Getting Started with Apache Spark

Need for Apache Spark

- Two to three decades back, most organizations were only working with structured data sources.

- We used to extract data from them using extract, transform, and load, or ETL tools. Clean and transform the data, and then load it into a data warehouse.

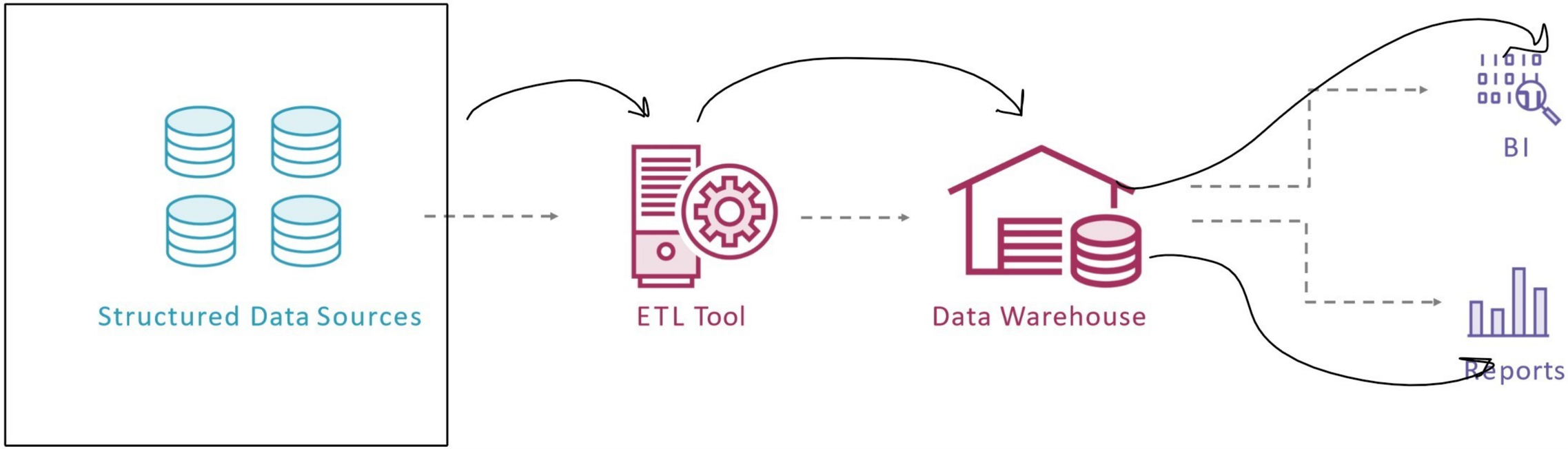
- So our data warehouse contains highly curated and structured data.

- From here, it was used for self- service BI and for reporting purposes.

- While this system worked well earlier, but the problem is that now structured data sources only account for 20% of organizations' data

structured / unstructured?

SAP HANA


Structured Data Sources
ETL Tool
Data Warehouse
BI
Reports

There is data available in a variety of file formats, like CSV, JSON, and log files. There is data from NoSQL databases.

There are audio and video files, and then there is streaming data as well. And this accounts for 80% of organizations' data.

So there has been an exponential growth, not just in types of data sources, but in data volumes as well.

To handle this, you also need to scale your infrastructure. While you may add more resources like CPU and RAM to your existing servers, it cannot be scaled up beyond a limit.

And finally, data pipelines have to quickly adapt to support other use cases, like machine learning and data science

5 V's of Big Data

Volume → Scale of Data

Variety → Different forms of data

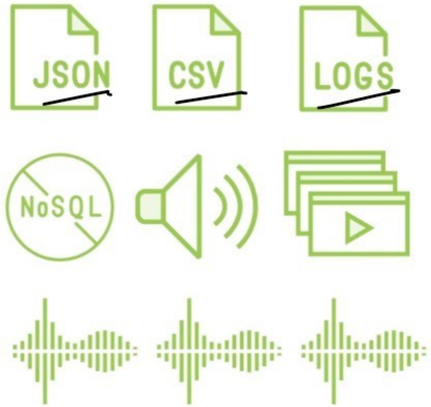
Velocity → Speed → Stock Market, Social → Streaming Applications

Value → generating a value from data

Veracity → eliminate uncertainty in data

→ Validation scenario

Structured Data Sources



Unstructured &Streaming Data Sources

Exponential Data Growth

ETL Tool

Data Warehouse

? No SQL → mongo DB
JSON -
Parquet

Infrastructure Scalability

BI

Reports

Machine Learning

Data Science

Evolving Business Needs

History

Apache Hadoop ?

Earlier

ETL Tools / RDBMS

2004

MapReduce / Distributed File System

Paper published by Google for distributed data processing

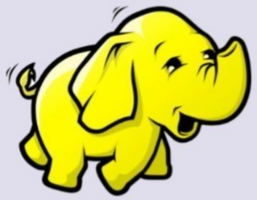
History

Earlier

ETL Tools / RDBMS

2006

Hadoop (MapReduce & HDFS)

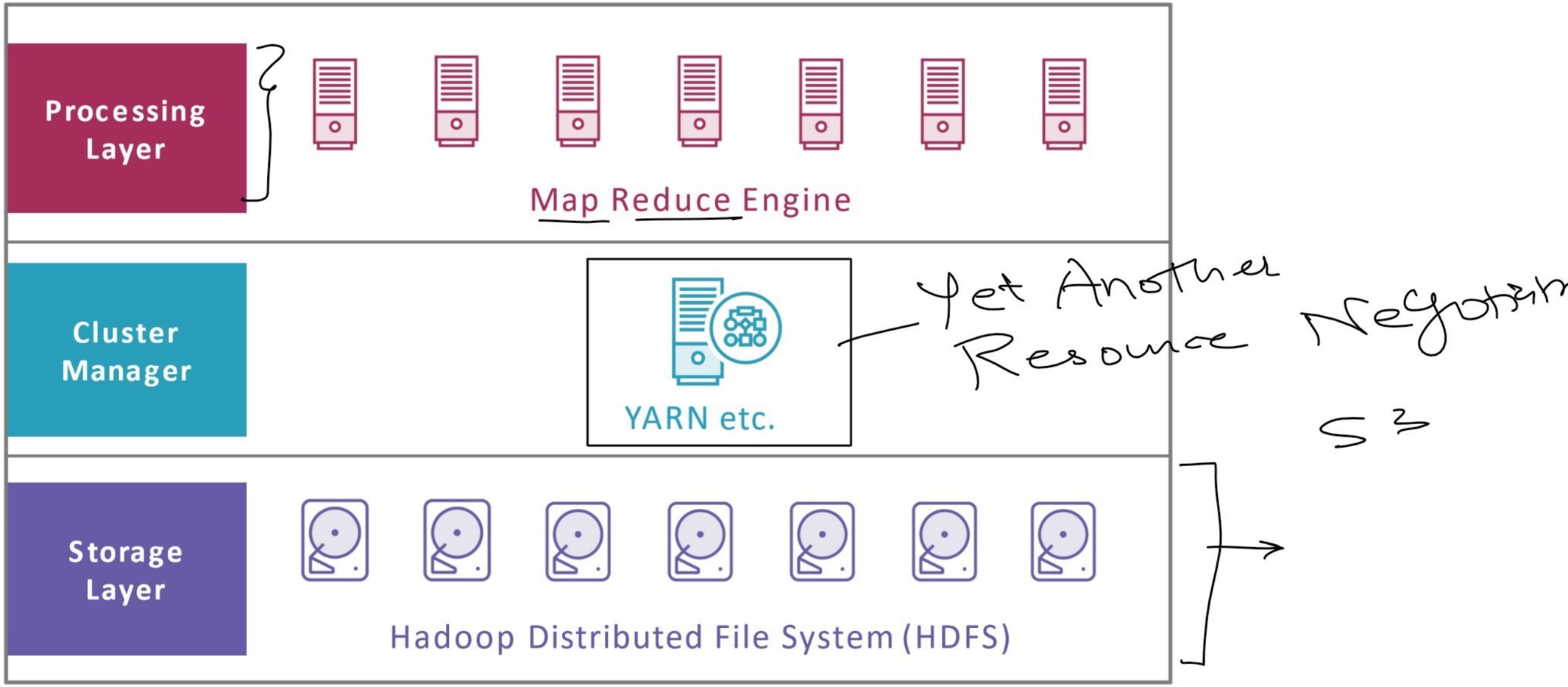


Framework for distributed storage & distributed data processing

2004

MapReduce / Distributed File System

Hadoop Components



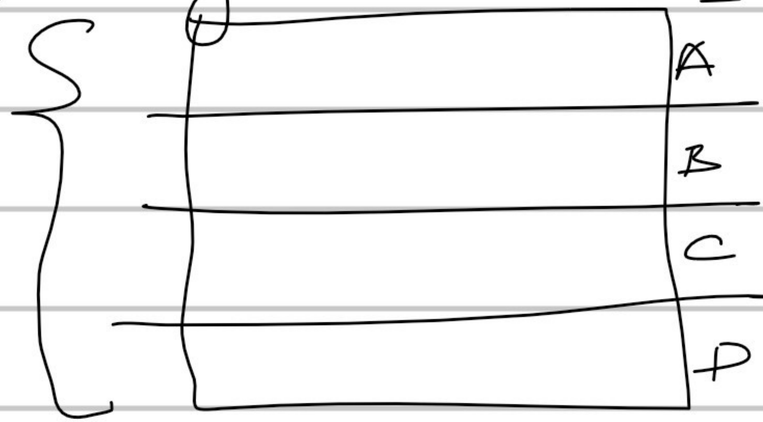
· Java / ——

dataset

part

Sectors

Survey · Dataset



population

A – 19K

B → 20K

C → 21K

D → 25K

Count

1 min

21 mins

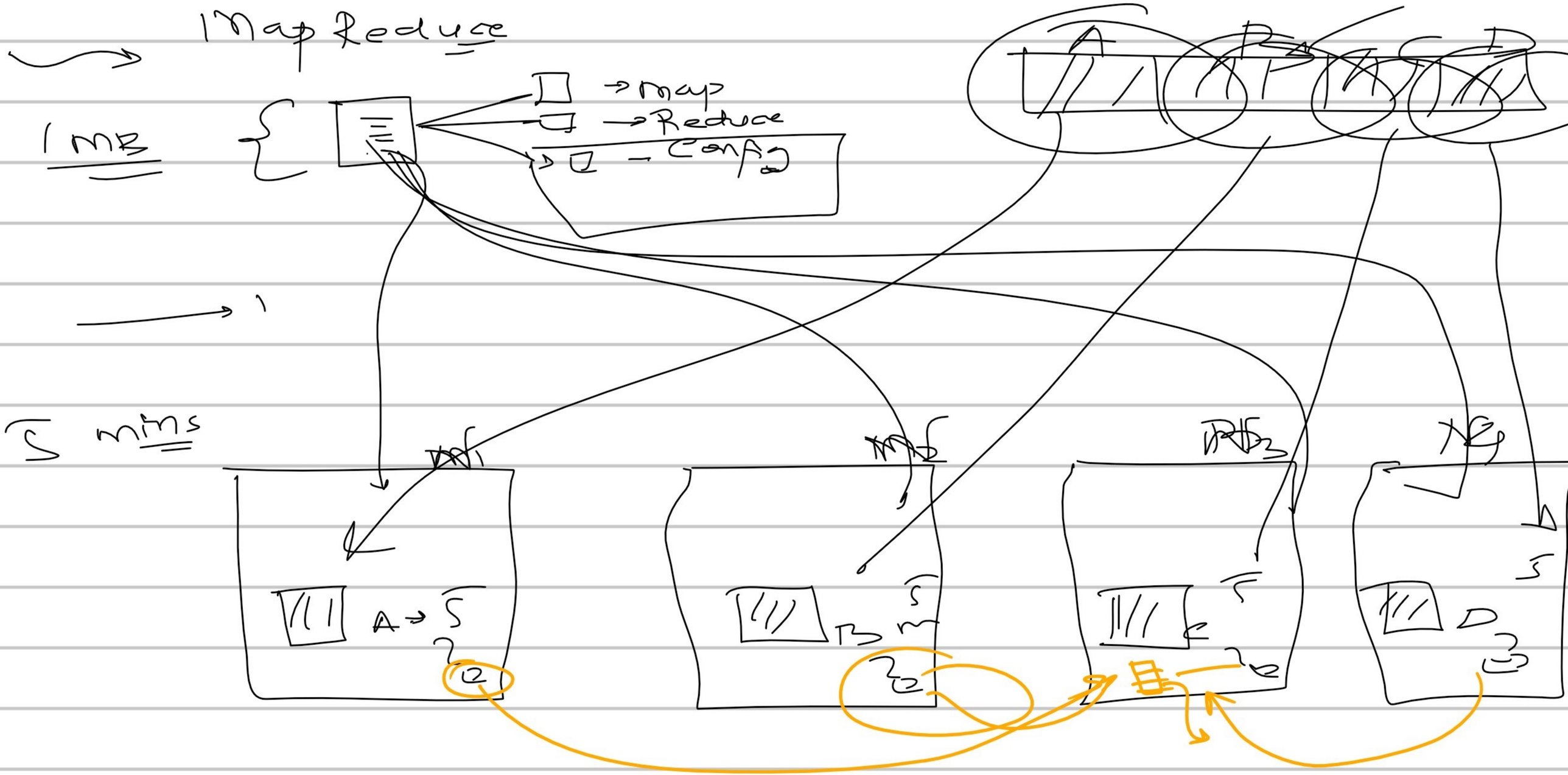
A → 5 mins

B → 5 mins

C · 5 mins

D · 5 mins

Map Reduce



HDFS

An HDFS, or Hadoop Distributed File System, is a distributed storage. This layer has multiple disks, and the files are stored here in a distributed manner. What does that mean? This means if you have a large file, it is split into multiple blocks, and the blocks are placed onto different disks. You can keep adding more hardware to virtually store unlimited data. So HDFS provides a scalable, fault tolerant, and low- cost solution to store the files.

MR

MapReduce engine is used for data processing. It can process structured, as well as unstructured data stored in HDFS. And this processing is done in parallel using multiple machines in the Hadoop cluster.

And finally, there is a cluster manager, like YARN, in short fo Yet Another Resource Negotiator. So cluster manager is responsible for managing and allocating the resources required for processing the data.

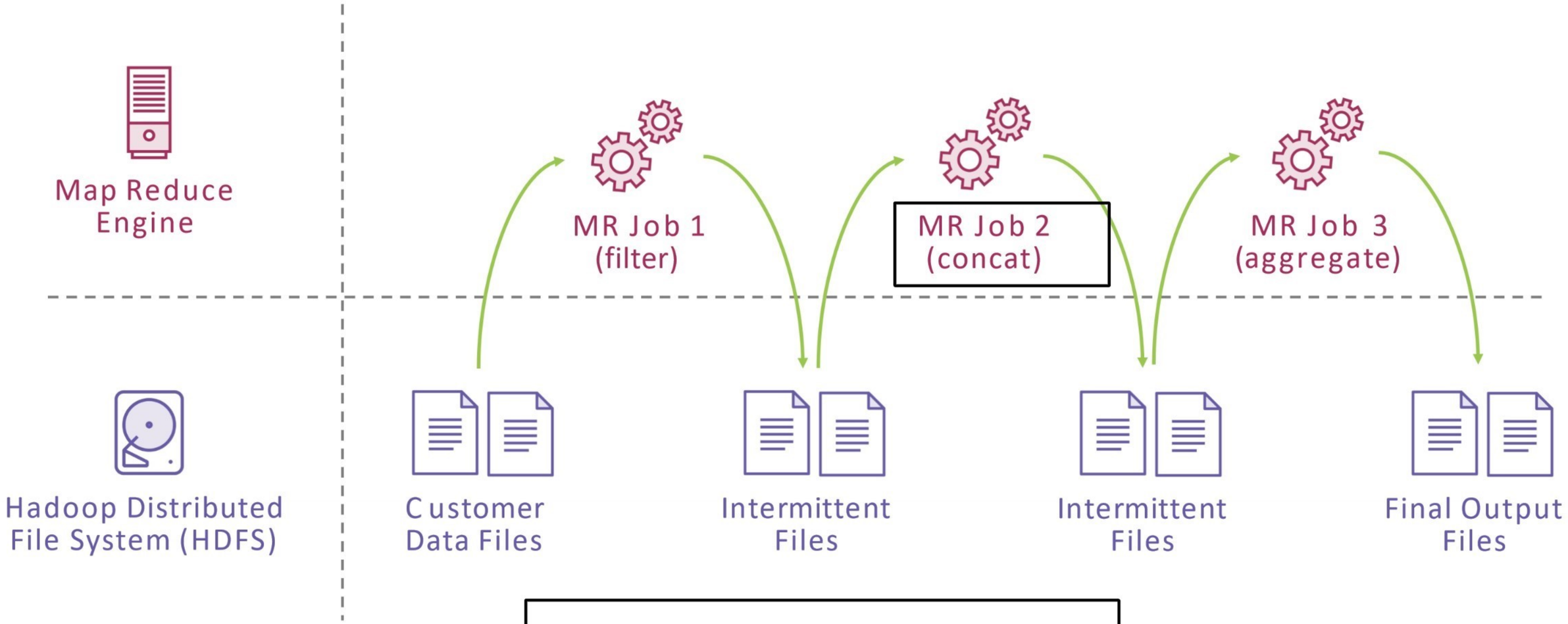
Let's say you have files stored in HDFS related to customers, and you want to filter customers who are new, combine their first and last names, and do some aggregates.

Now to process the data, you will write code that defines your MapReduce jobs.

So let's say the first job is to filter new customers. So MapReduce job will read the files and filter the data, and then write the intermittent output to HDFS.

Then for the next operation to combine first and last names, the second MapReduce job will read these intermittent files, apply the operation, and again write the partial output to disk.

How Hadoop Works?

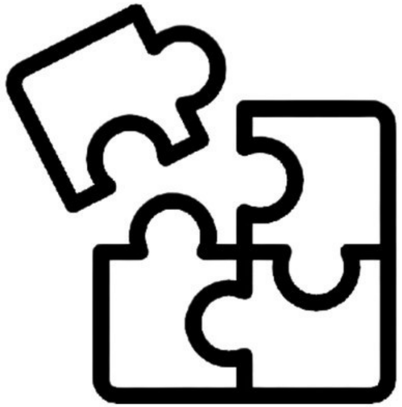


Disk-heavy Architecture

And for final operation, it will do the same and produce final output files. So if you carefully notice, each MapReduce job is reading and writing to disk.

This is what is known as disk- heavy architecture of Hadoop. So while Hadoop can process huge volumes of data, there are certain challenges as well.

