# Apache Spark 3 Fundamentals

## Getting Started with Apache Spark

**Mohit Batra**
Founder, Crystal Talks

linkedin.com/in/mohitbatra

Understand what is Apache Spark and how it works

Setup Apache Spark environment

Work with native Spark API - RDDs

Clean & transform data with DataFrames

Work with Spark SQL, UDFs & common operations

Perform optimizations in Spark

New features in Spark 3

Handle streaming data with Structured Streaming

Work with Spark in cloud

# Overview

Need for Apache Spark

Spark architecture and ecosystem

How execution happens in Spark?

Spark supported APIs

# Version Check

# Version Check

**This version was created by using:**

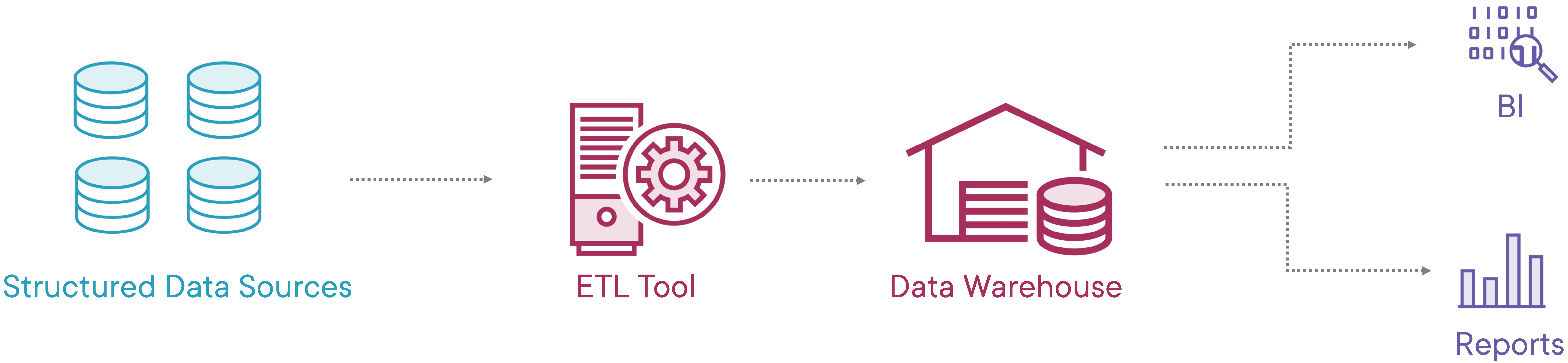
- Apache Spark 3.3.1
- Apache Hadoop 3.3
- Java v11
- Scala 2.13
- Python 3.7

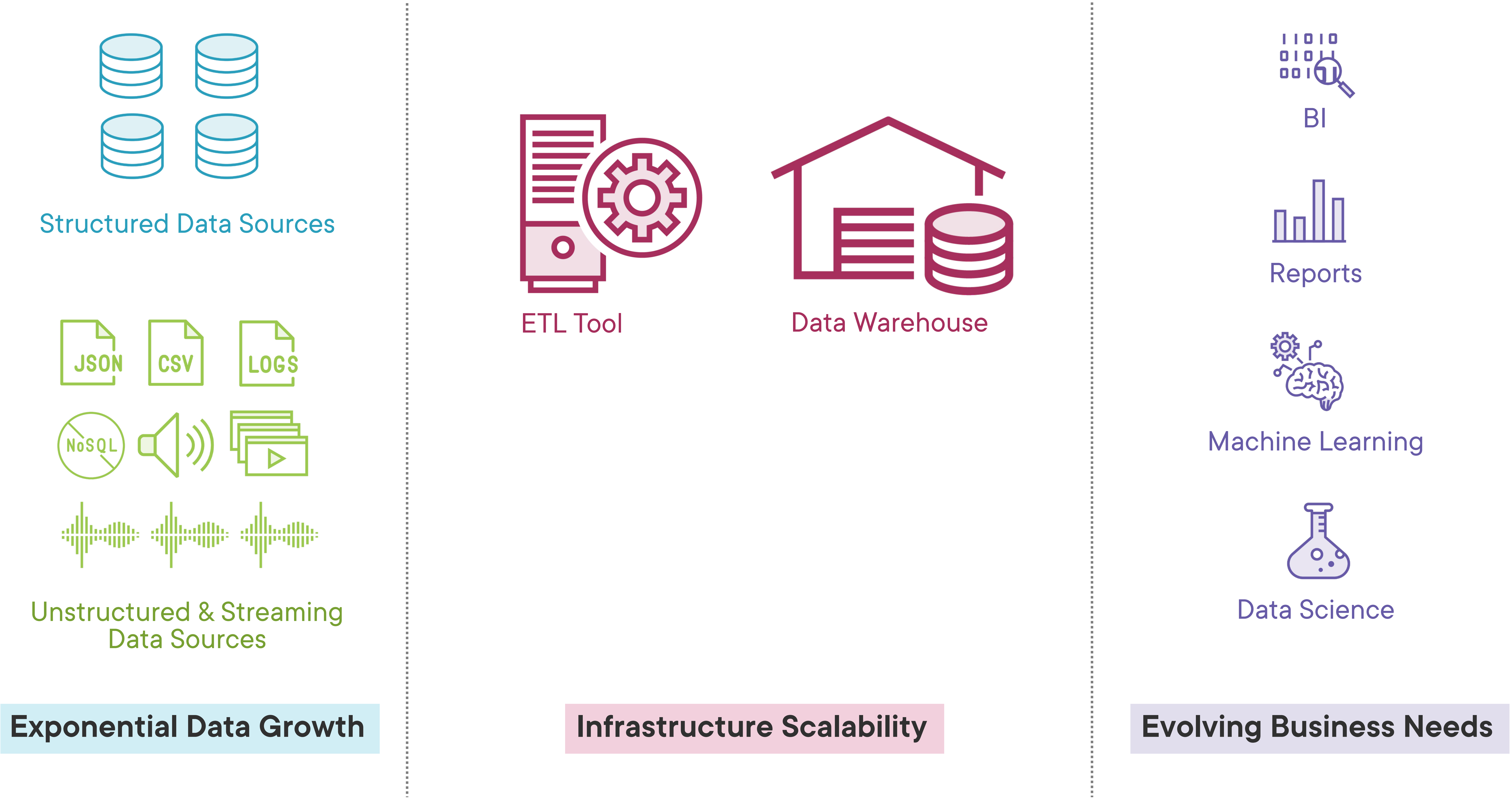
# Version Check

**This course is 100% applicable to:**

- Apache Spark 3.0 onwards
- Java v8 (8u92+), v11 and v17
- Scala 2.12 onwards
- Python 3.7 onwards

# Need for Apache Spark


Structured Data Sources
ETL Tool
Data Warehouse
BI
Reports



Structured Data Sources
JSON
CSV
LOGS
NoSQL
Unstructured & Streaming Data Sources
Exponential Data Growth
ETL Tool
Data Warehouse
Infrastructure Scalability
BI
Reports
Machine Learning
Data Science
Evolving Business Needs

# History

**Earlier**

ETL Tools / RDBMS

**2004**

**MapReduce / Distributed File System**

Paper published by Google for distributed data processing

# History

**Earlier**

ETL Tools / RDBMS

**2004**

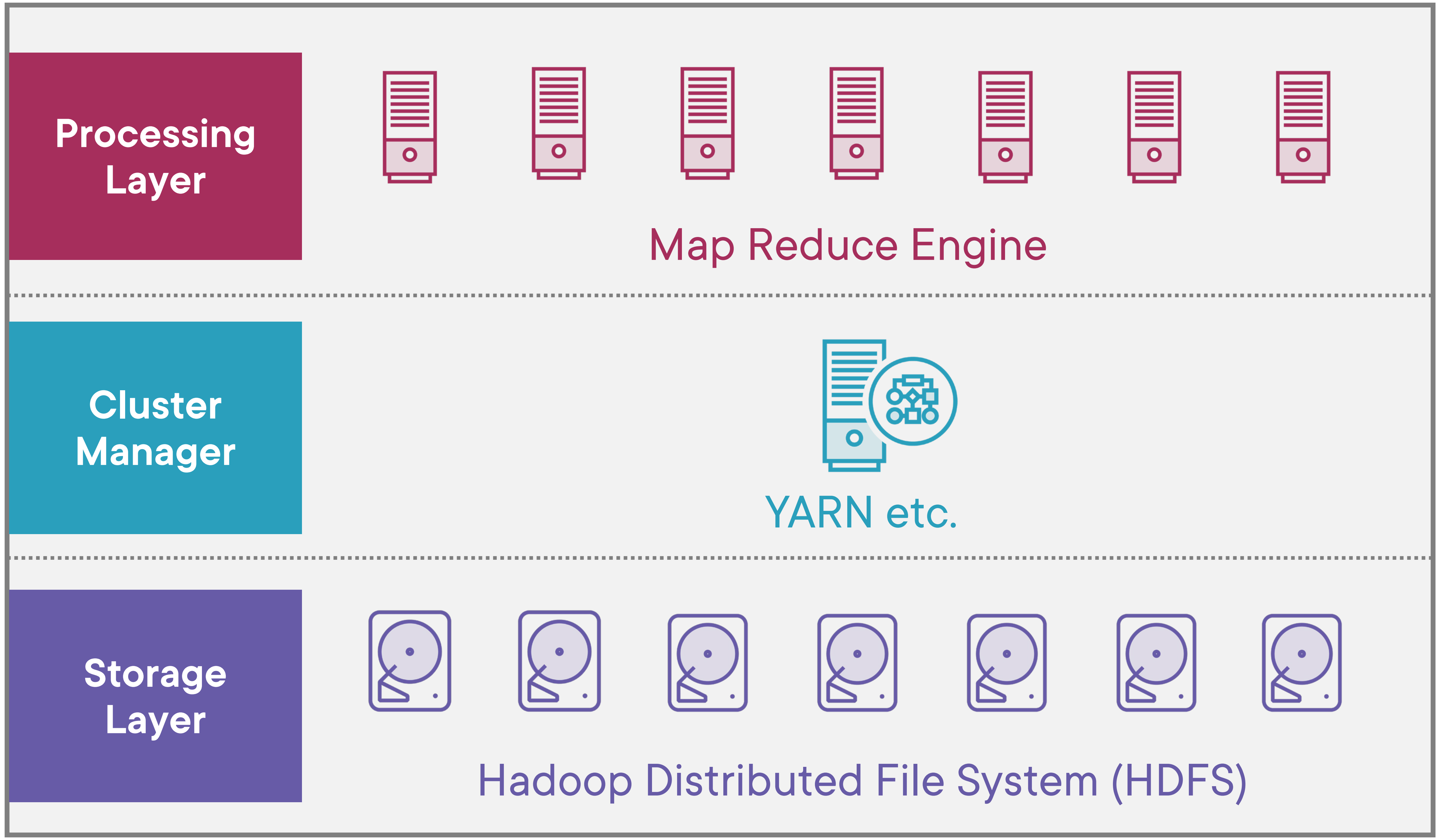
**MapReduce / Distributed File System**

**2006**

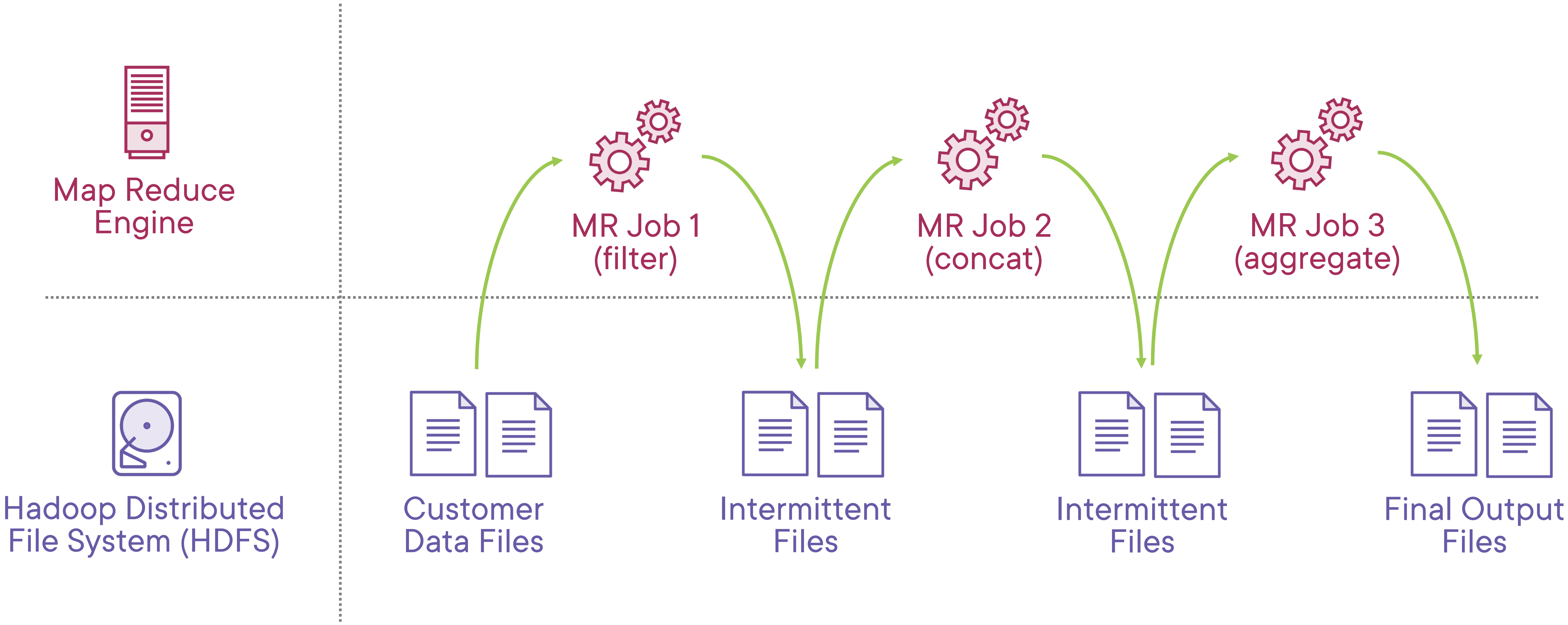
Hadoop (MapReduce & HDFS)

Framework for distributed storage & distributed data processing

# Hadoop Components

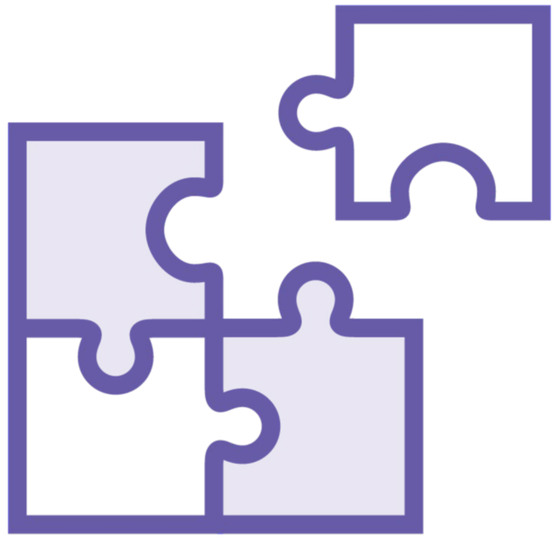


# How Hadoop Works?




**Disk-heavy Architecture**

# Challenges with Hadoop



**Infrastructure & environment complexity**

**Need to write a lot of code**

**Disk-heavy architecture: Multiple IO operations slows down processing**

**No built-in support for streaming, ML etc.**

- New tools were built to handle them adding to complexity

# History

**Earlier**
ETL Tools / RDBMS

**2004**
**MapReduce / Distributed File System**

**2006**
Hadoop (MapReduce & HDFS)

**2009**
**Apache Spark**

In-memory engine for distributed data processing

# Apache Spark

Extremely powerful analytics engine for large-scale distributed data processing, whether structured or unstructured

In-memory engine can run workloads up to 100x faster than Hadoop

Simplified code and multiple language support

Unifies variety of use cases – batch processing, stream processing, machine learning & advanced analytics

# History

**Earlier**
ETL Tools / RDBMS

**2004**
**MapReduce / Distributed File System**

**2006**
Hadoop (MapReduce & HDFS)

**2009**
**Apache Spark**

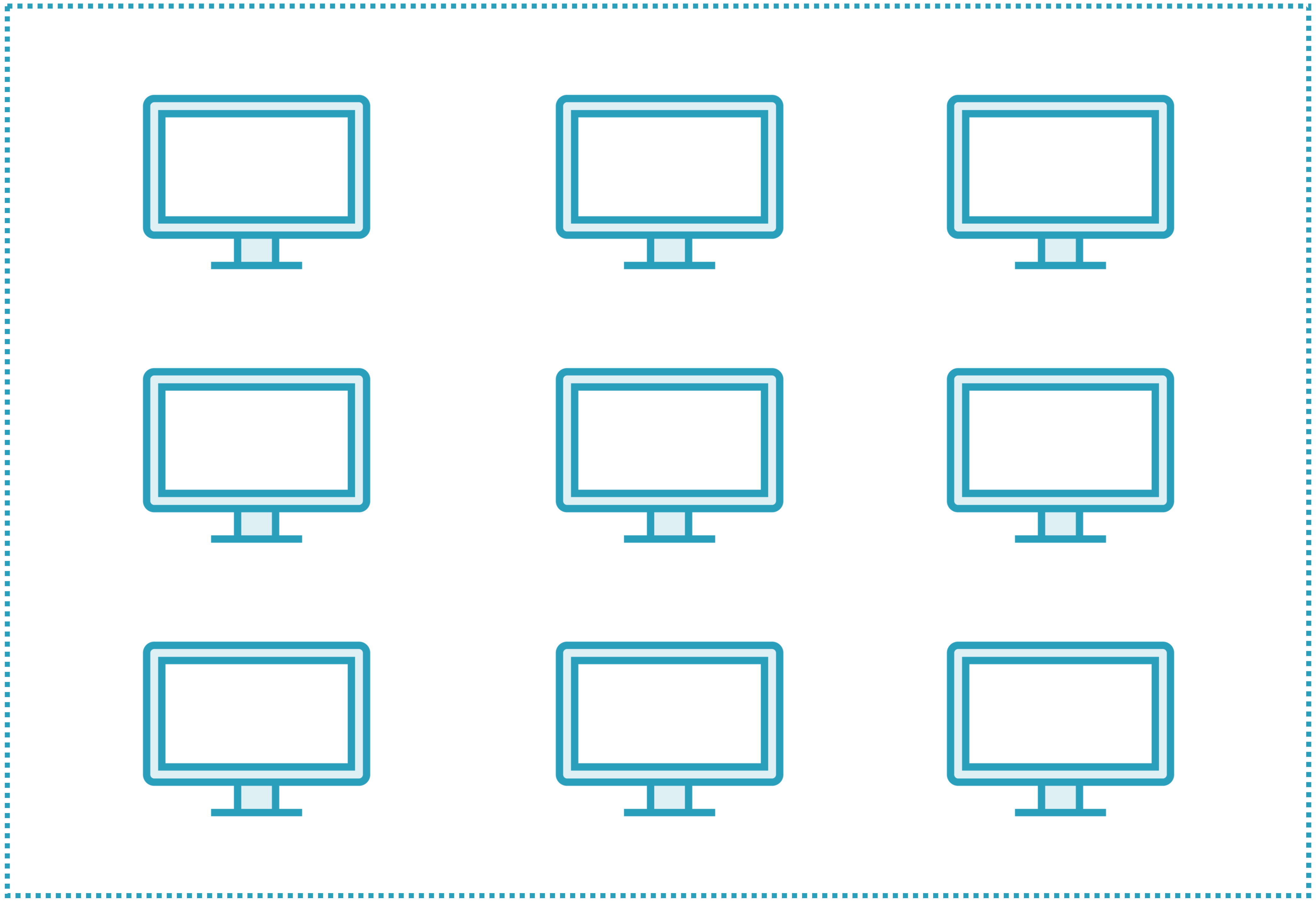
**Now**
Cloud-based Spark platforms

Databricks
Azure Synapse Analytics
Amazon EMR etc.

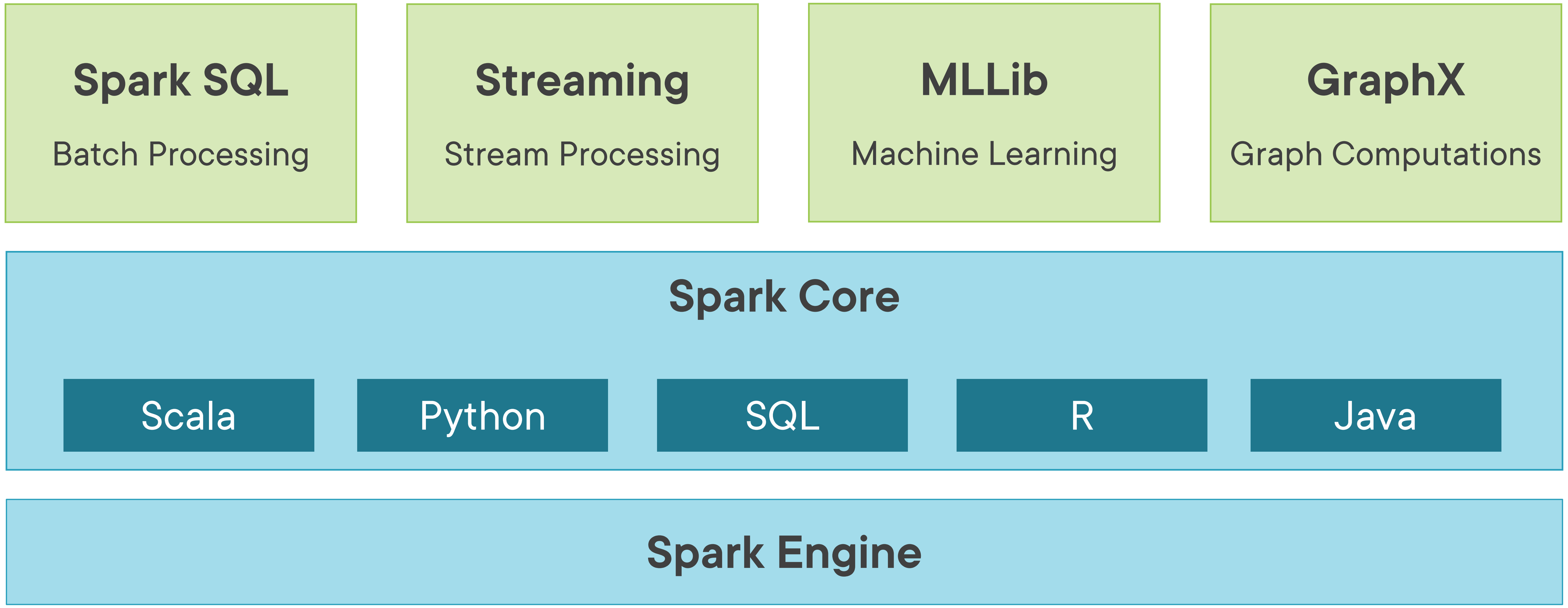
# Understanding Spark Architecture & Ecosystem

Apache Spark is an extremely powerful, in-memory analytics engine built on cluster computing technology

