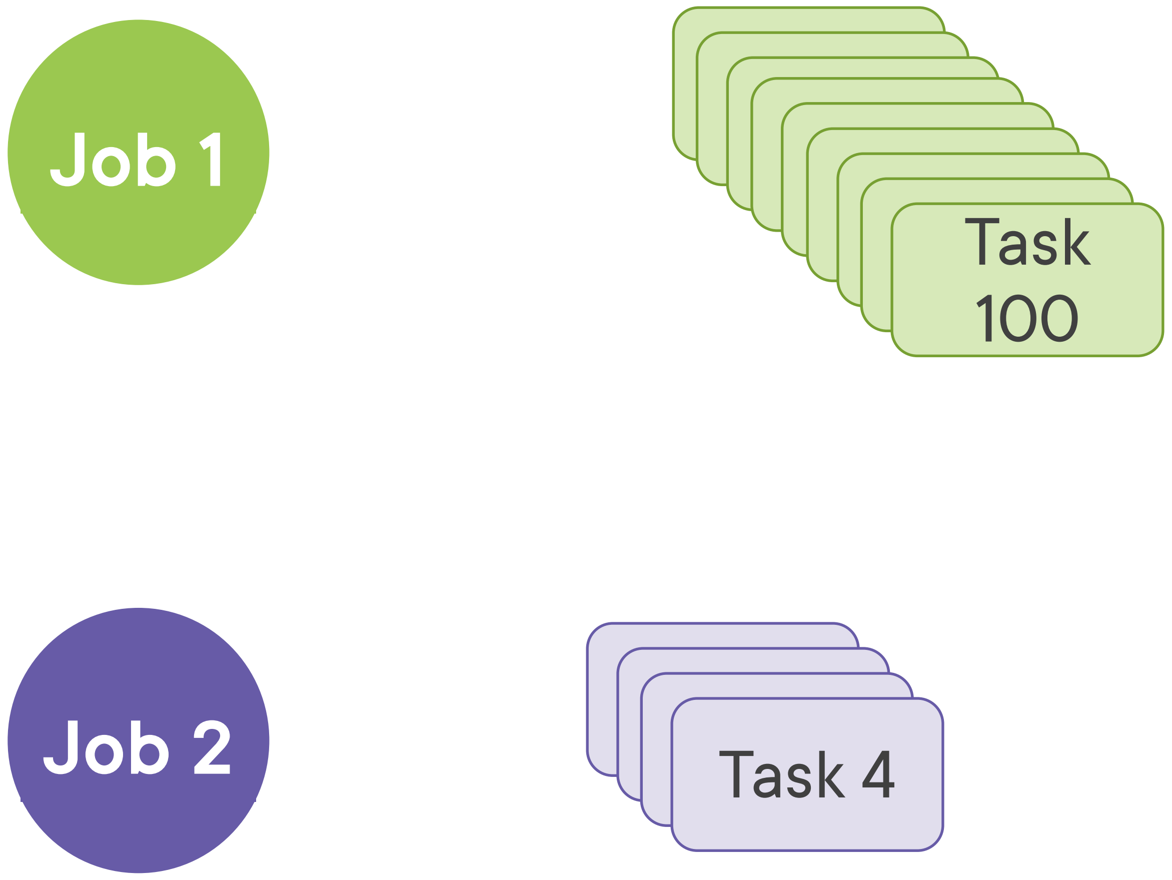
FIFO Scheduling

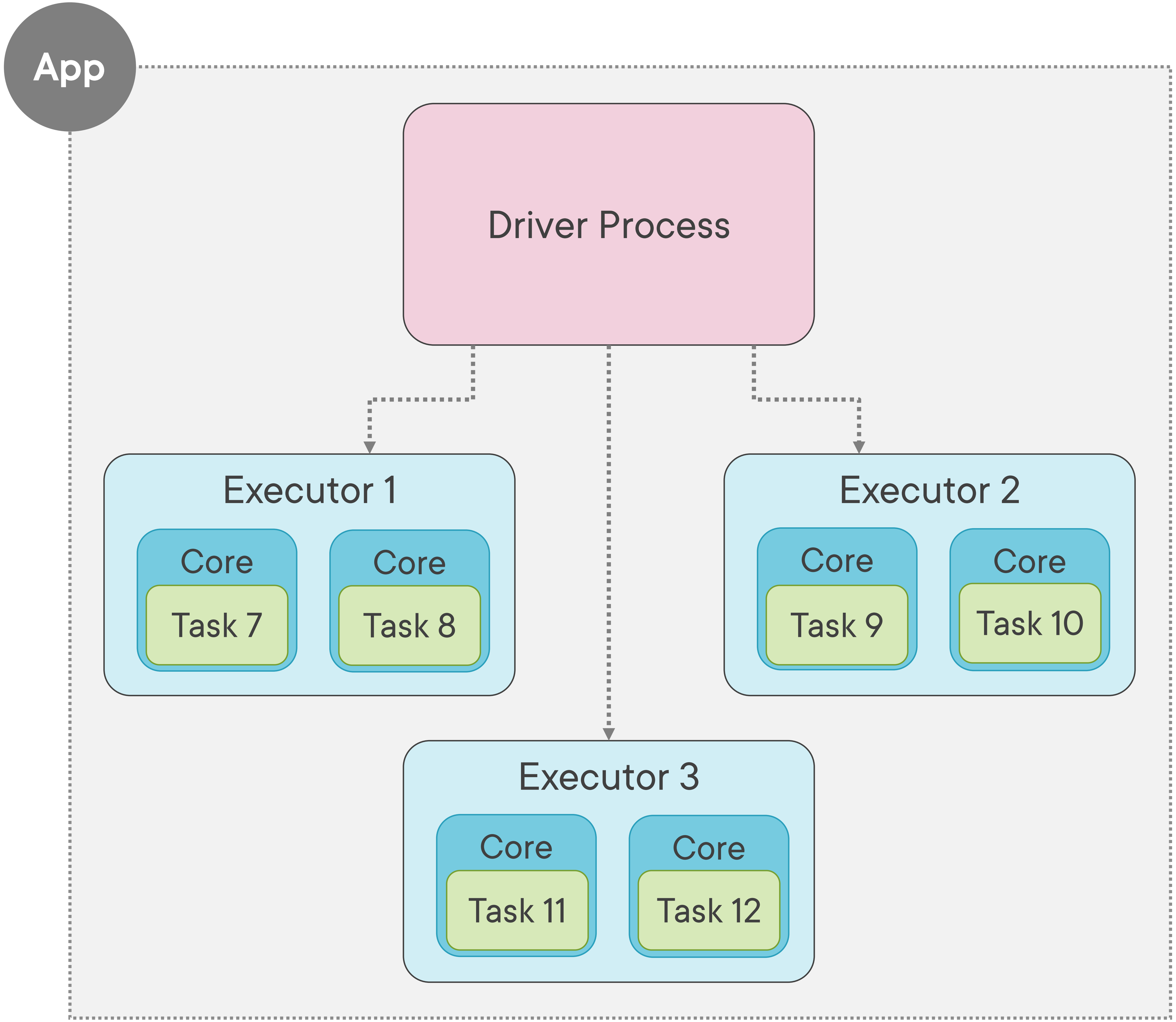
FAIR Scheduling

FIFO Scheduling



FIFO Scheduling



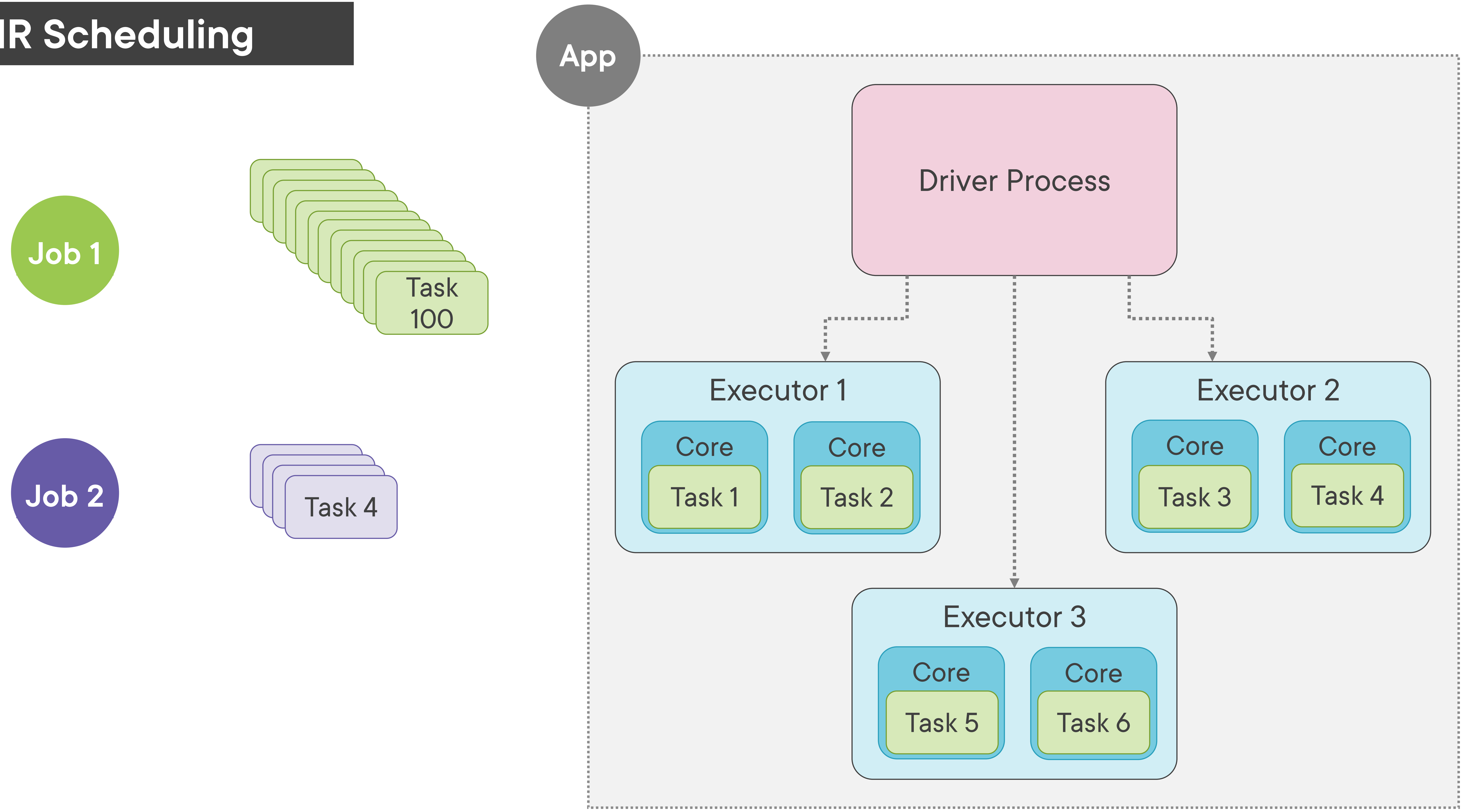


Default scheduling