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

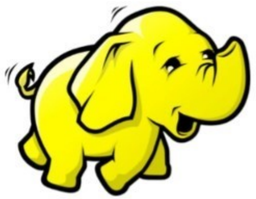
History

Earlier

ETL Tools / RDBMS

2006

Hadoop (MapReduce & HDFS)



2004

MapReduce / Distributed File System

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

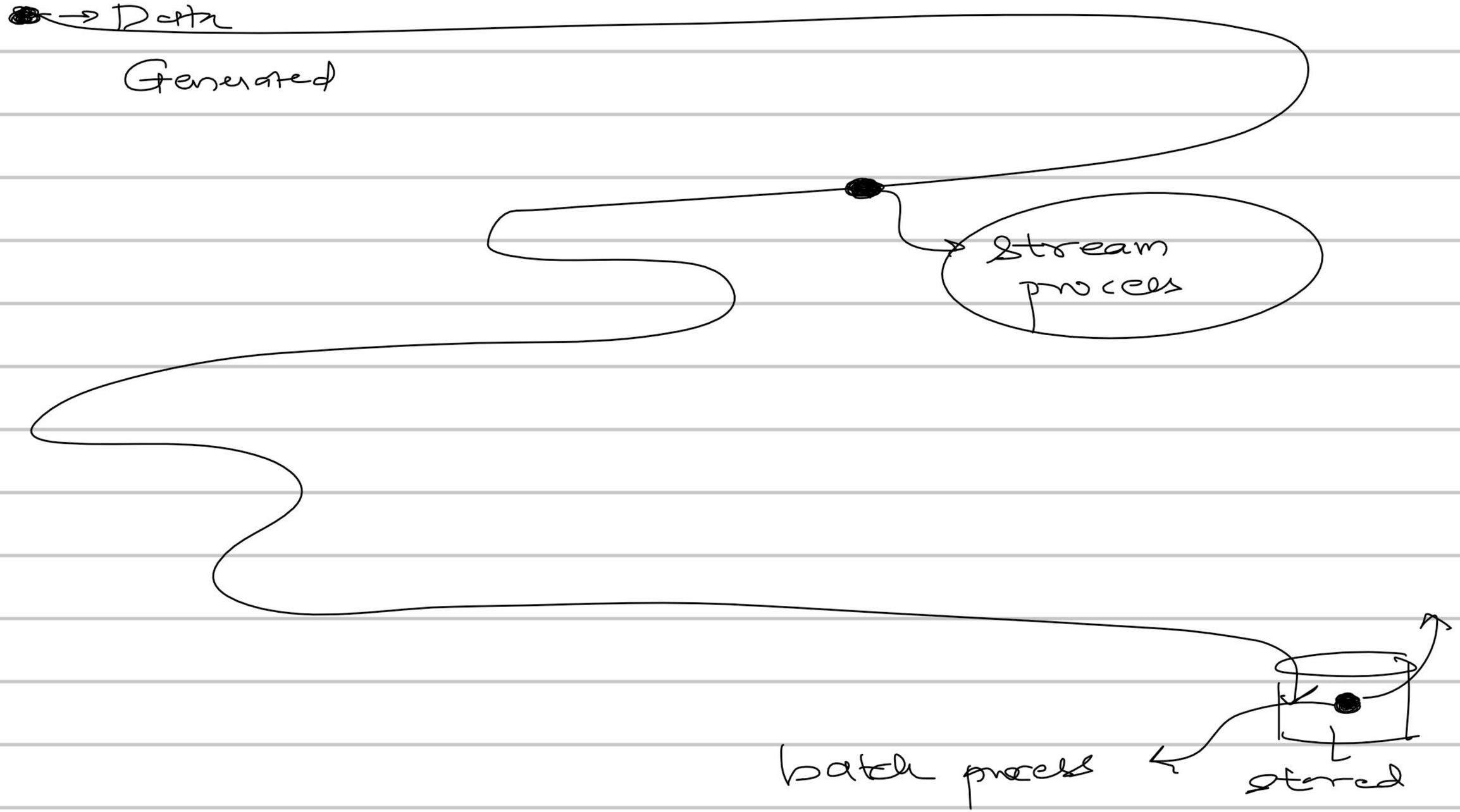
Scala

Dev → Scala
Java
Python
R

SQL

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

APL



Apache Hadoop

→ open source

→ Cloudera

→ Hortonworks

AWS, Azure, GCP

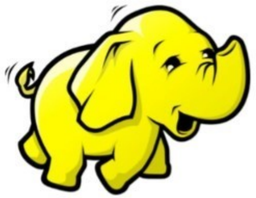
Apache Spark

History

Earlier
ETL Tools / RDBMS

2004
MapReduce / Distributed File System

2006
Hadoop (MapReduce & HDFS)



2009
Apache Spark

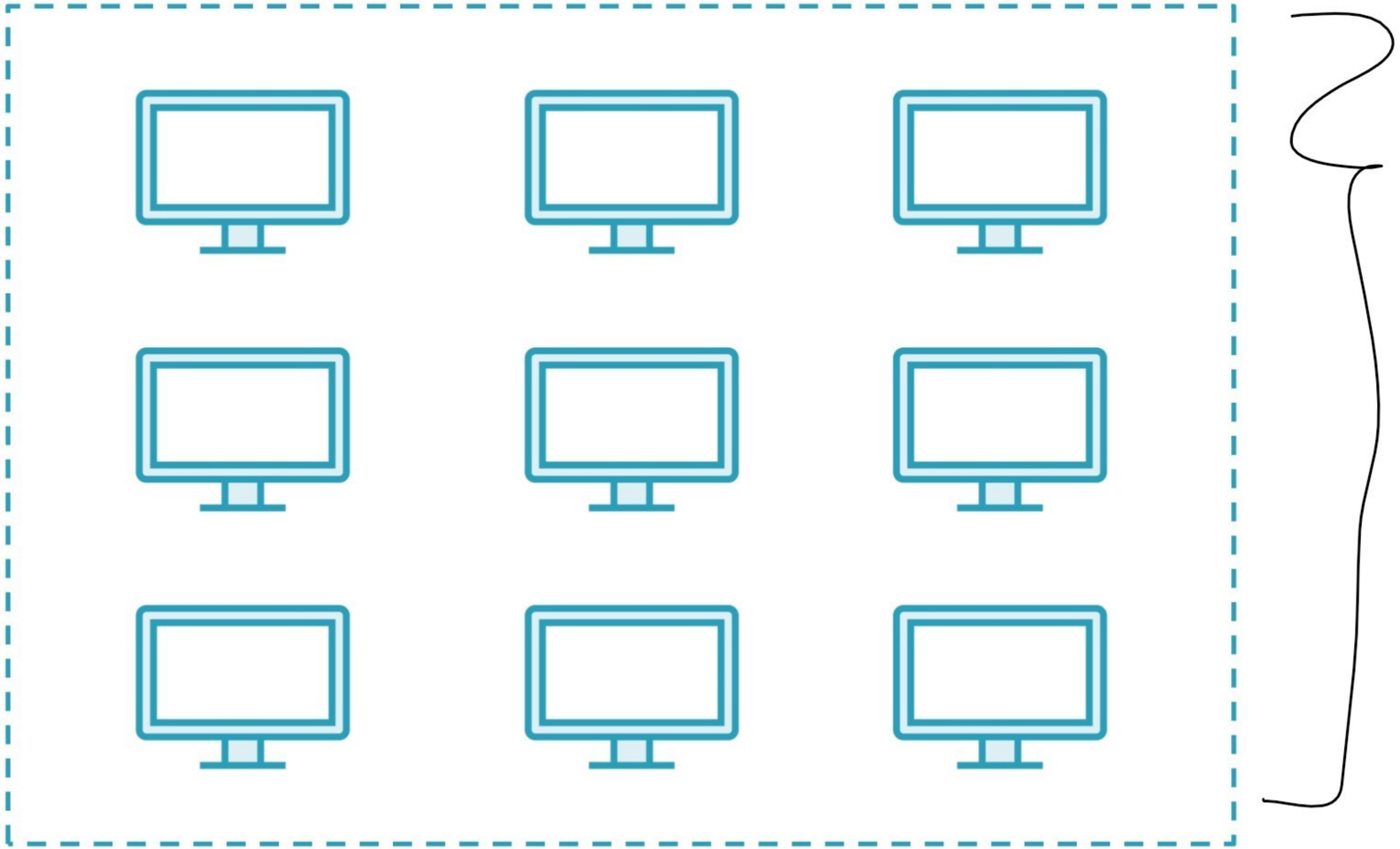
Now
Cloud-based Spark platforms

Databricks
Azure Synapse Analytics
Amazon EMR etc.

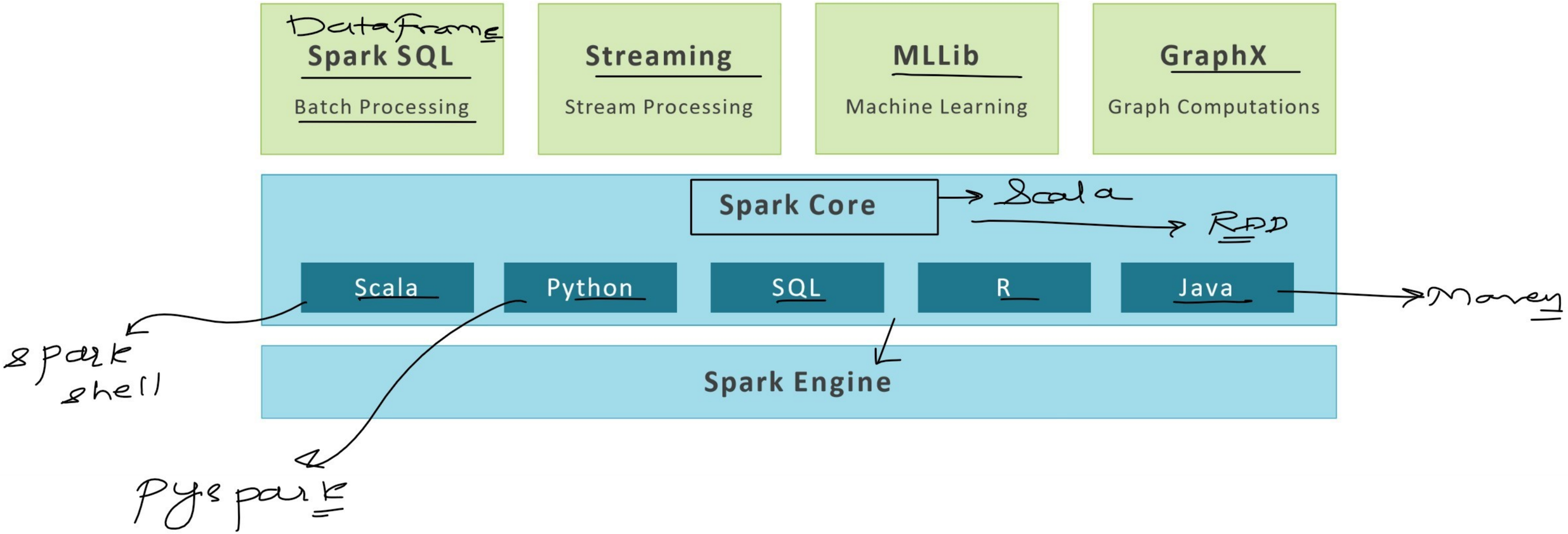
Understanding Spark Architecture & Ecosystem

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

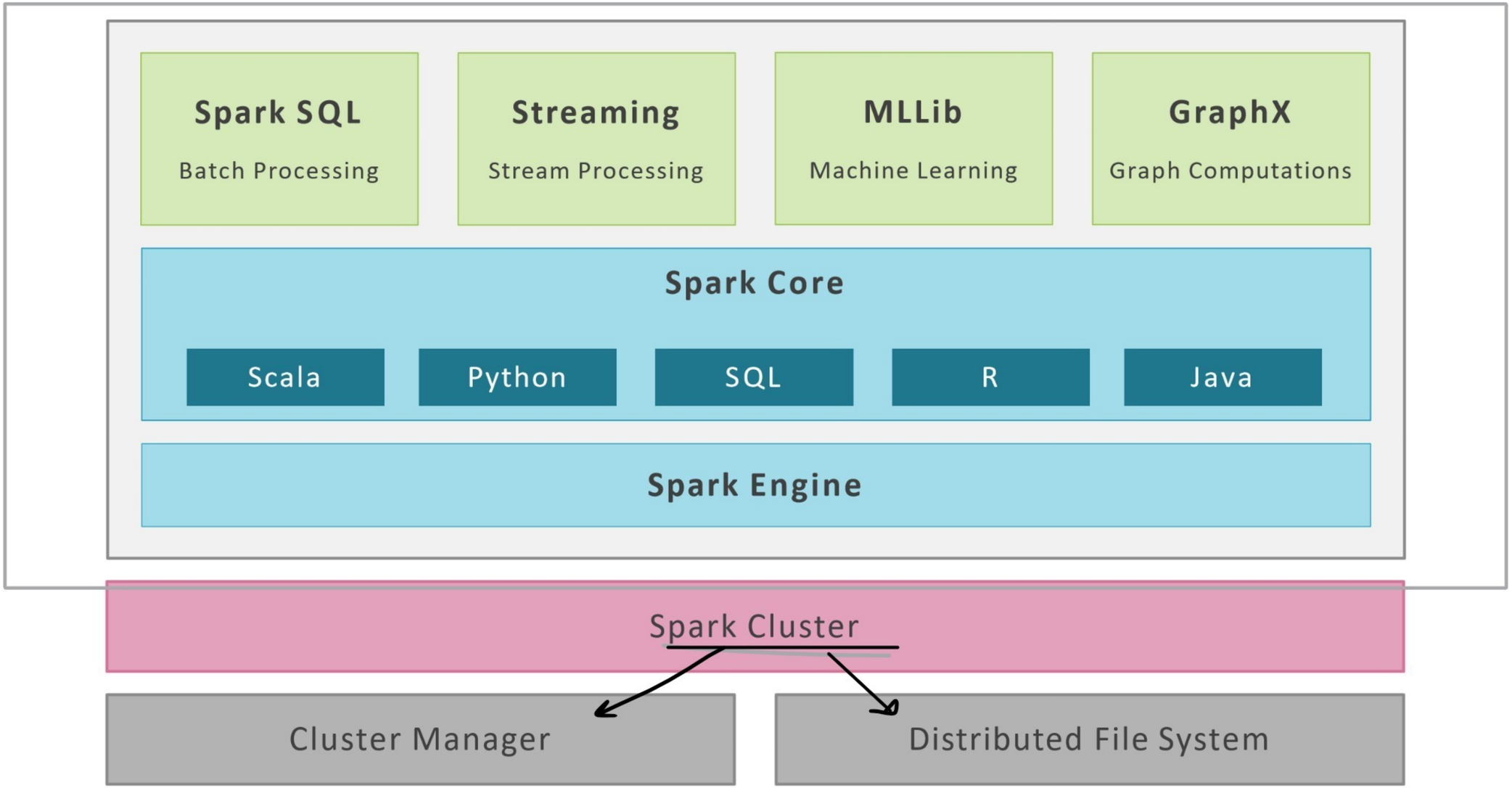
Spark Cluster



Spark Architecture



Spark Architecture



Spark Ecosystem

Cluster Managers

- YARN, Kubernetes, Mesos etc. , standalone

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?

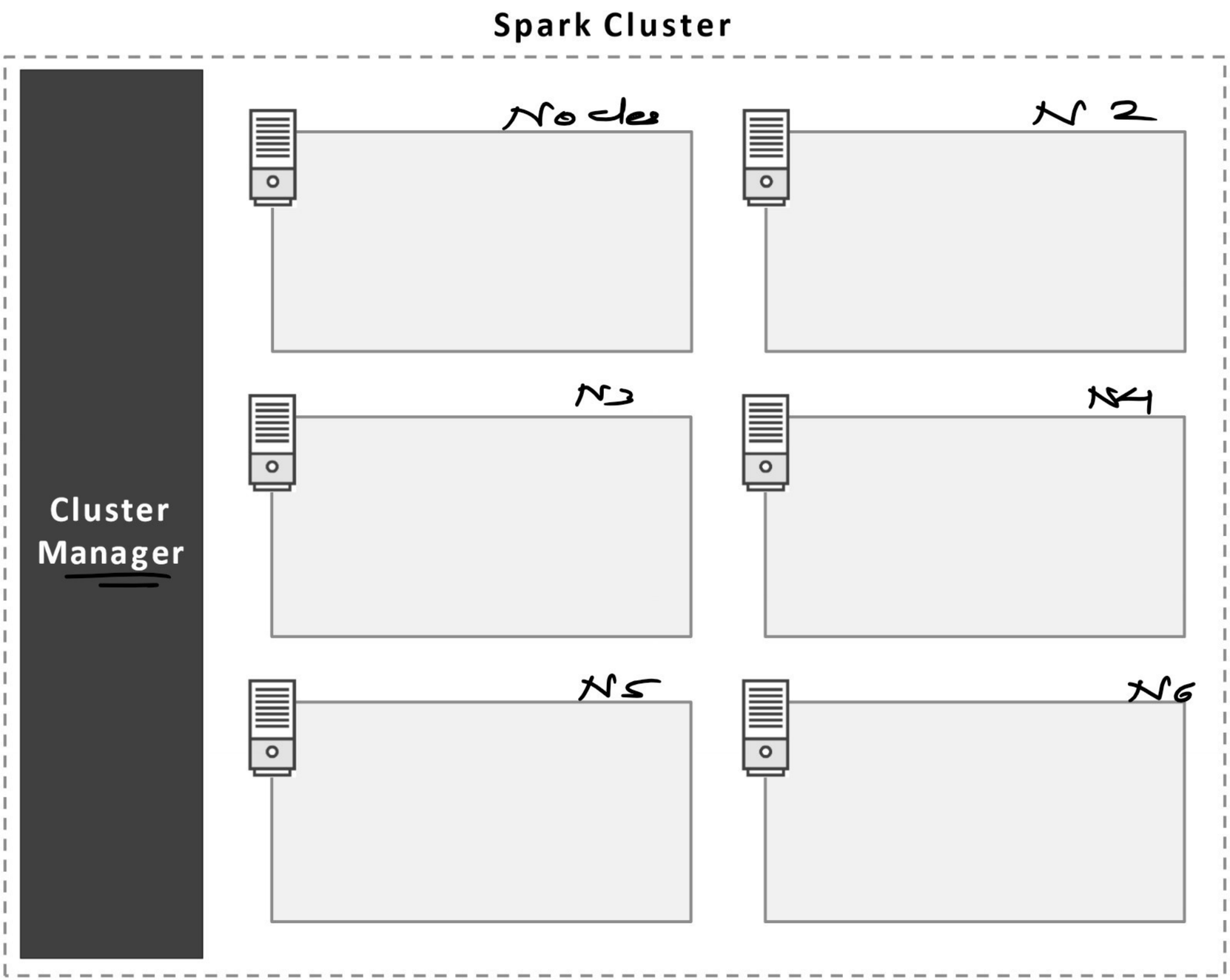
You can install Spark on a single machine, or you can install and run it together on multiple machines.

So a cluster is basically a group of machines or nodes. They may be virtual machines as well.

Then, it also has a cluster manager. The cluster manager allocates resources on the cluster to Spark applications.