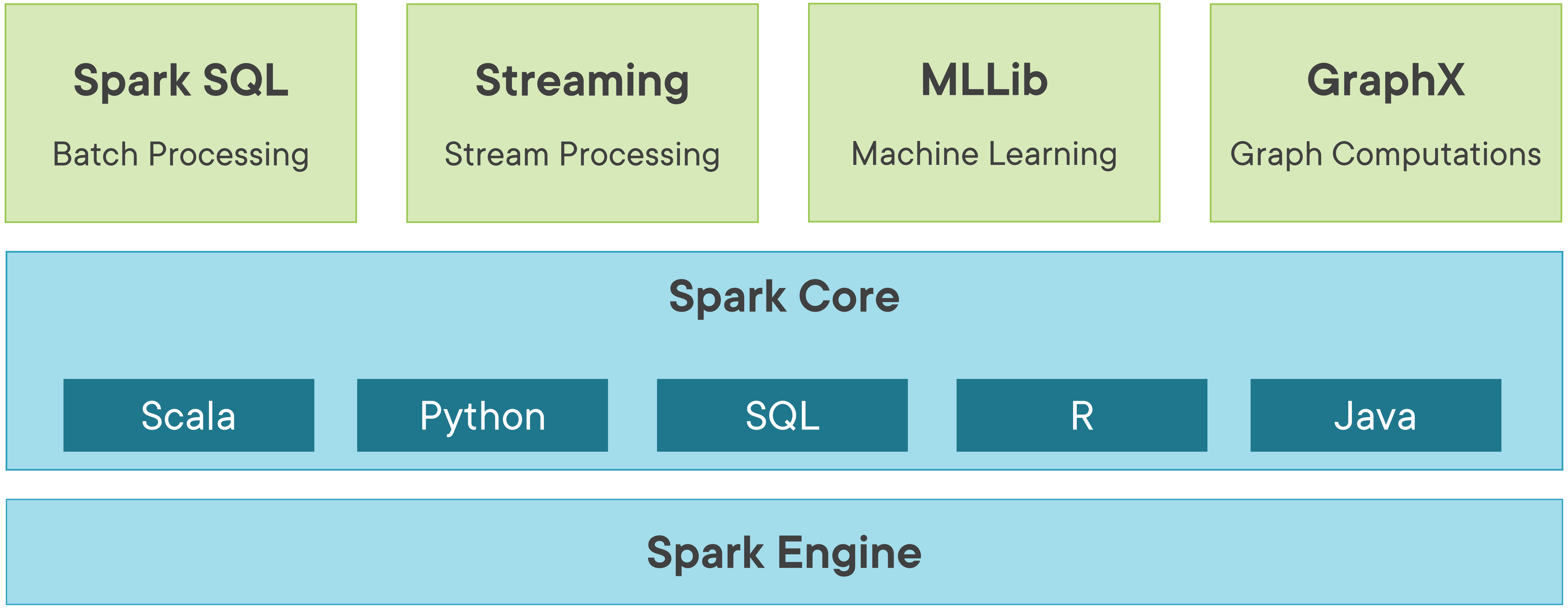
# Spark Cluster



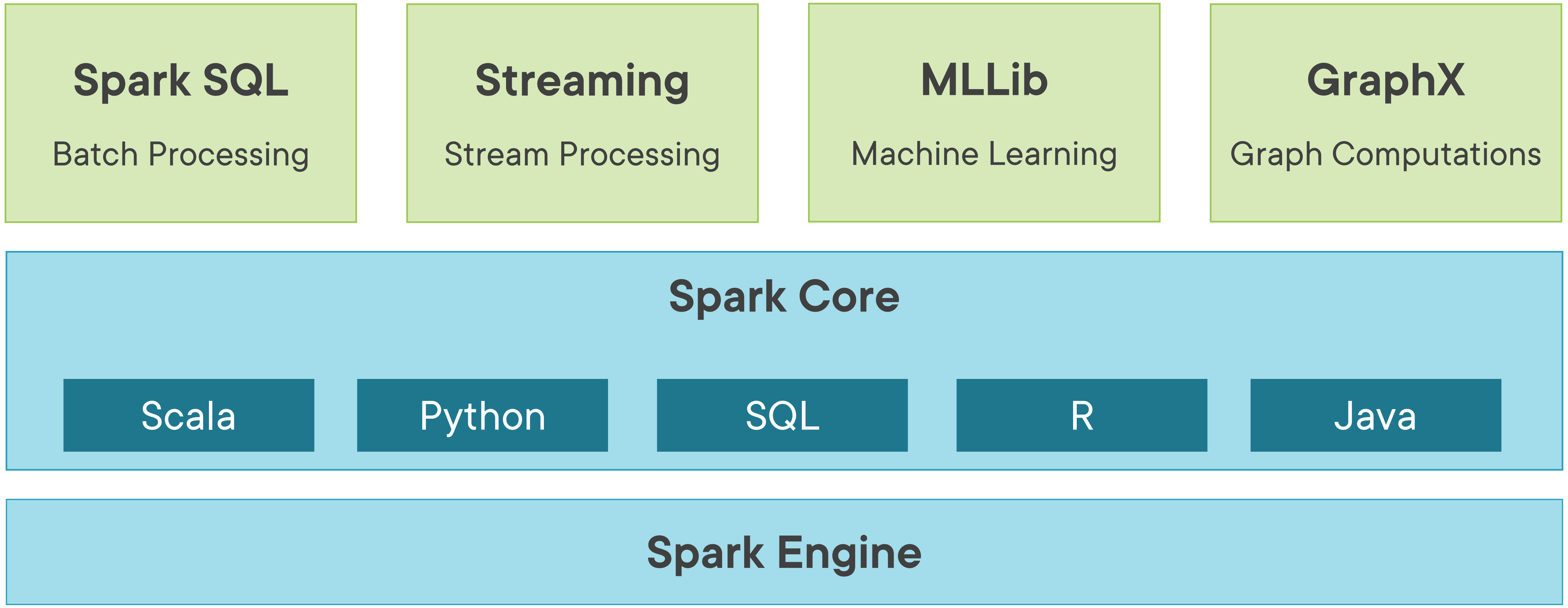
# Spark Architecture



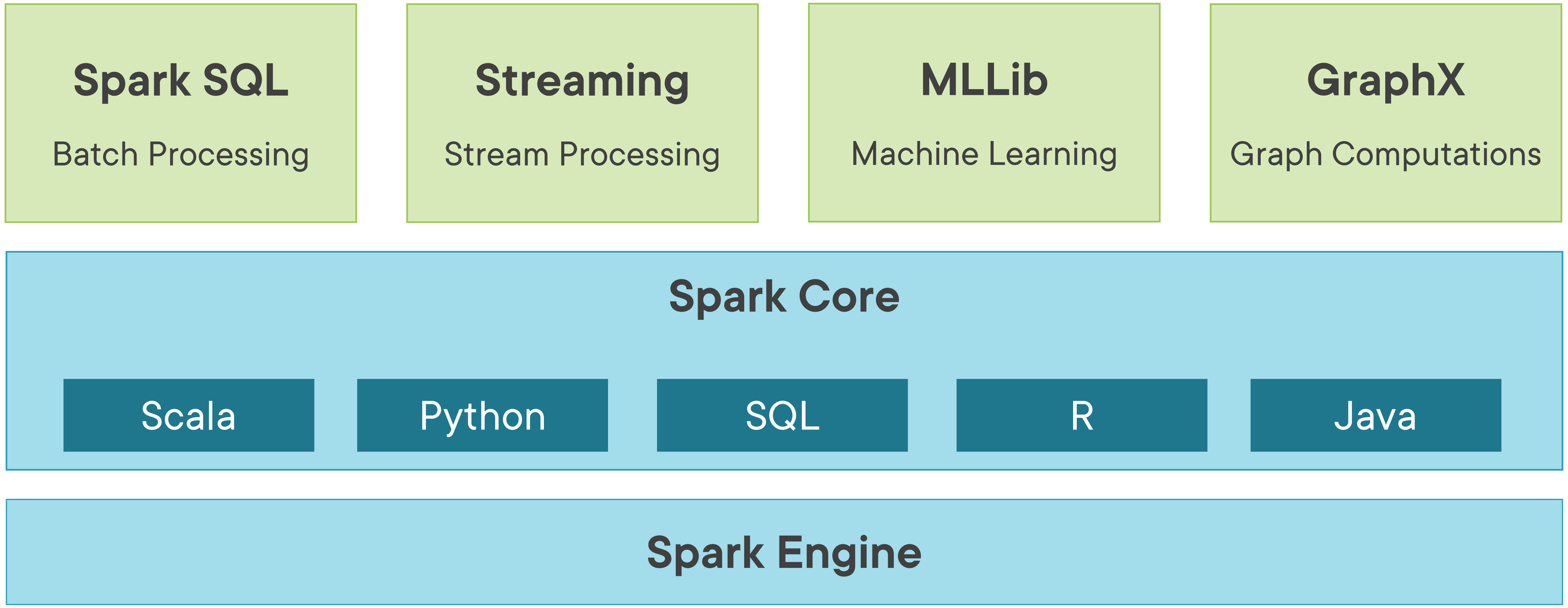
# Spark Architecture



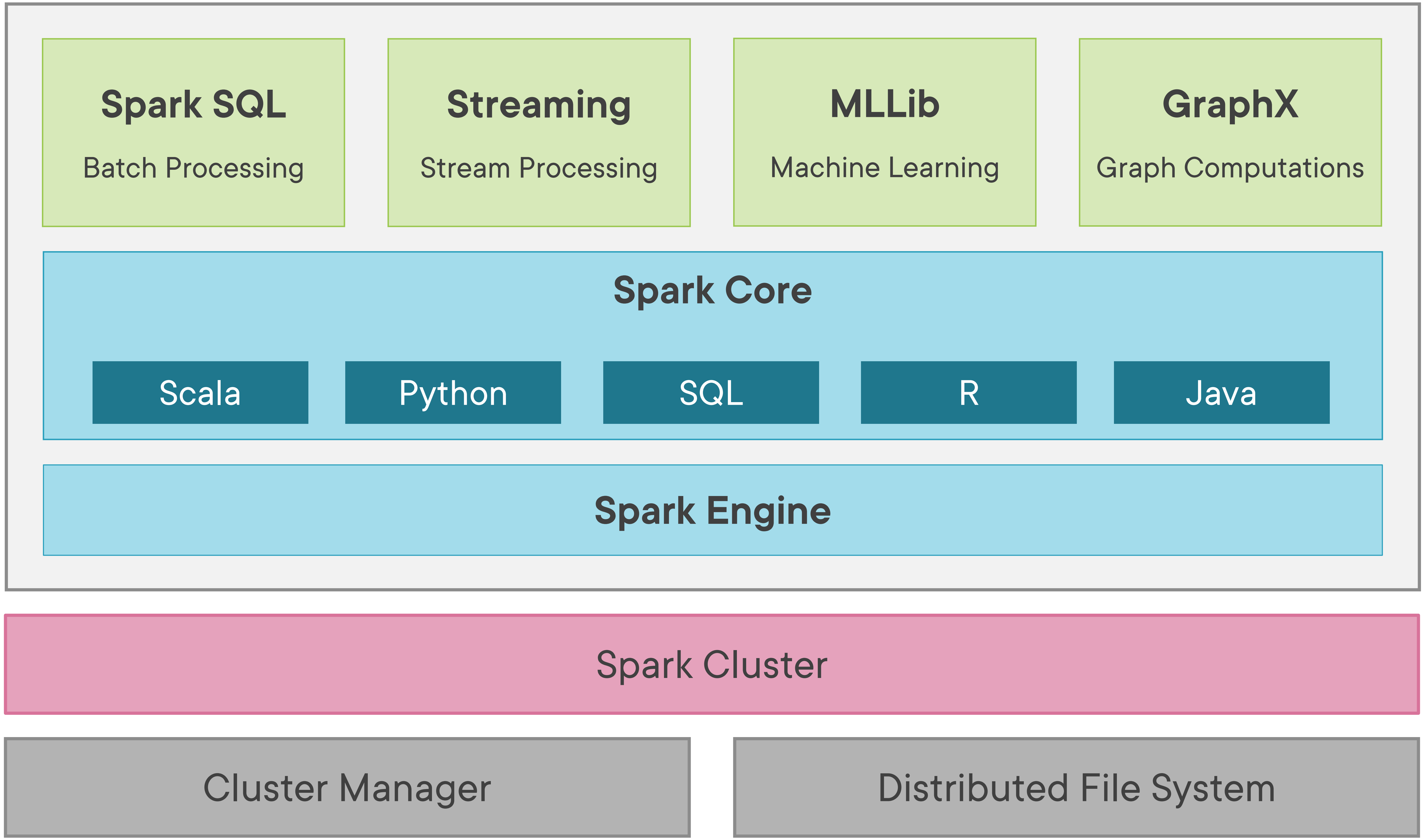
# Spark Architecture



# Spark Architecture



# Spark Architecture



# Spark Ecosystem

**Cluster Managers**

- YARN, Kubernetes, Mesos etc.

**Distributed File System options**

- HDFS, Azure Data Lake Store, Amazon S3 or Google Cloud Storage

**Multiple language support**

- Scala, Python, SQL, R & Java
- Open-source support for C#

**Development options**

- Console, IDEs (PyCharm, VS Code), Notebooks (Jupyter, Zeppelin) etc.

**Available in Cloud Platforms**

- Databricks, Cloudera, Azure Synapse Analytics, Azure HDInsight, Amazon EMR etc.

# Open-source Connectors

**Relational databases – SQL Server, Oracle**

**NoSQL – MongoDB, Cassandra, Azure Cosmos DB**

**Apache Hadoop, HBase, Hive**

**Cloud storage**

**MPP engines – AWS Redshift, Azure Dedicated SQL**

**Visualization tools – Power BI, Tableau**

**Streaming – Kafka, Azure Event Hubs, AWS Kinesis**

***...and much more***

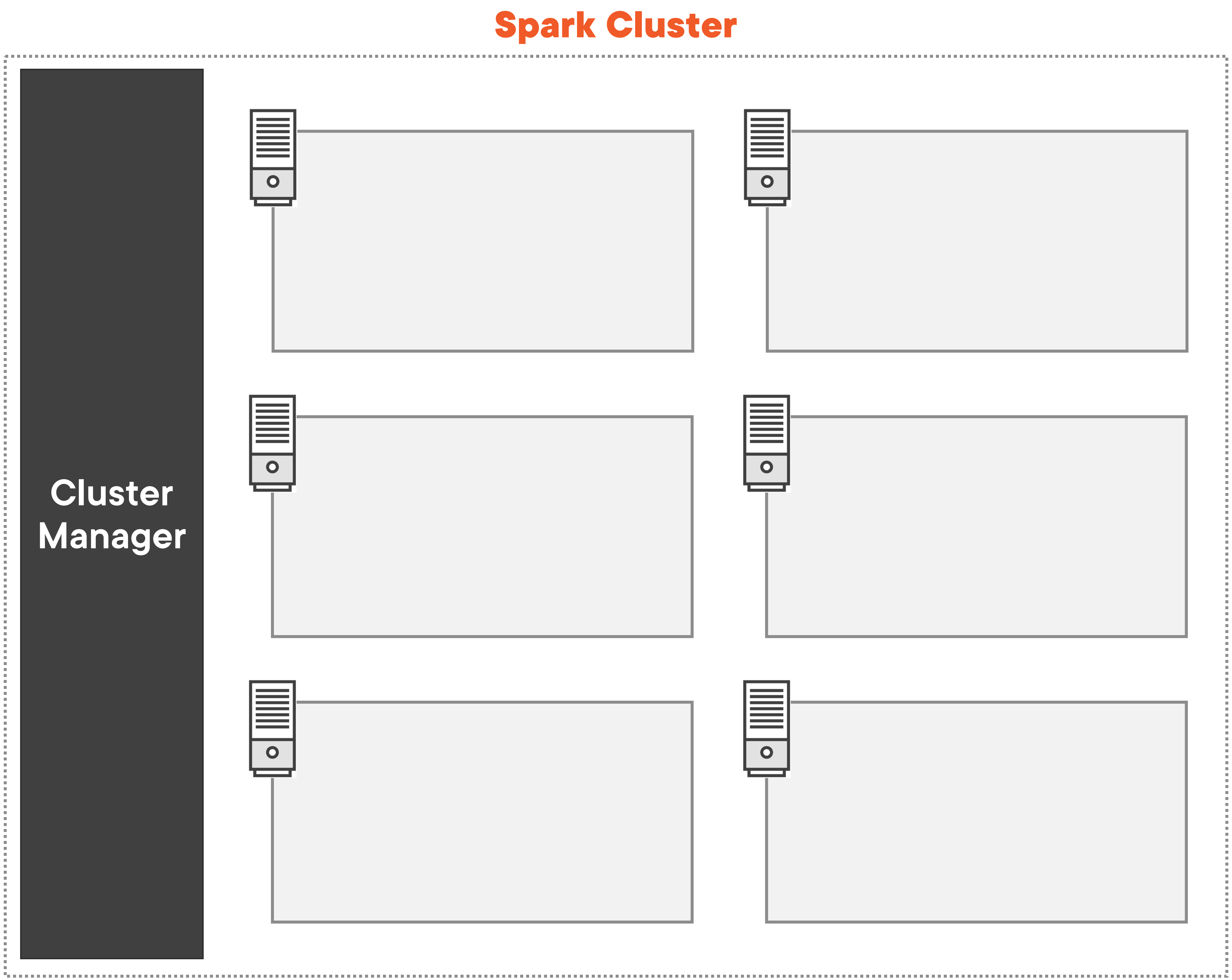
# Check out more open-source projects...

https://spark.apache.org/third-party-projects.html

# How Execution Happens in Spark?

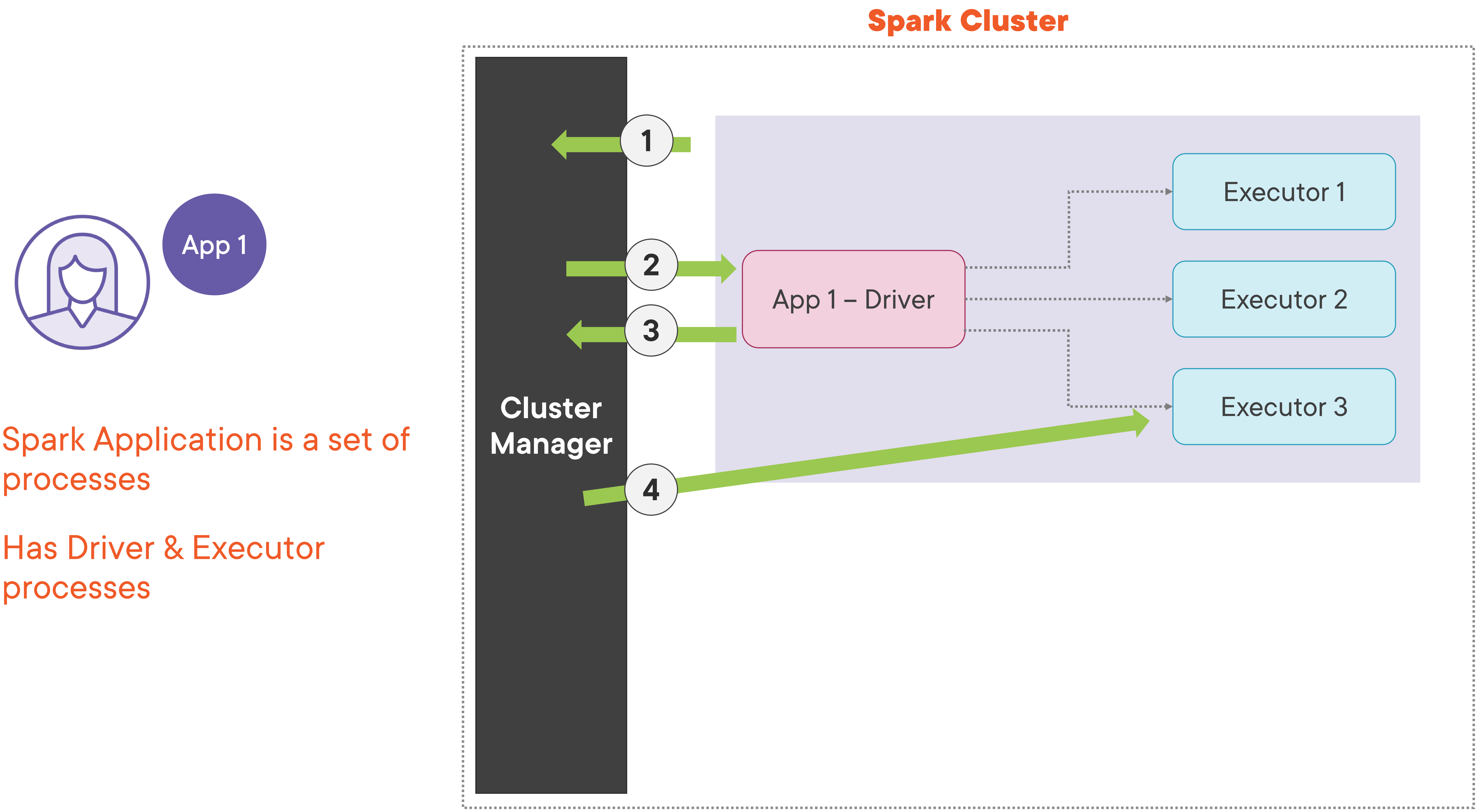
Cluster is a group of machines / nodes

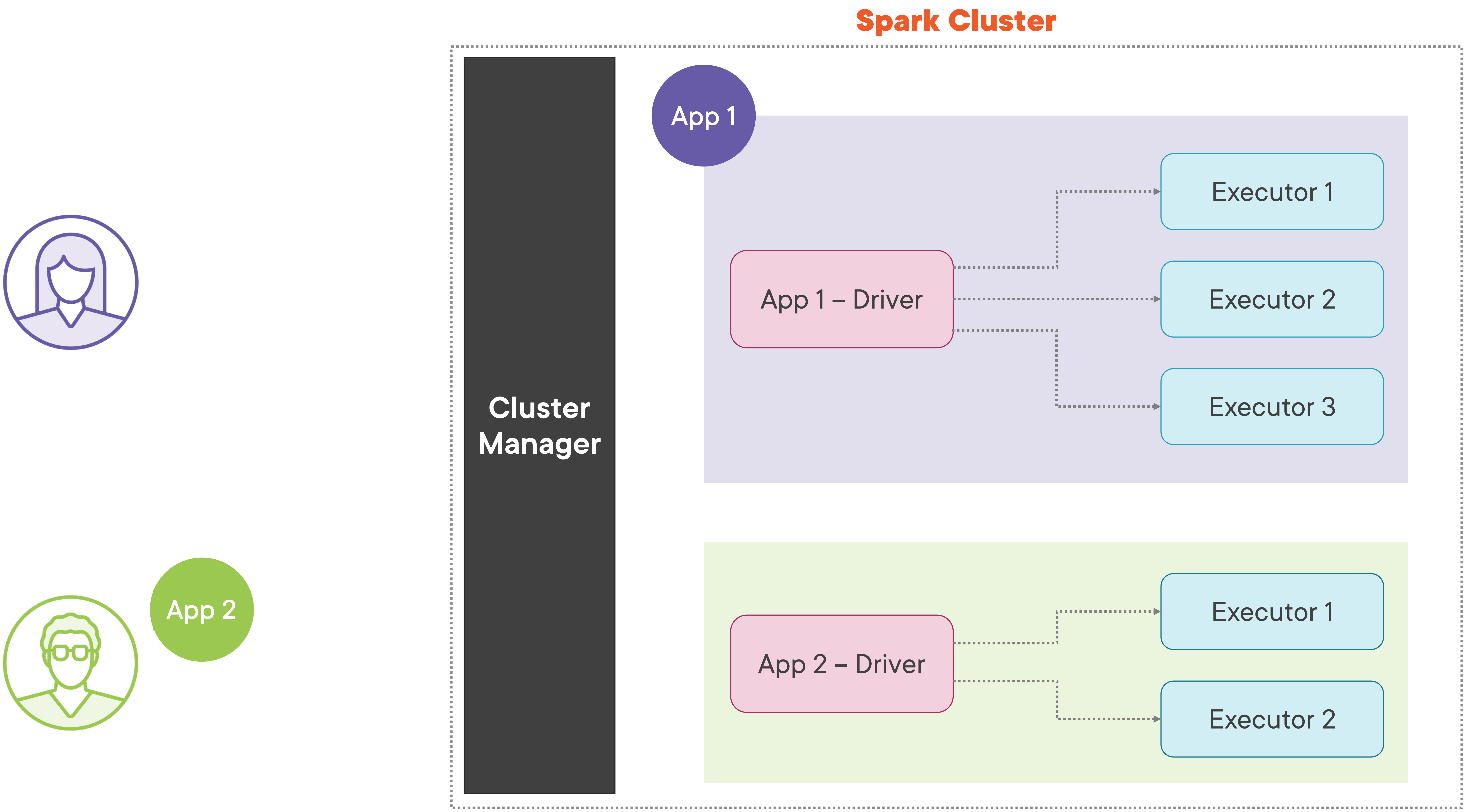
Cluster Manager allocates resources to Spark Applications on Cluster

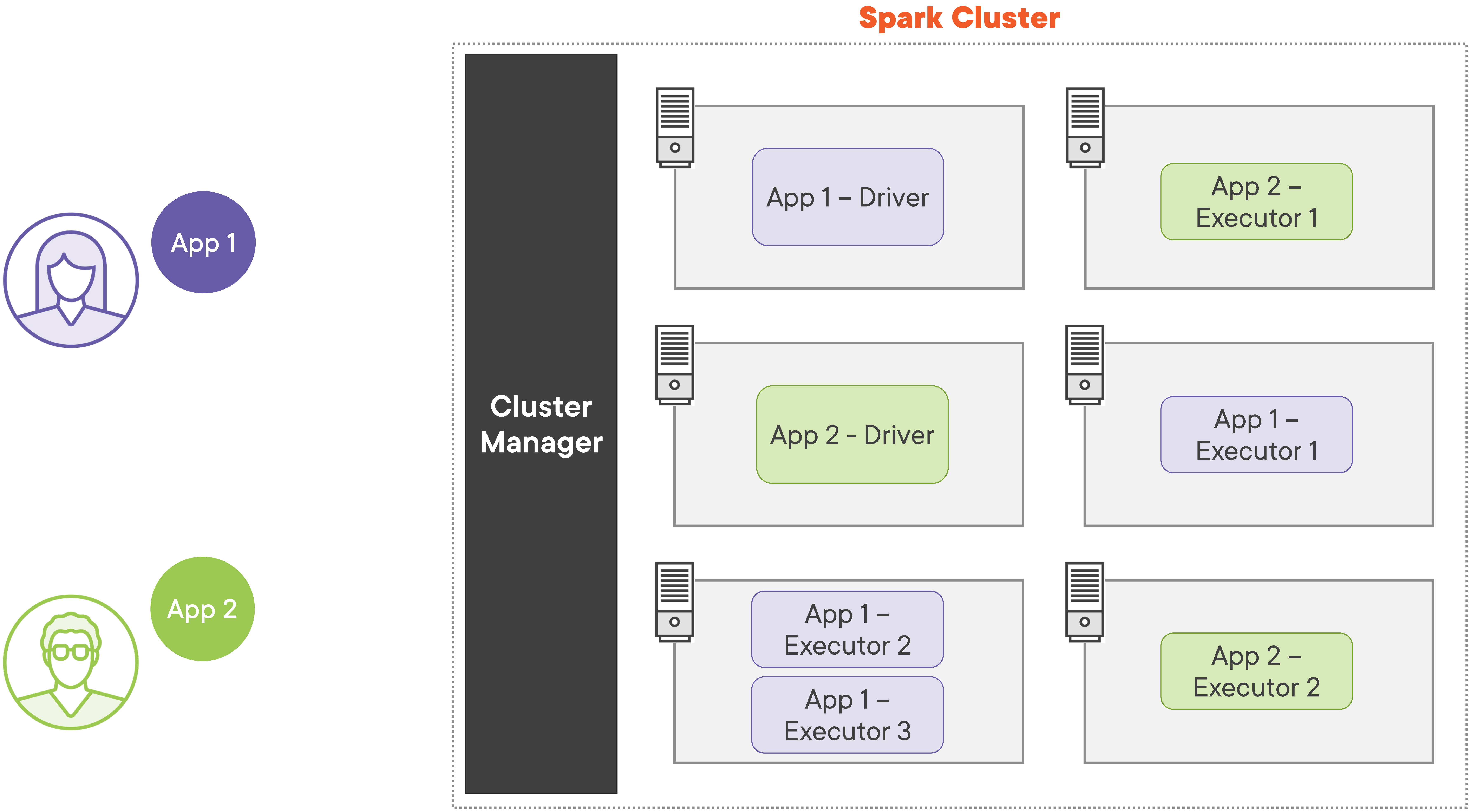


Spark Application is a set of processes

Has Driver & Executor processes



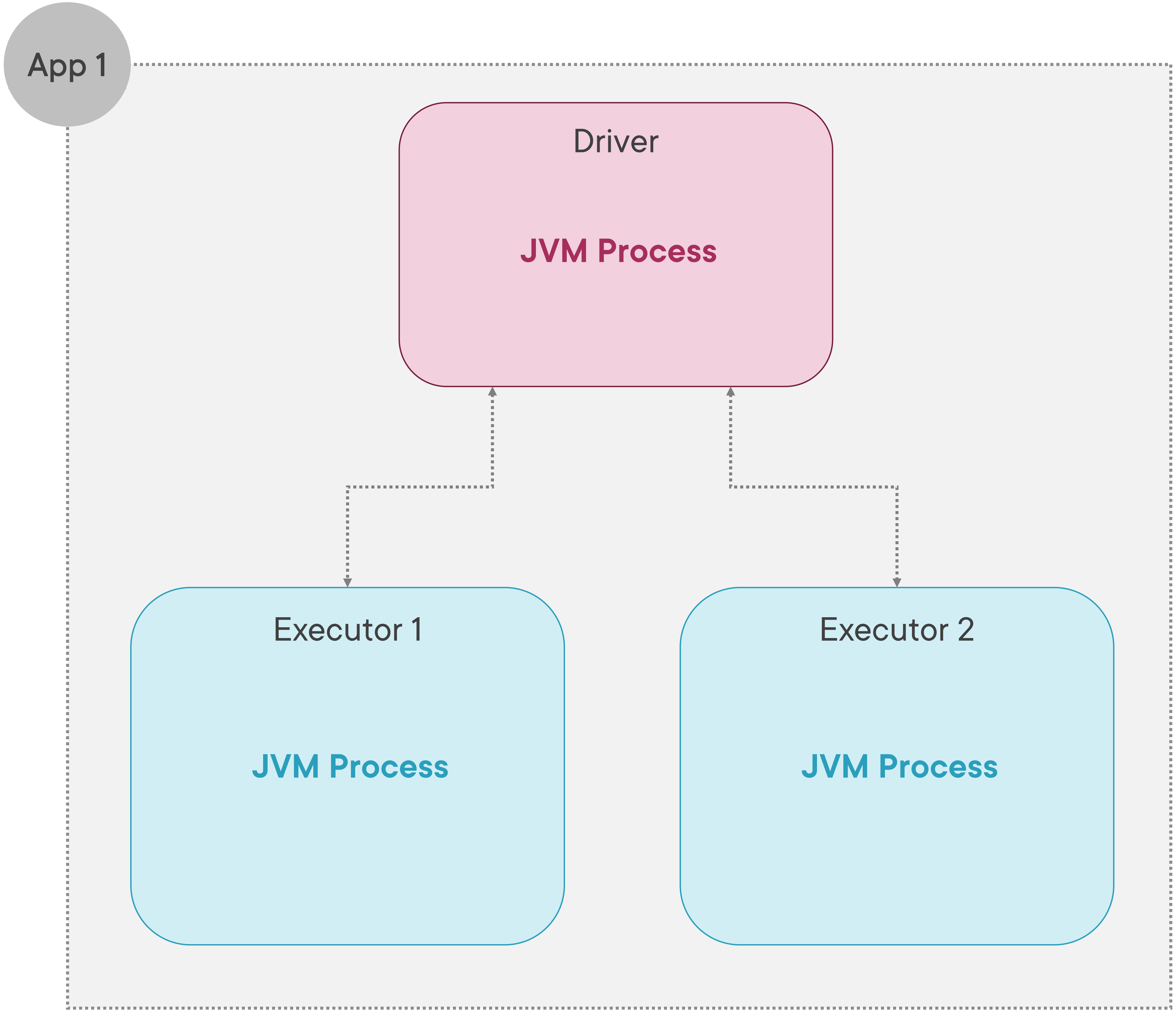

Spark Cluster
Cluster Manager
App 1
App 1 – Driver
Executor 1
Executor 2
Executor 3
App 2
App 2 – Driver
Executor 1
Executor 2


Spark Cluster
App 1
App 2
Cluster Manager
App 1 – Driver
App 2 – Executor 1
App 2 - Driver
App 1 – Executor 1
App 1 – Executor 2
App 1 – Executor 3
App 2 – Executor 2