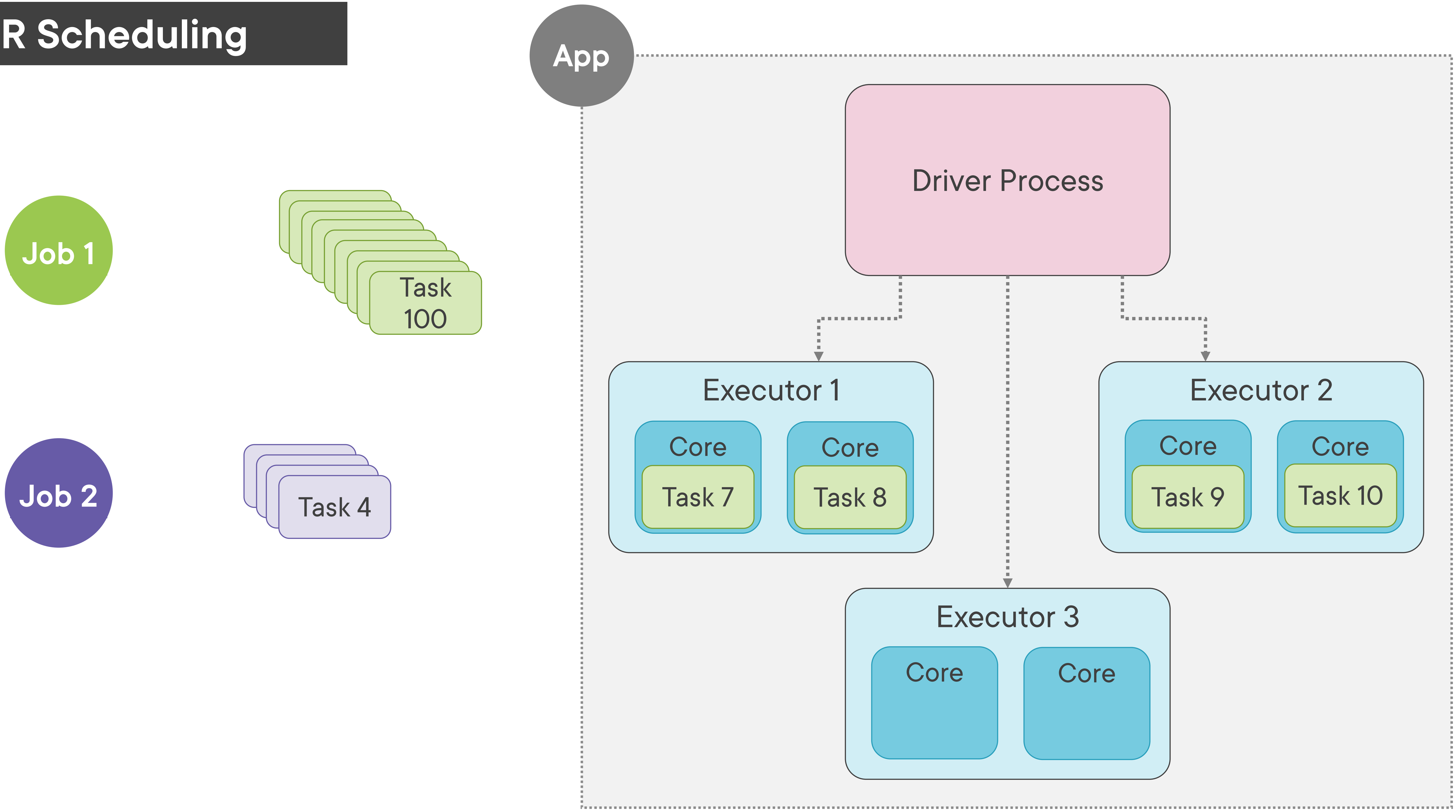
First job gets priority on resources

Other jobs in queue are delayed

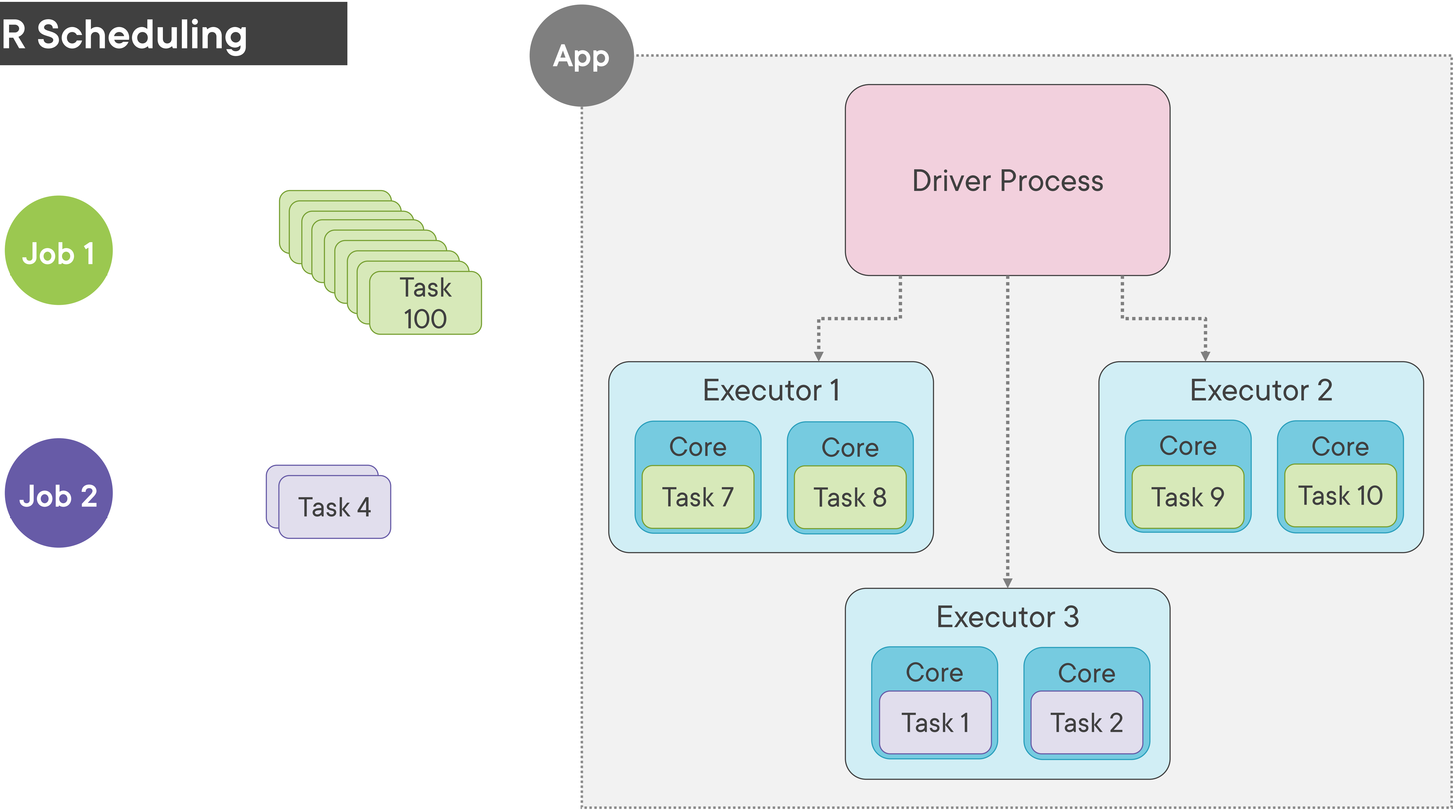
FAIR Scheduling



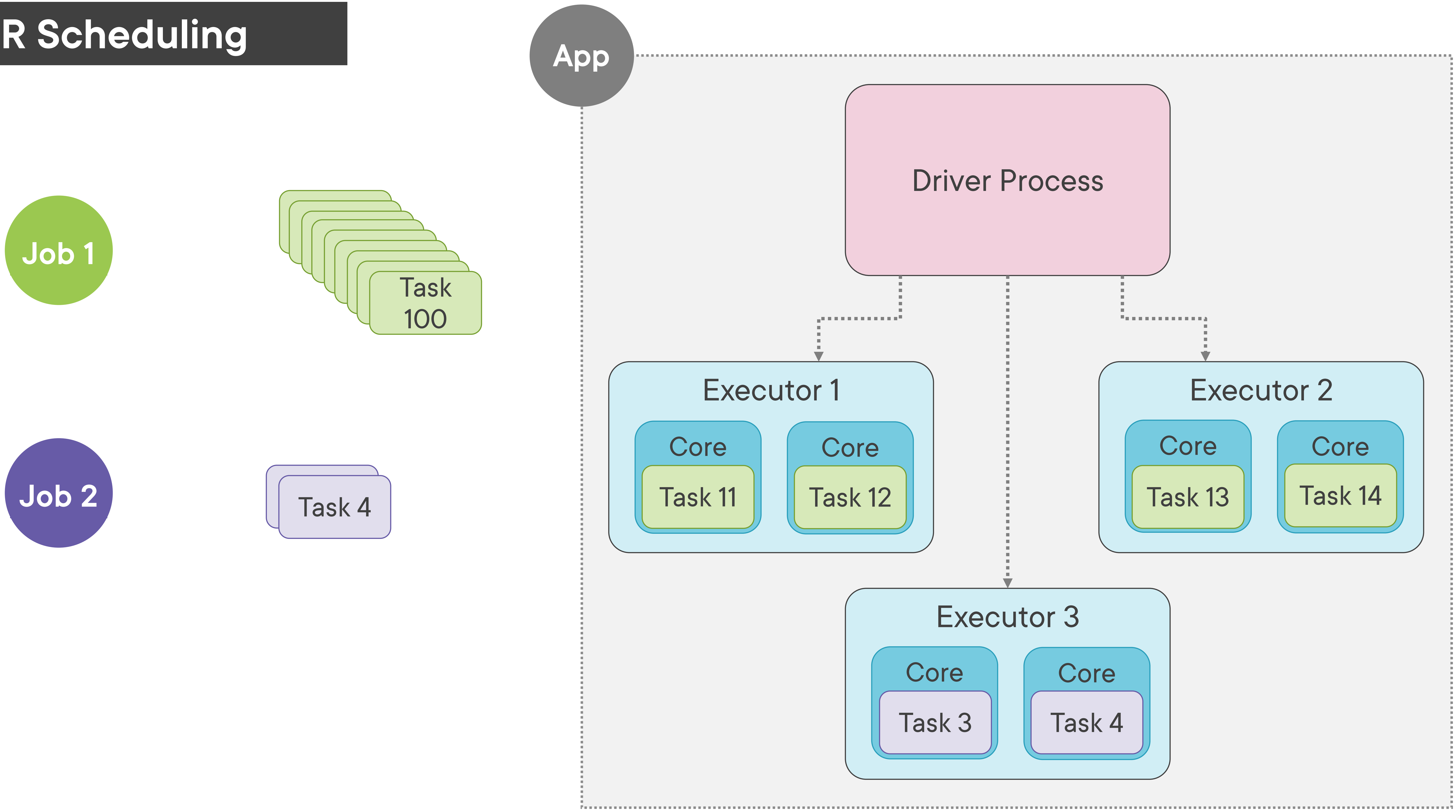