Cluster is a group of machines / nodes

Cluster Manager allocates resources to Spark Applications on Cluster

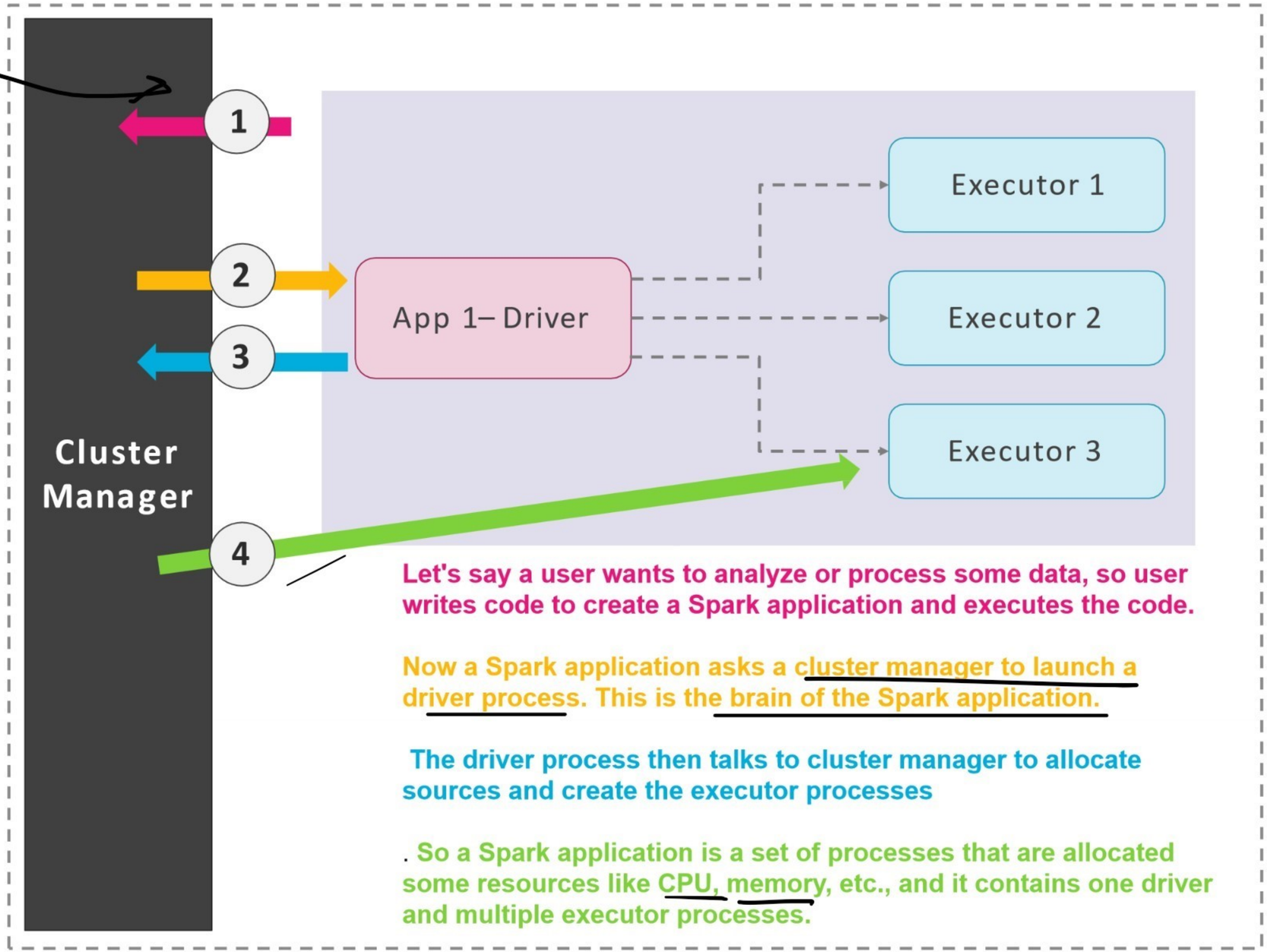


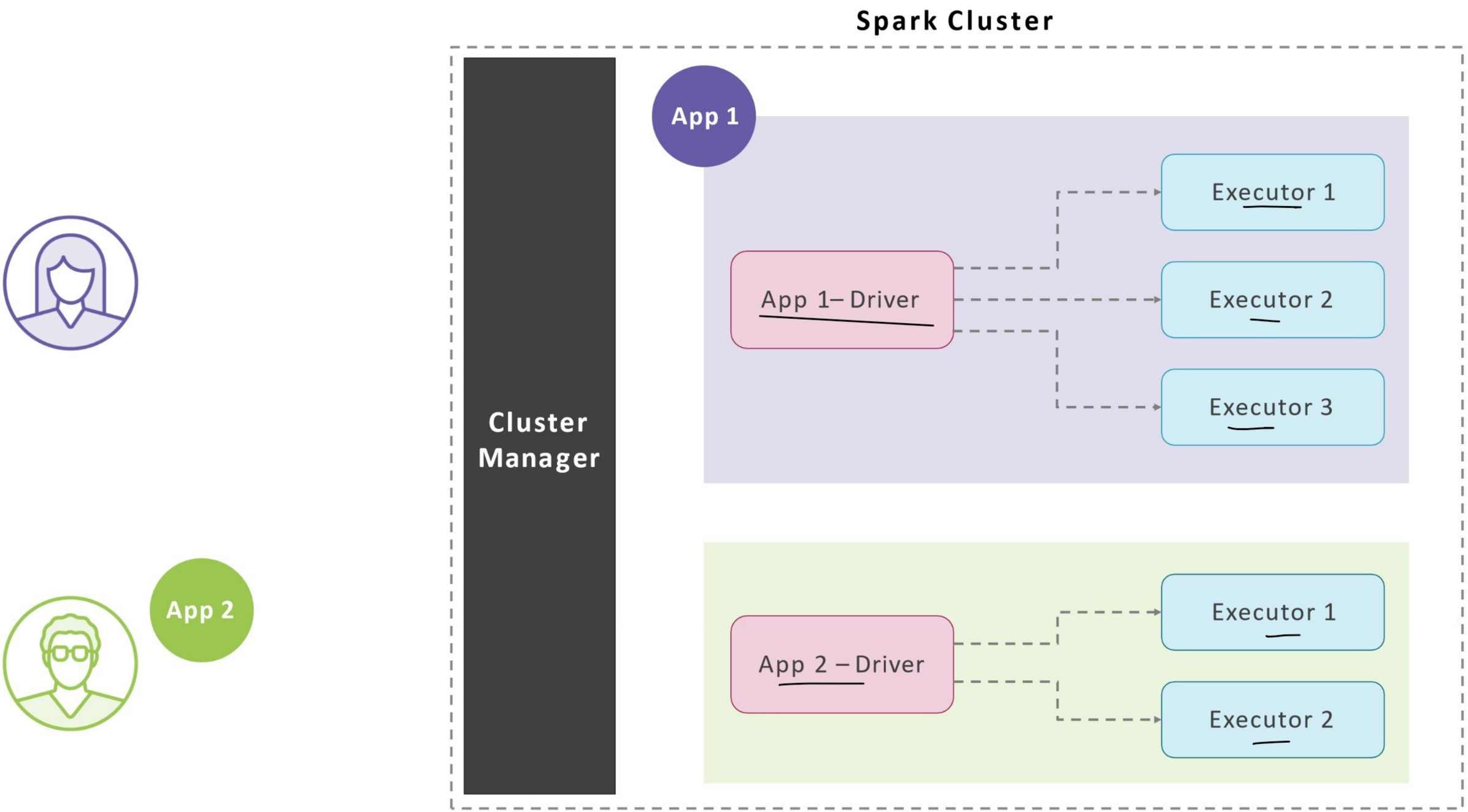
submit the spark code

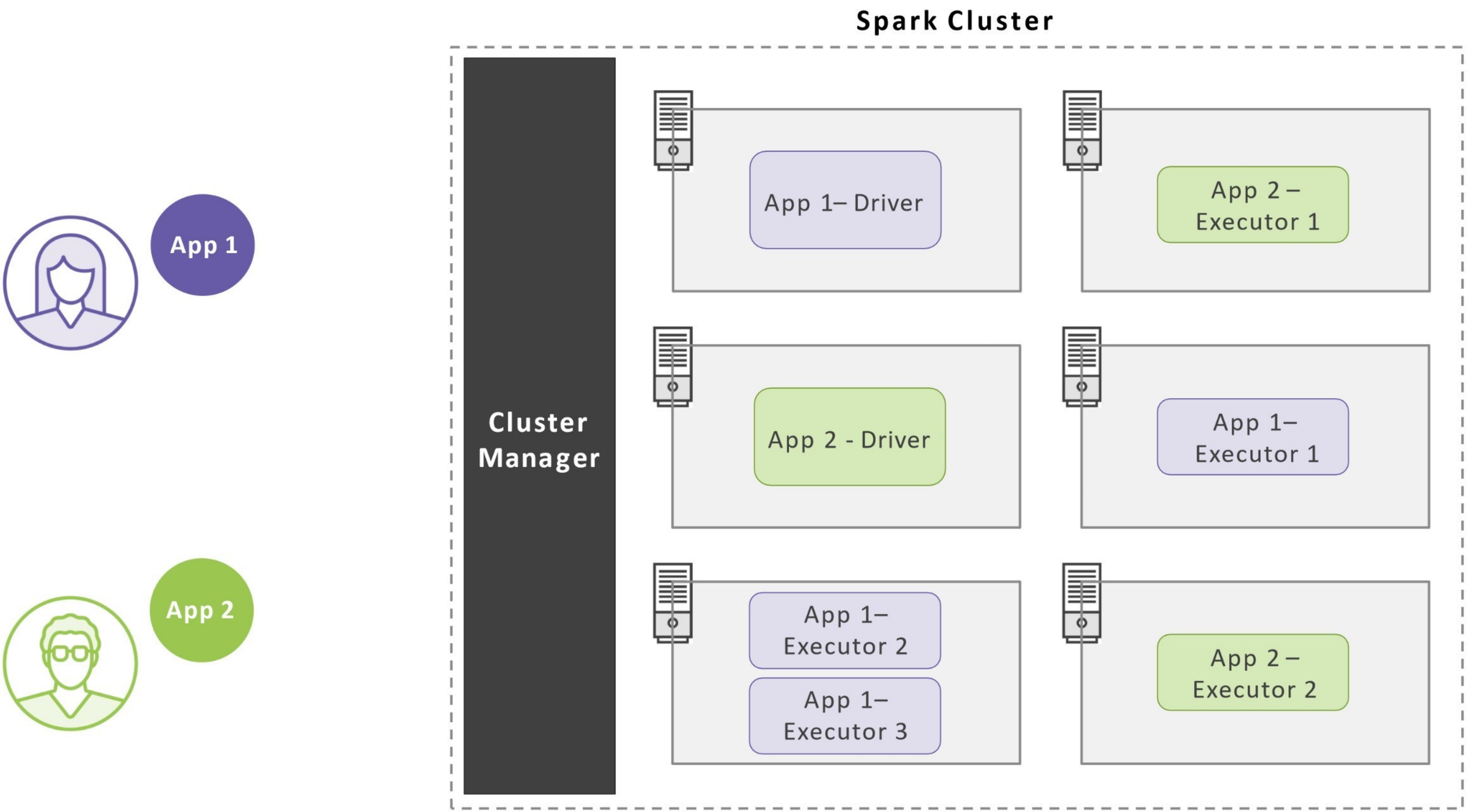
App 1

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

Driver takes input from user

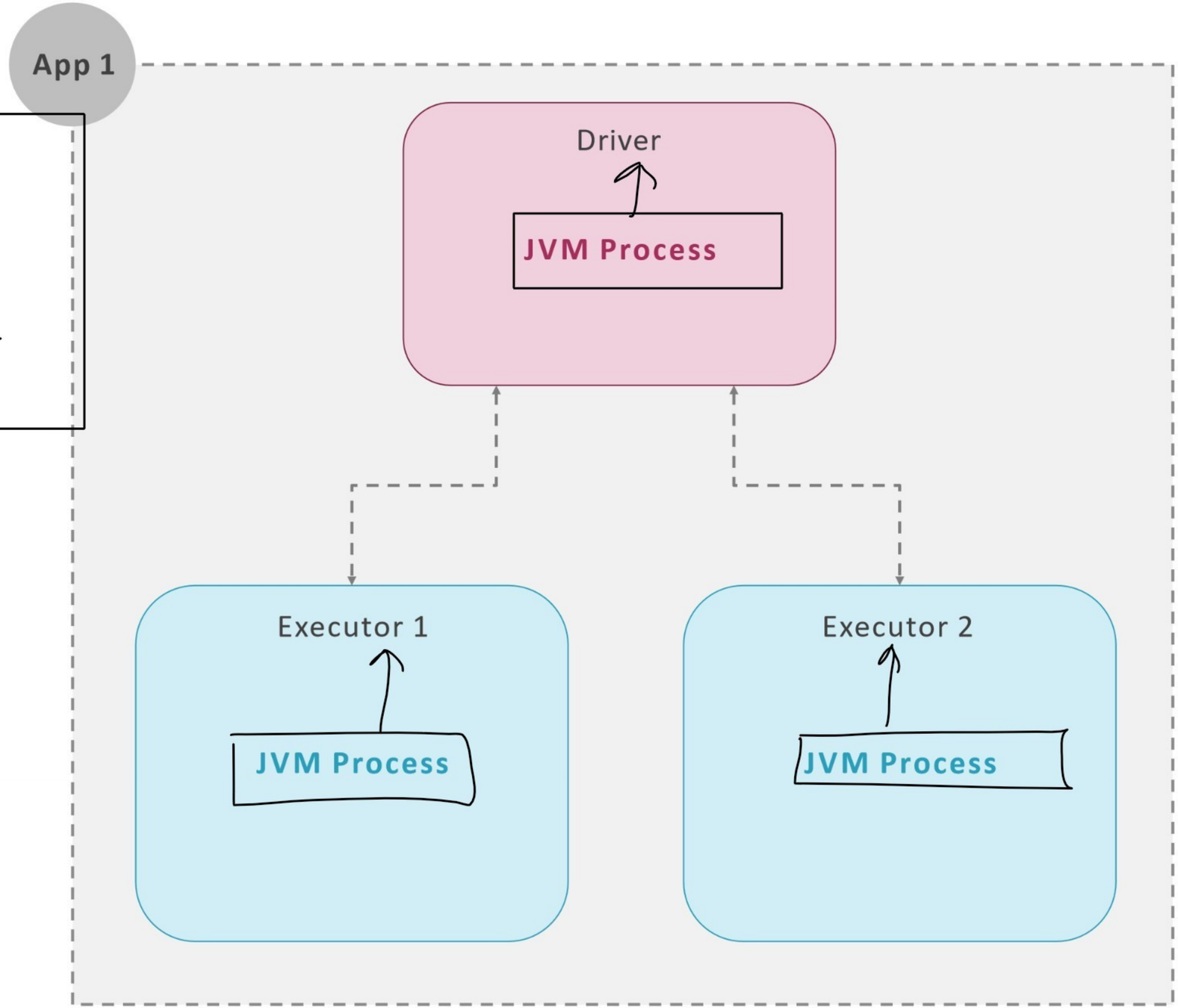
Determines how to execute a job

Analyzes job & distributes work to executors

EXECUTOR

Executors are responsible for executing the work (or code)

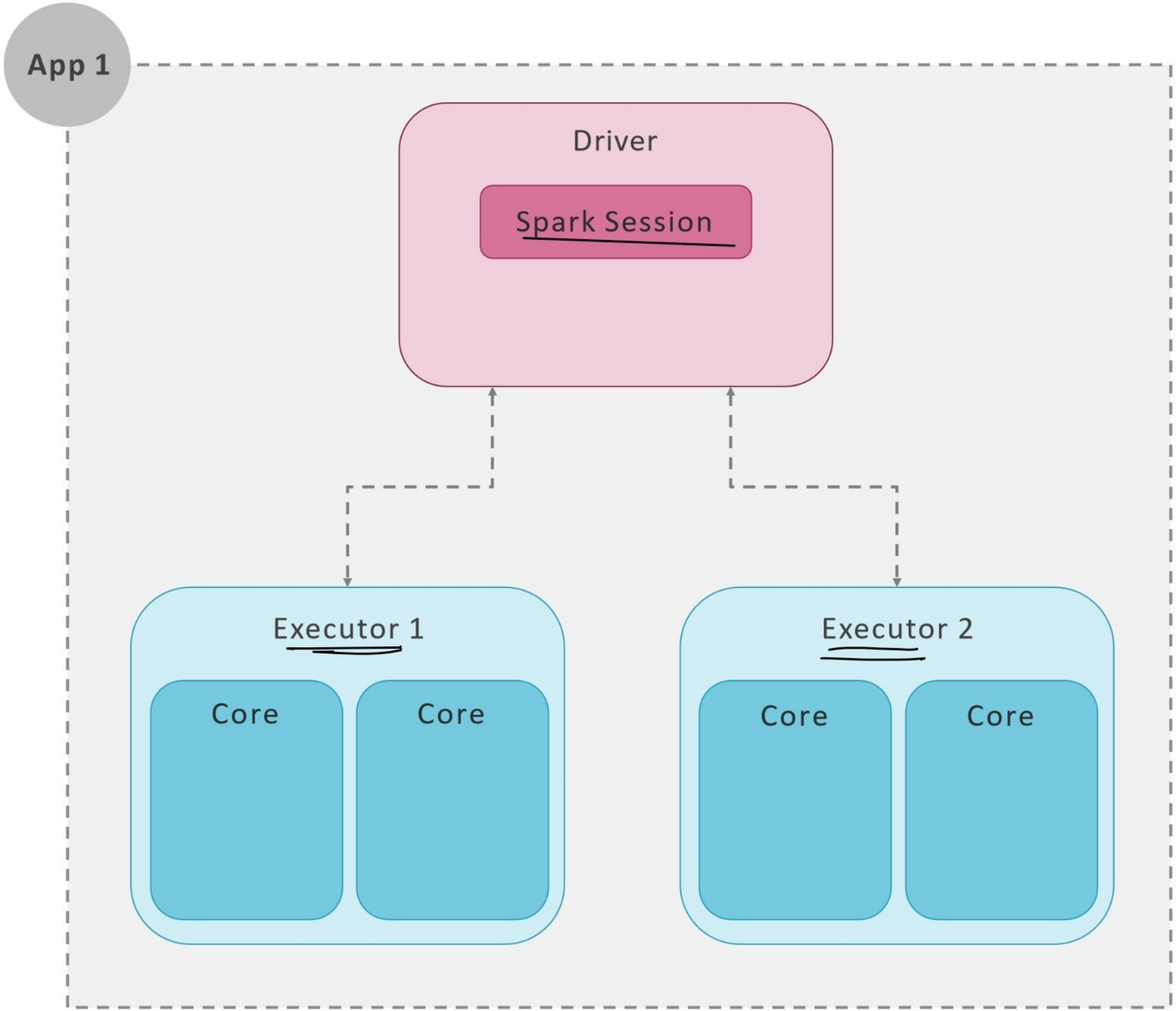
Returns result back to Driver



Spark Session is the entry point to all functionality of Spark

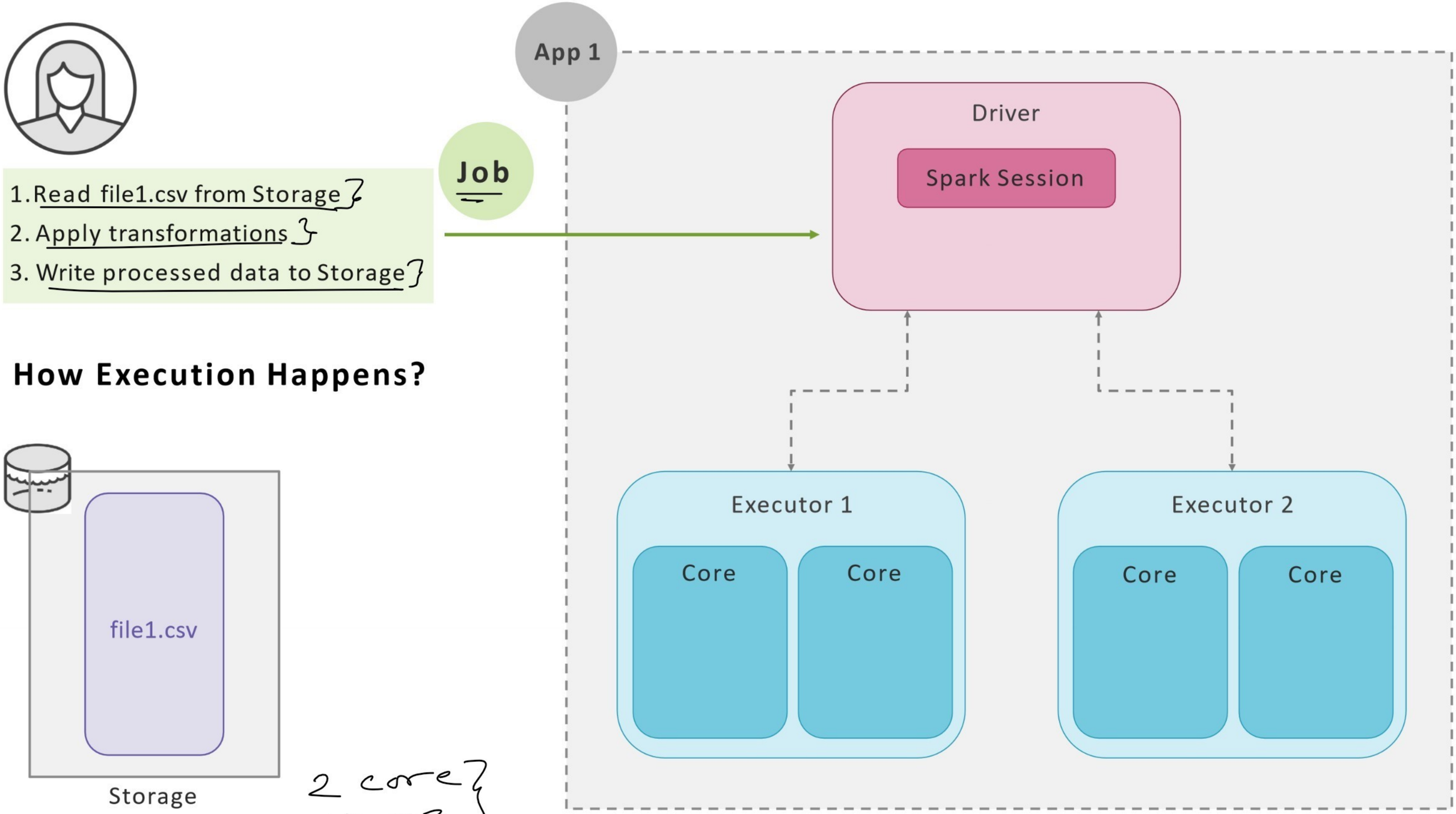
Use Spark Session to read file, create objects, run queries etc.

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?