Driver takes input from user

Determines how to execute a job

Analyzes job & distributes work to executors



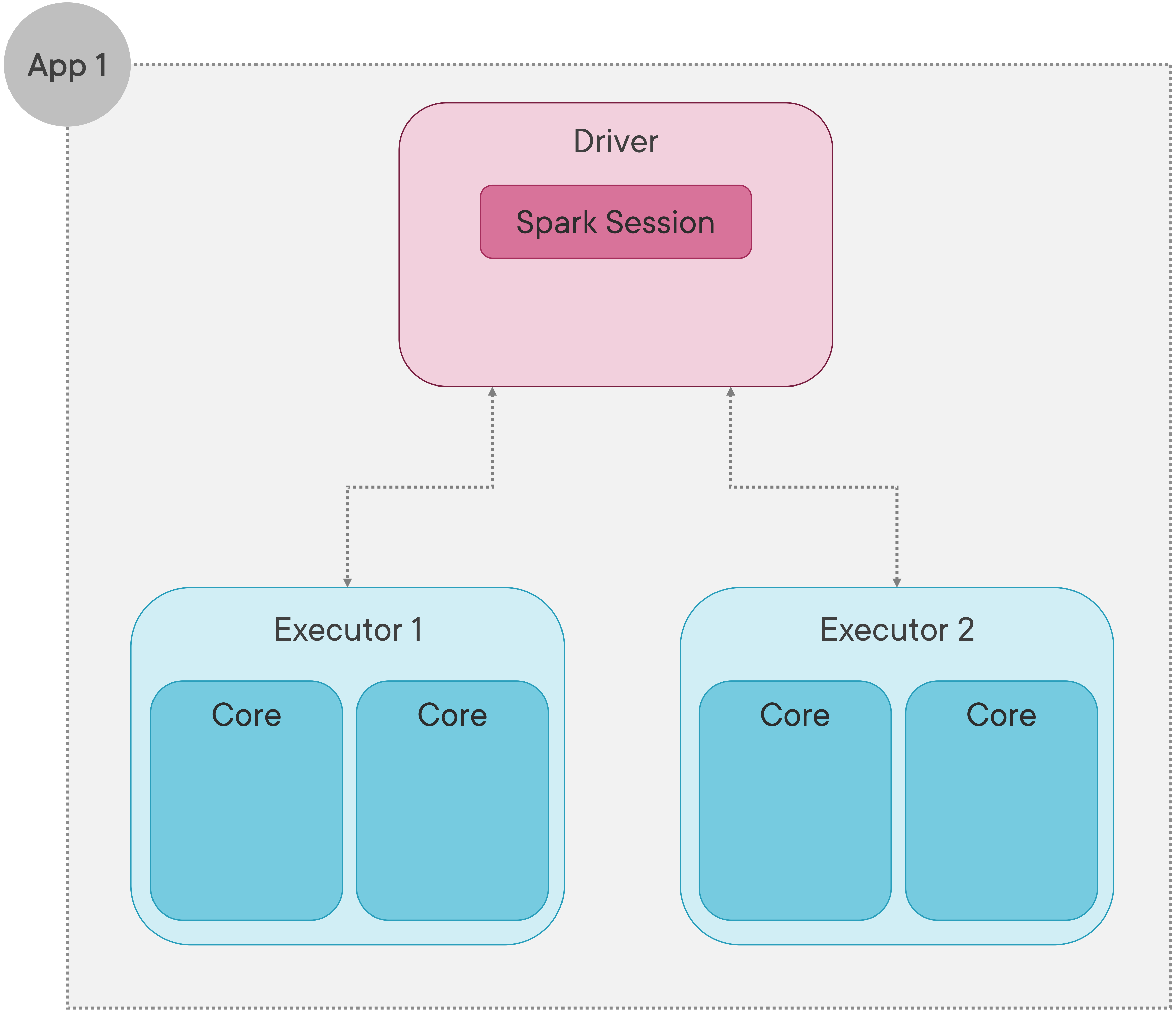

Executors are responsible for executing the work (or code)

Returns result back to Driver

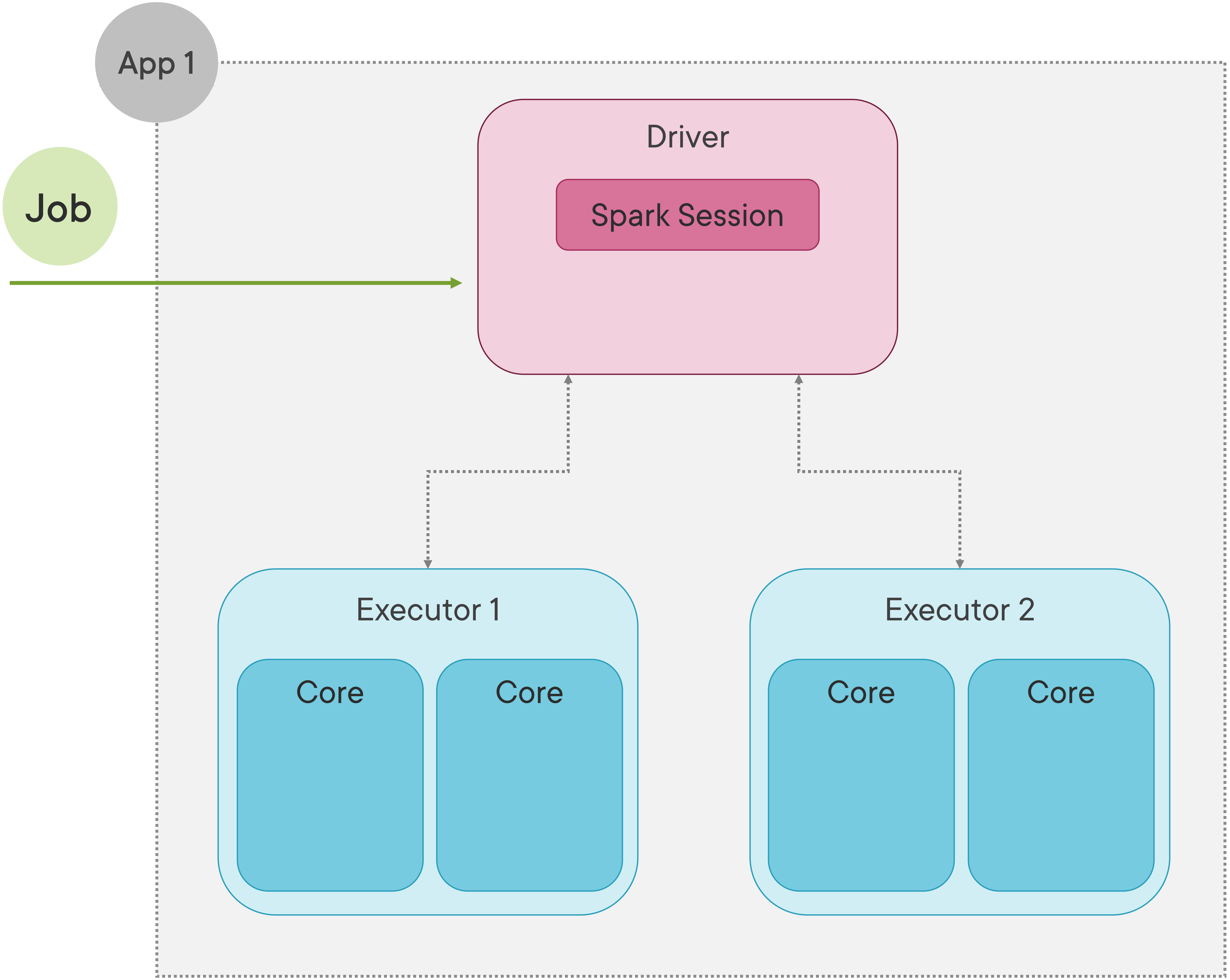
Spark Session is the entry point to all functionality of Spark

Use Spark Session to read file, create objects, run queries etc.

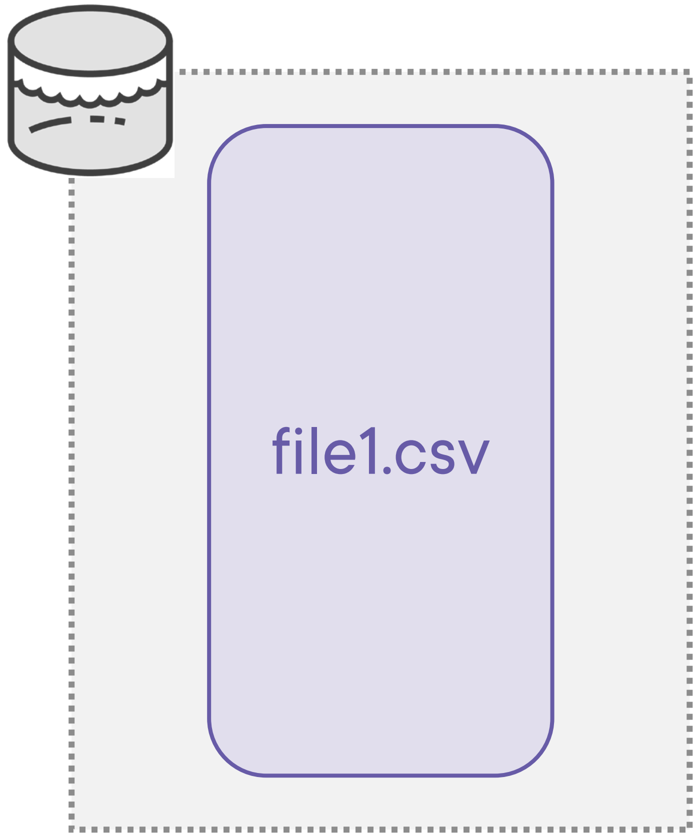
Executor Size
2 Cores & 14 GB RAM
*(example)*



1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

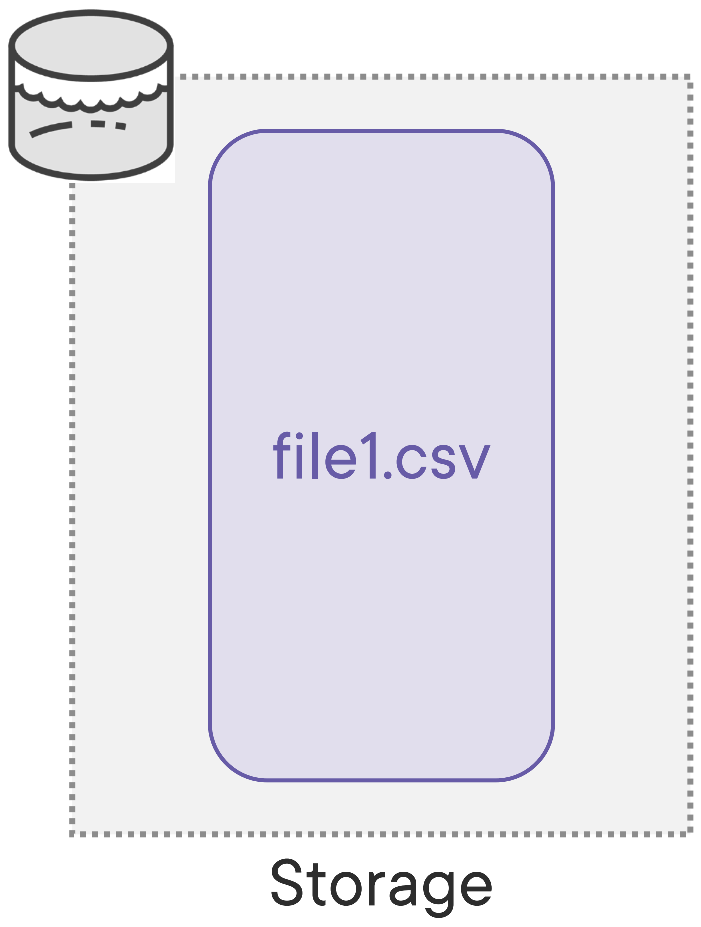


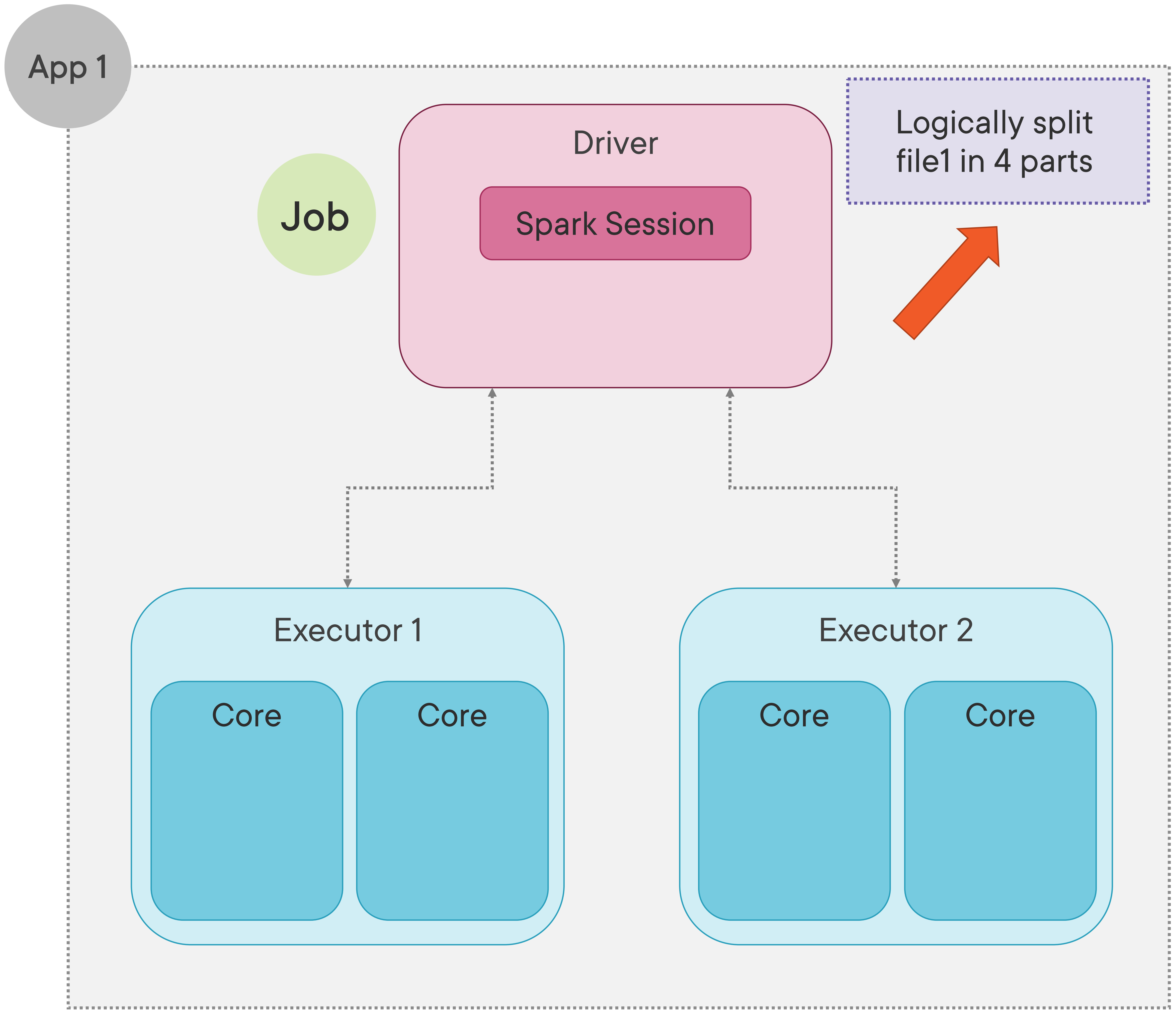

## How Execution Happens?



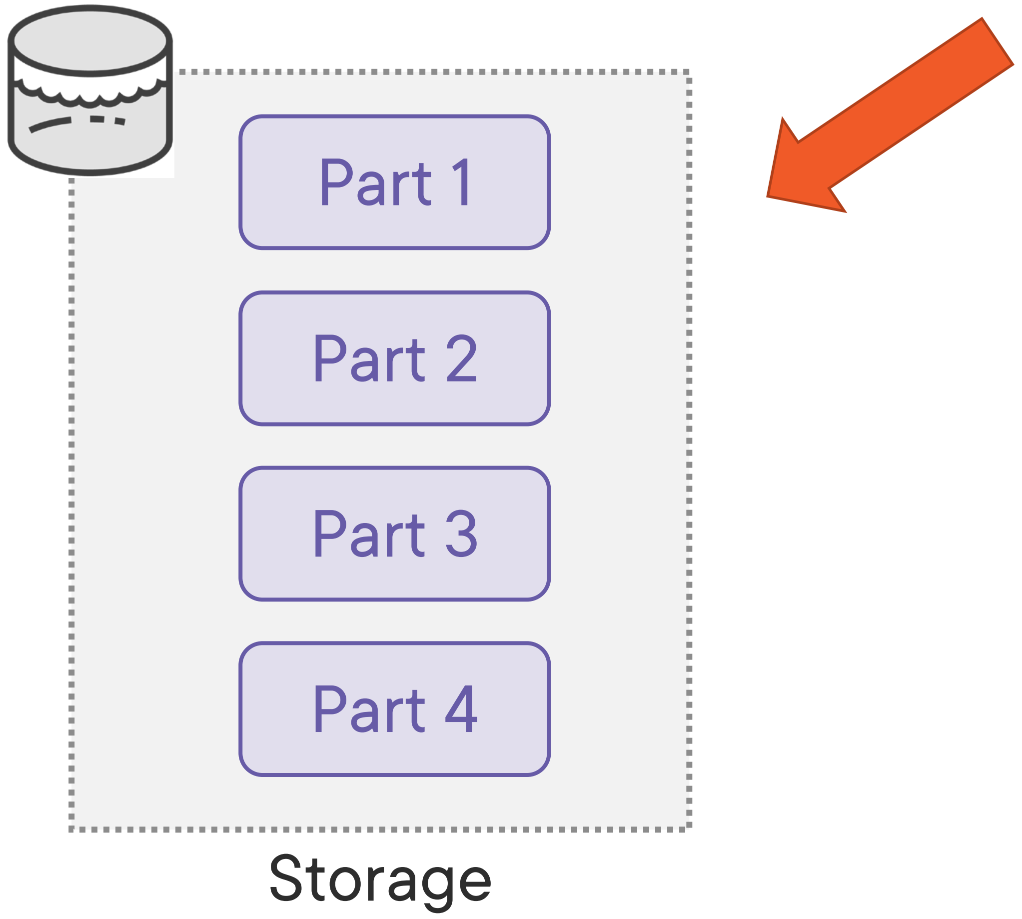

Storage

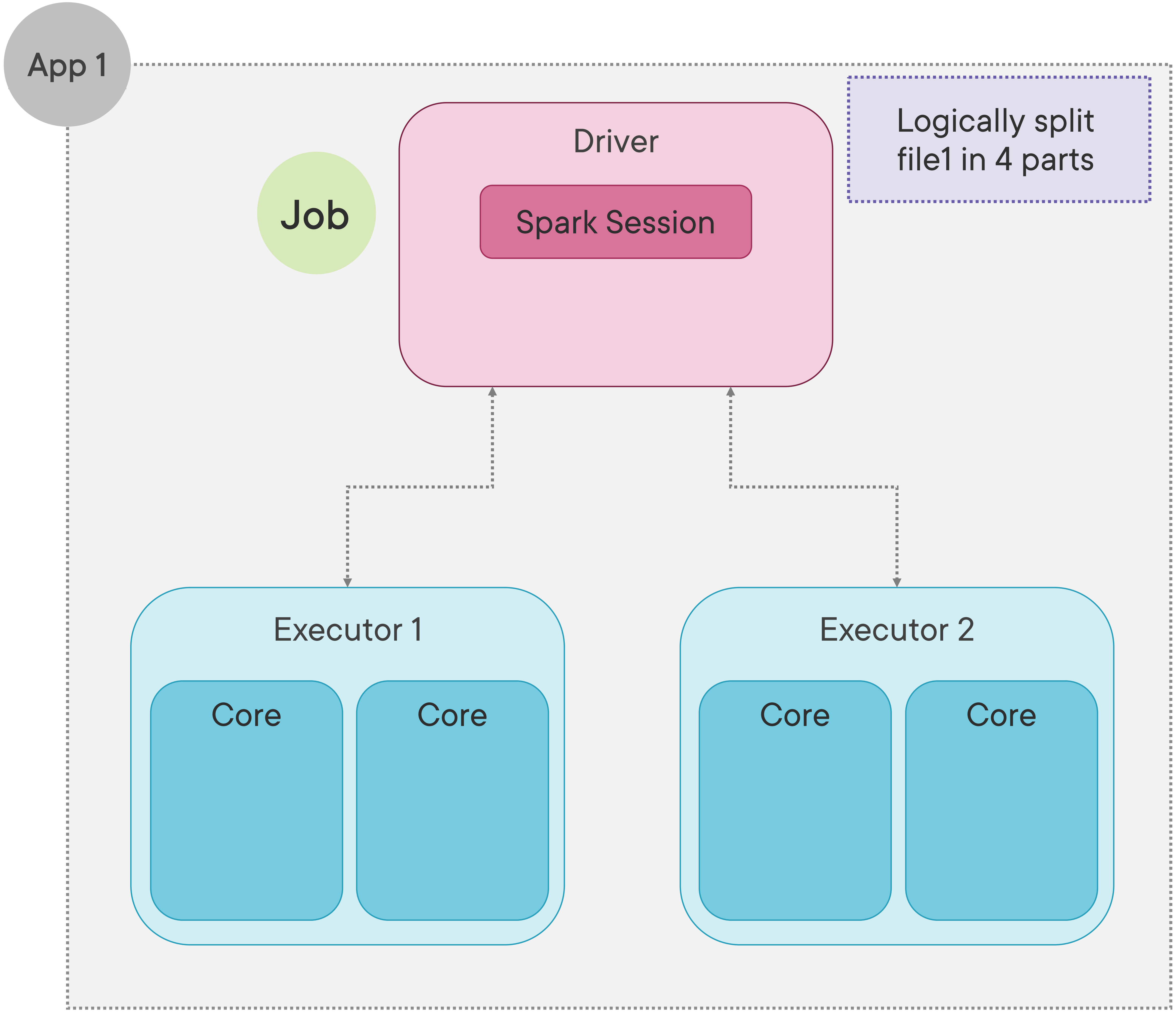
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

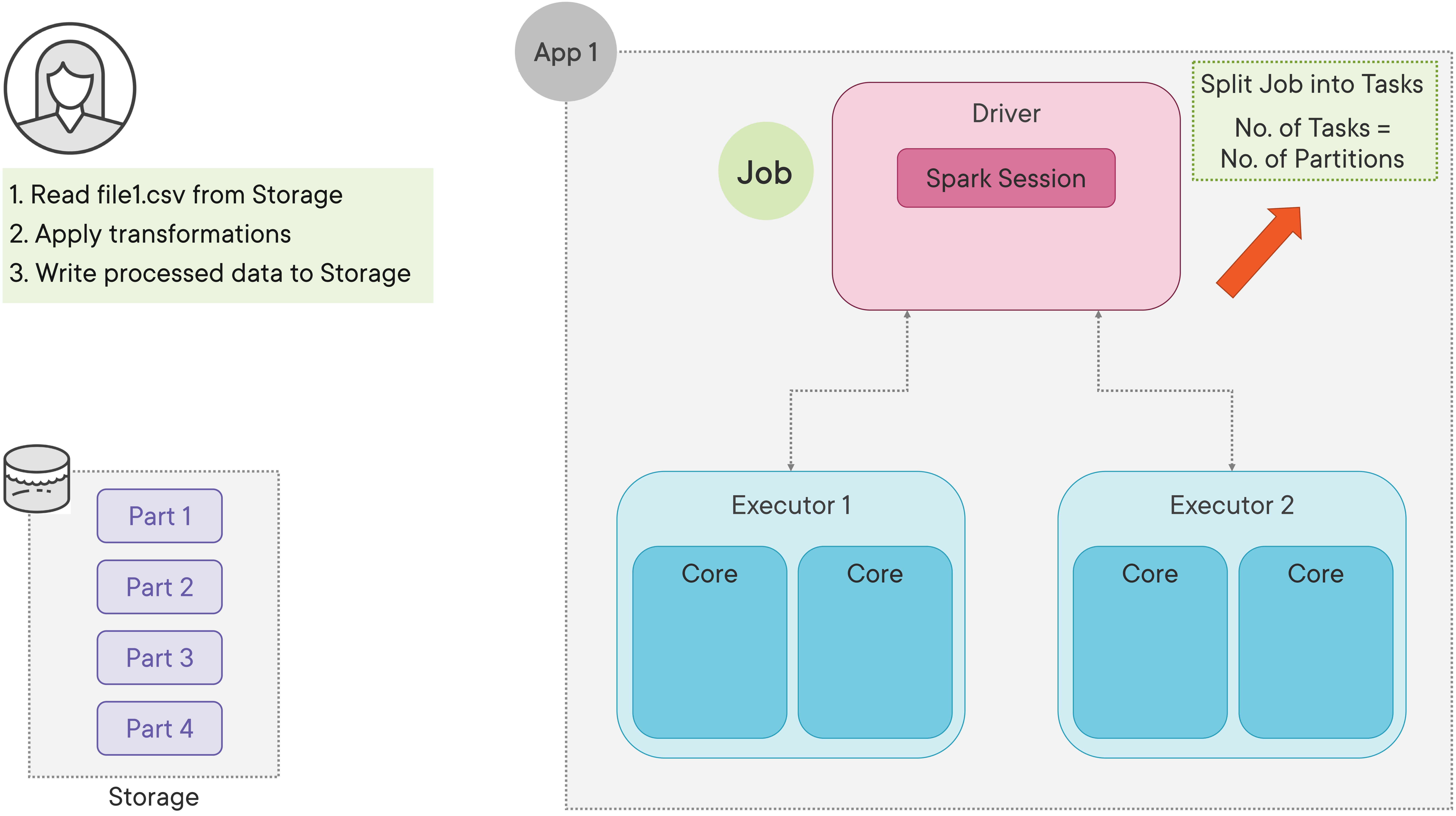


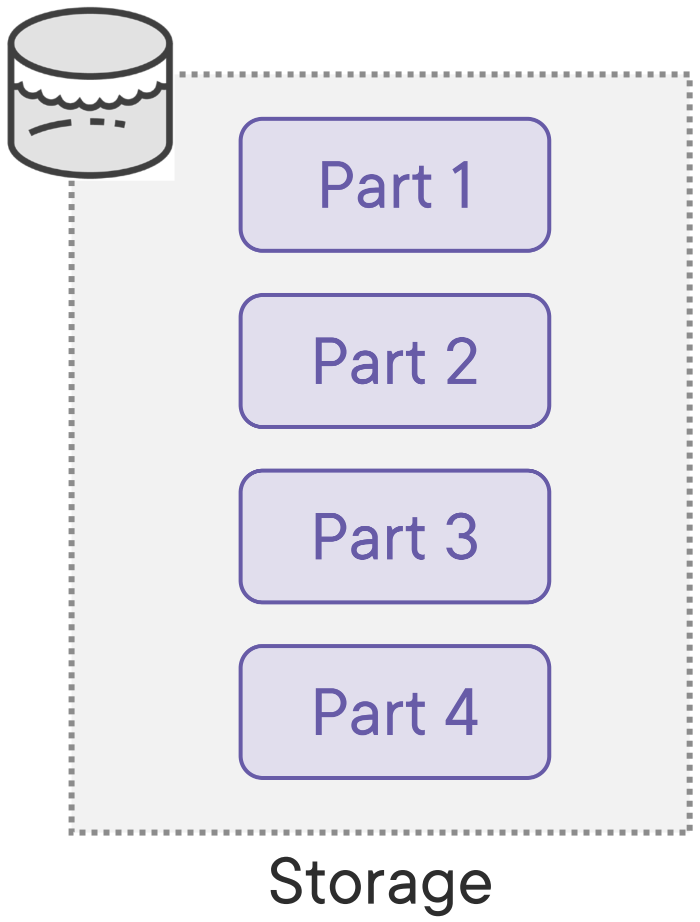
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