FAIR Scheduling



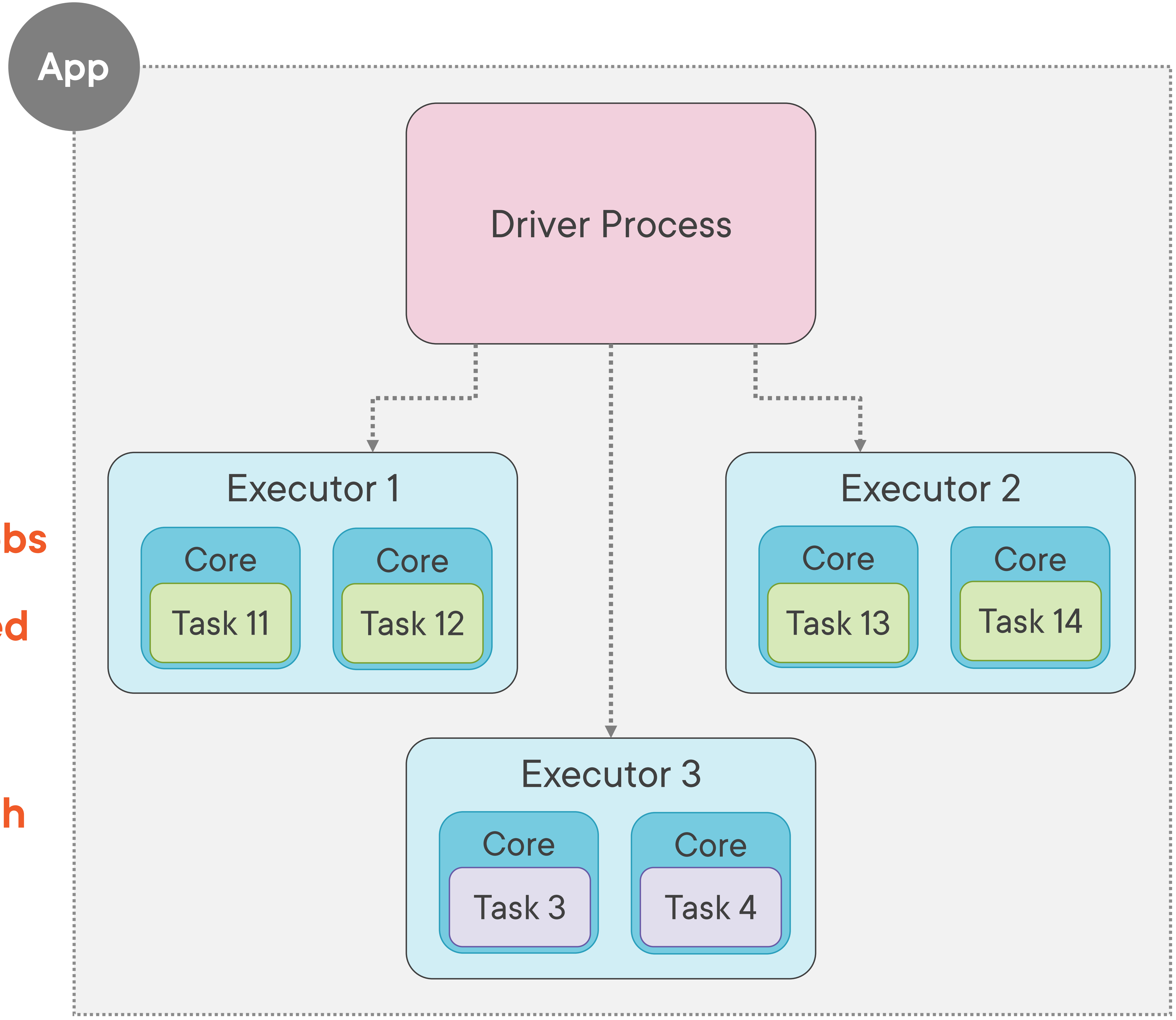
FAIR Scheduling



FAIR Scheduling



FAIR Scheduling



Fair sharing of resources between jobs

Tasks from different jobs are assigned in a round-robin fashion

Small jobs can finish faster, without waiting for long-running jobs to finish

To configure, set:

- spark.scheduler.mode = FAIR

Summary

In-memory partitions

- Settings to control partitions while reading & shuffling
- Change DF partitions using coalesce & repartition

Each driver & executor container has memory allocated

- Allocated JVM heap & non-heap memory
- Flexible memory usage b/w execution & caching

Persist data using cache() and persist()

- Avoids re-running transformations with every Action

Spark supports multiple join strategies

- Broadcast hash join: Large-small dataset join
- Shuffle sort merge join: Large-large dataset join
- Shuffle sort merge join can be improved by Bucketing

Spark supports dynamic allocation of resources

- Resource scheduling across applications
- Resource scheduling within application

Up Next:
Features in Apache Spark 3