2 core
14 GB

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 4 parts

Executor 1

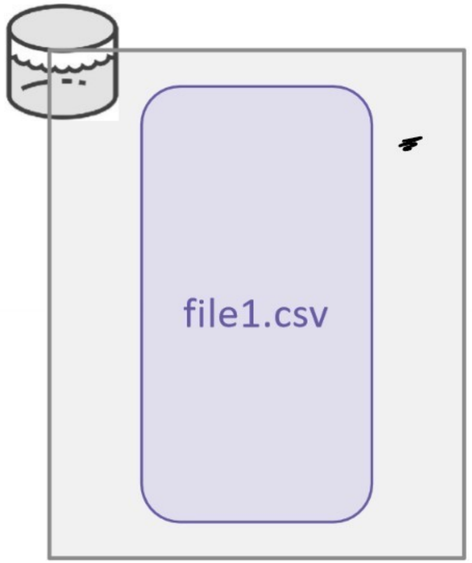
Core

Core

Executor 2

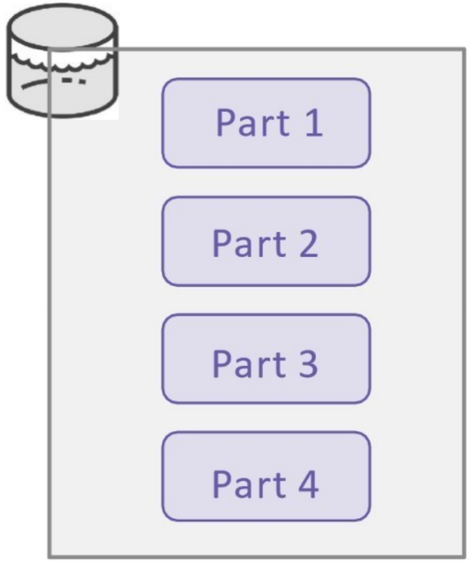
Core

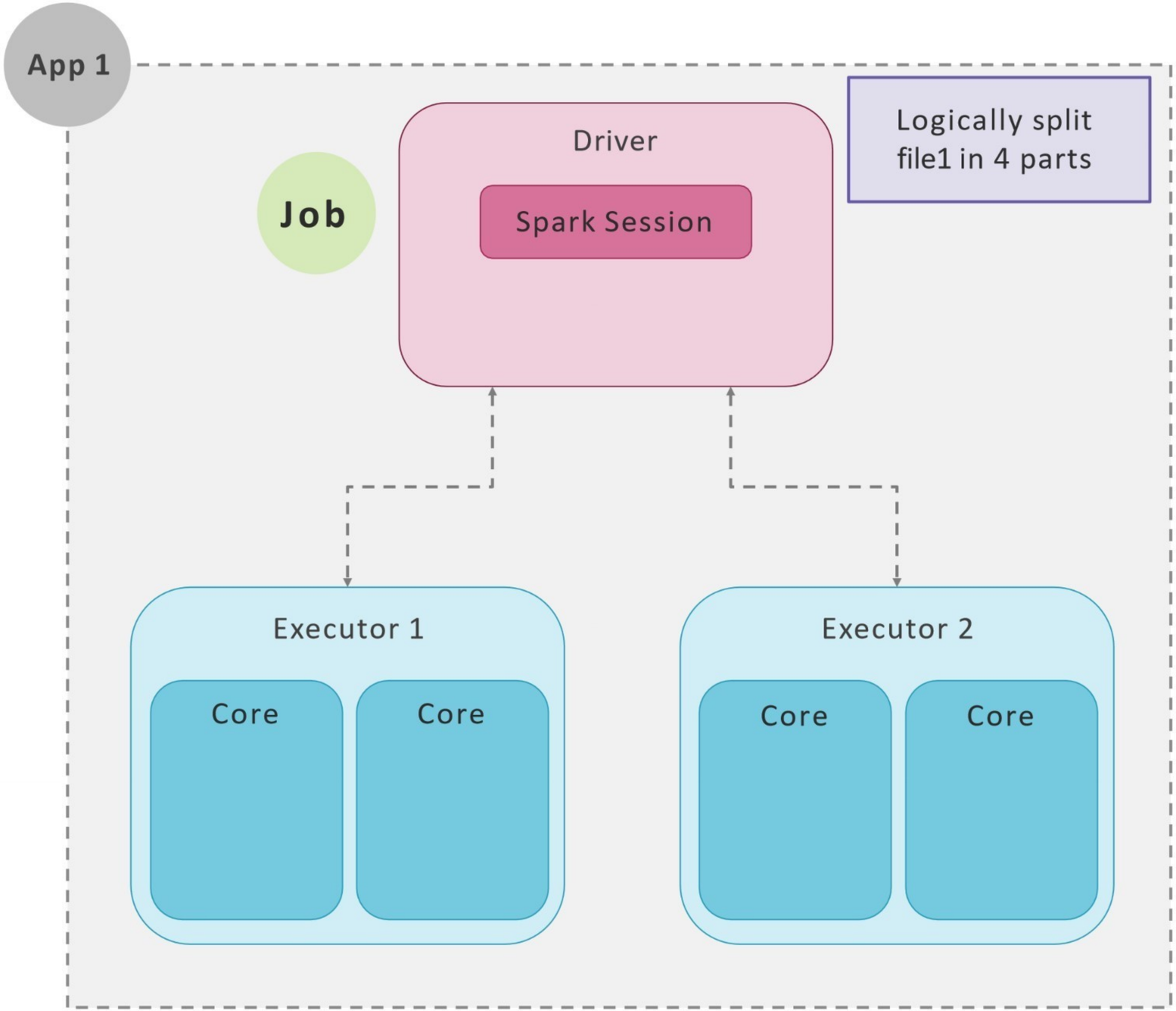
Core



Storage

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

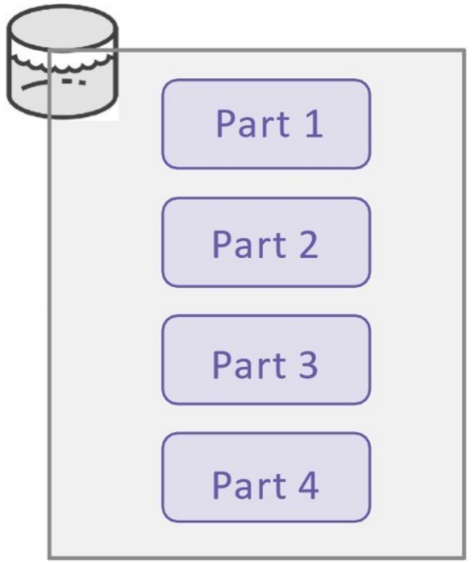


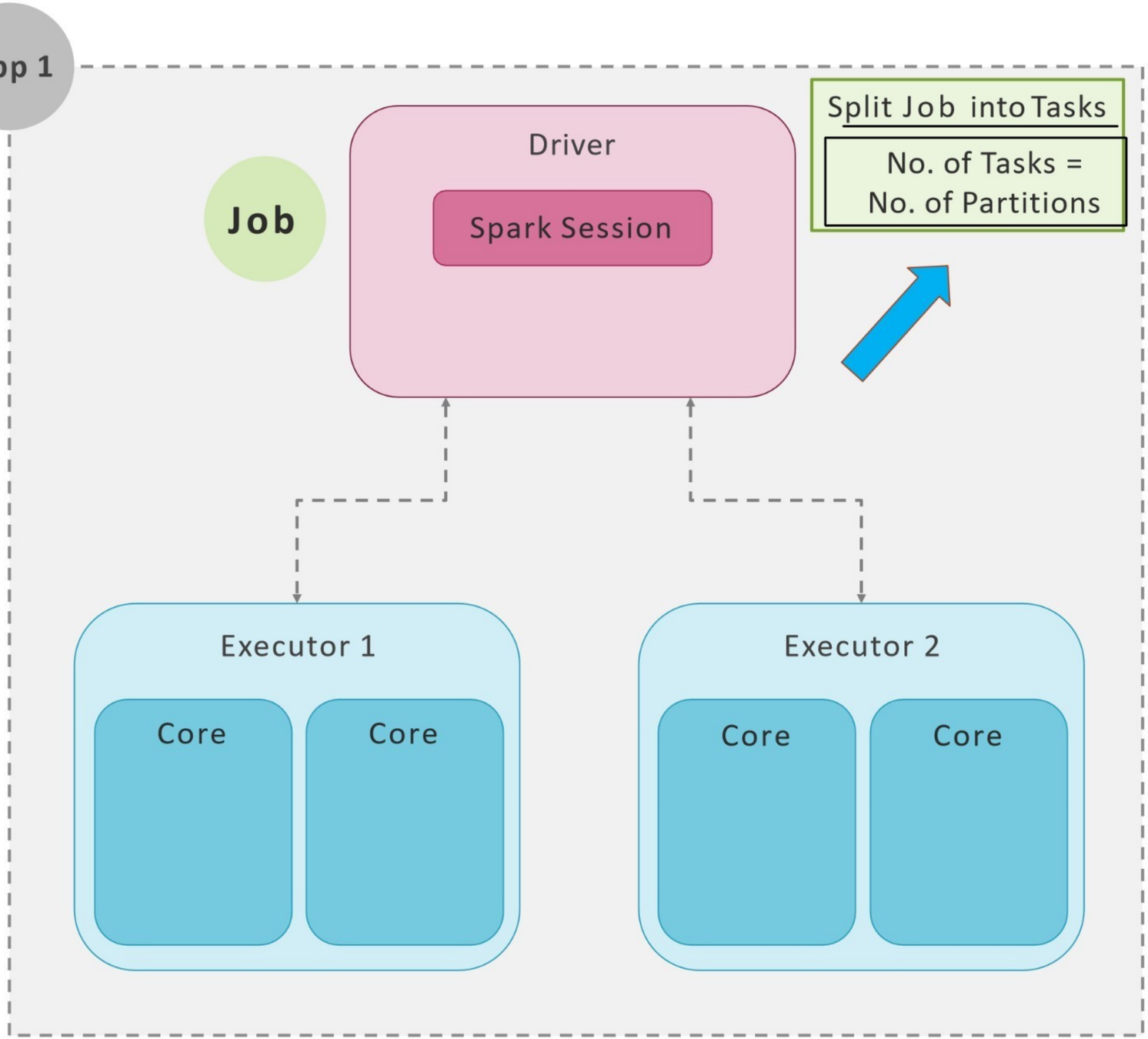


1 Job → 4 Task

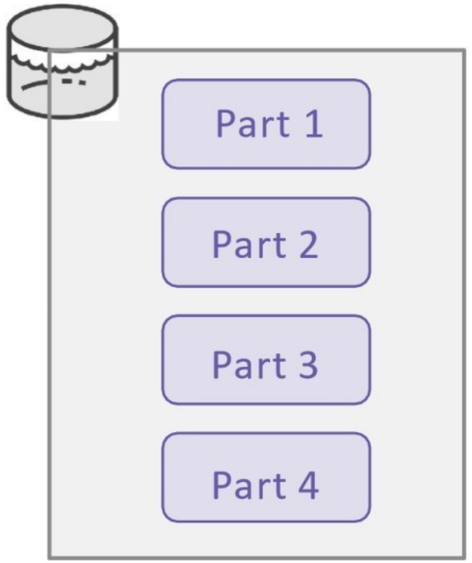
No. of partitions 4

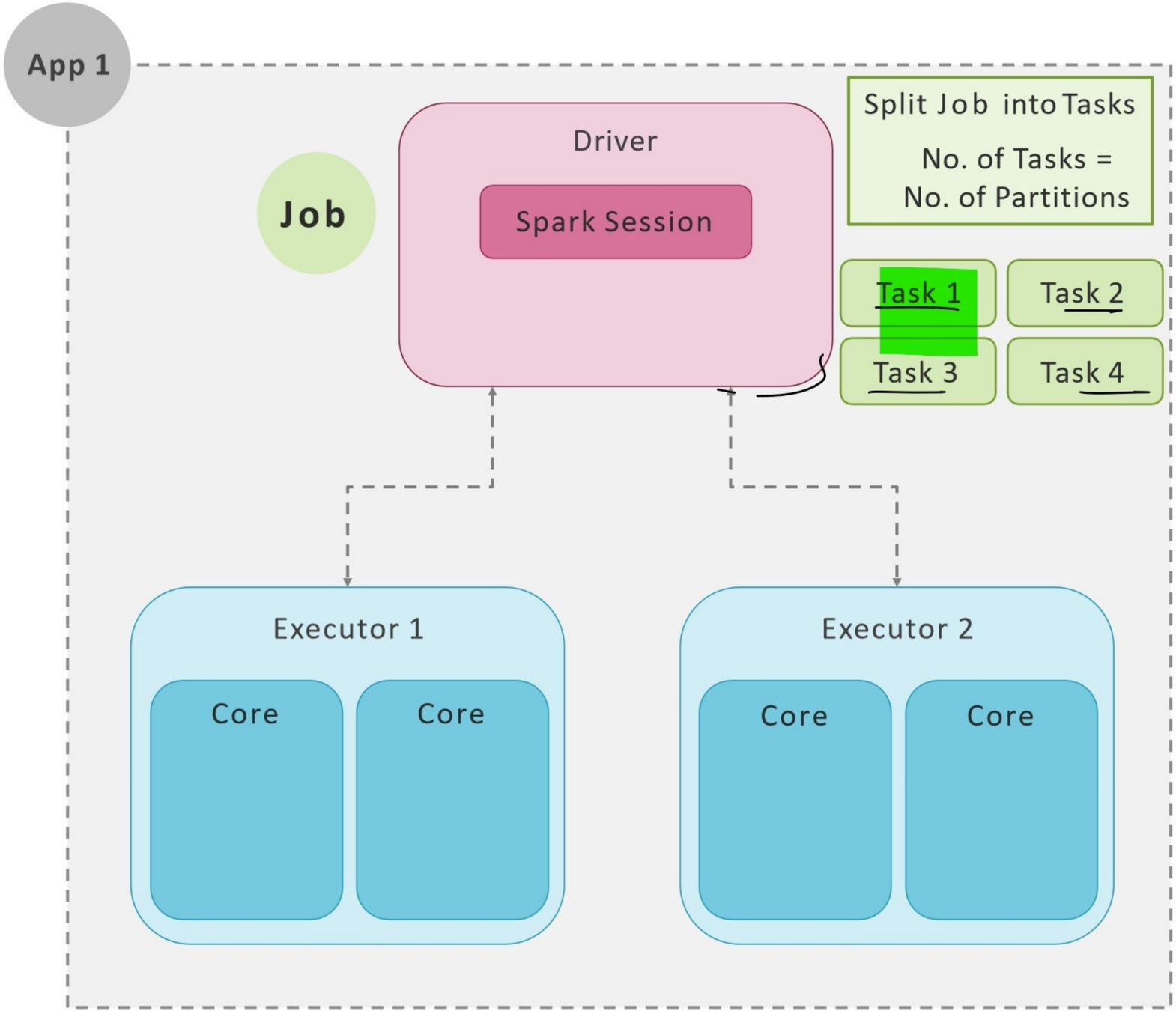
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



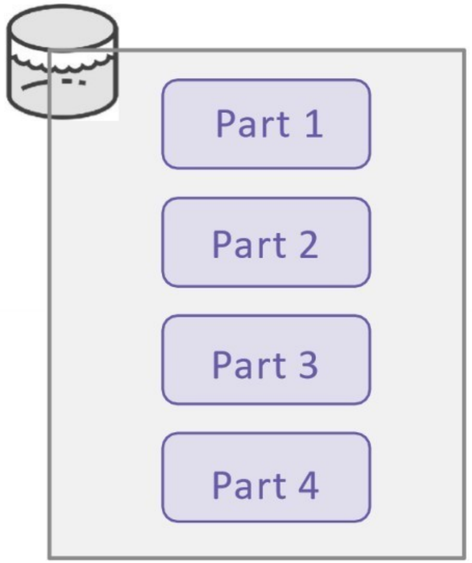


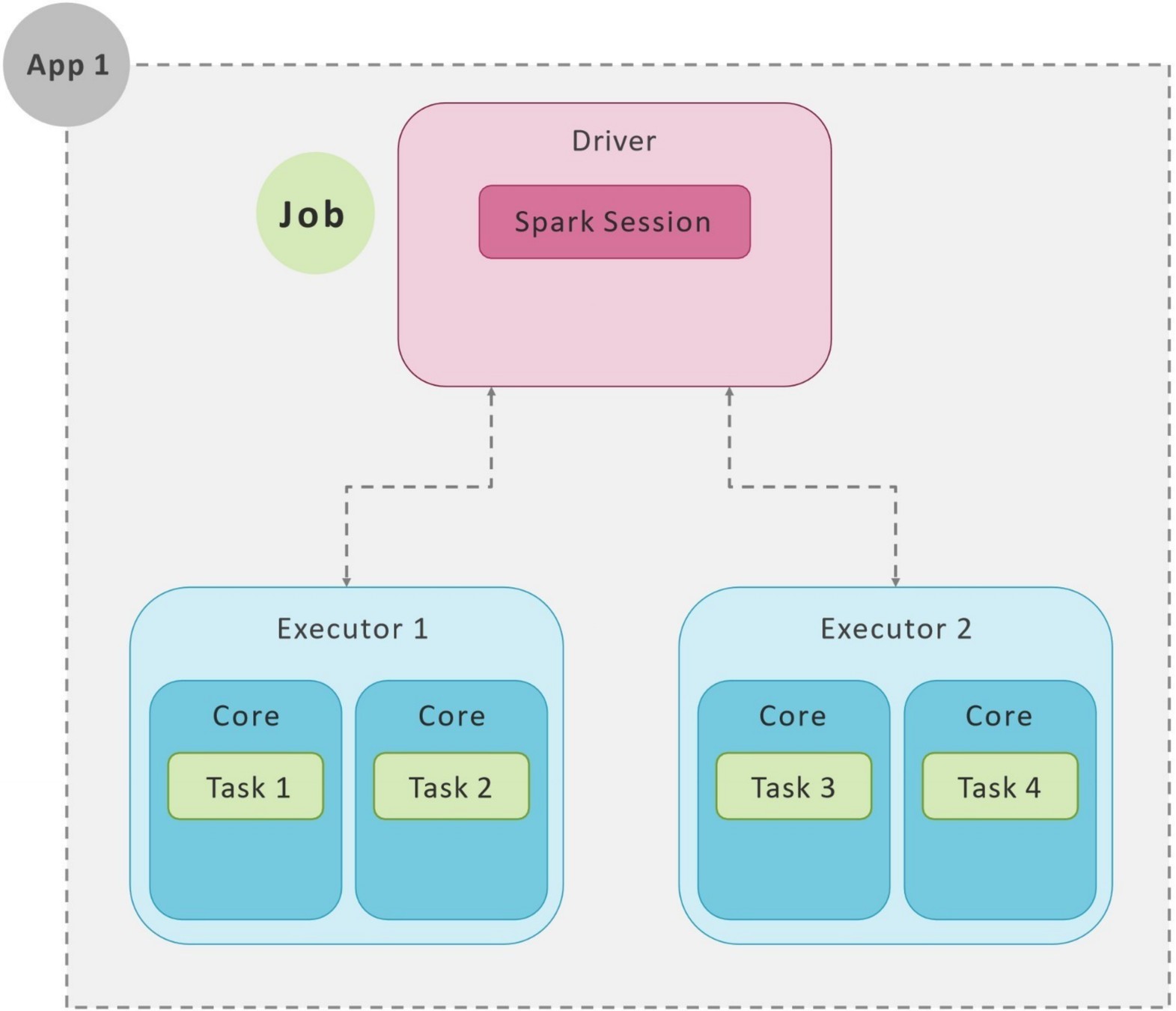
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



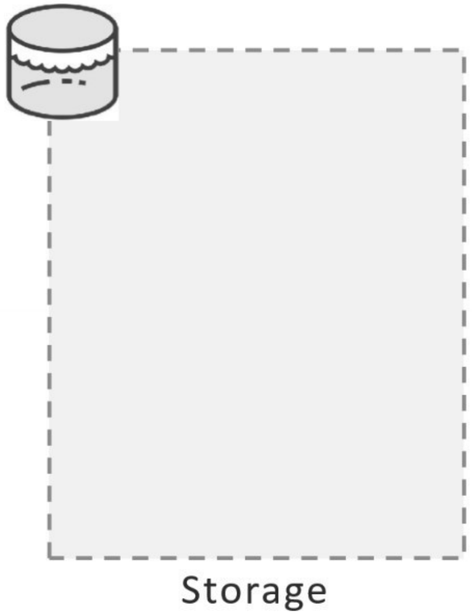


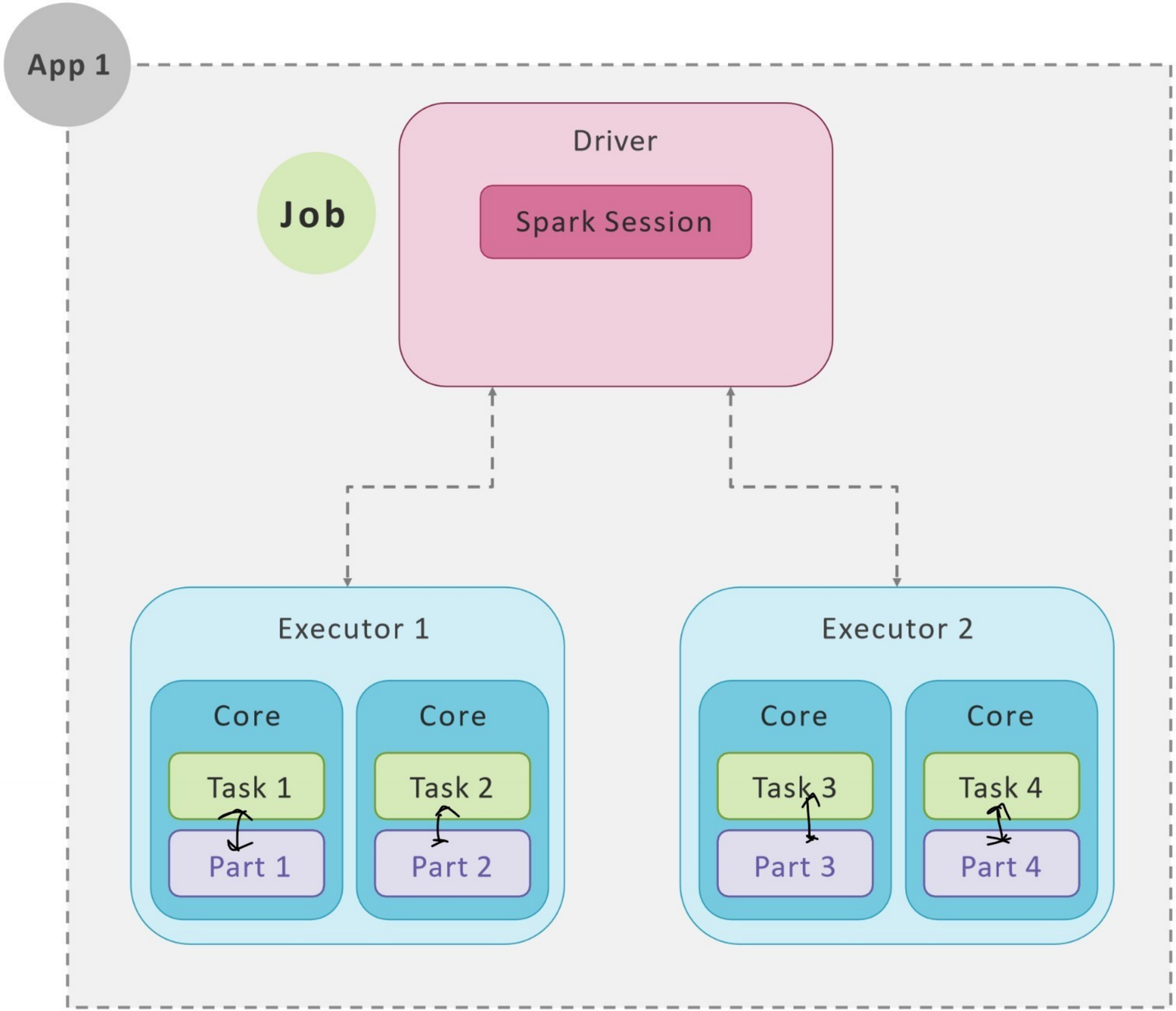
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage