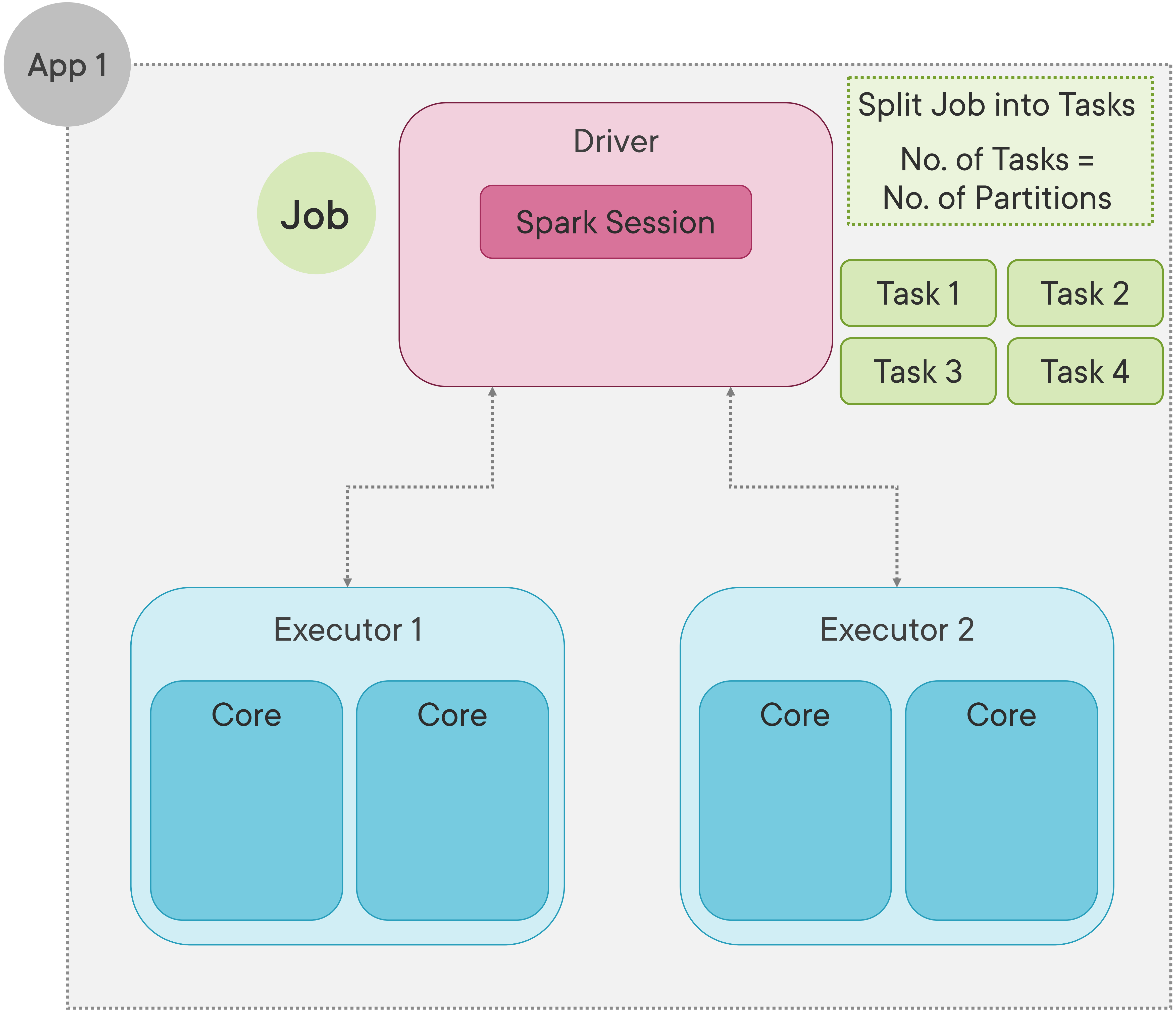


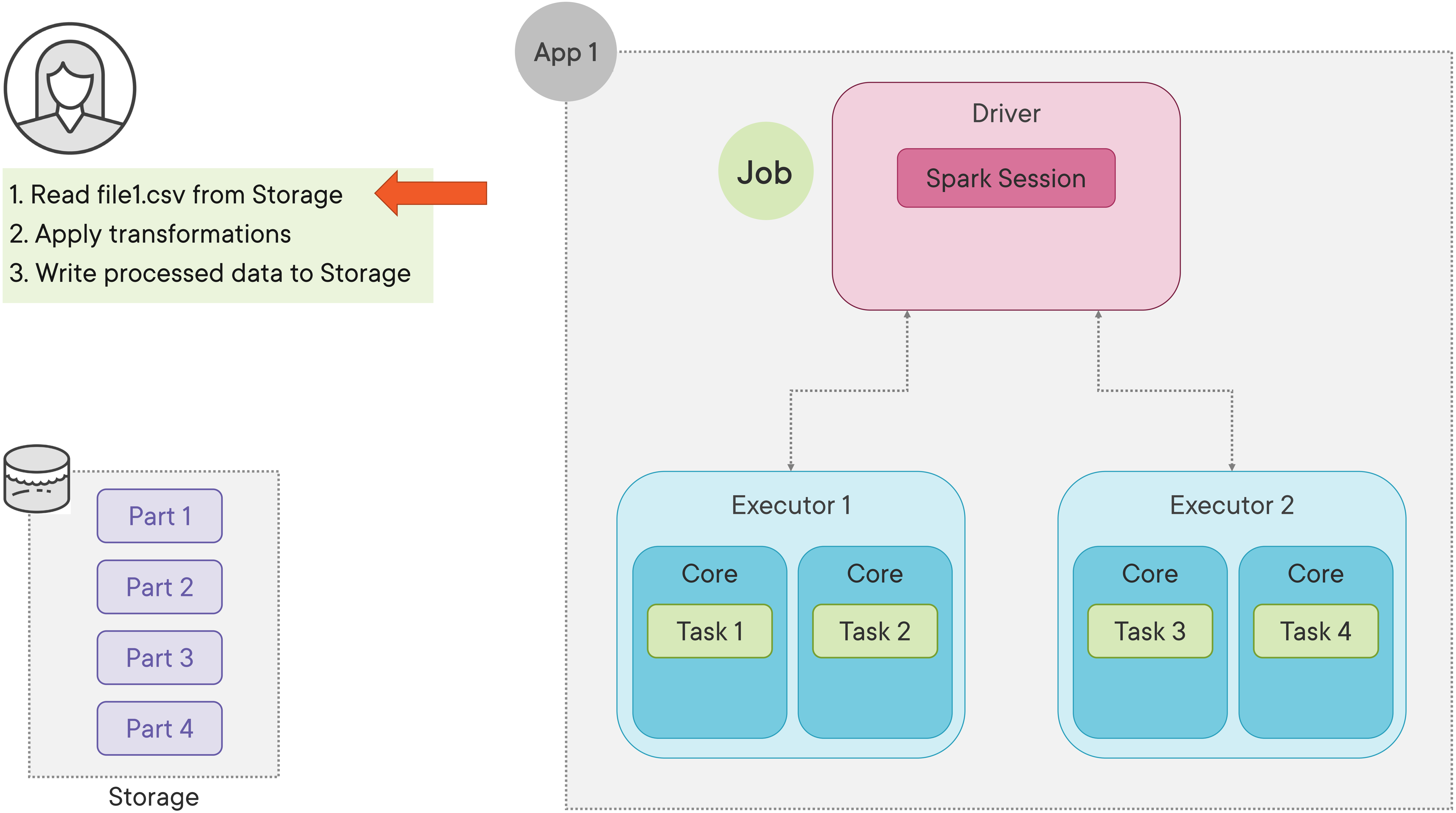




1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Part 1
Part 2
Part 3
Part 4
Storage
App 1
Job
Driver
Spark Session
Split Job into Tasks
No. of Tasks =
No. of Partitions
Executor 1
Core
Core
Executor 2
Core
Core

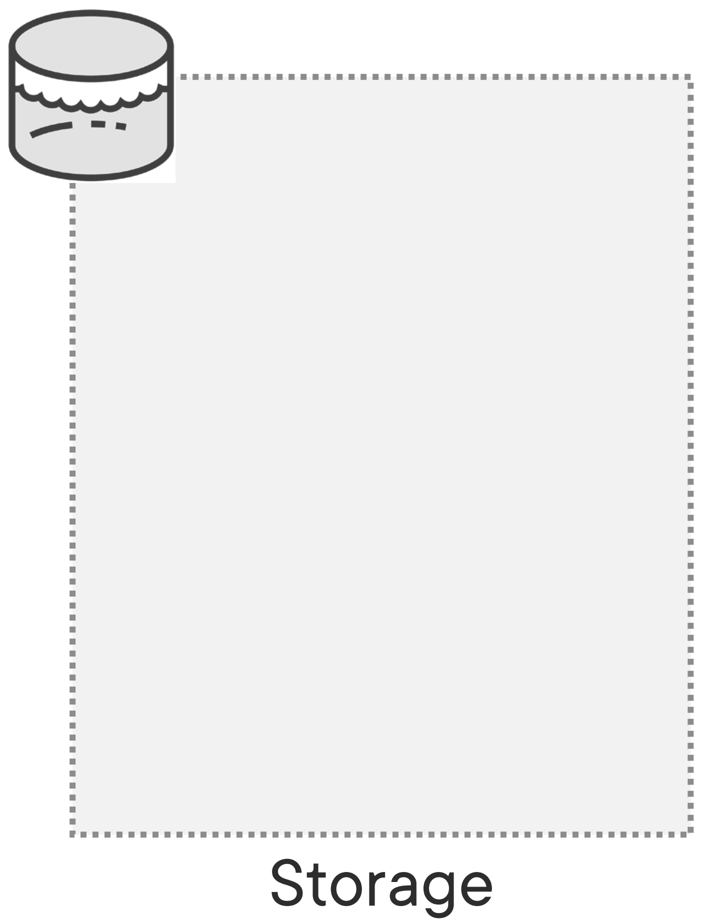
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

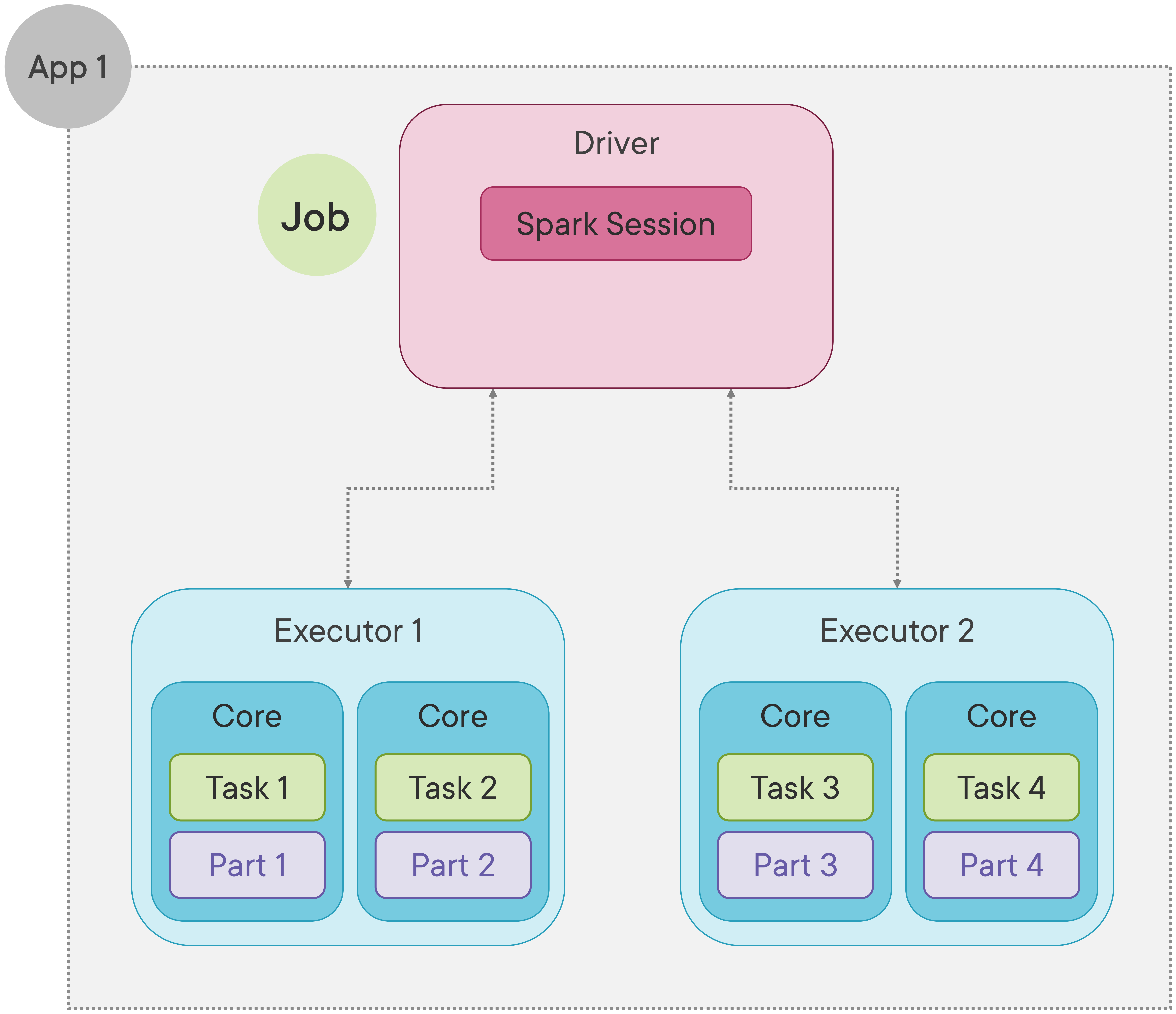


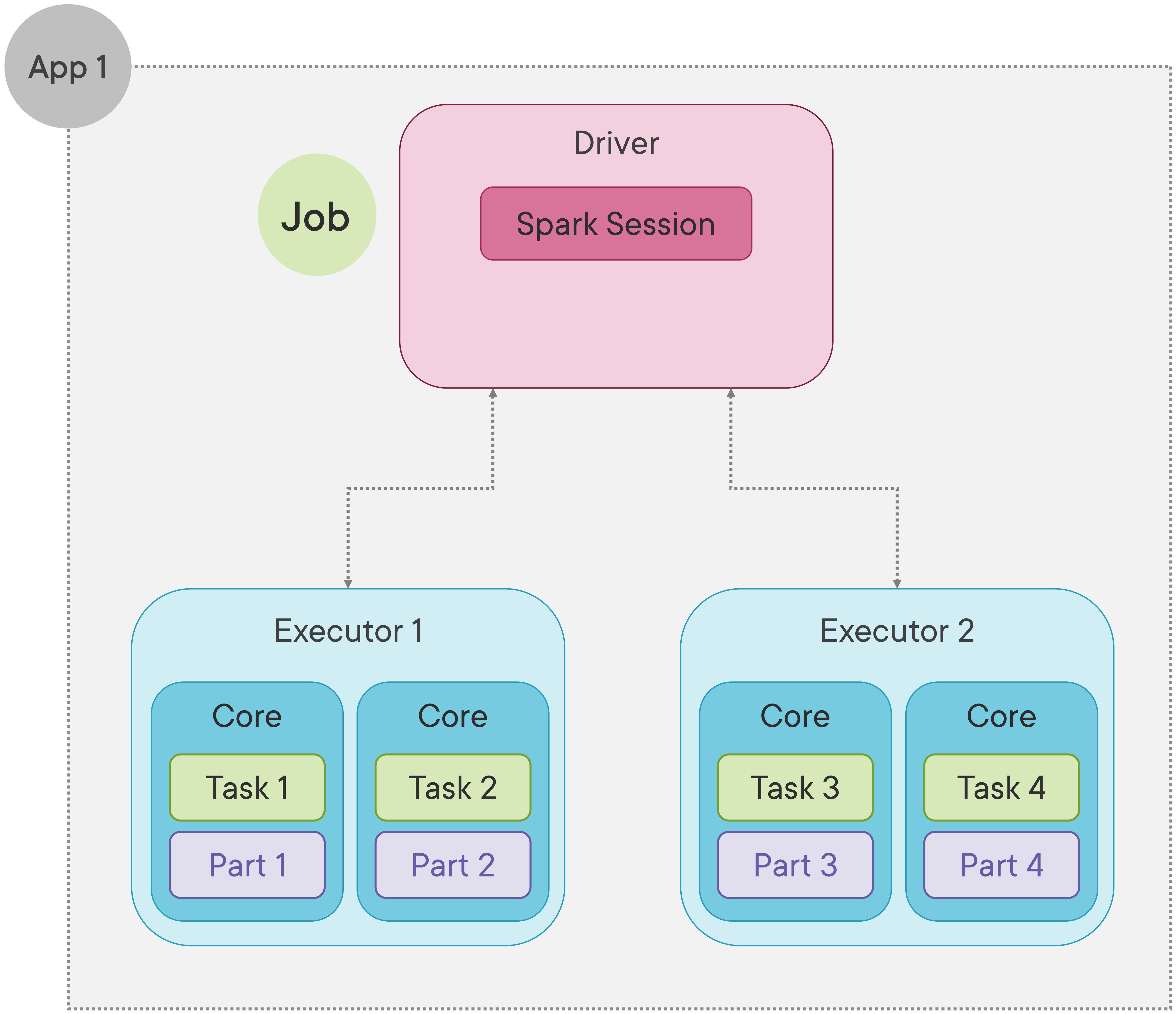

1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Part 1
Part 2
Part 3
Part 4
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Core
Task 2
Executor 2
Core
Task 3
Core
Task 4

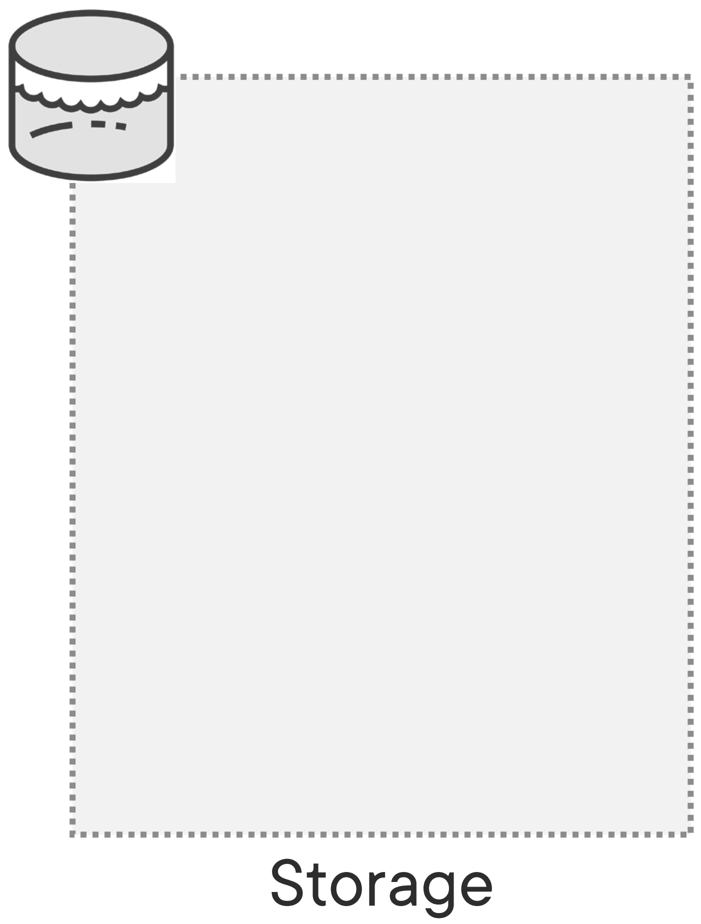
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

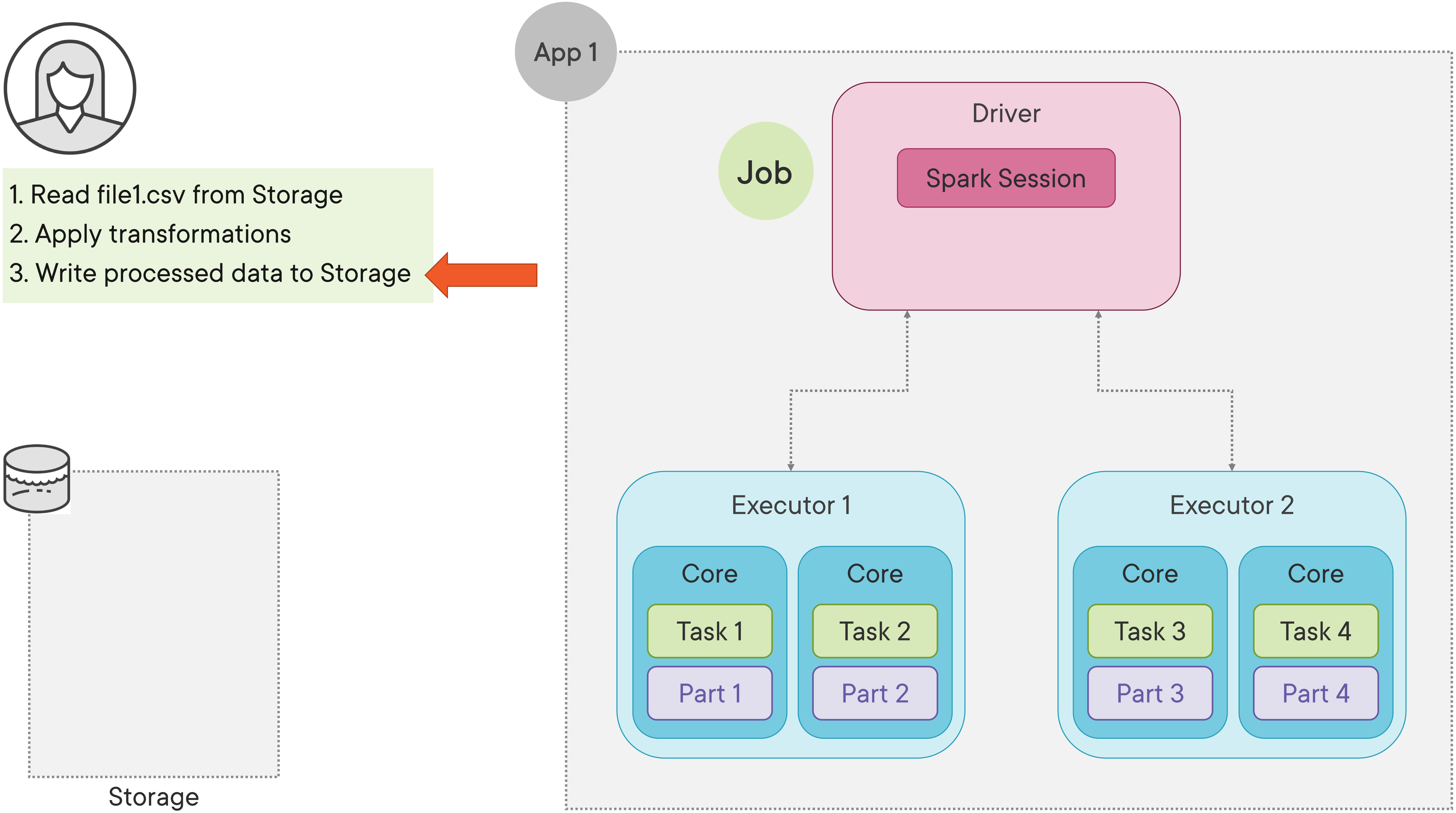


1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

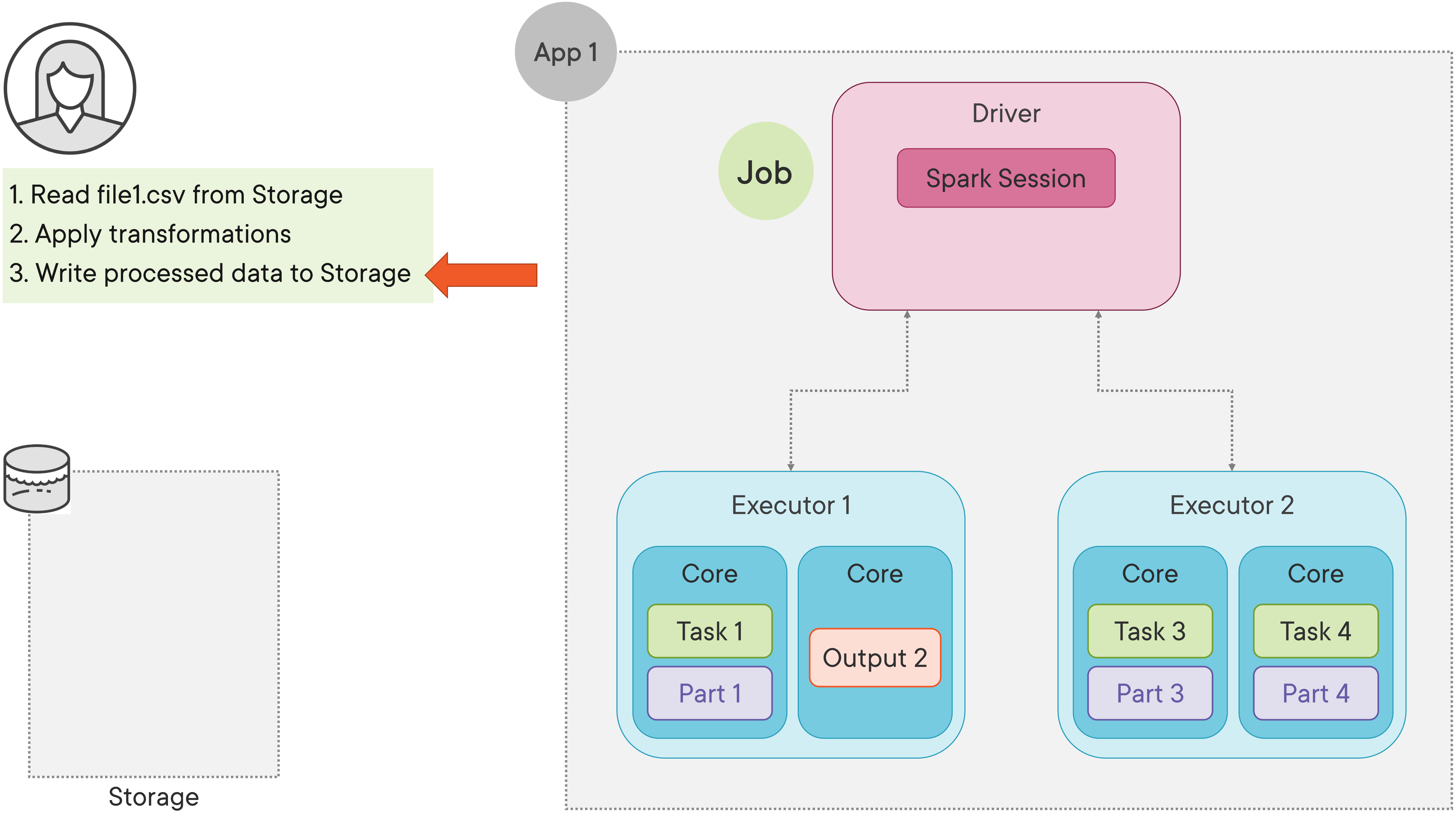


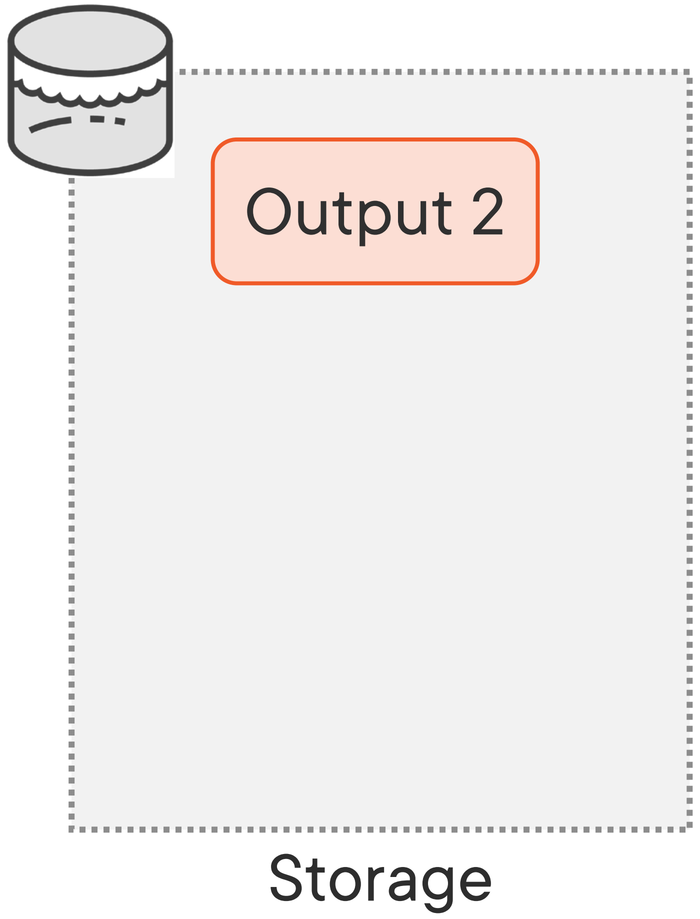


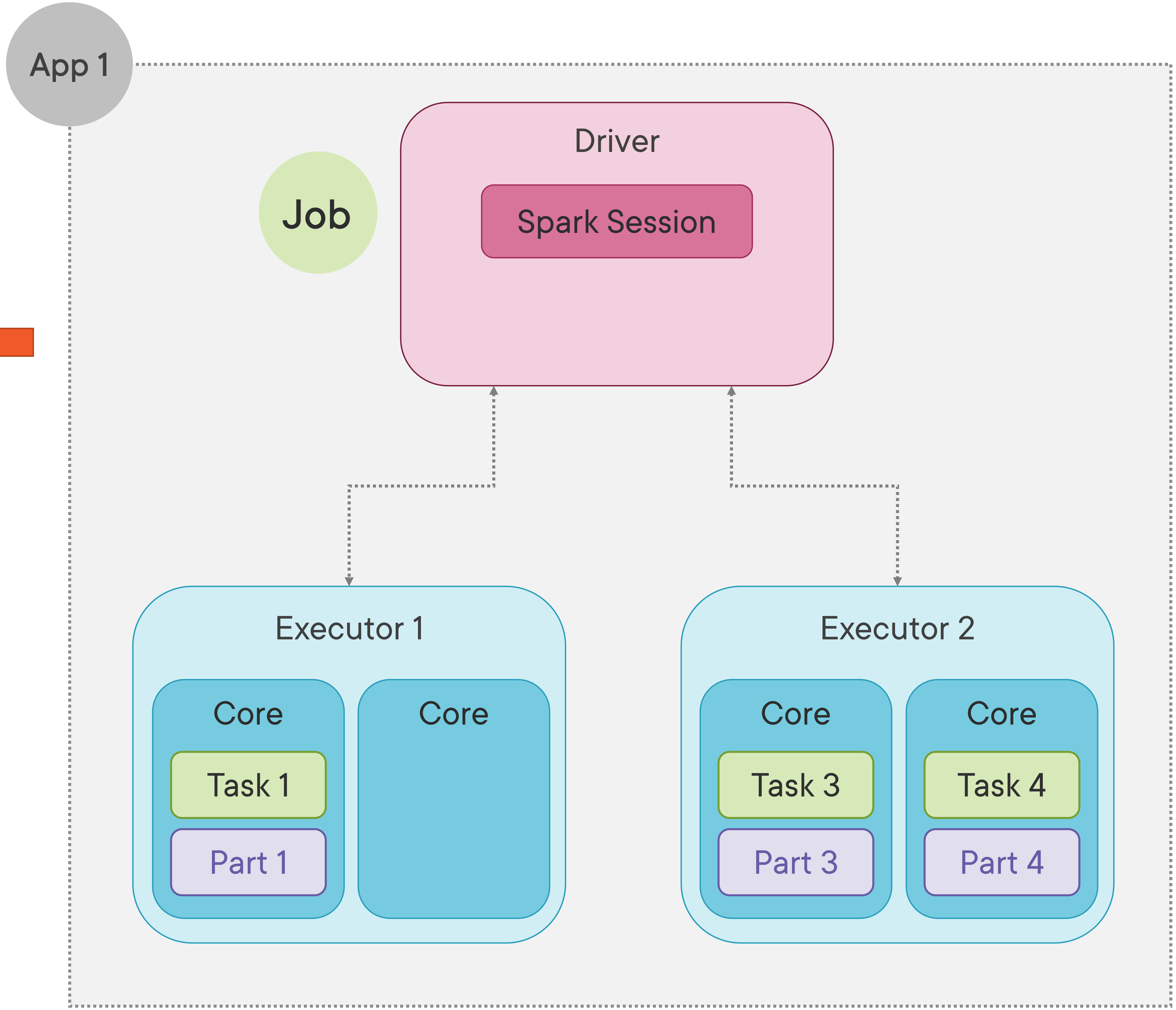
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Part 1
Core
Task 2
Part 2
Executor 2
Core
Task 3
Part 3
Core
Task 4
Part 4