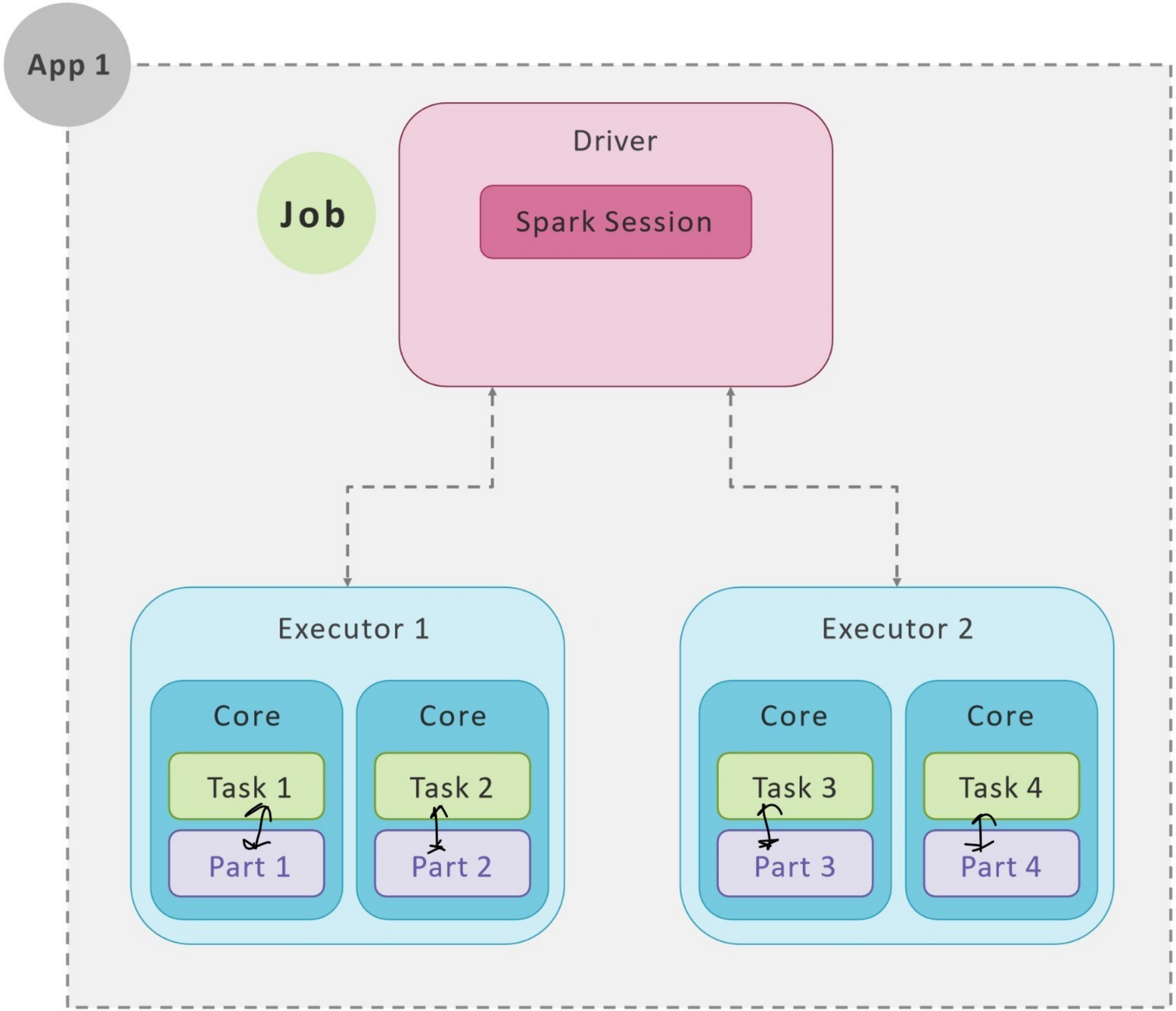
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

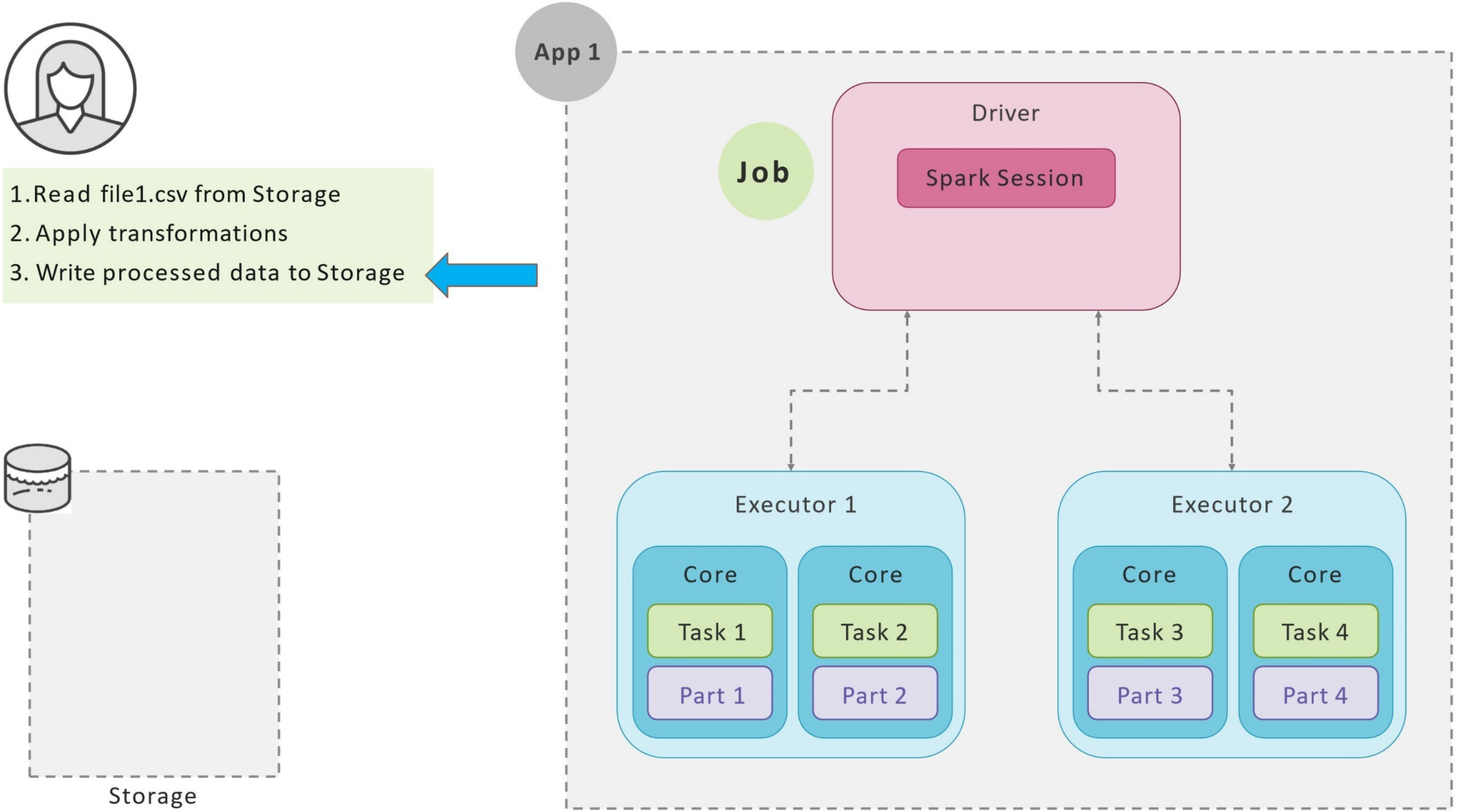


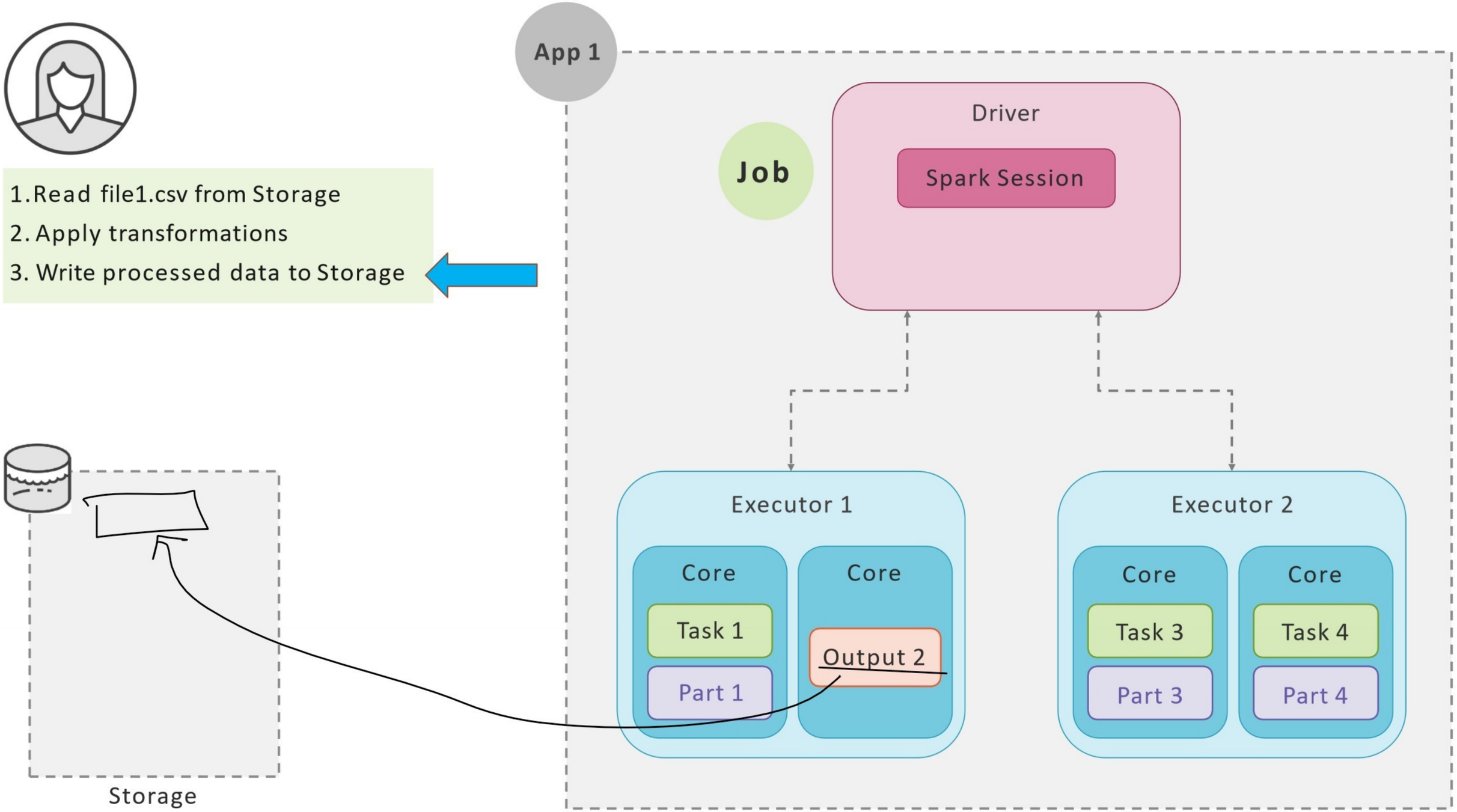


1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

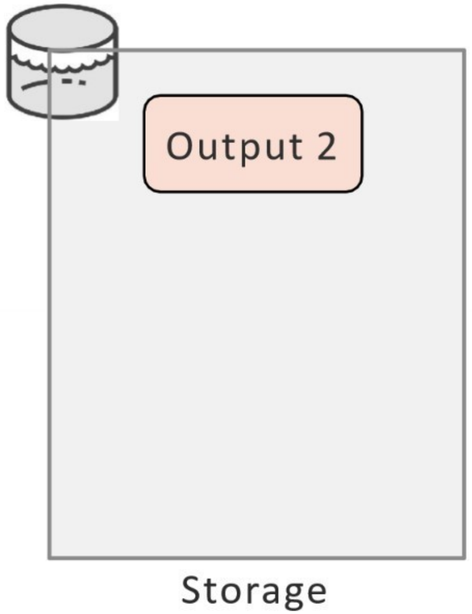
Storage

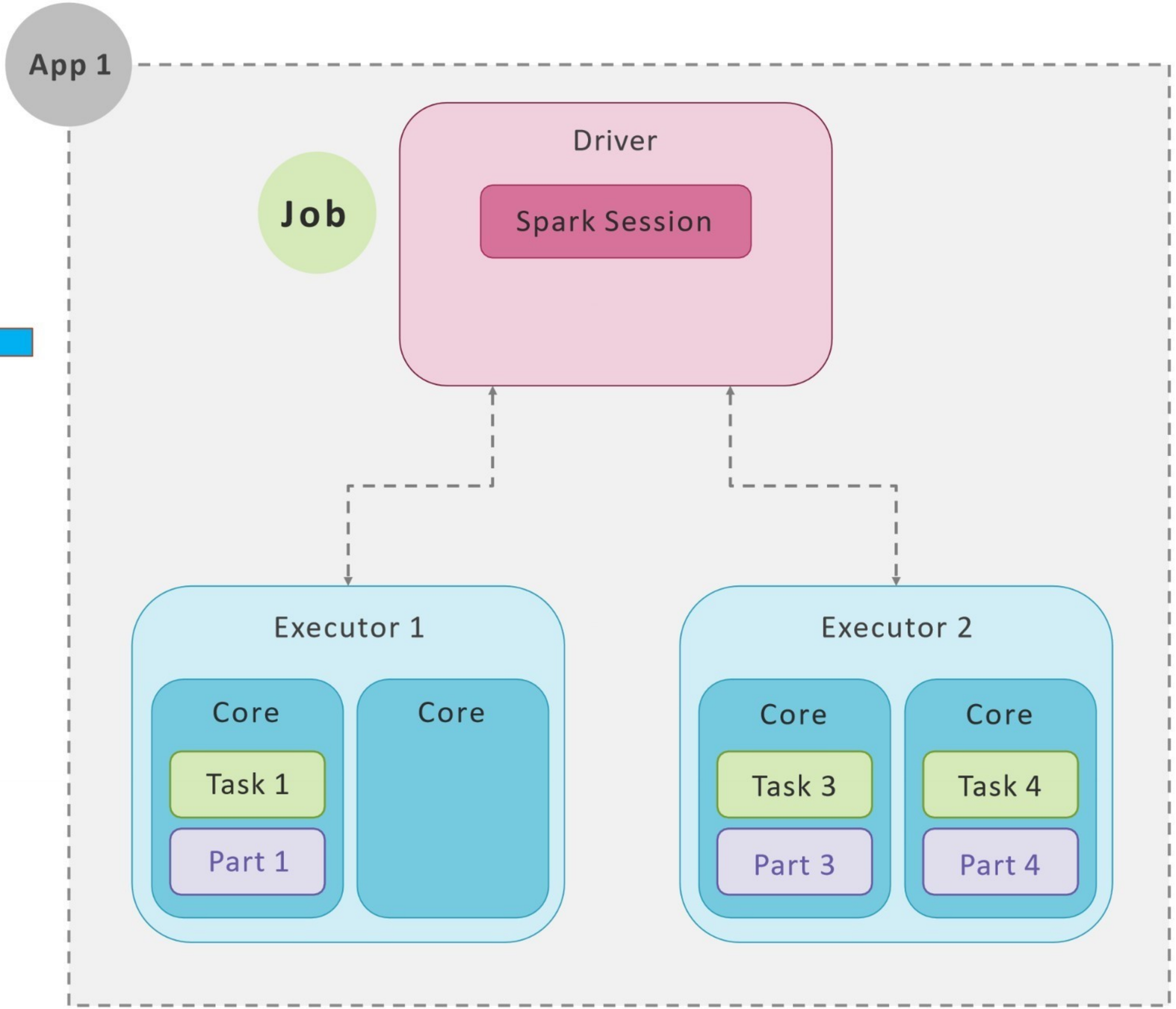



1.Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Part 1
Core
Task 2
Part 2
Executor 2
Core
Task 3
Part 3
Core
Task 4
Part 4
Storage

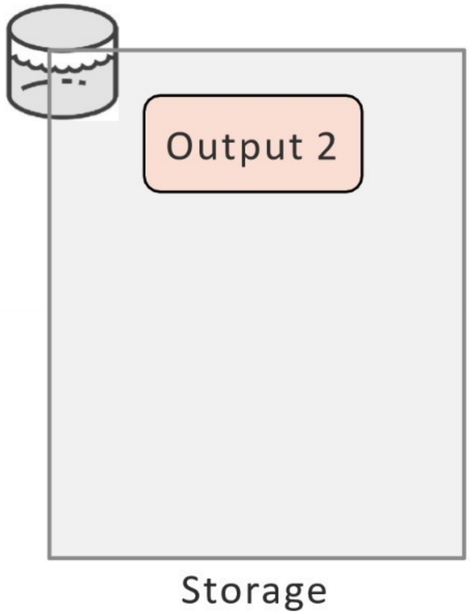

App 1
Job
Driver
Spark Session
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage
Executor 1
Core
Task 1
Part 1
Core
Output 2
Executor 2
Core
Task 3
Part 3
Core
Task 4
Part 4
Storage