1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

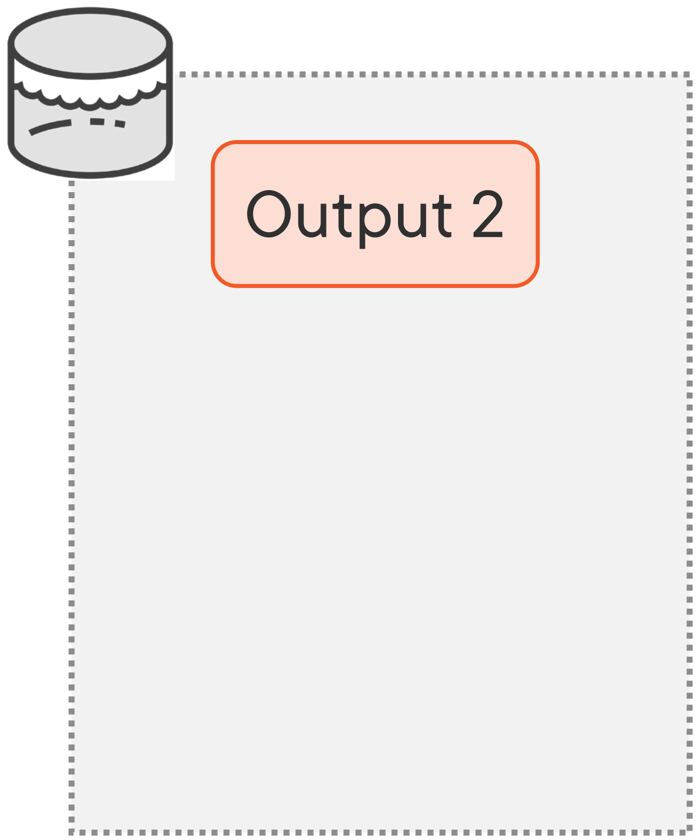


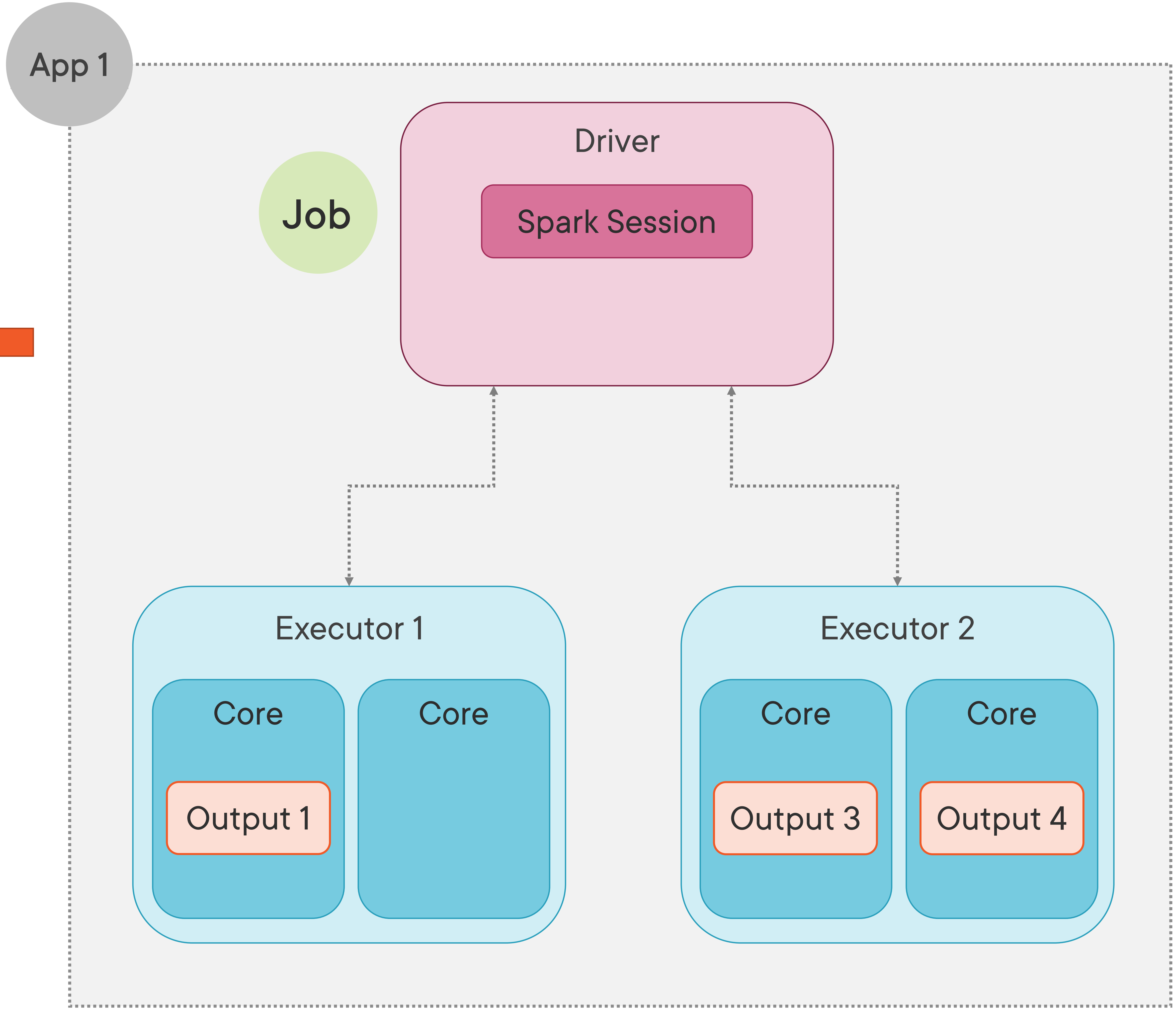
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



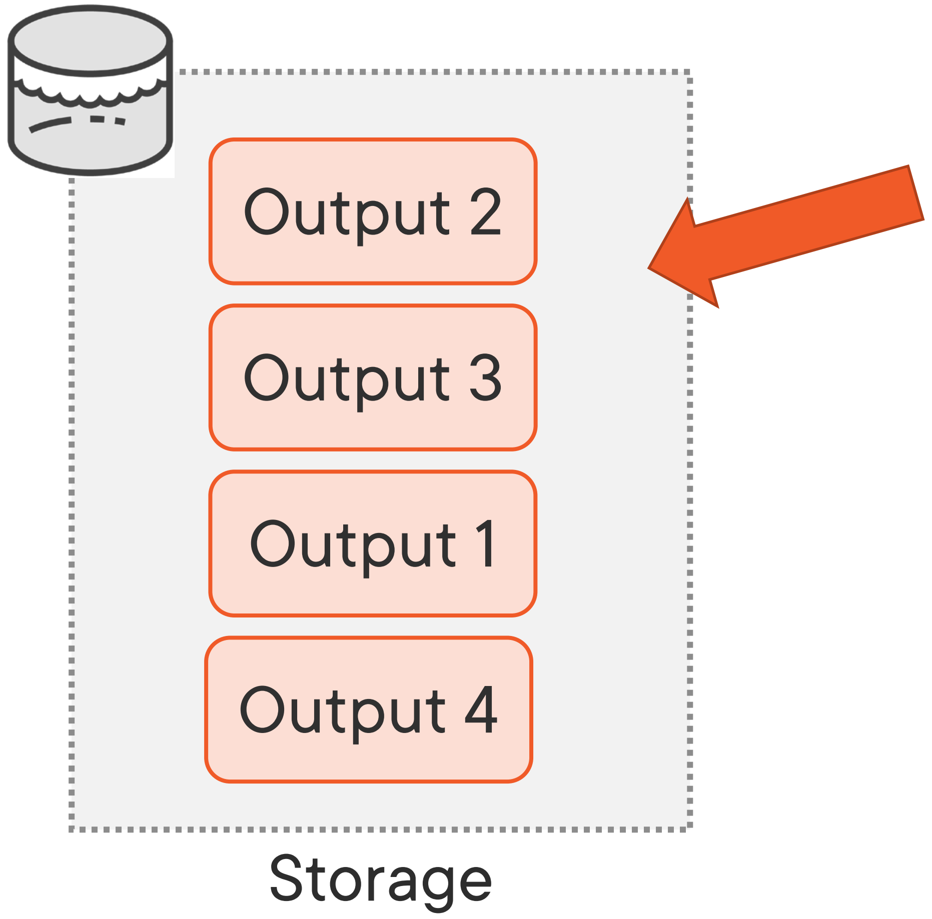



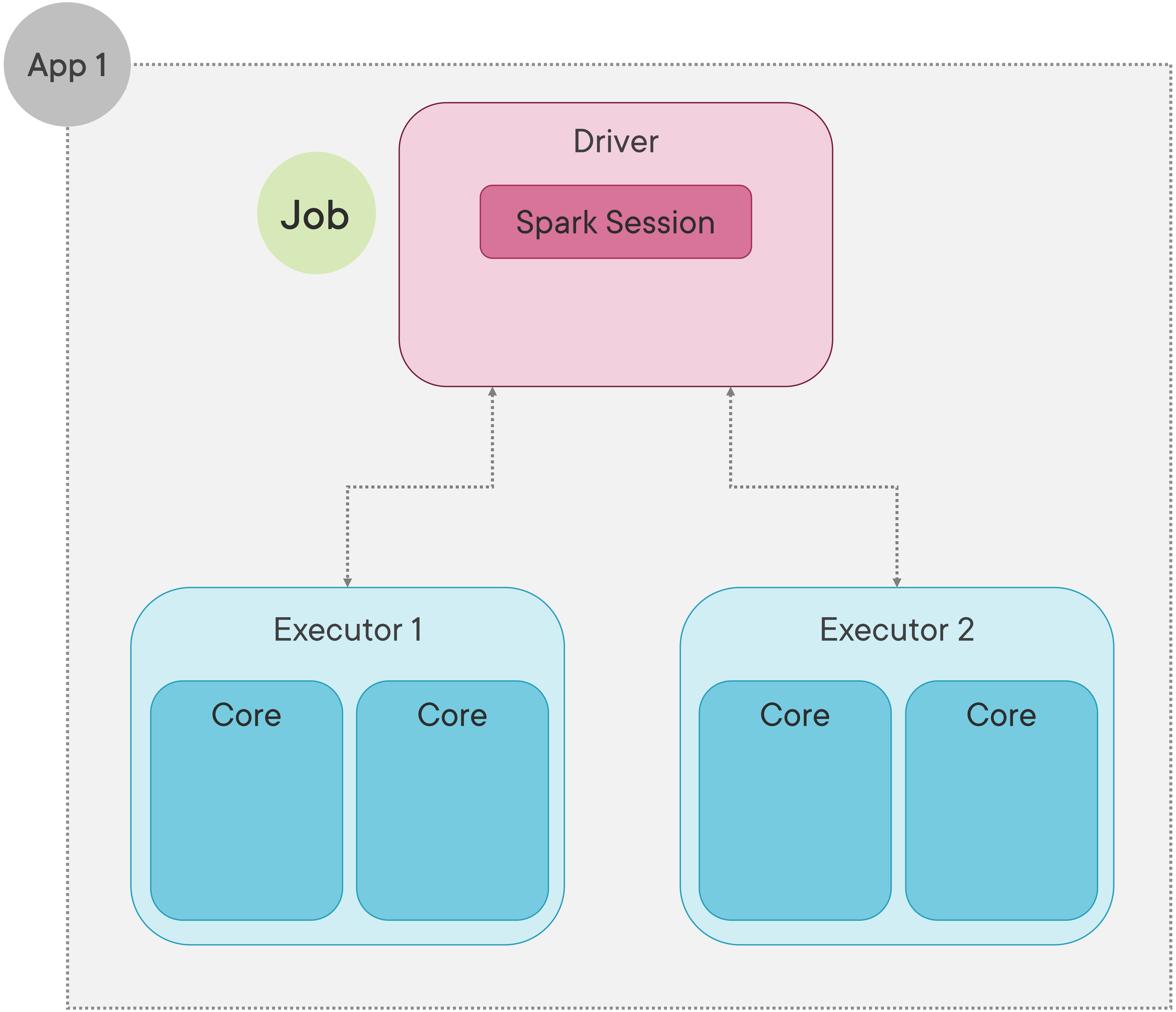
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



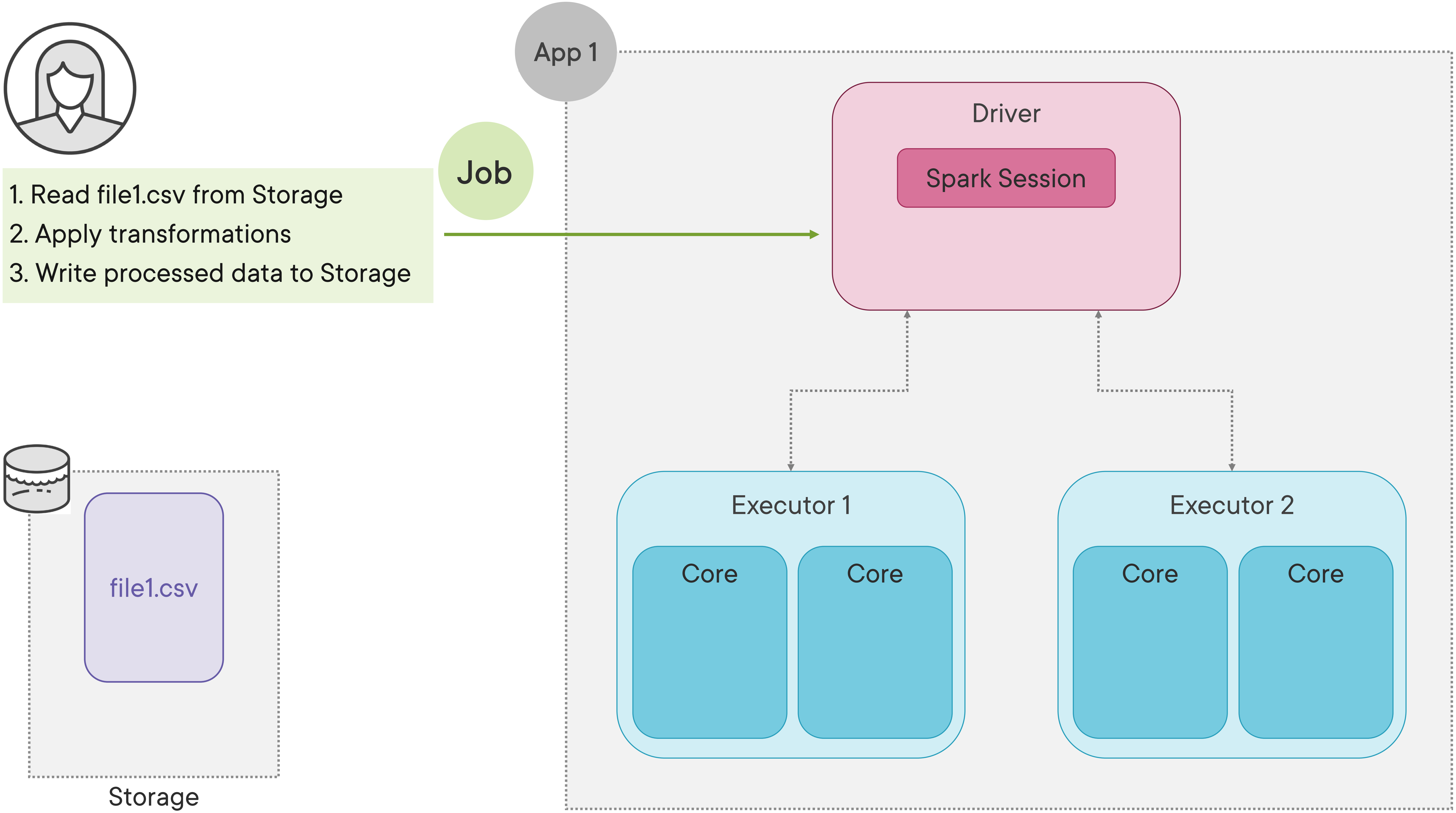



1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

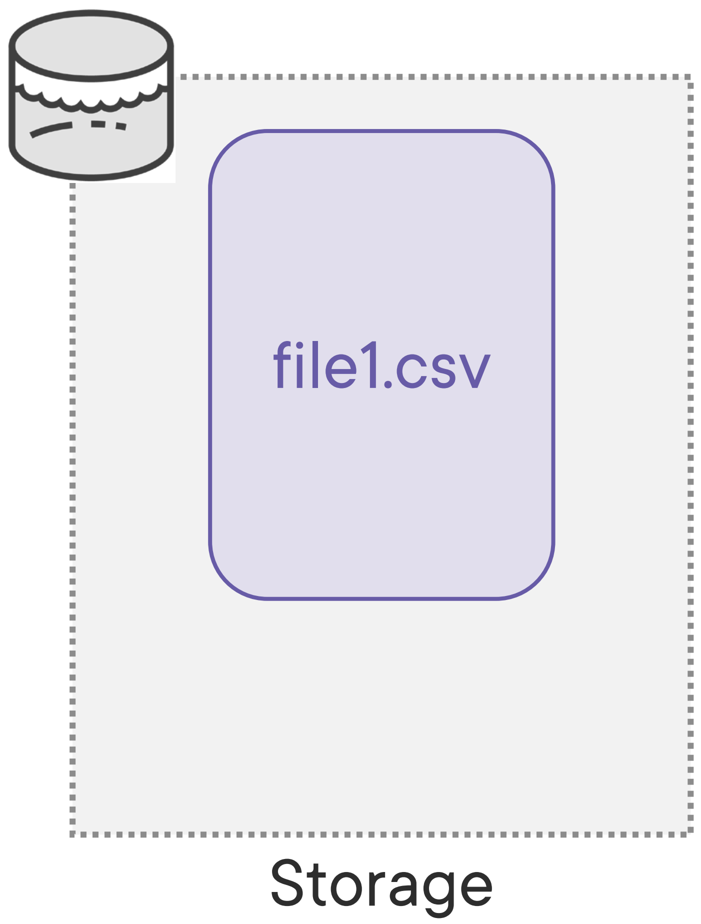


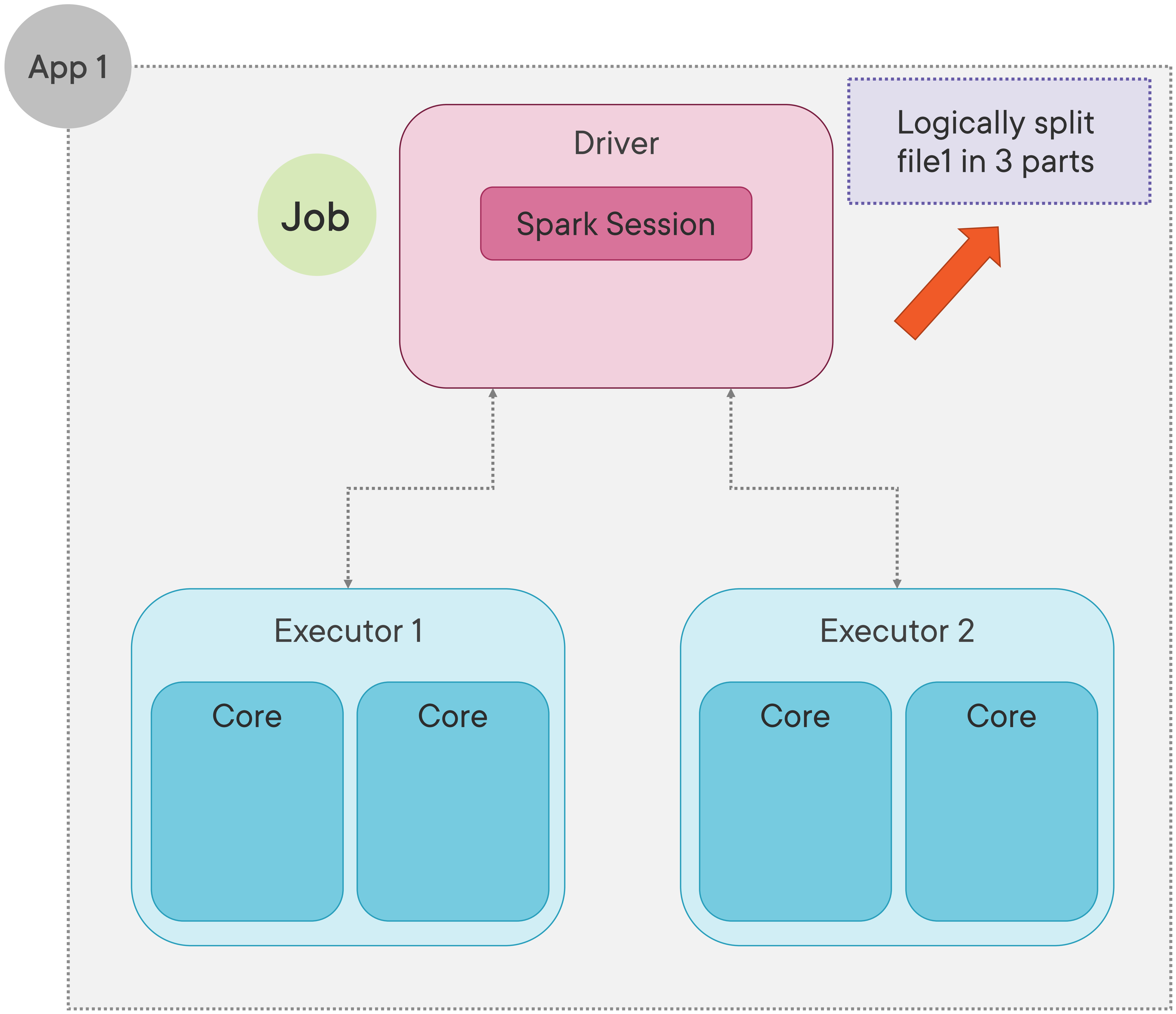



What if number of Tasks are lesser than available Cores?

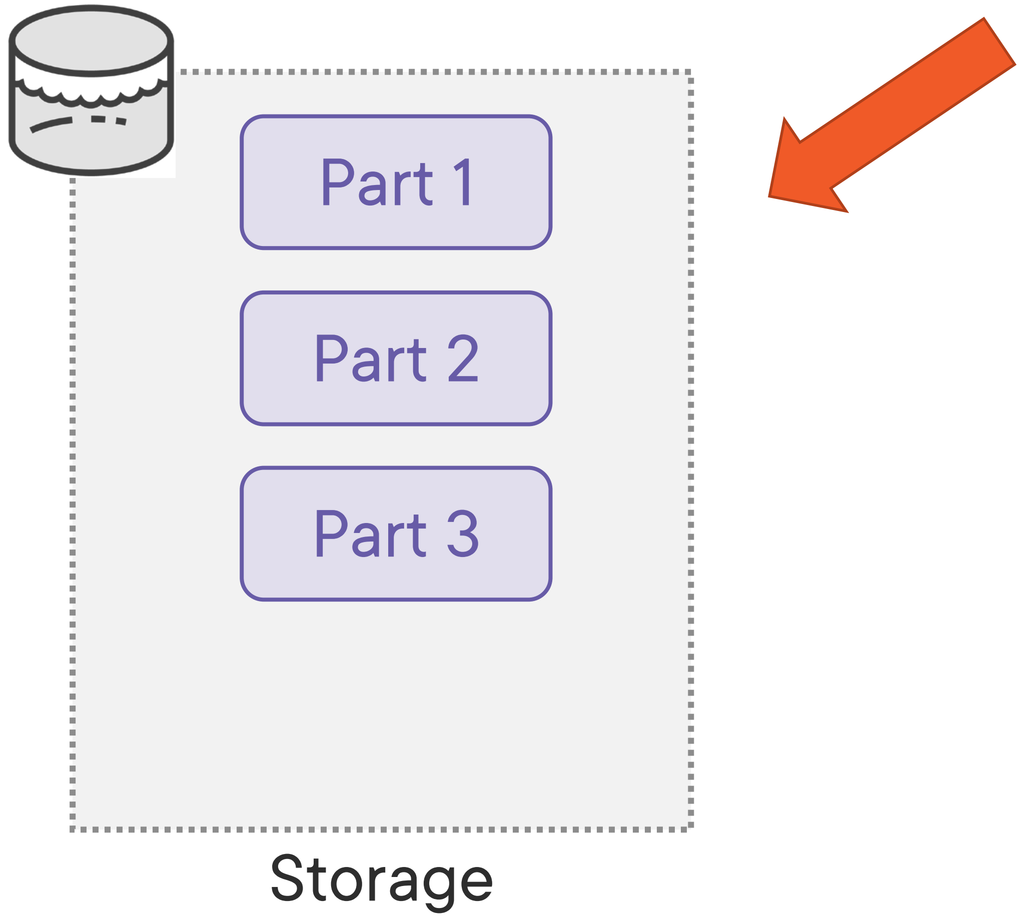

App 1
Job
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Driver
Spark Session
Executor 1
Core
Core
Executor 2
Core
Core
file1.csv
Storage