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

Executor 1

Core

Output 1

Core

Executor 2

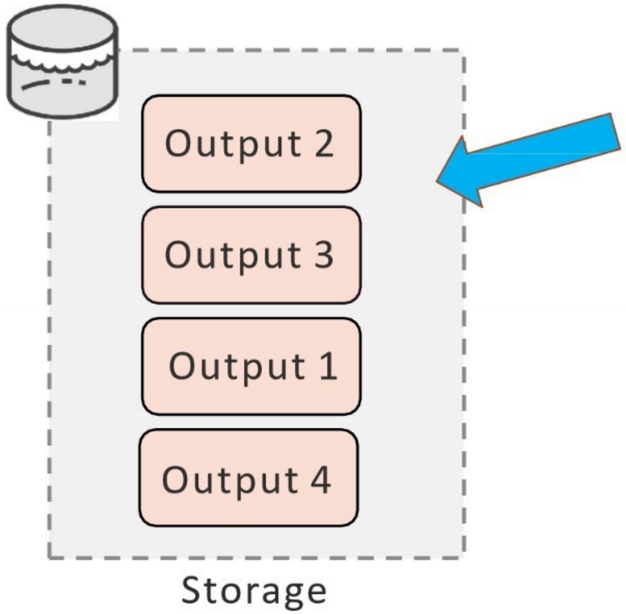
Core

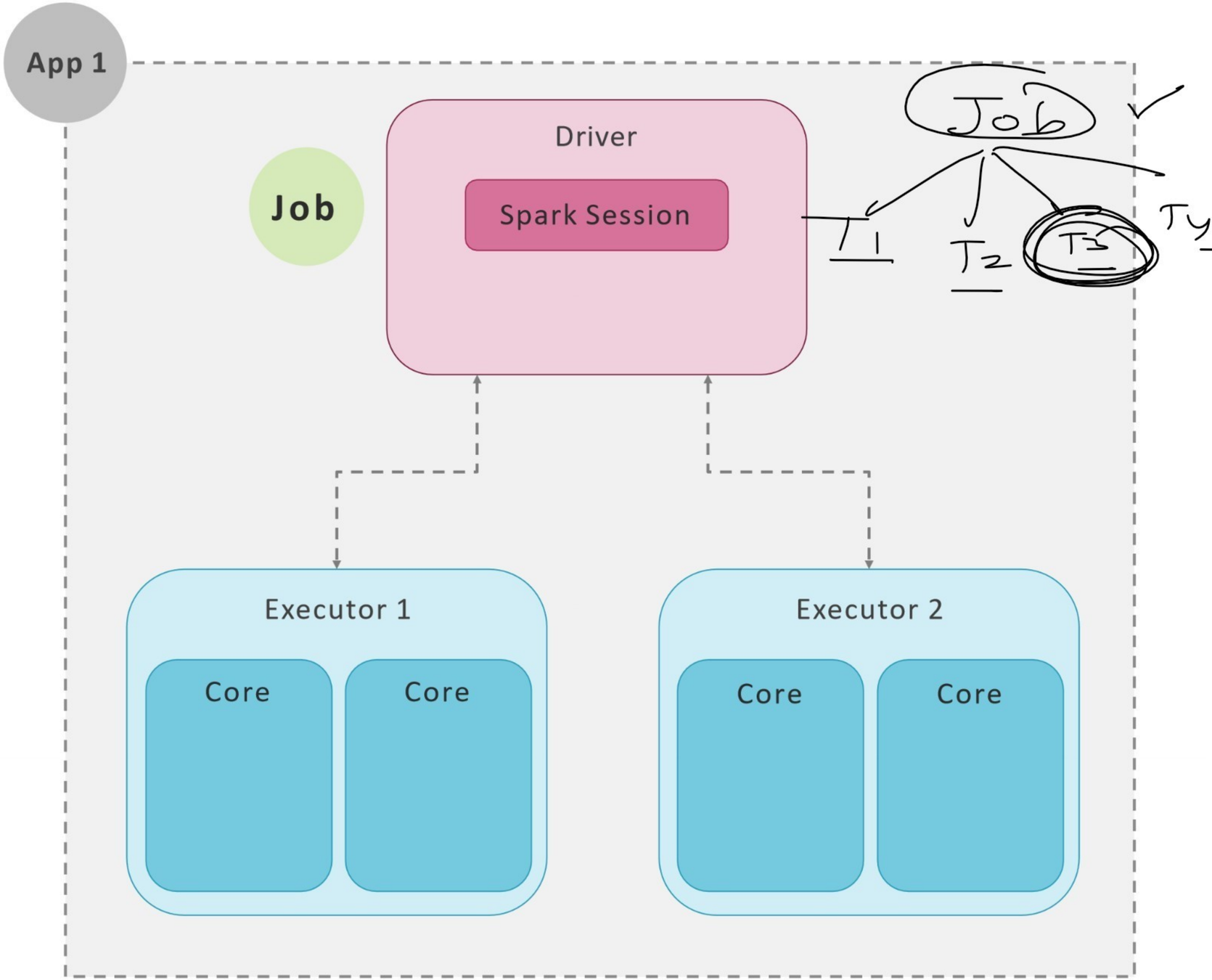
Output 3

Core

Output 4

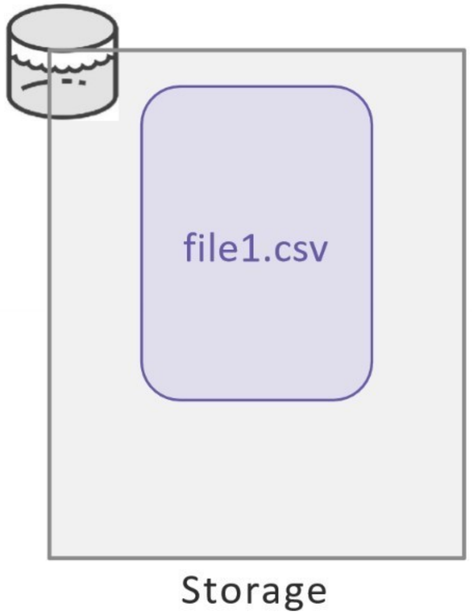
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

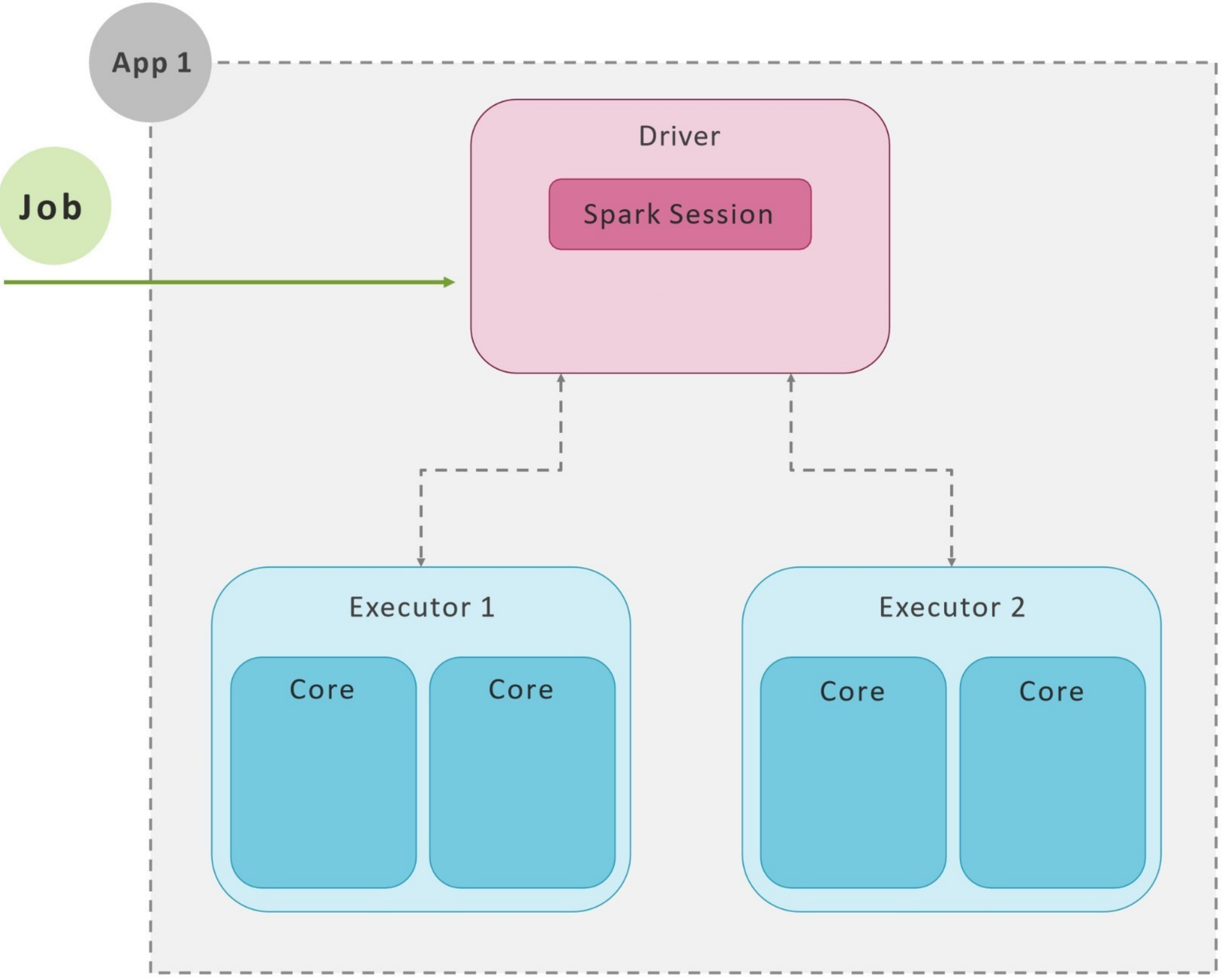




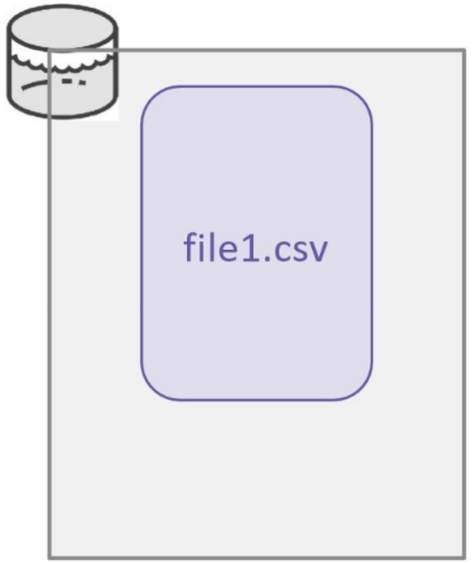
What if number of Tasks are lesser than available Cores?

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

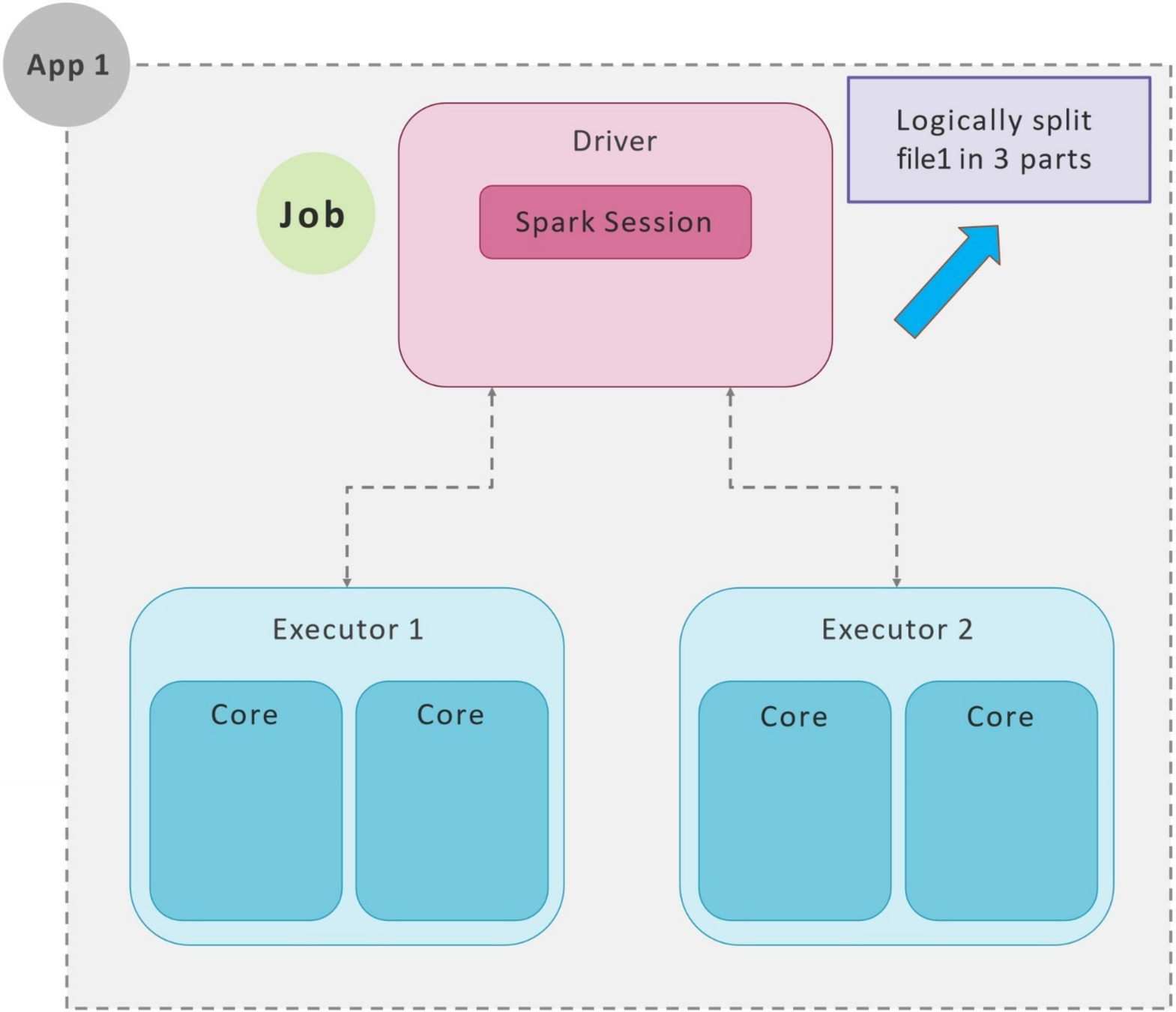




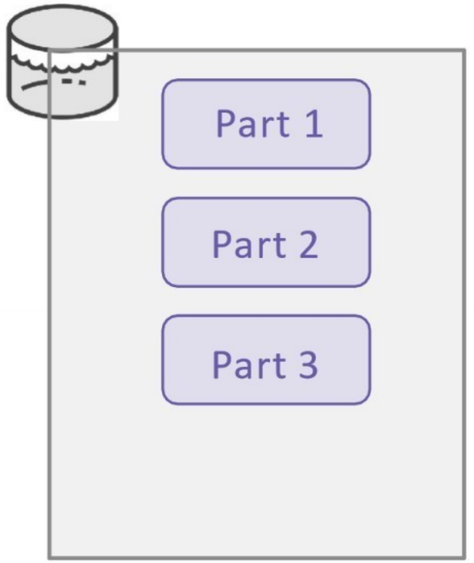
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



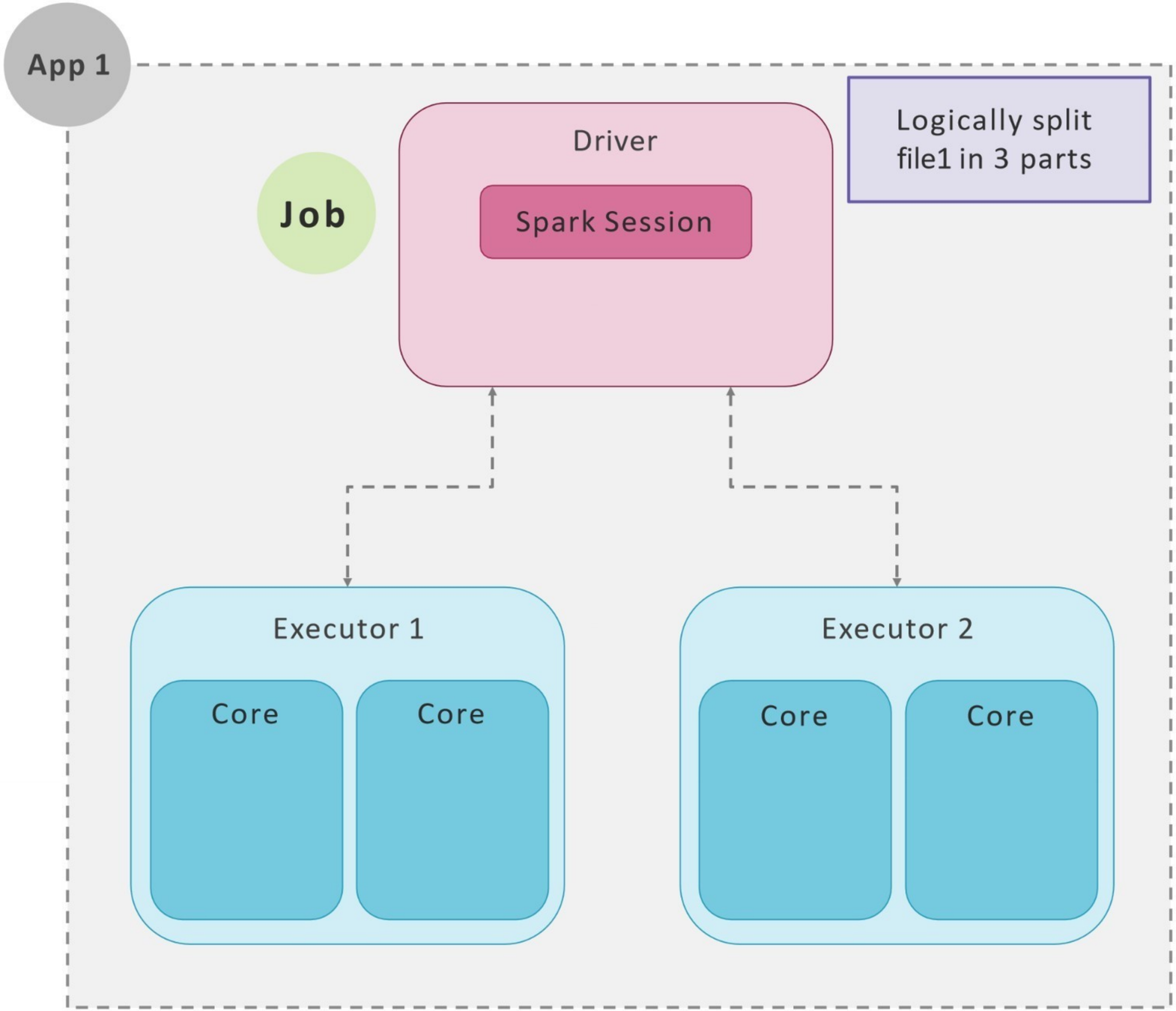
Storage



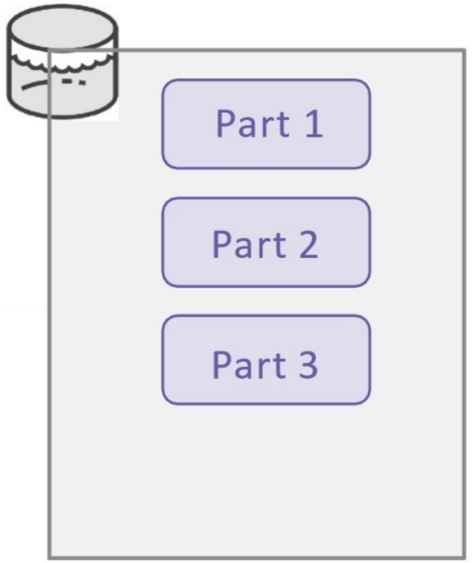
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