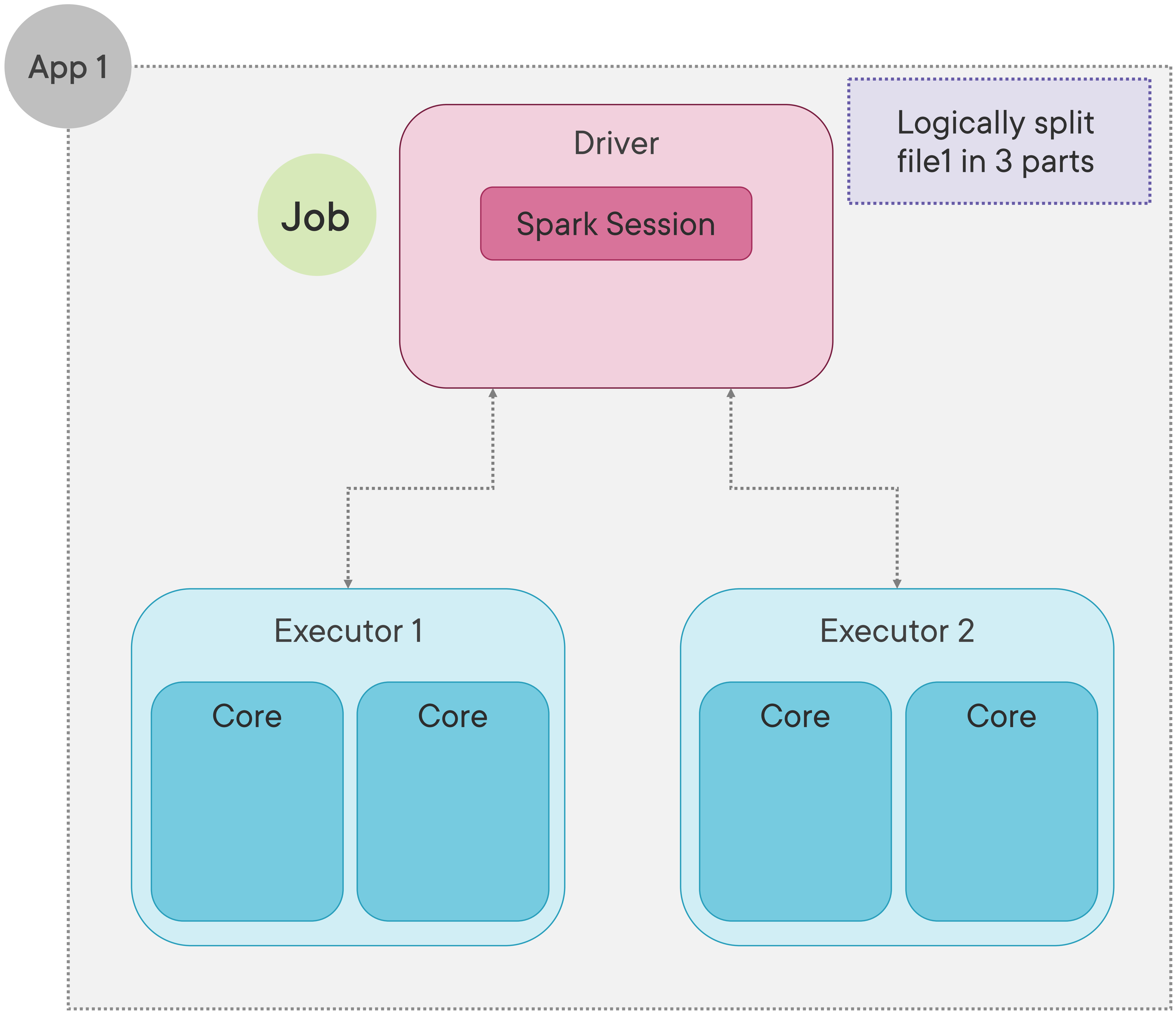
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

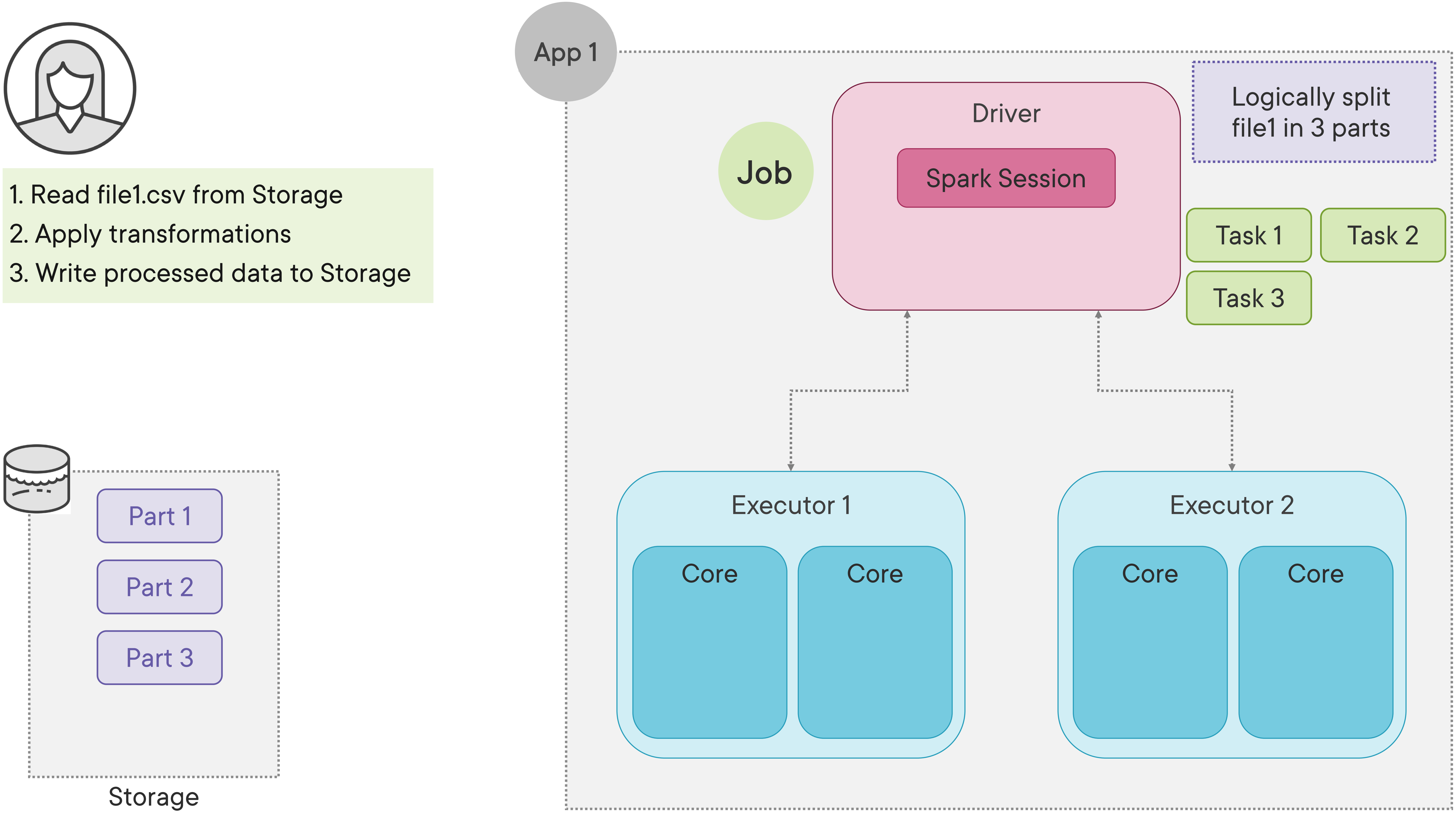


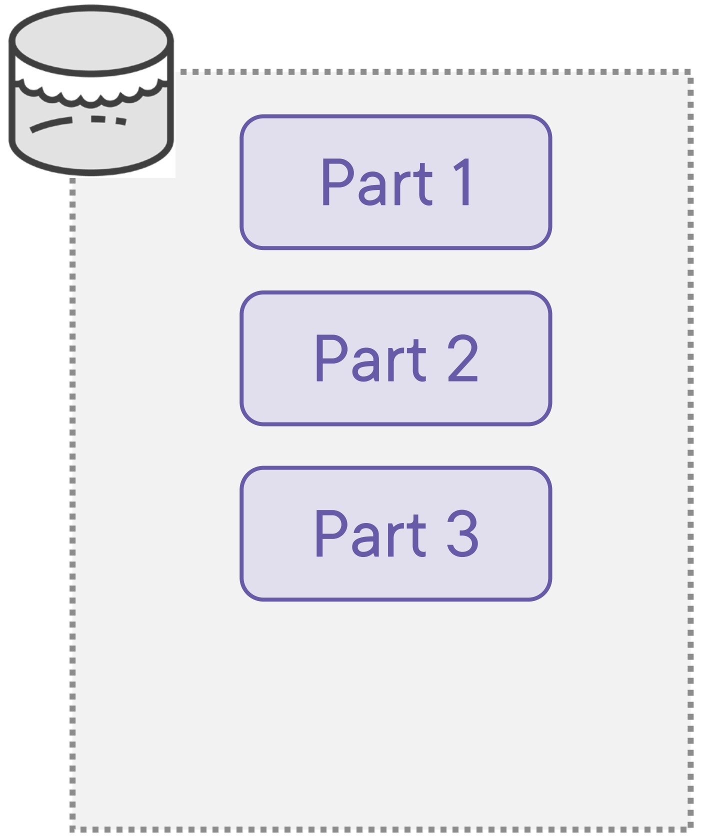
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



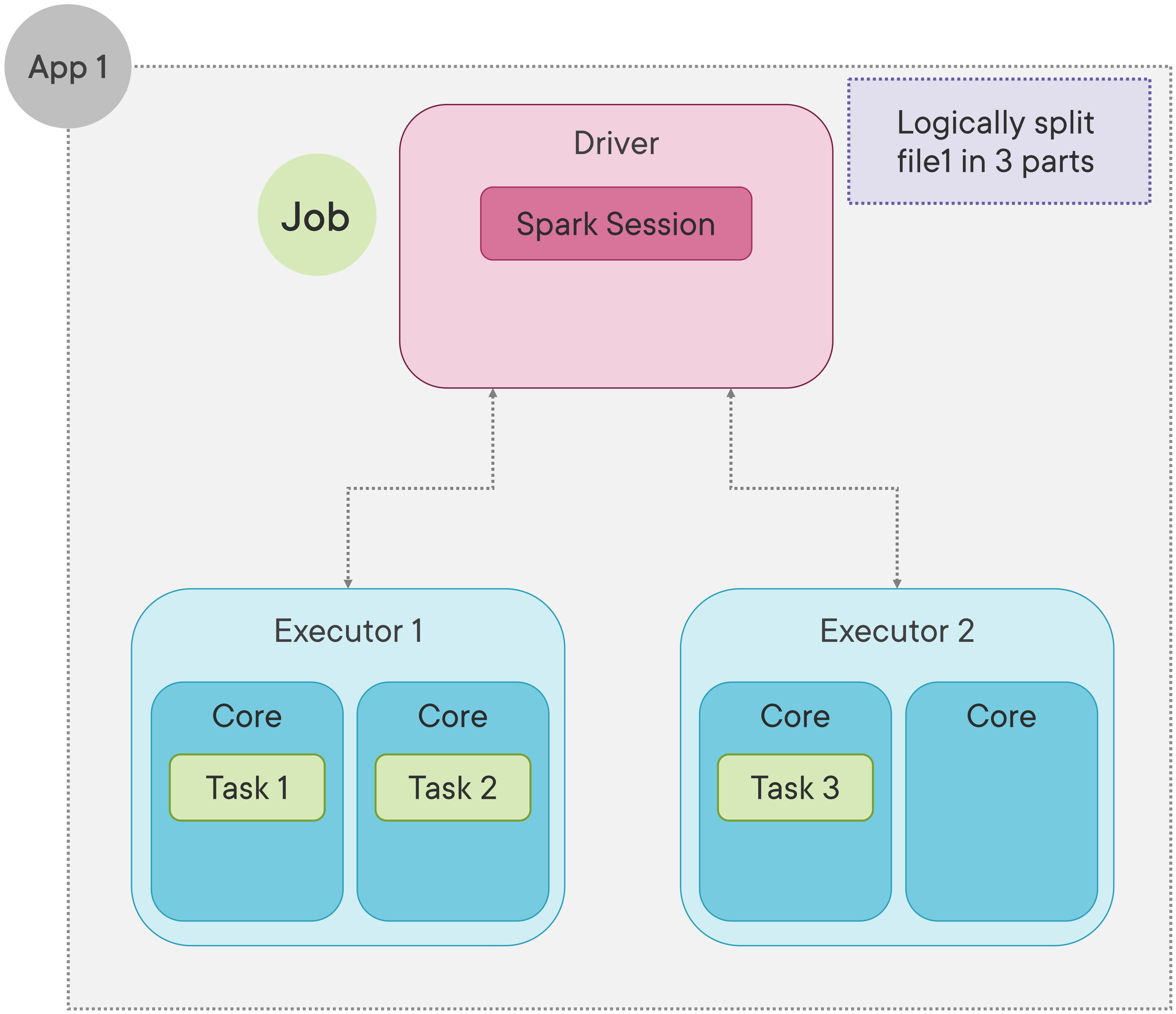




1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
App 1
Job
Driver
Spark Session
Logically split file1 in 3 parts
Task 1
Task 2
Task 3
Executor 1
Core
Core
Executor 2
Core
Core
Part 1
Part 2
Part 3
Storage

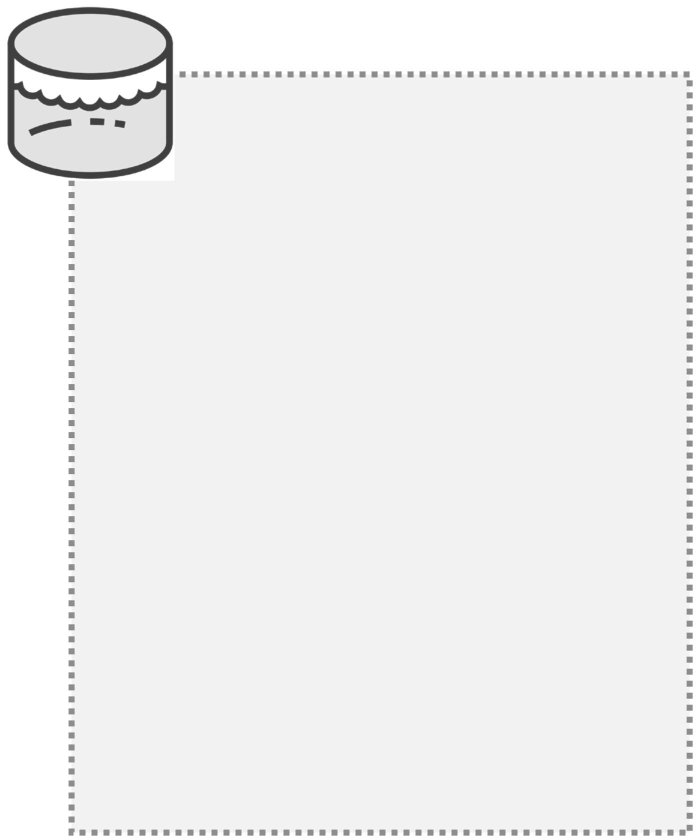
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



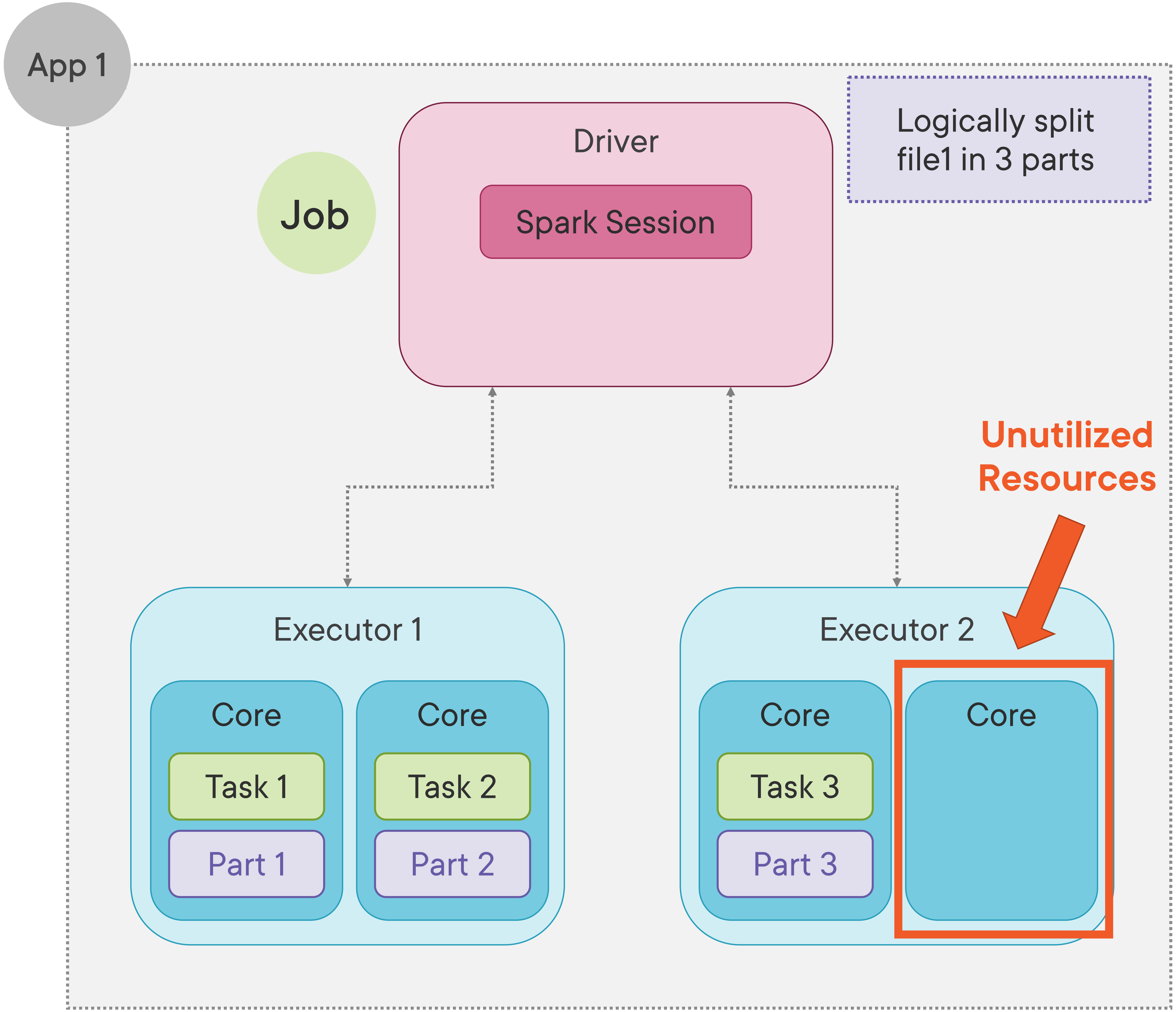

Storage



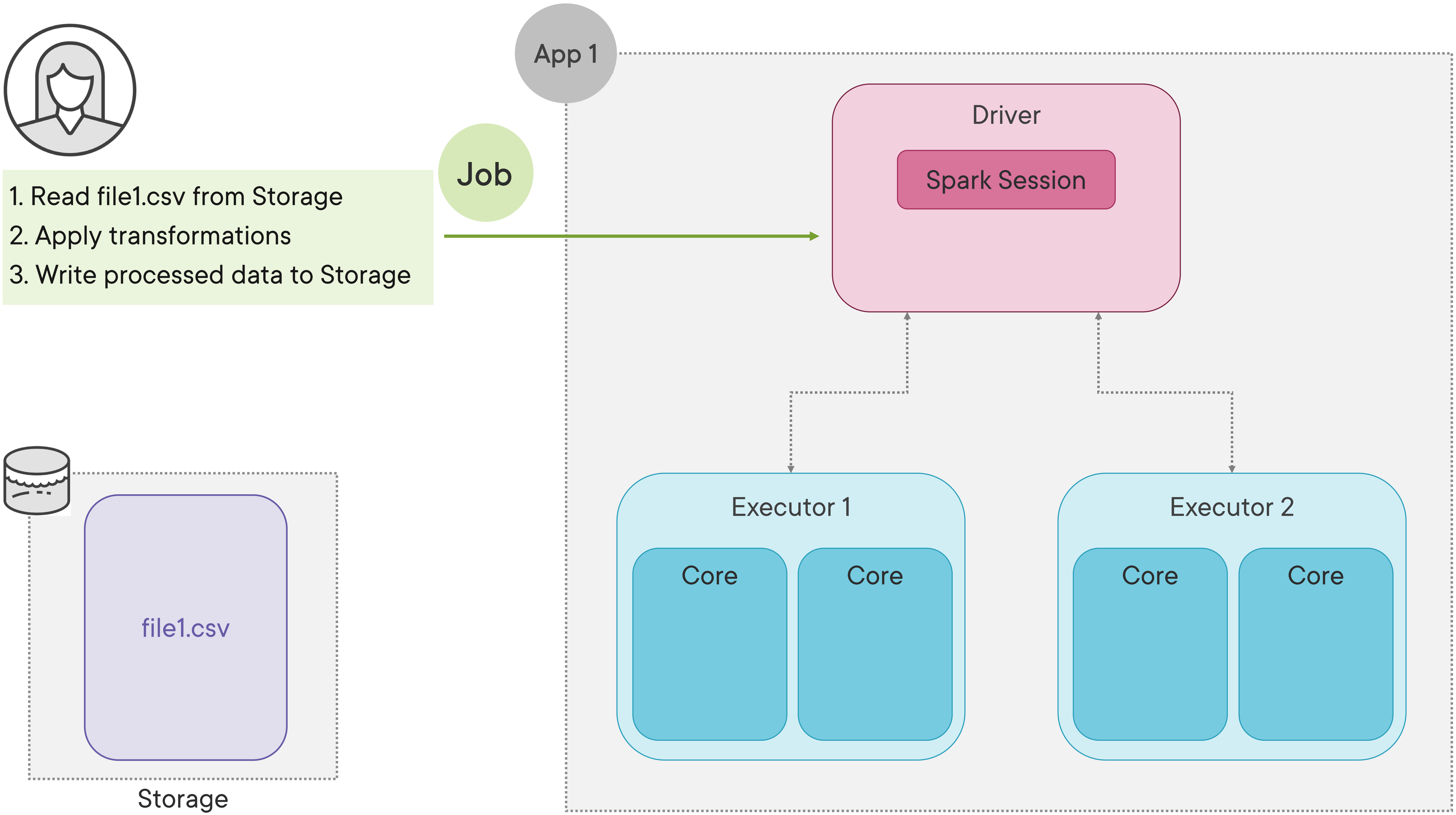
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



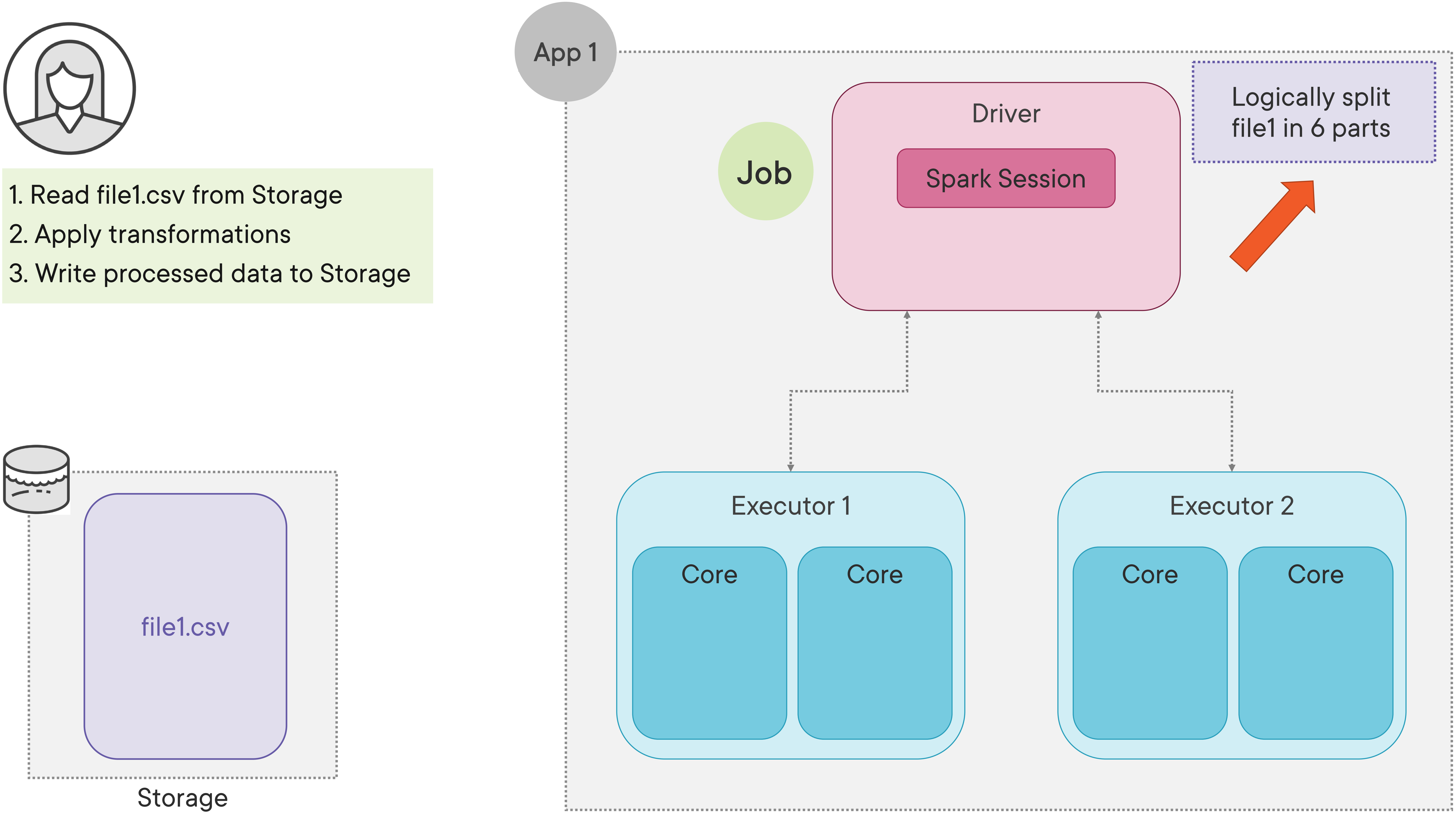
Storage



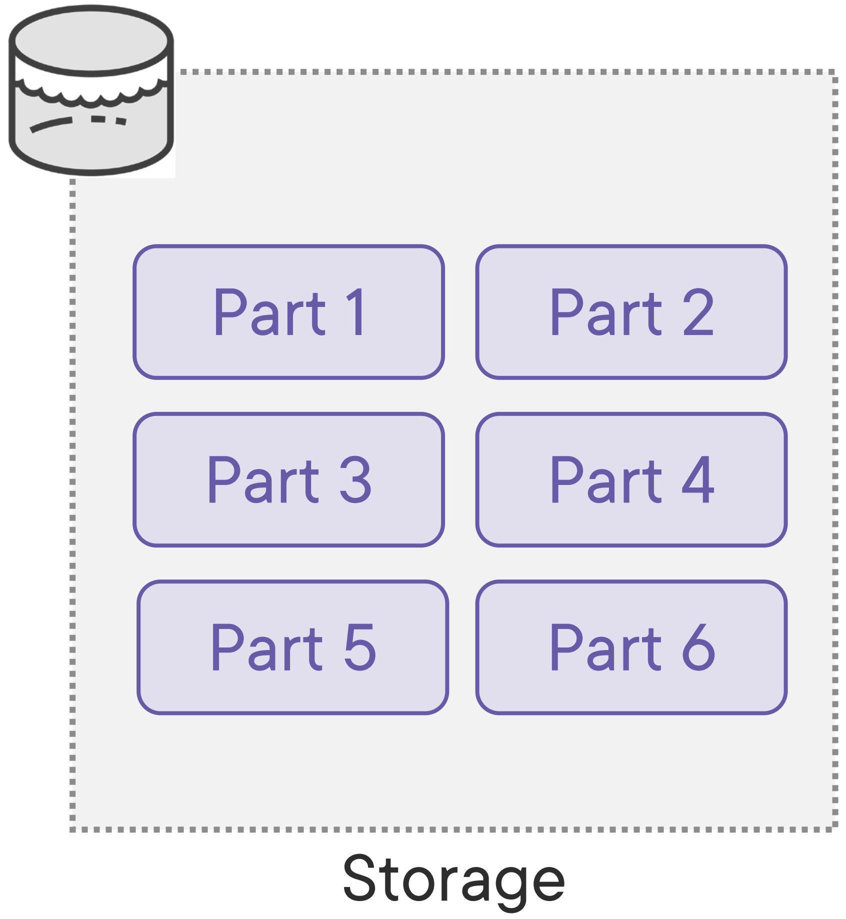
What if number of Tasks are more than available Cores?