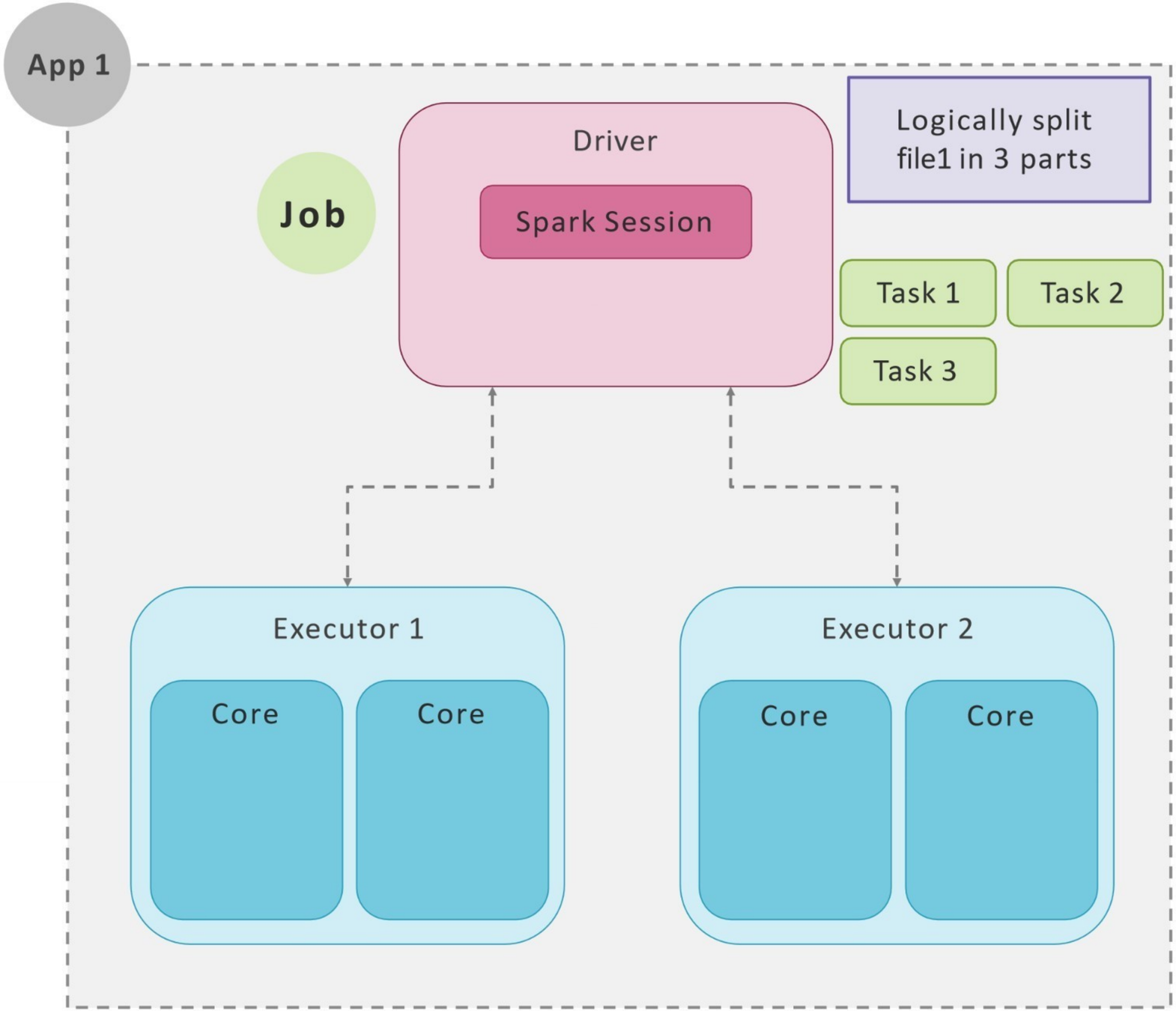
Storage



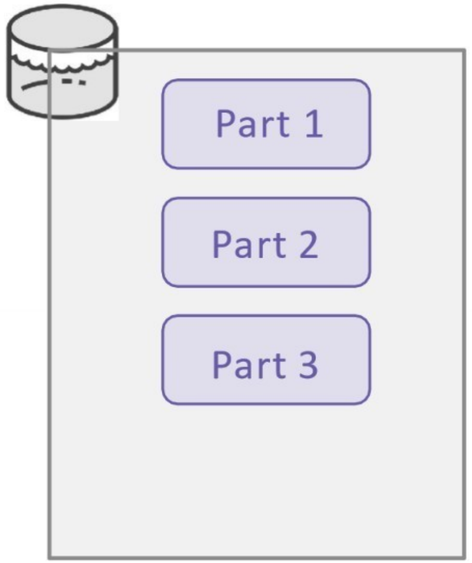
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

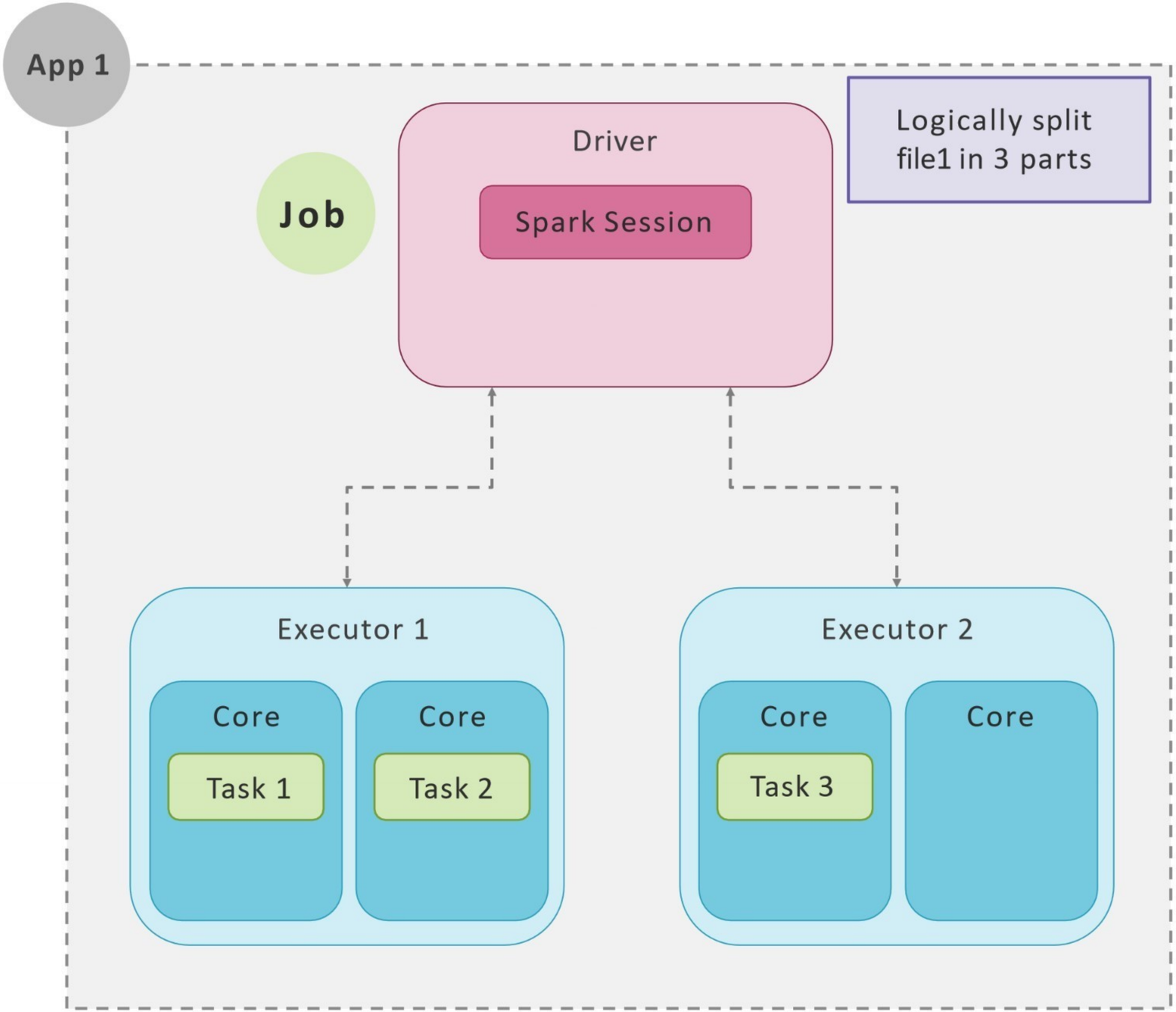


Storage

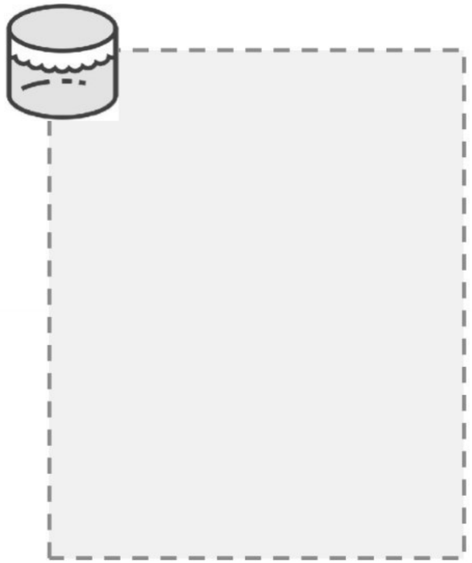


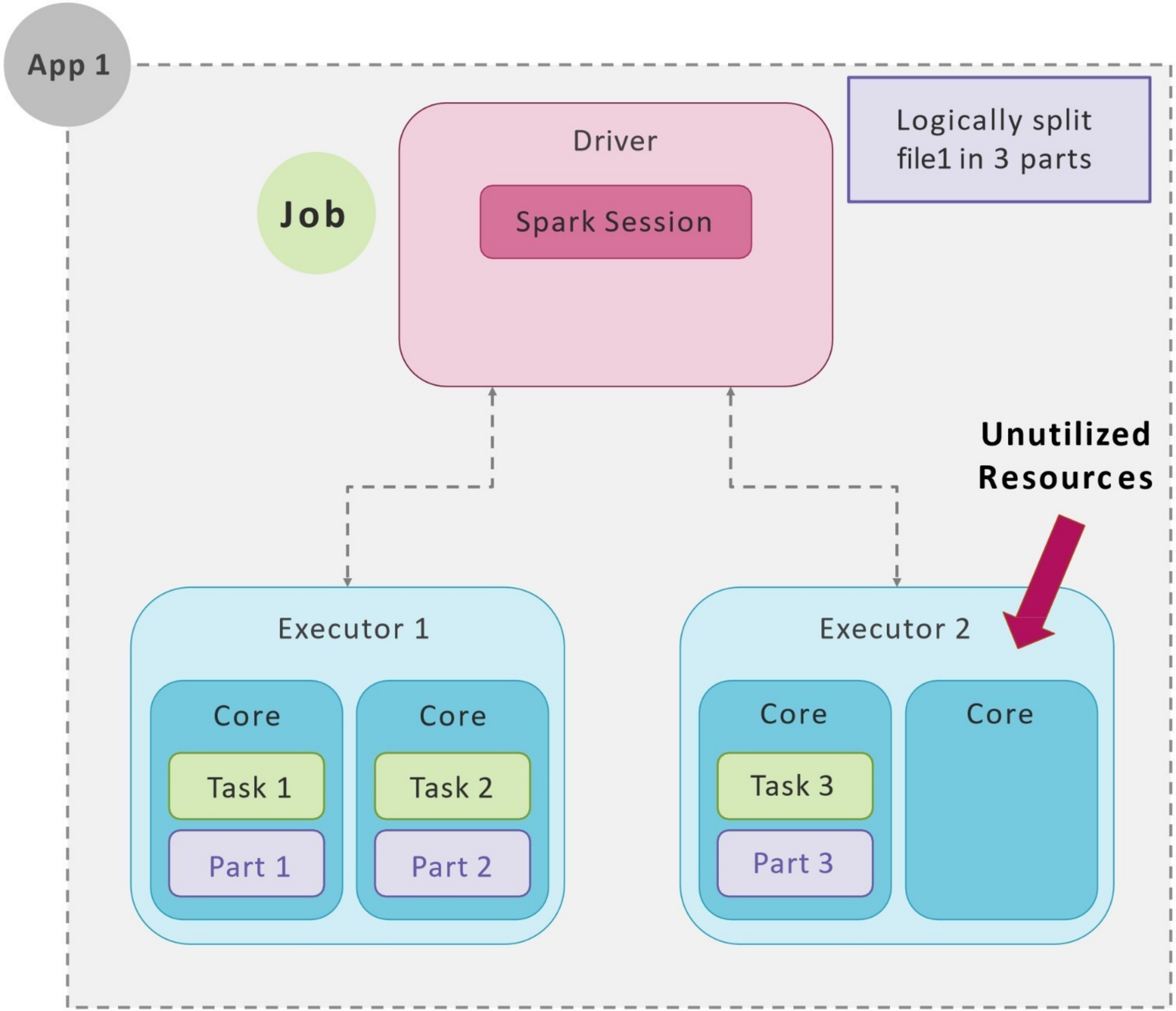
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage





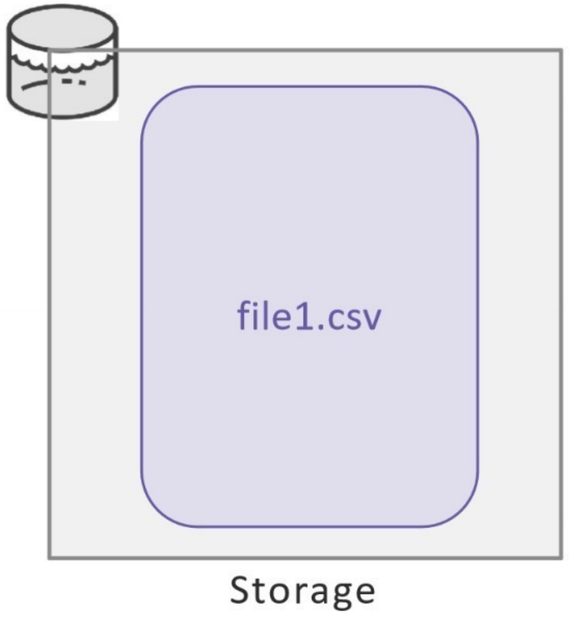
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

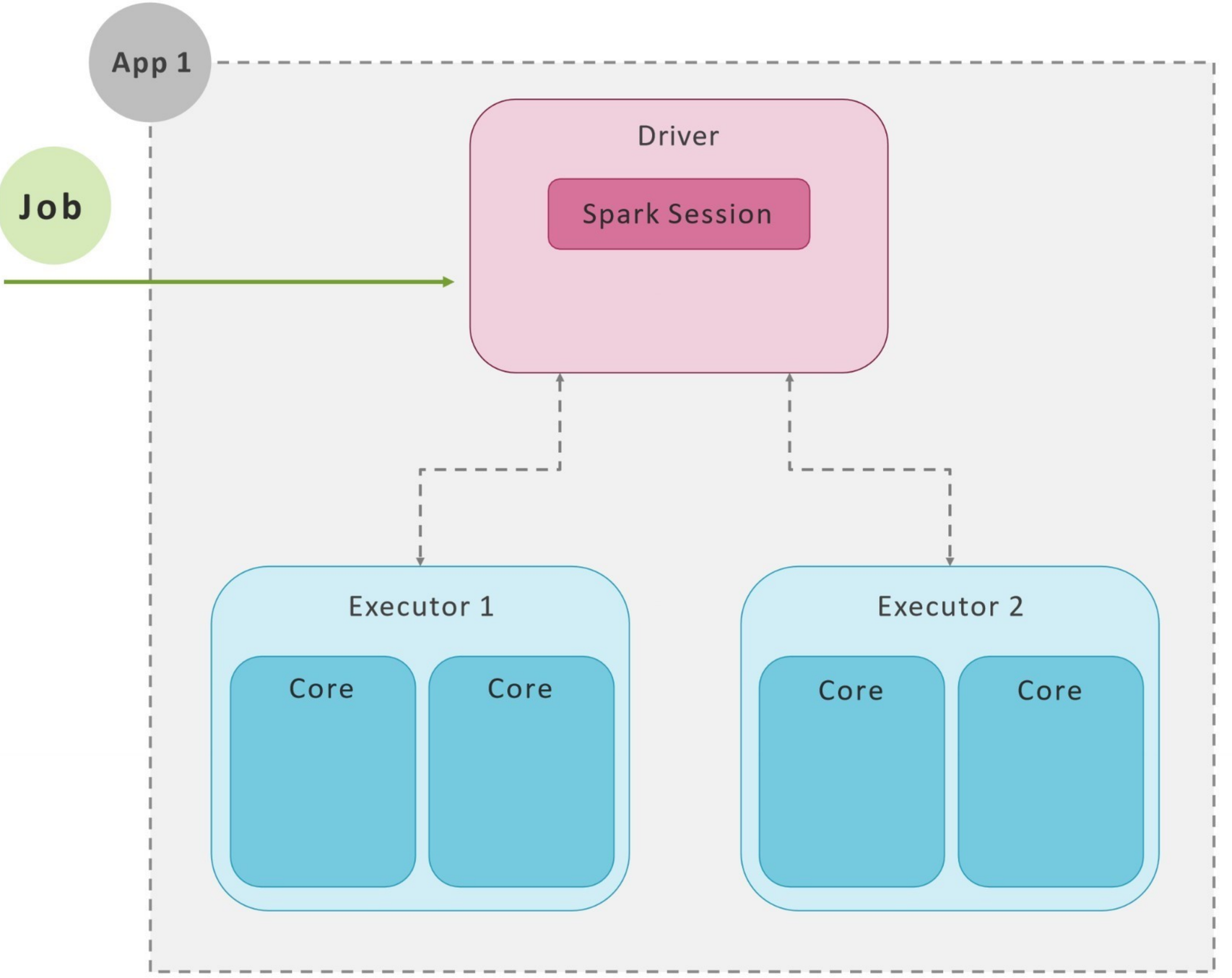




What if number of Tasks are more than available Cores?

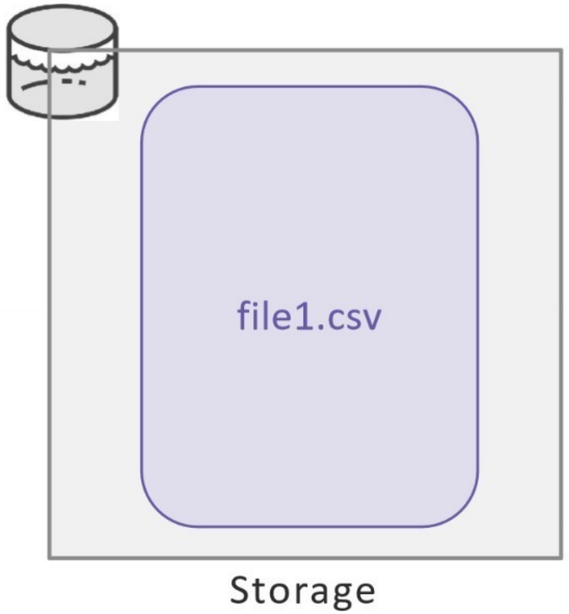
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