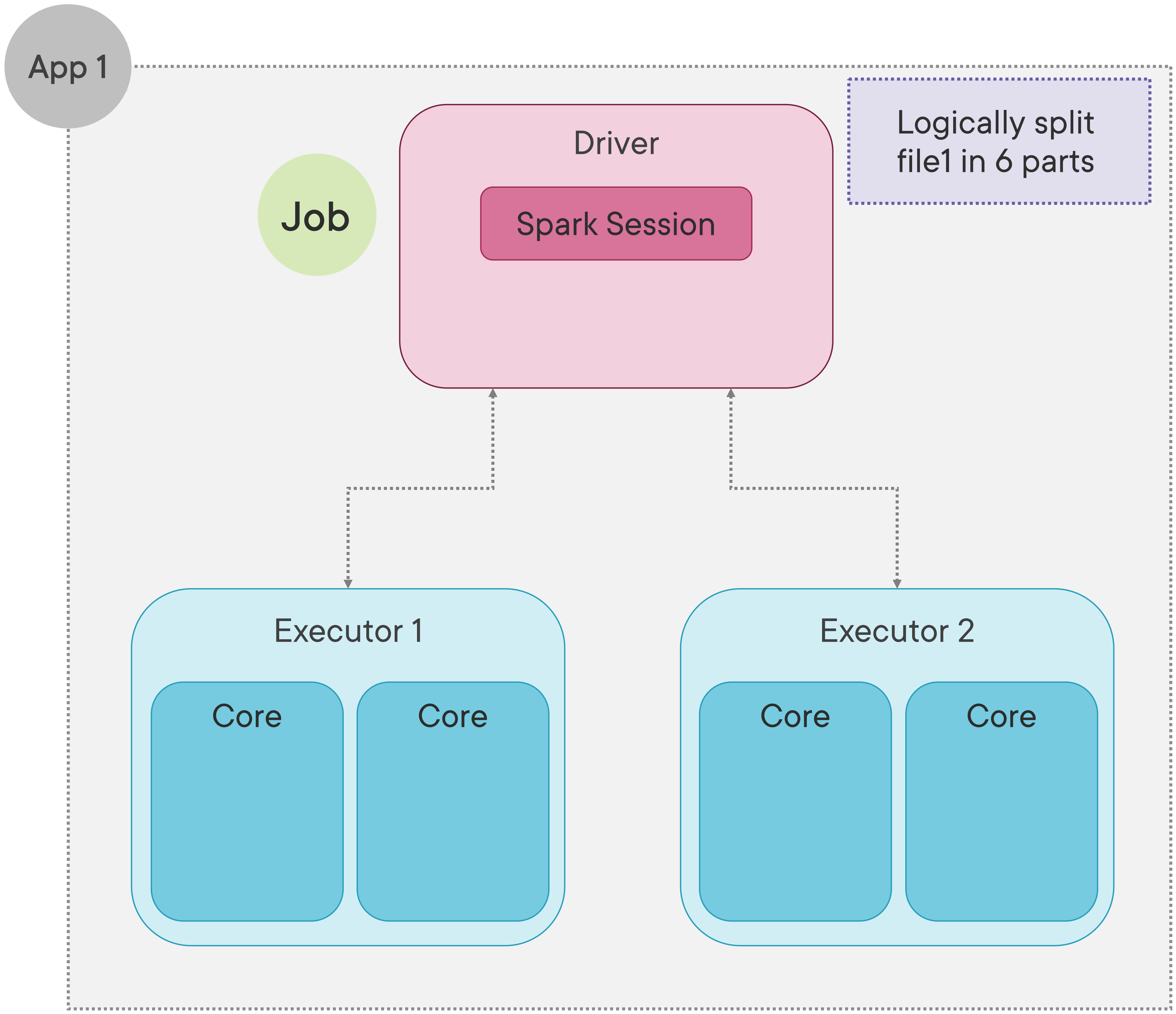

App 1
Driver
Spark Session
Job
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Executor 1
Core
Core
Executor 2
Core
Core
file1.csv
Storage

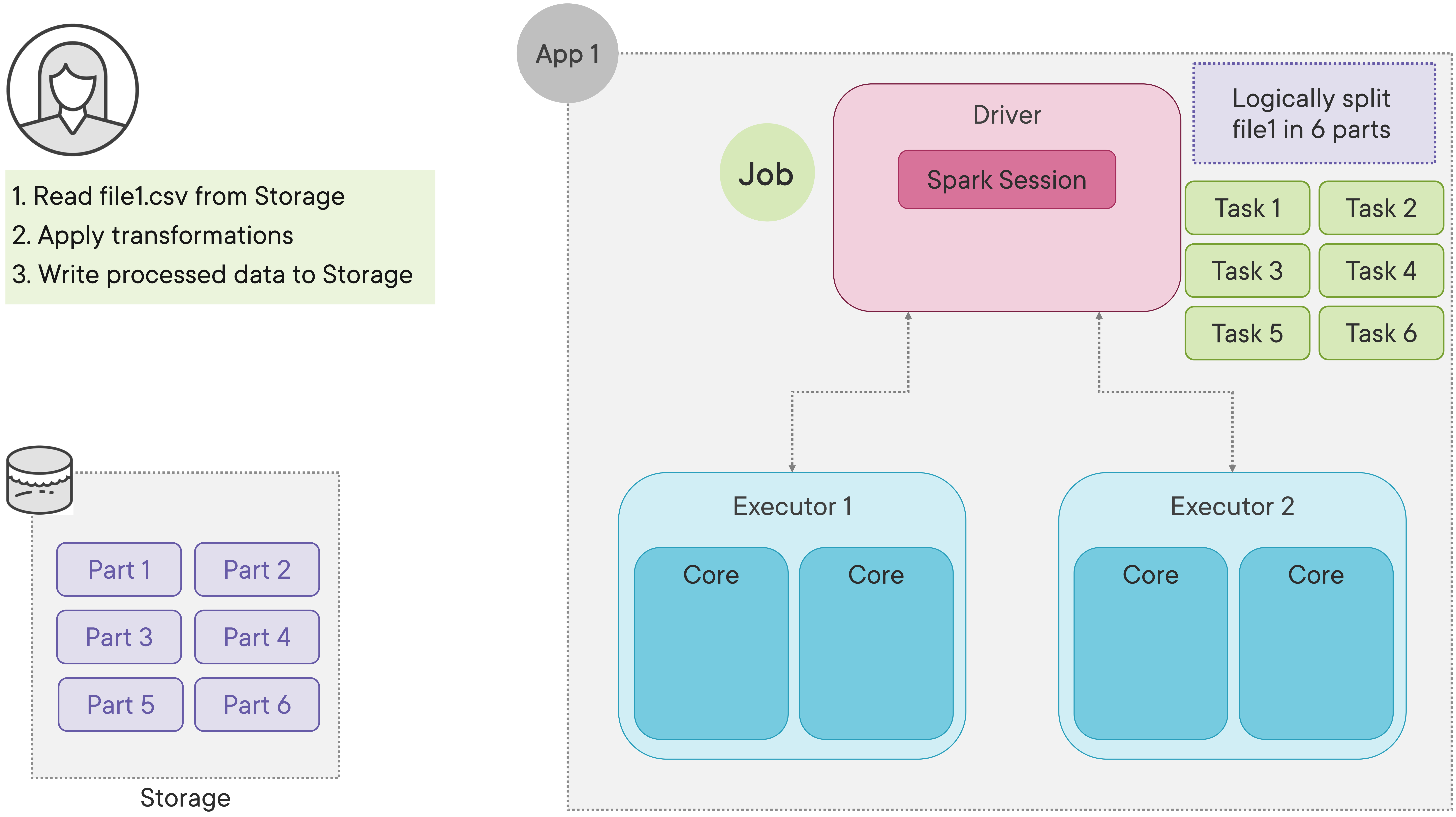
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



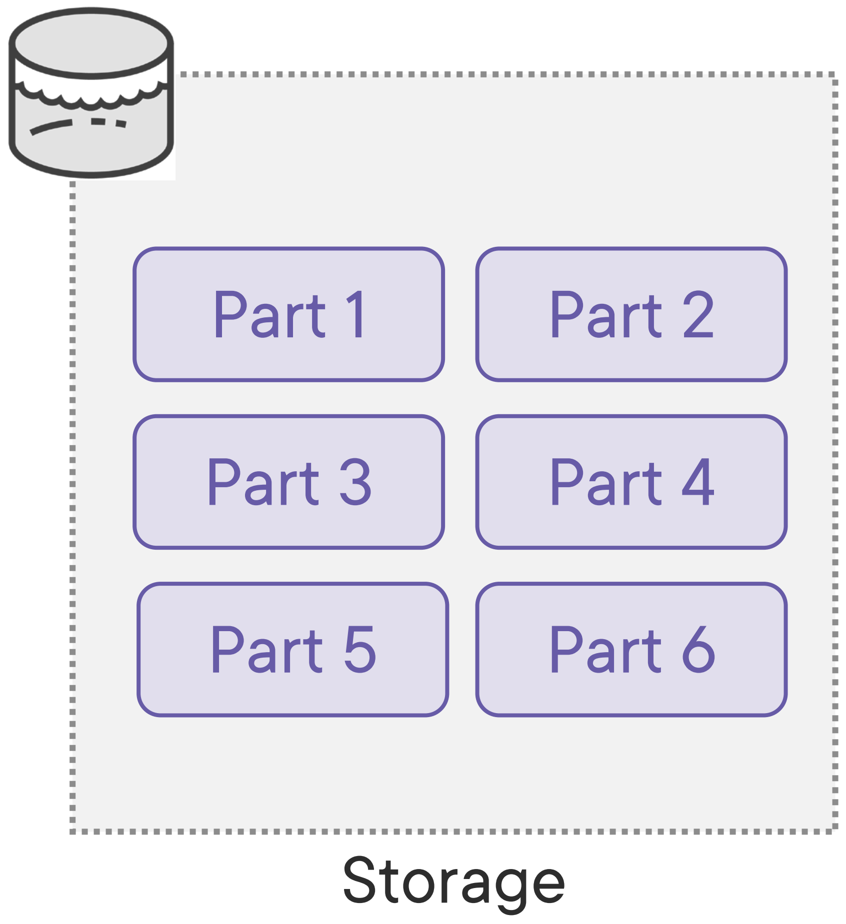
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

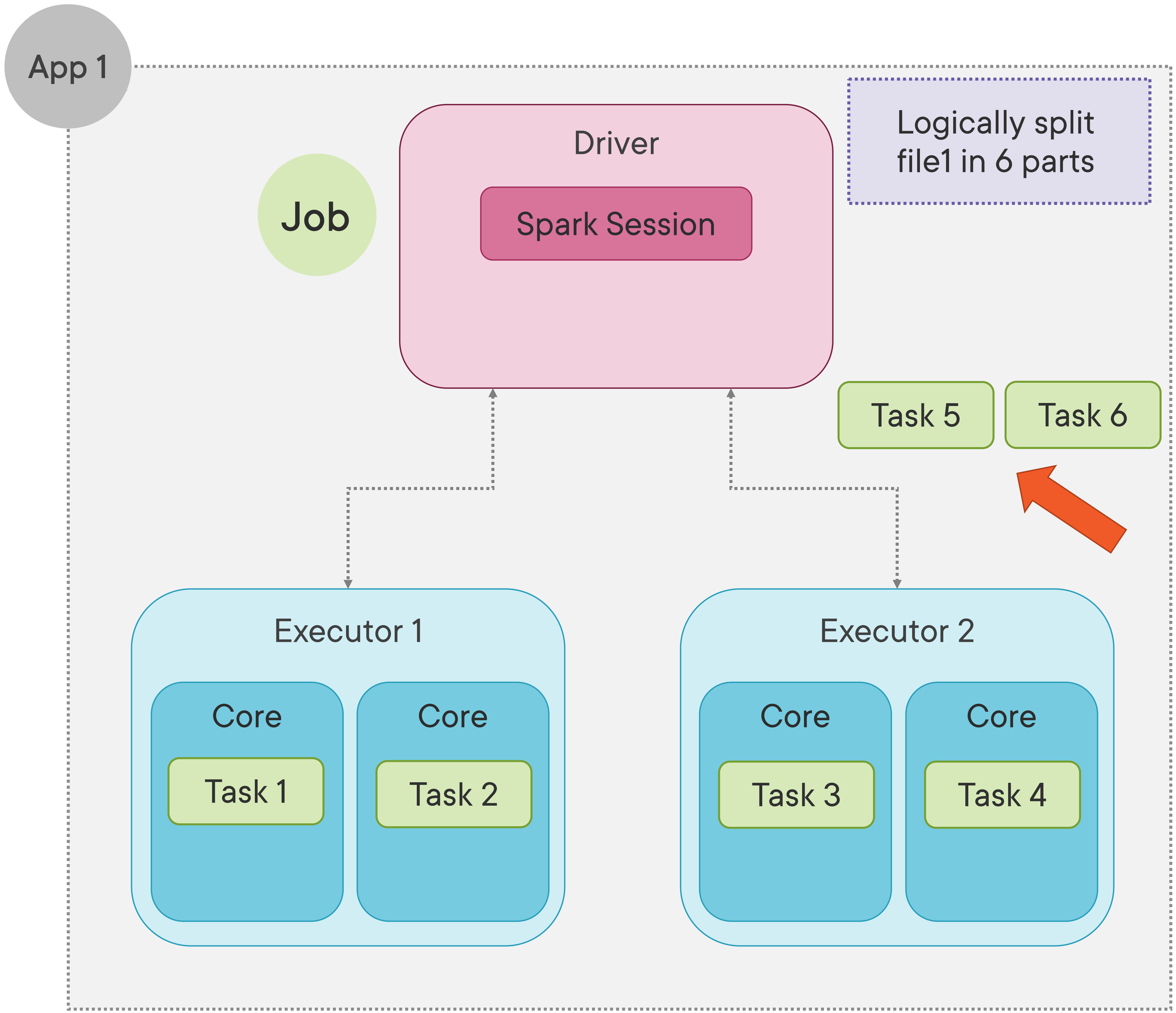


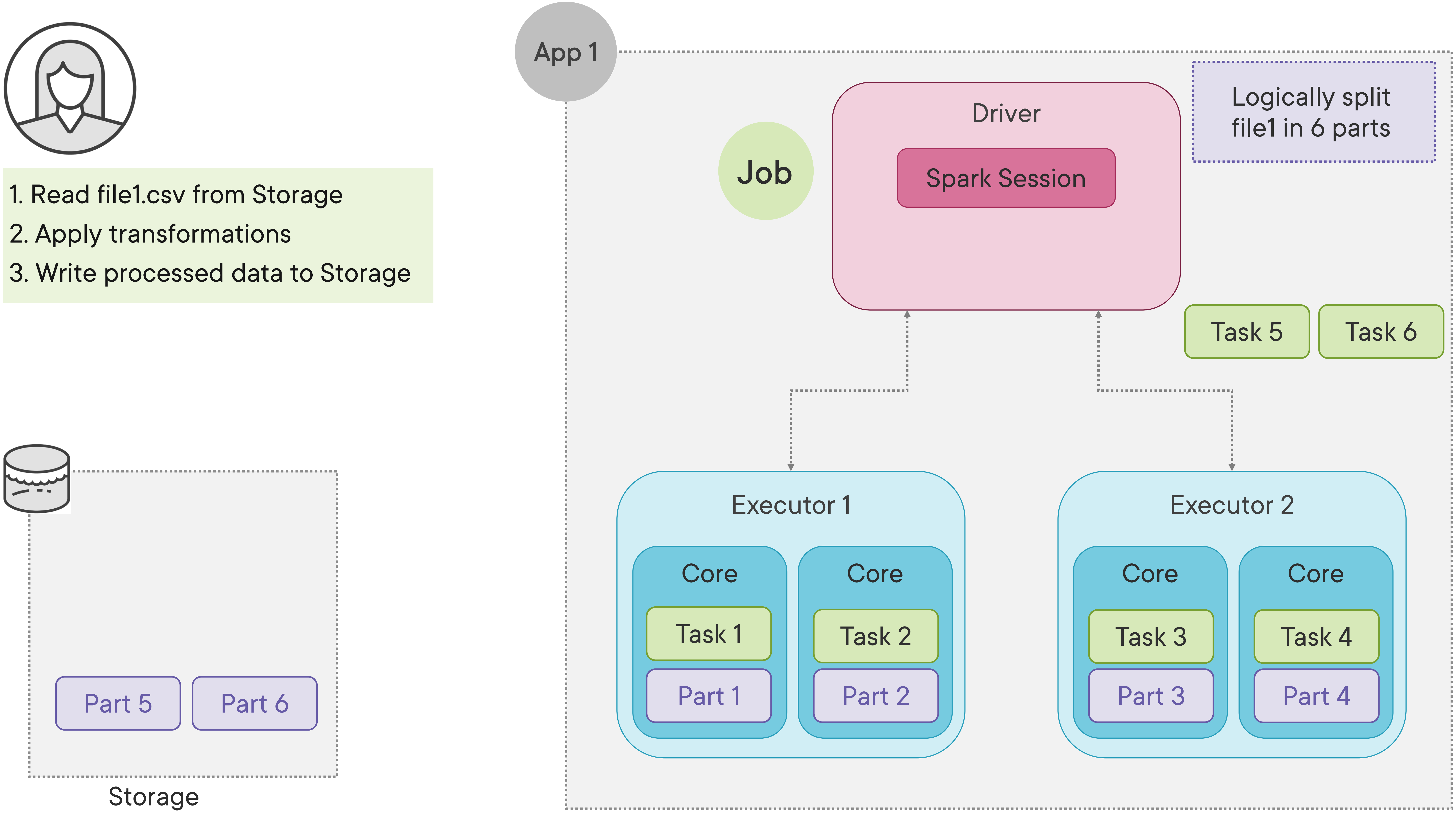




1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Part 1
Part 2
Part 3
Part 4
Part 5
Part 6
Storage
App 1
Job
Driver
Spark Session
Logically split file1 in 6 parts
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Executor 1
Core
Core
Executor 2
Core
Core

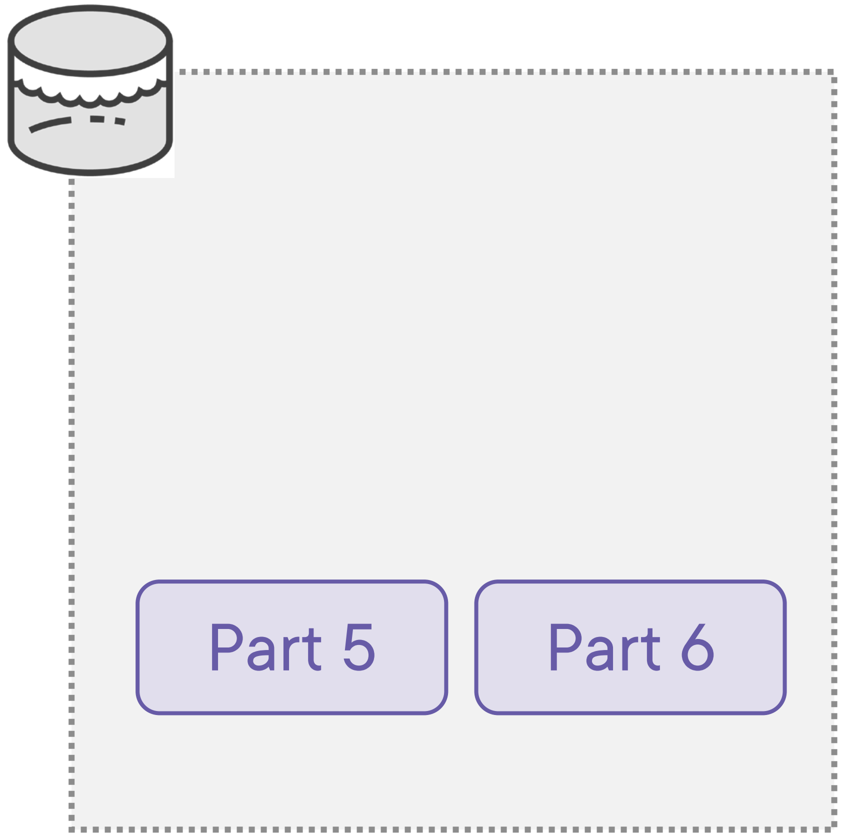
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



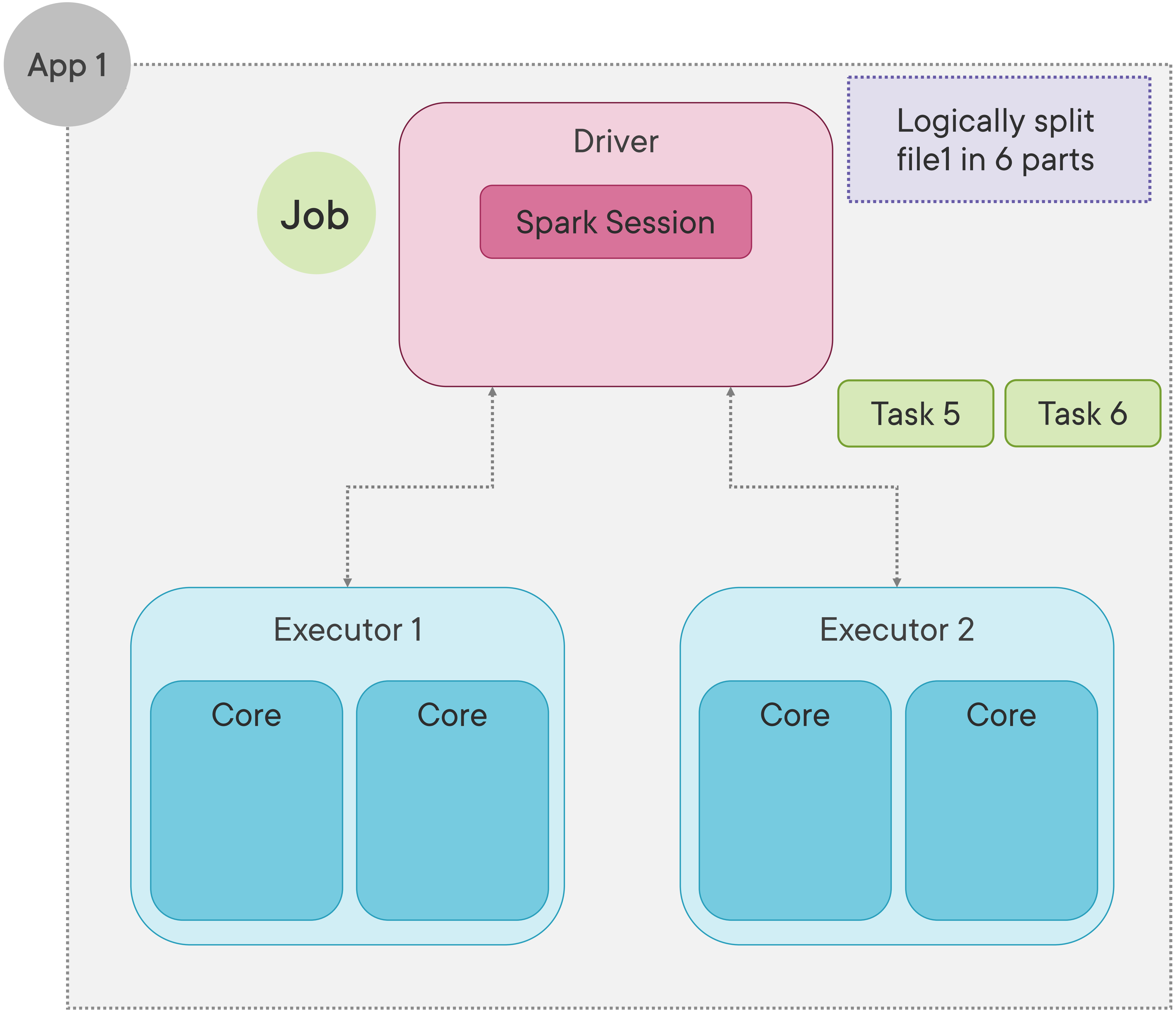




1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
App 1
Job
Driver
Spark Session
Logically split file1 in 6 parts
Task 5
Task 6
Executor 1
Core
Task 1
Part 1
Core
Task 2
Part 2
Executor 2
Core
Task 3
Part 3
Core
Task 4
Part 4
Part 5
Part 6
Storage

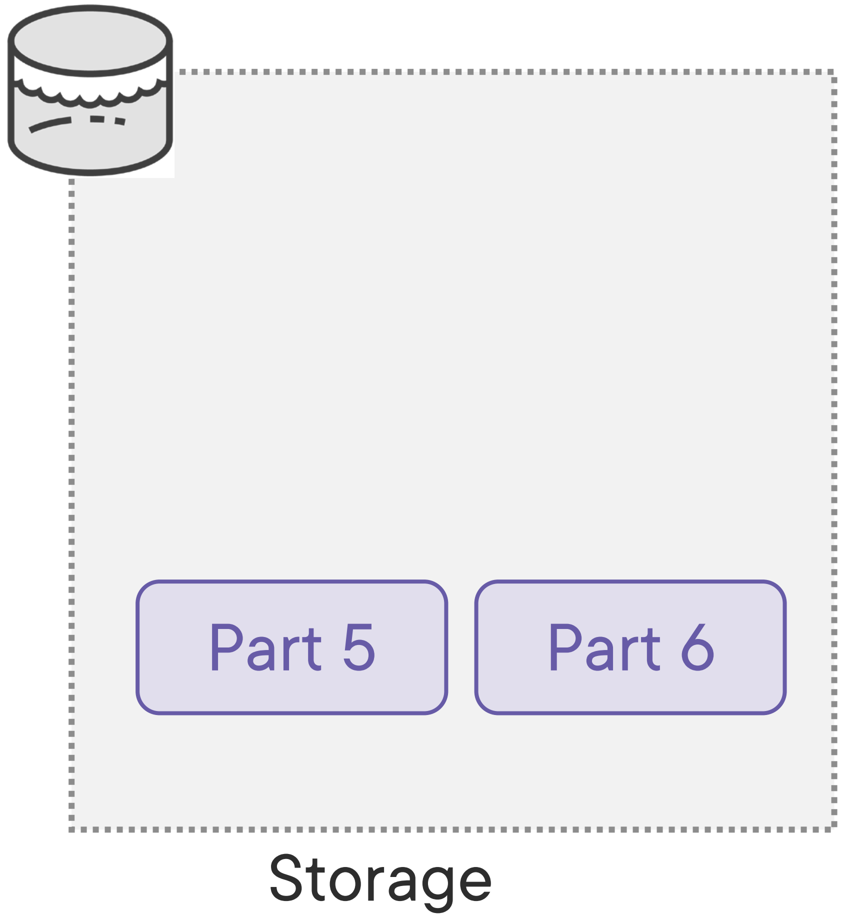
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

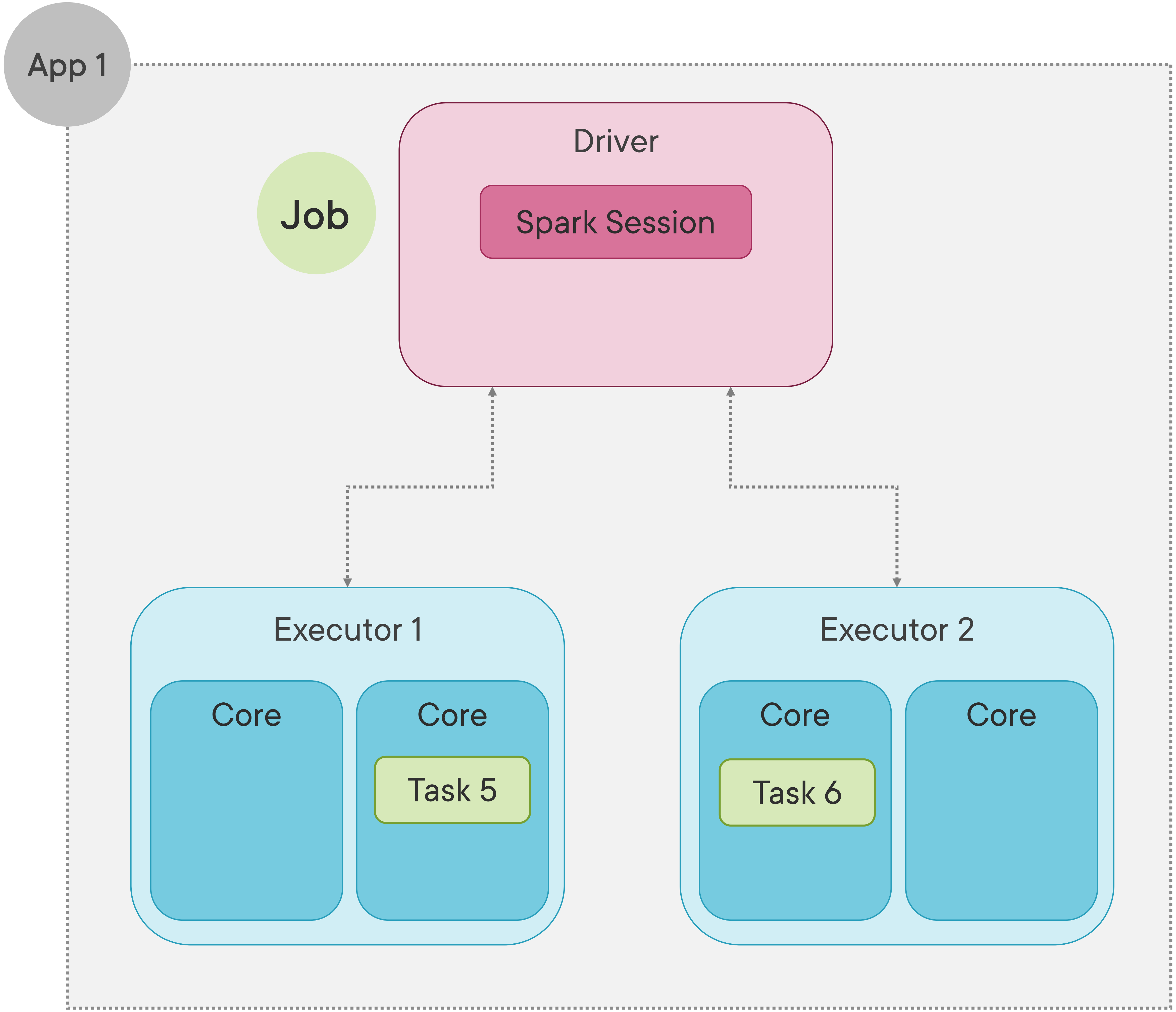



Storage

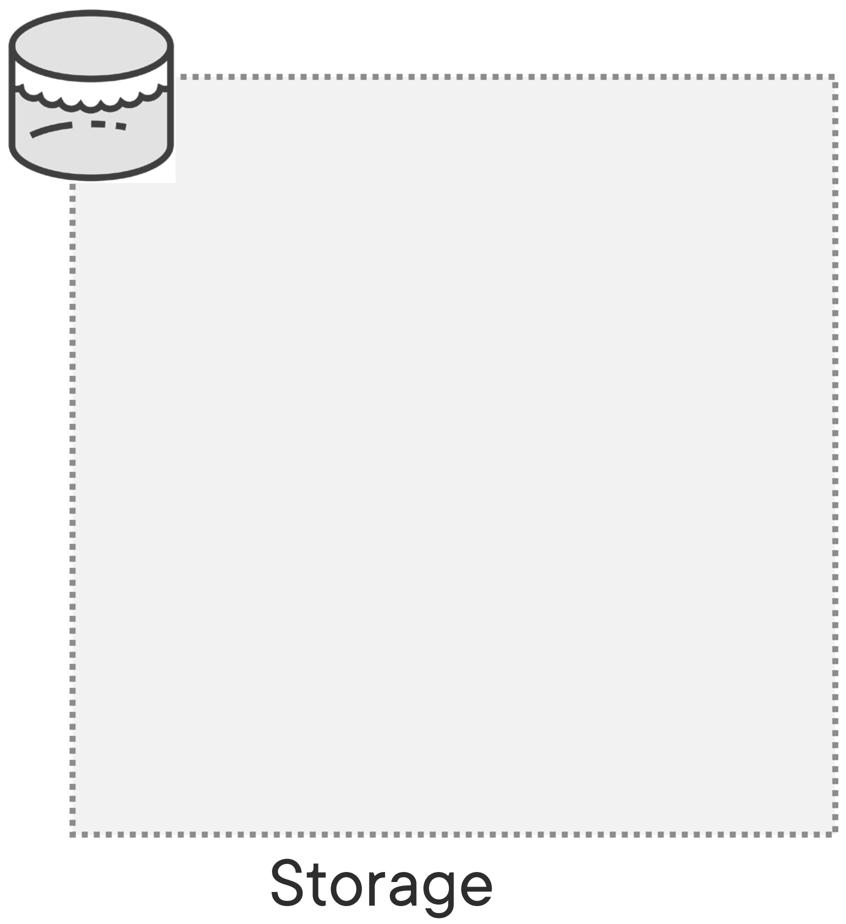


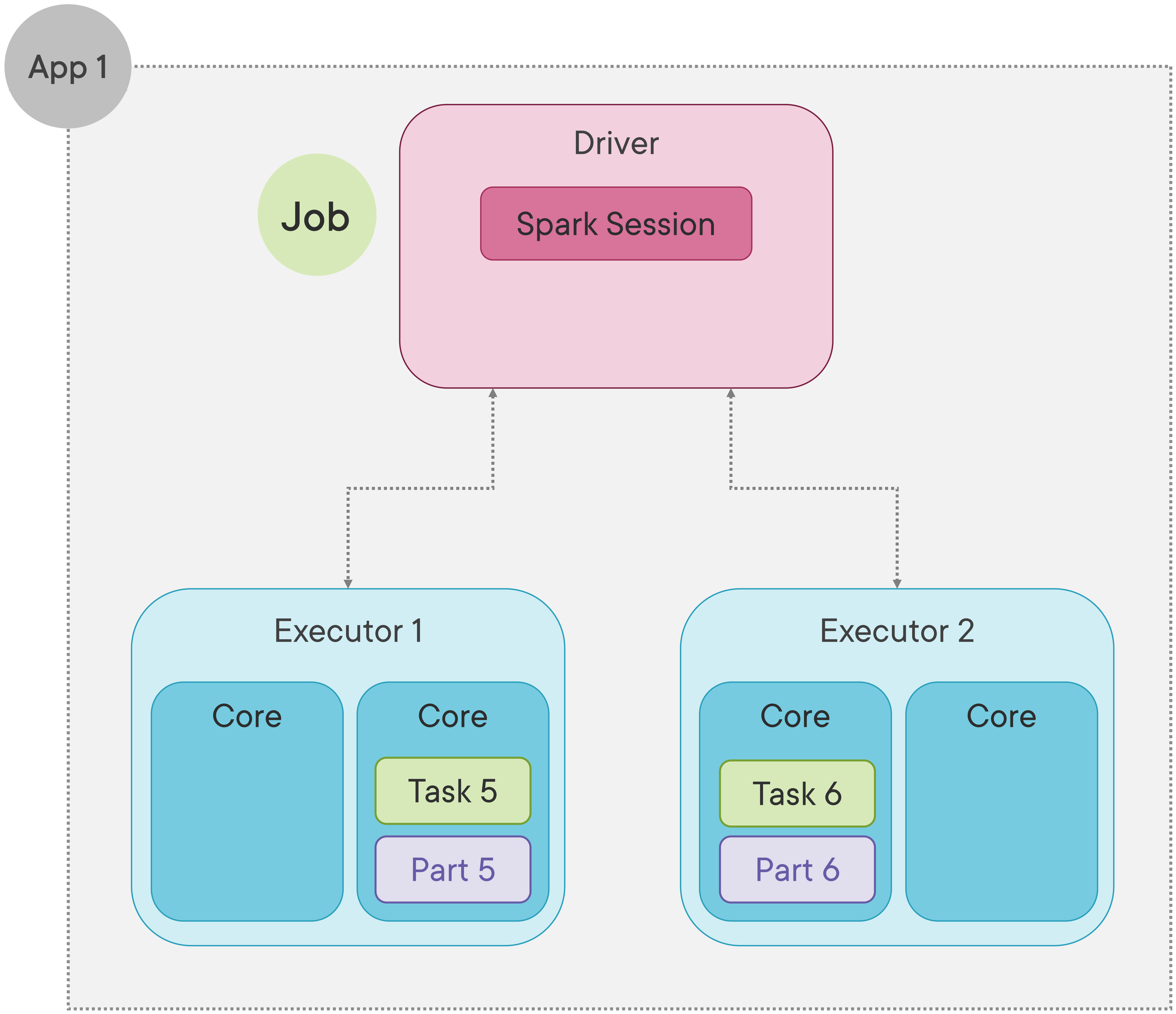
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



## To improve Parallelization

- Increase number of Executors
- Add more cores to each executor

But provisioning too many resources results in under utilization (wastage)

# Spark Application

**Job**

- Job is created when you need to execute code & take action (getting back results)

**Partitions**

- A partition is a chunk of data
- Driver decides how many partitions to be created
- Number of tasks = number of partitions
- Each task processes only one partition
- How many partitions to be created? How Spark decides that? Can we control? – we'll learn later!

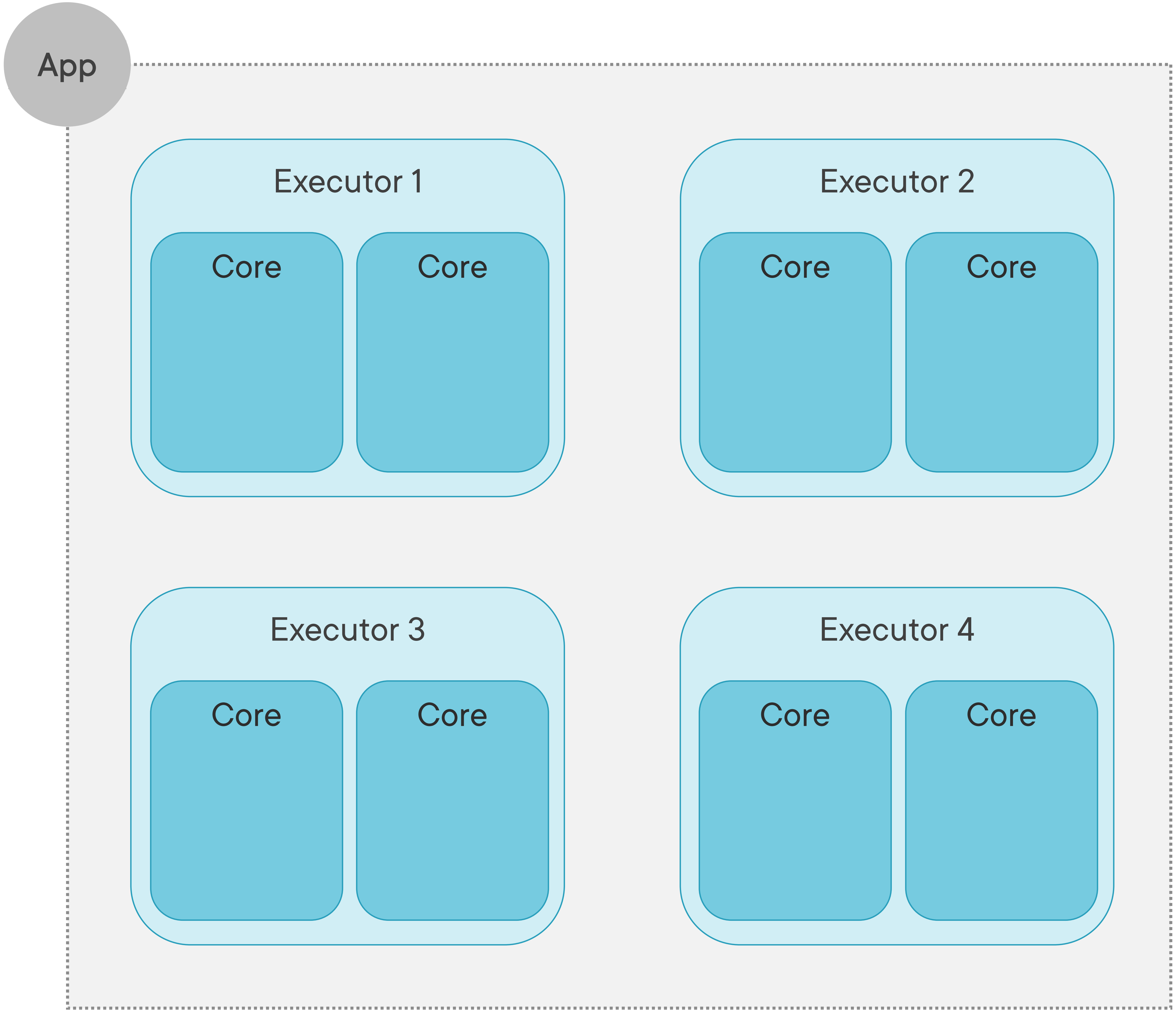
**Cores / Threads / Slots**

- Each core can execute only one task at a time
- Number of parallel tasks = Number of cores

Executor Size = 2 cores
14 GB RAM

Total Cores = 2 X 4
= 8 cores

Parallel Tasks = Total Cores
→ 8 tasks can execute in parallel



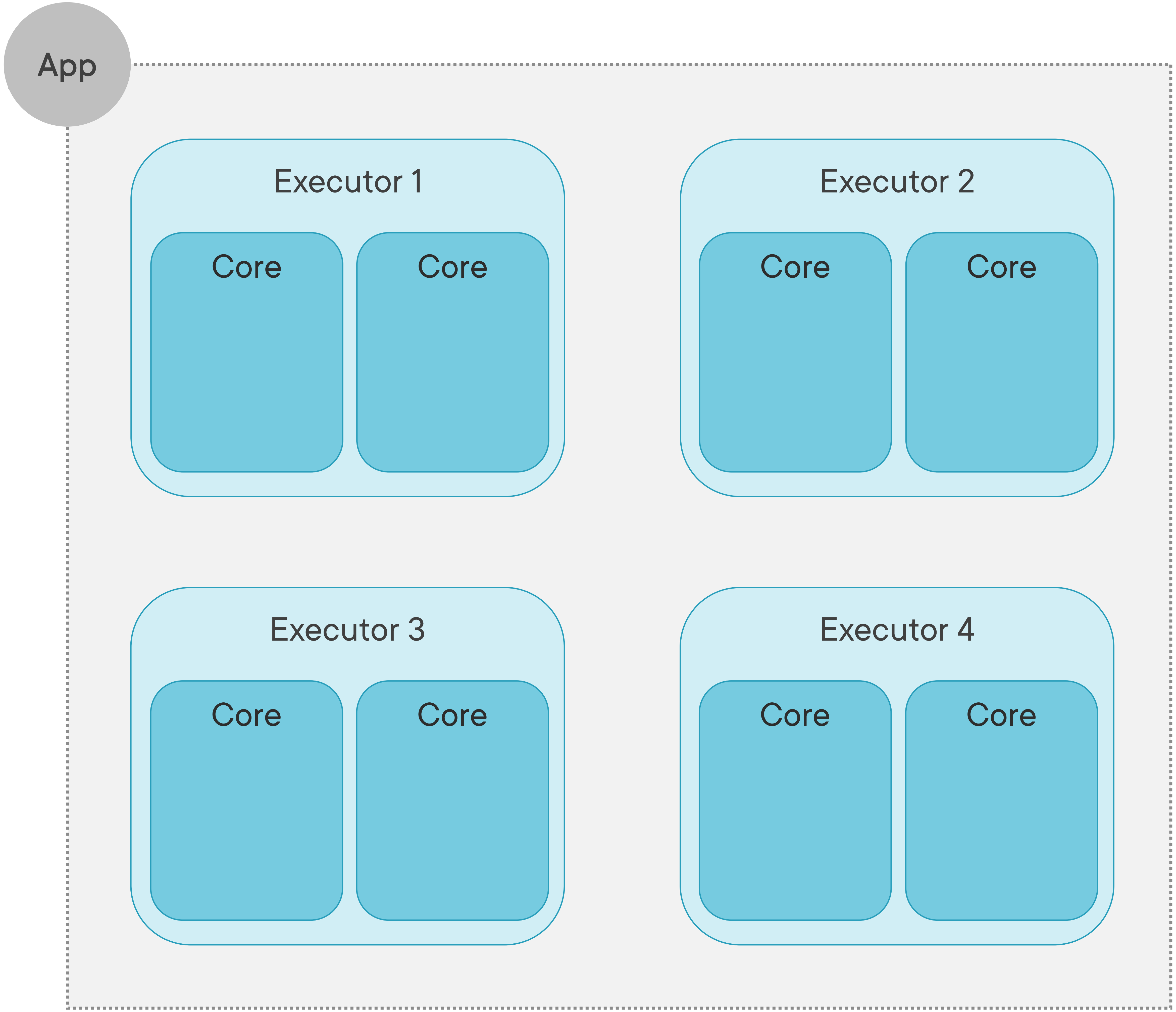
Partitions created by Driver = 16

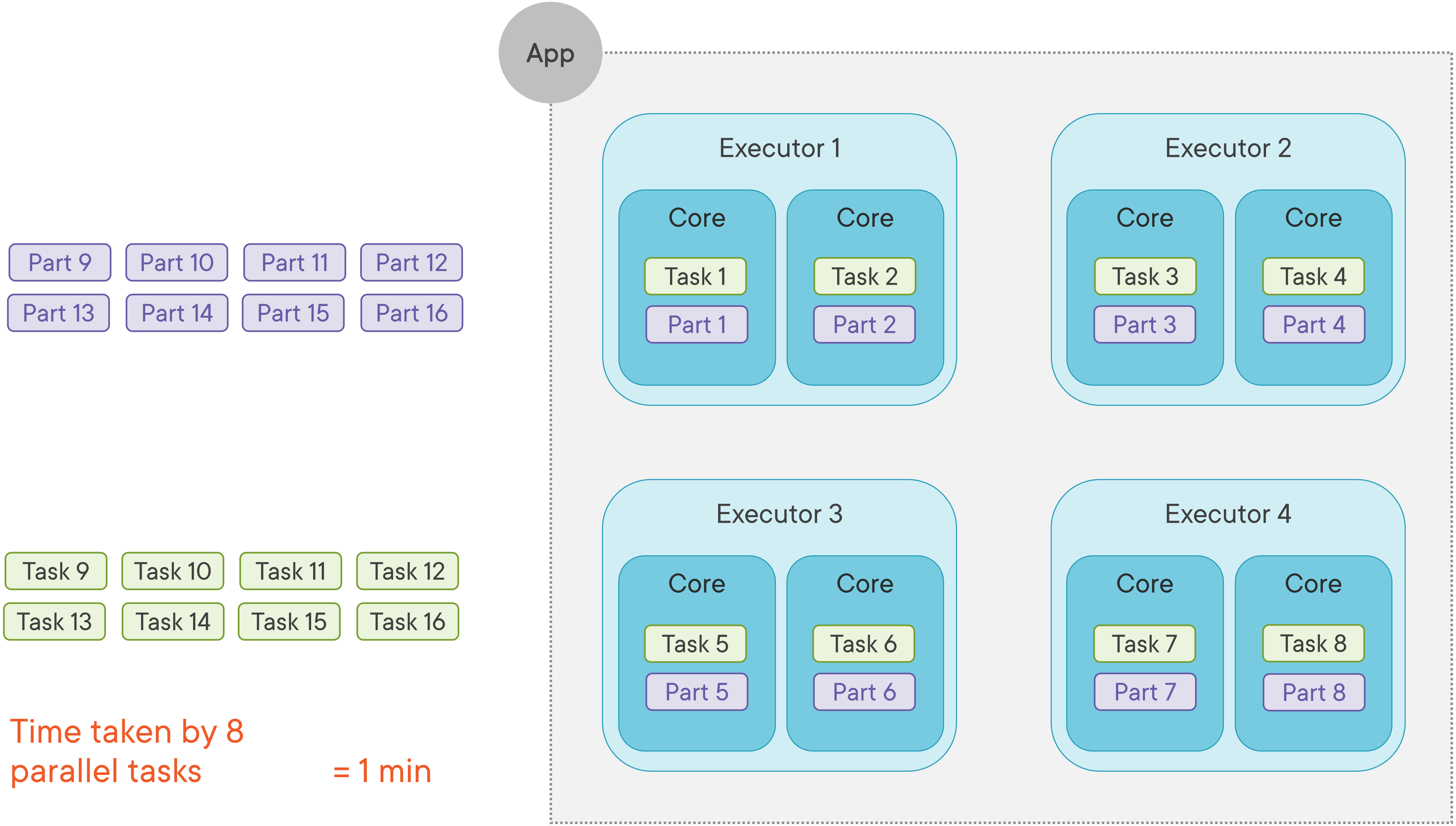
| | | | |
|---|---|---|---|
| Part 1 | Part 2 | Part 3 | Part 4 |
| Part 5 | Part 6 | Part 7 | Part 8 |
| Part 9 | Part 10 | Part 11 | Part 12 |
| Part 13 | Part 14 | Part 15 | Part 16 |

Total Tasks = Total Partitions

| | | | |
|---|---|---|---|
| Task 1 | Task 2 | Task 3 | Task 4 |
| Task 5 | Task 6 | Task 7 | Task 8 |
| Task 9 | Task 10 | Task 11 | Task 12 |
| Task 13 | Task 14 | Task 15 | Task 16 |

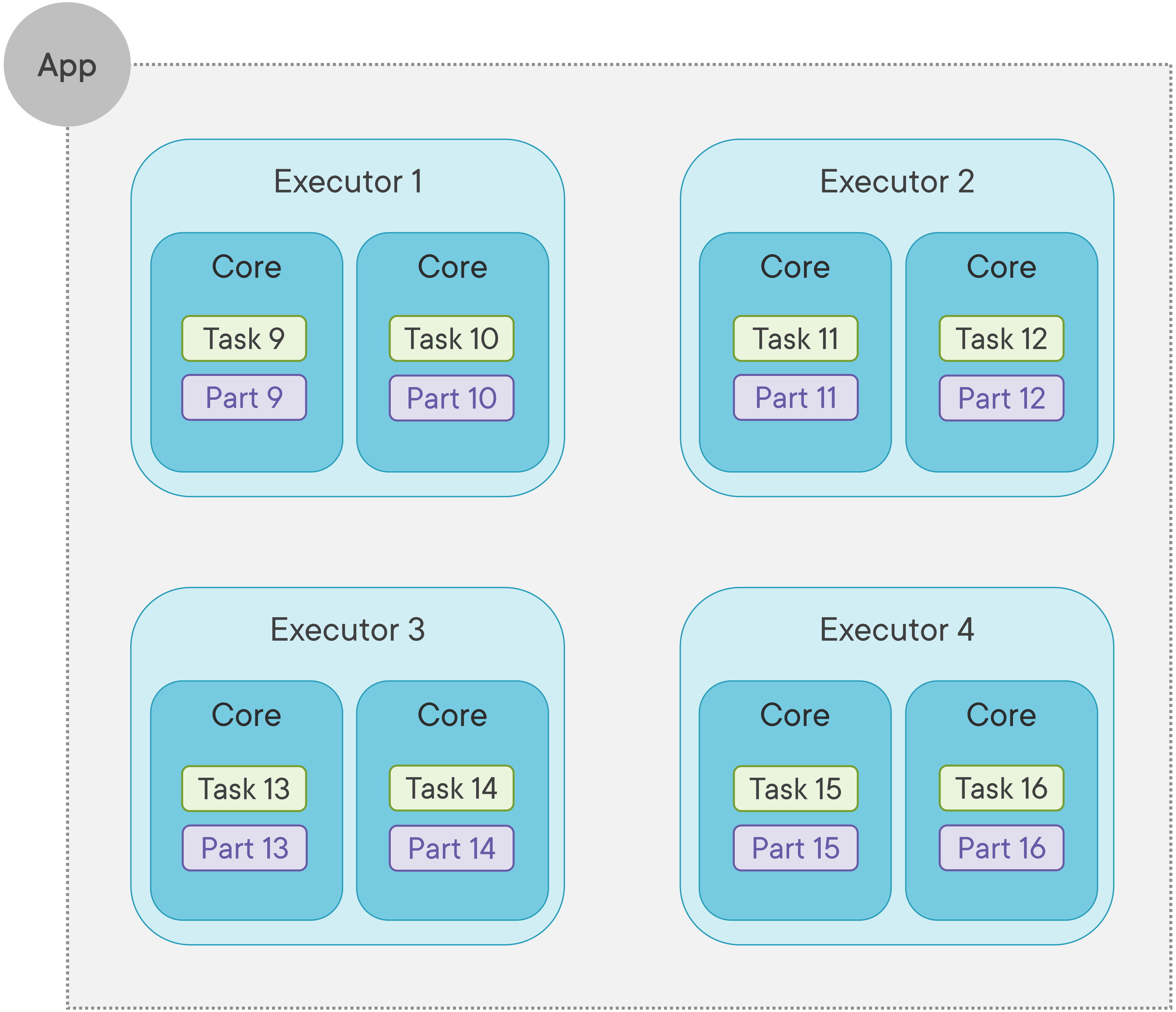
Time to complete task = 1 min






Time taken by 8 parallel tasks = 1 min

Total time taken
by 16 tasks = 2 mins



# Spark APIs – RDDs, DataFrames & Datasets

# Spark APIs

Resilient Distributed Datasets (RDDs)

DataFrames

Datasets

# Resilient Distributed Datasets

**Introduced with inception of Spark**

**Native data structure (Spark Core library)**

**RDD represents collection of data in memory**

- Ex - When file is loaded in memory, it is called RDD

**All processing in Spark happens on RDDs**

**Write code using low-level RDD APIs**

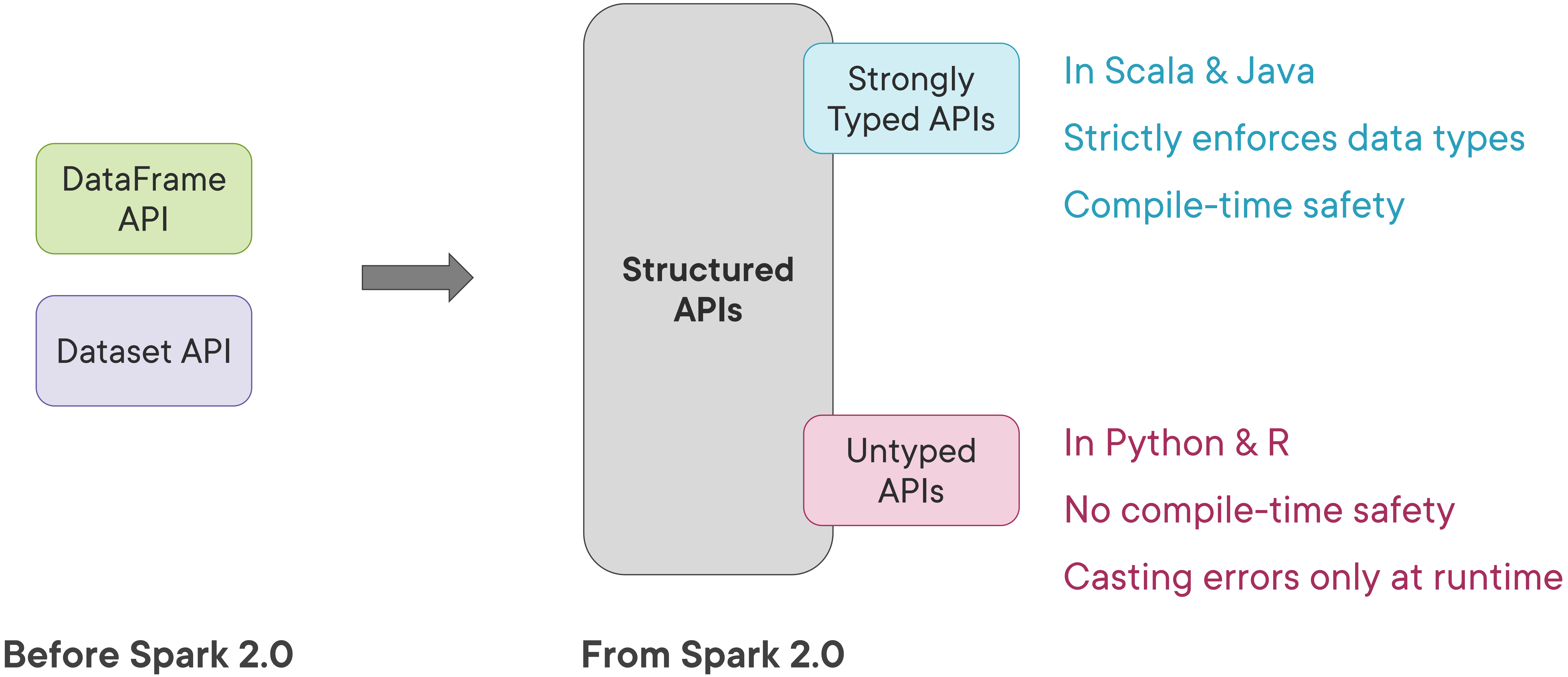
- APIs load data in memory as RDDs & process them
- All languages supported – Scala, Java, R & Python

**Spark does not apply any optimization to RDD code**

# Higher-Level Spark APIs

| DataFrames | Datasets |
| --- | --- |
| Introduced with Spark 1.3 | Introduced with Spark 1.6 |
| Based on RDDs | Based on RDDs |
| Collection of data, but in tabular format | Combination of RDDs and DataFrames |
| No compile-time safety | Provides compile-time safety |
| Spark applies optimizations to code | Spark applies optimizations to code |
| Supported in all languages | Supported in Java & Scala |

# Higher-Level Spark APIs



Since we are going to use PySpark, we'll work with Untyped APIs (generally referred as DataFrames)

# Summary

**Background of Apache Spark**

- Spark fits the bill for modern data processing needs
- Overcomes challenges faced by Hadoop

**In-memory analytics engine that runs on a cluster**

- Spark Core & Engine performs distributed processing
- Built-in libraries for various uses cases – batch processing, streaming, ML, graph computations
- Uses cluster manager & HDFS

**Distributed execution architecture**

- Spark application creates Driver & Executors
- Driver creates a Spark Session
- Cluster Manager handles allocation of resources
- Data is divided into Partitions
- Task is allocated to a core to process a data partition

**Supports 3 APIs – RDDs, DataFrames & Datasets**

# Up Next:
# Setting up Spark Environment