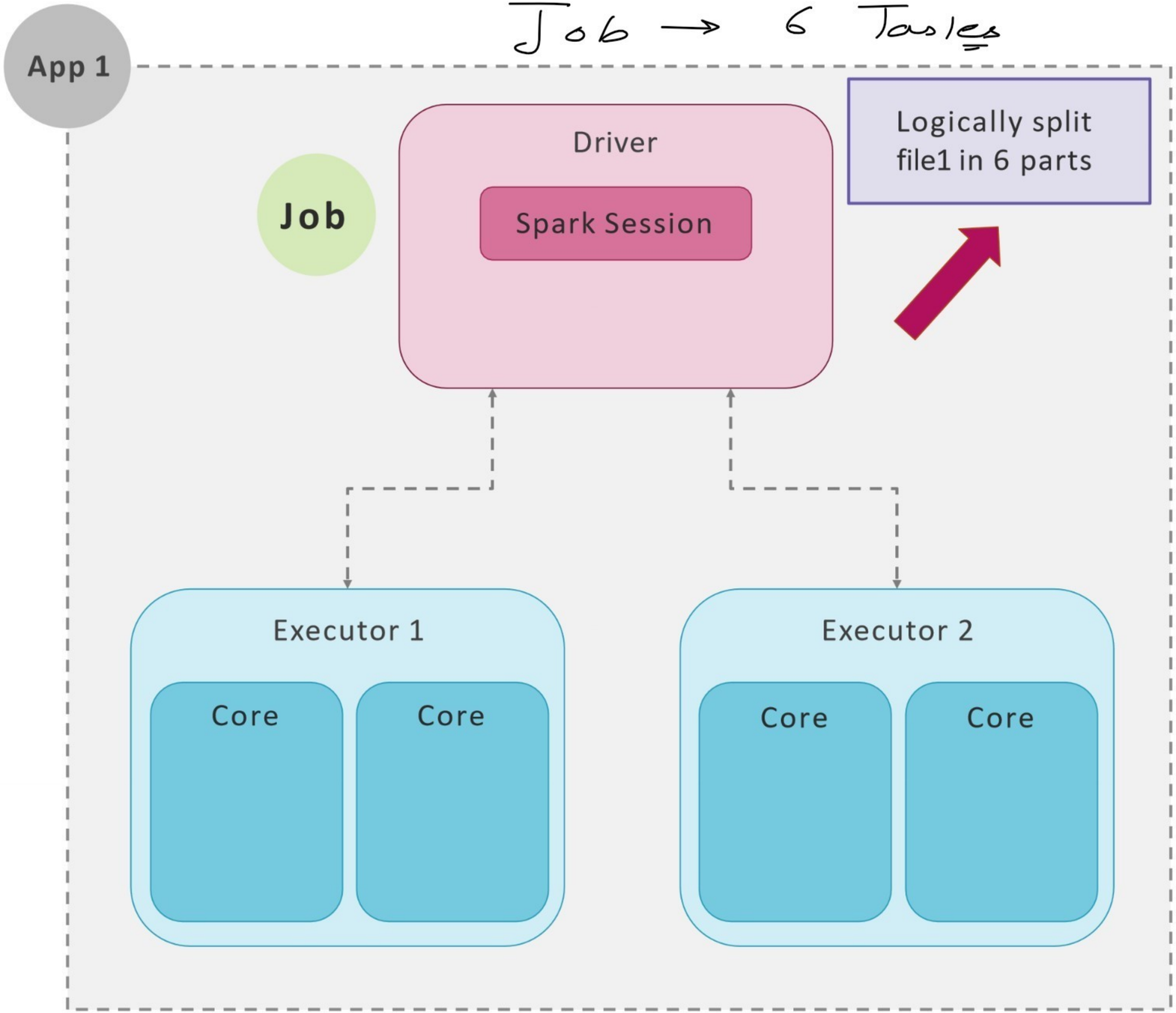




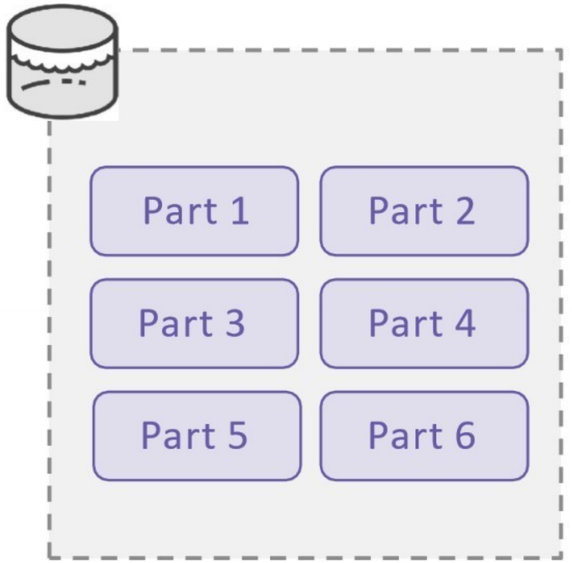
Job → 6 Tasks

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

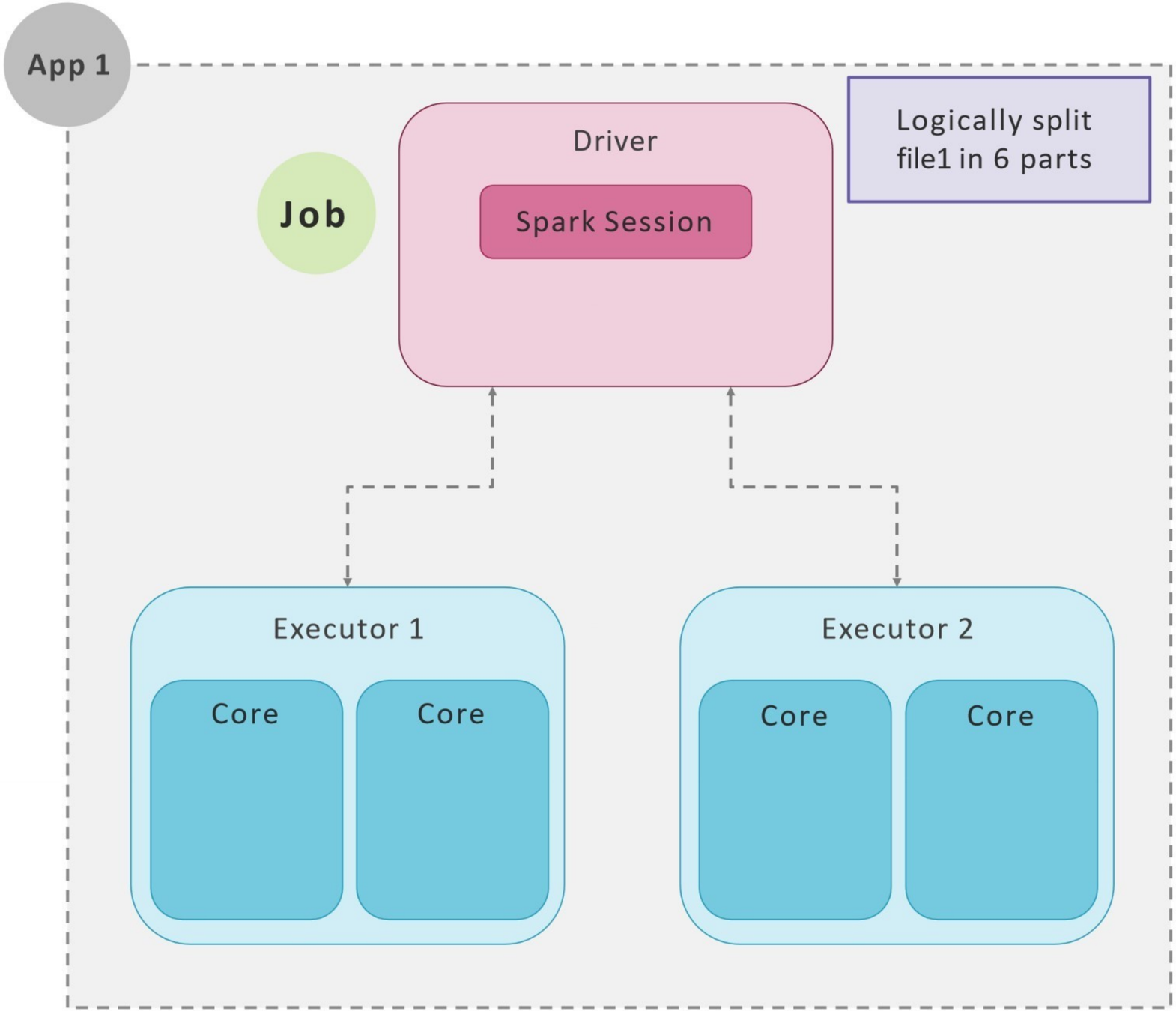




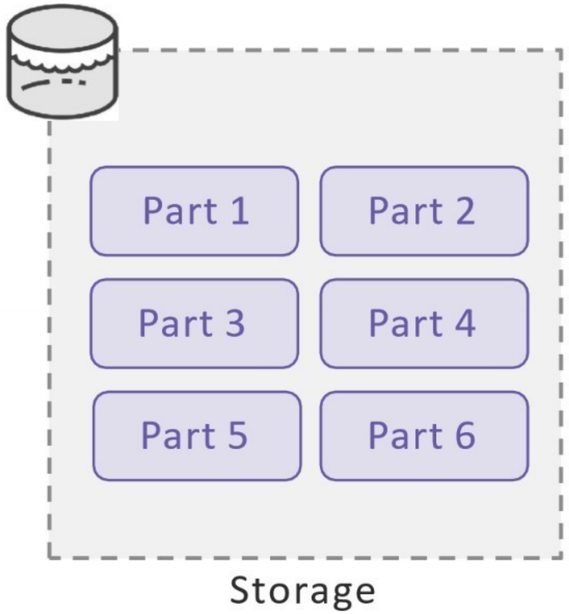
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage

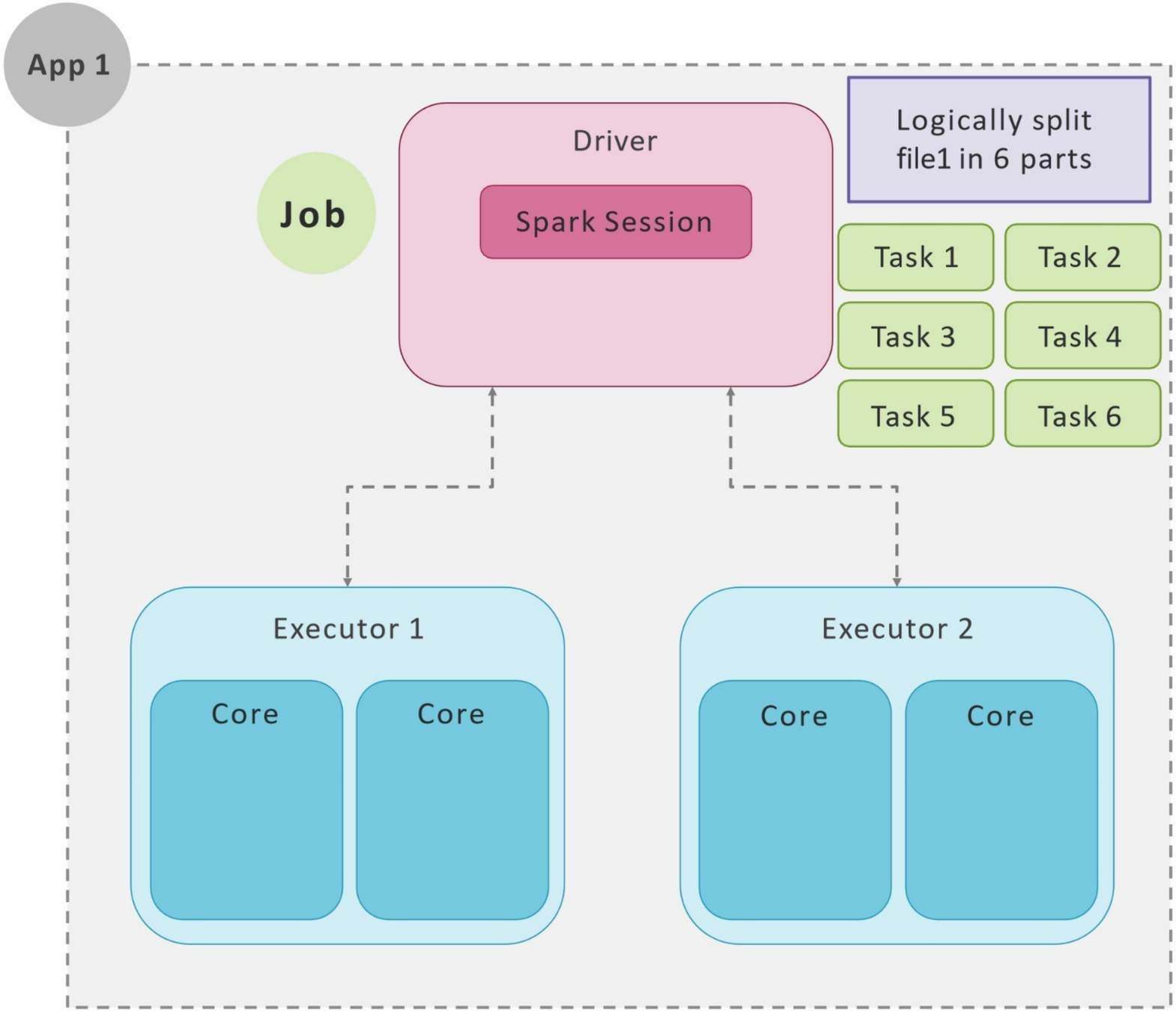


Storage

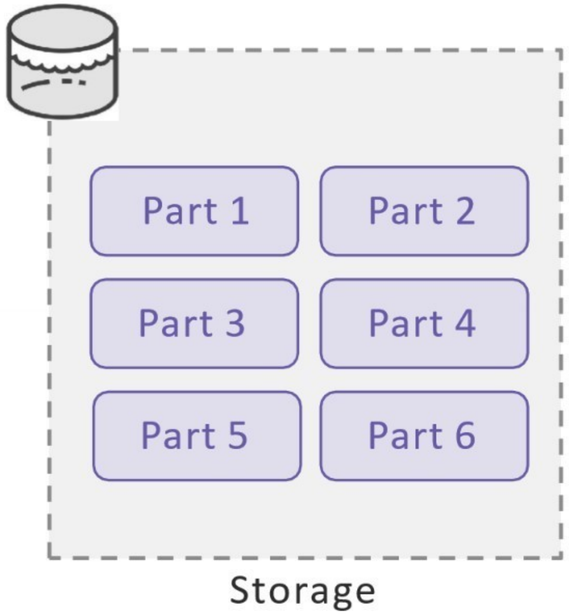


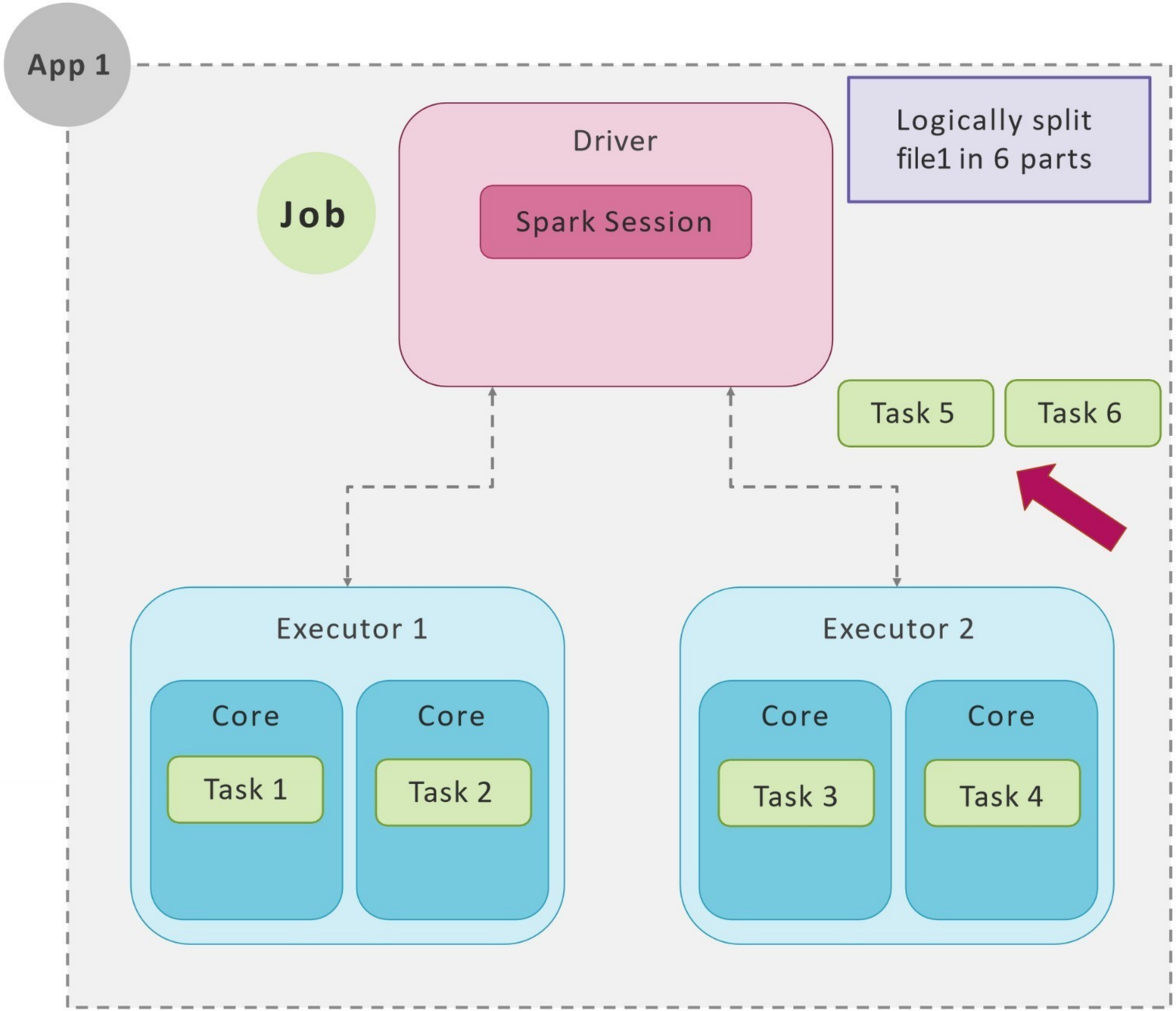
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



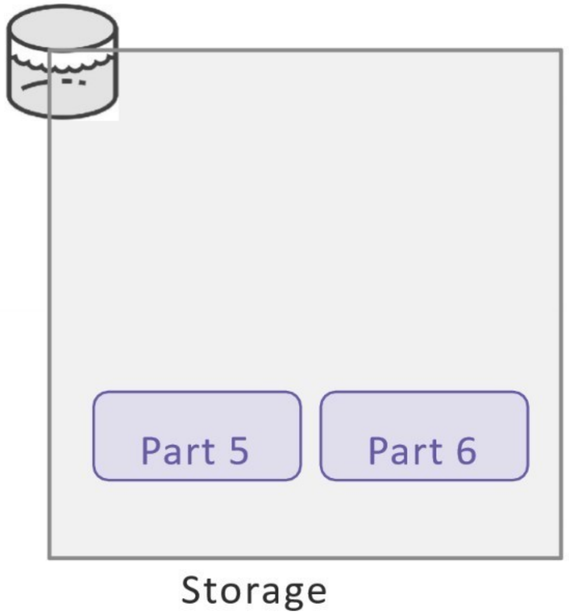


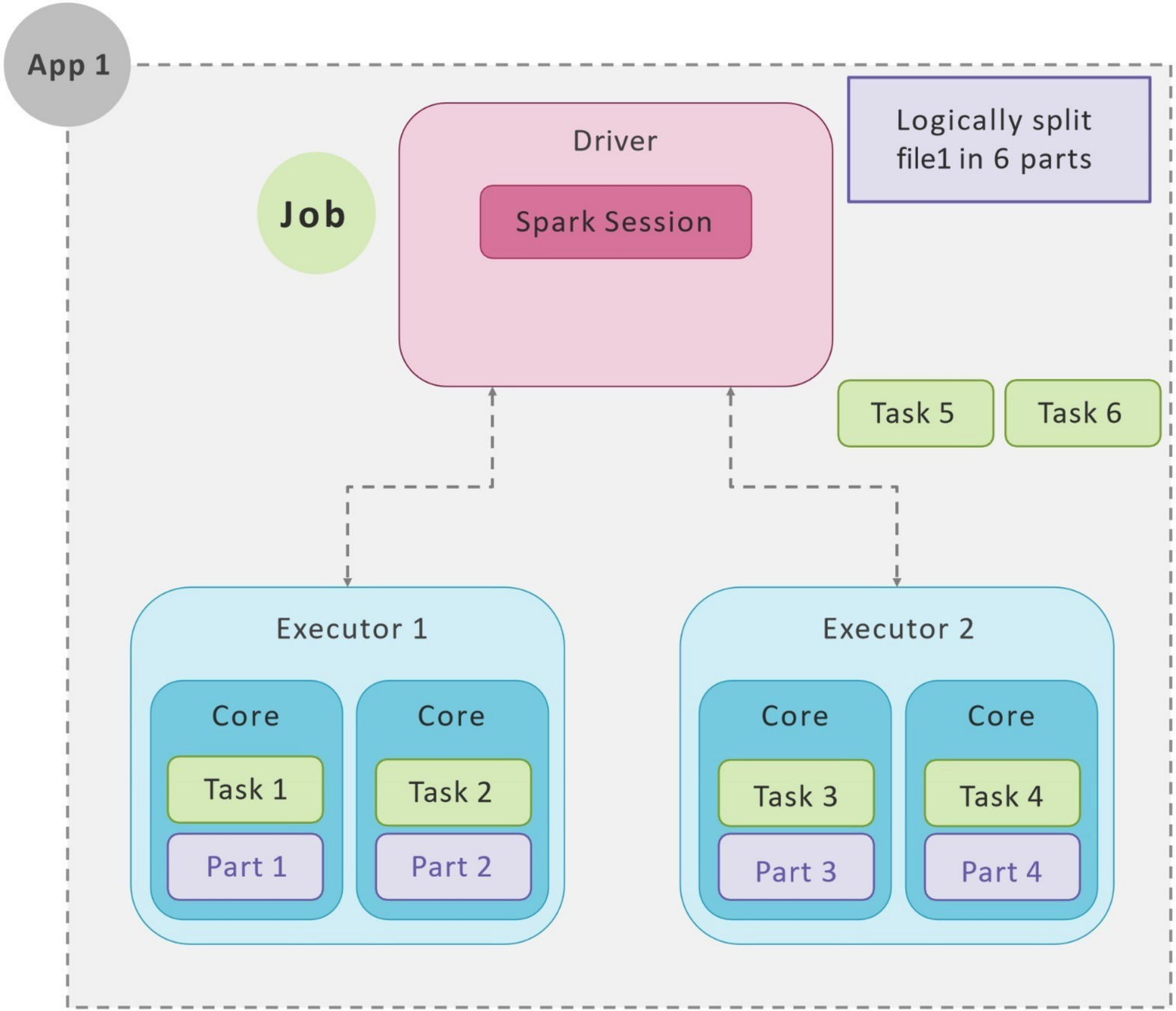
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



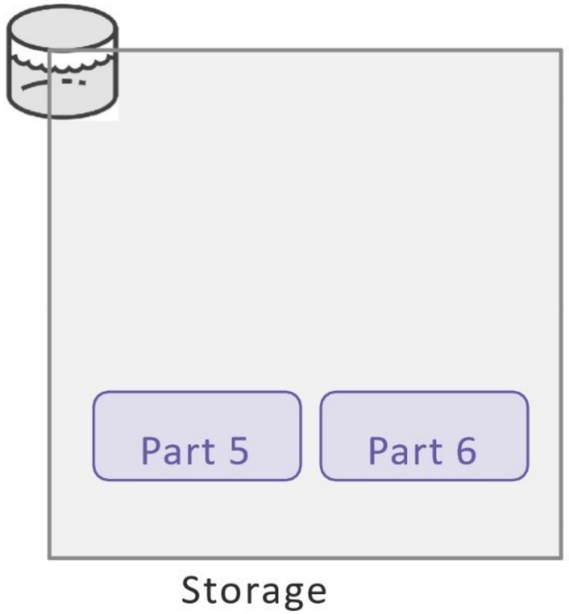


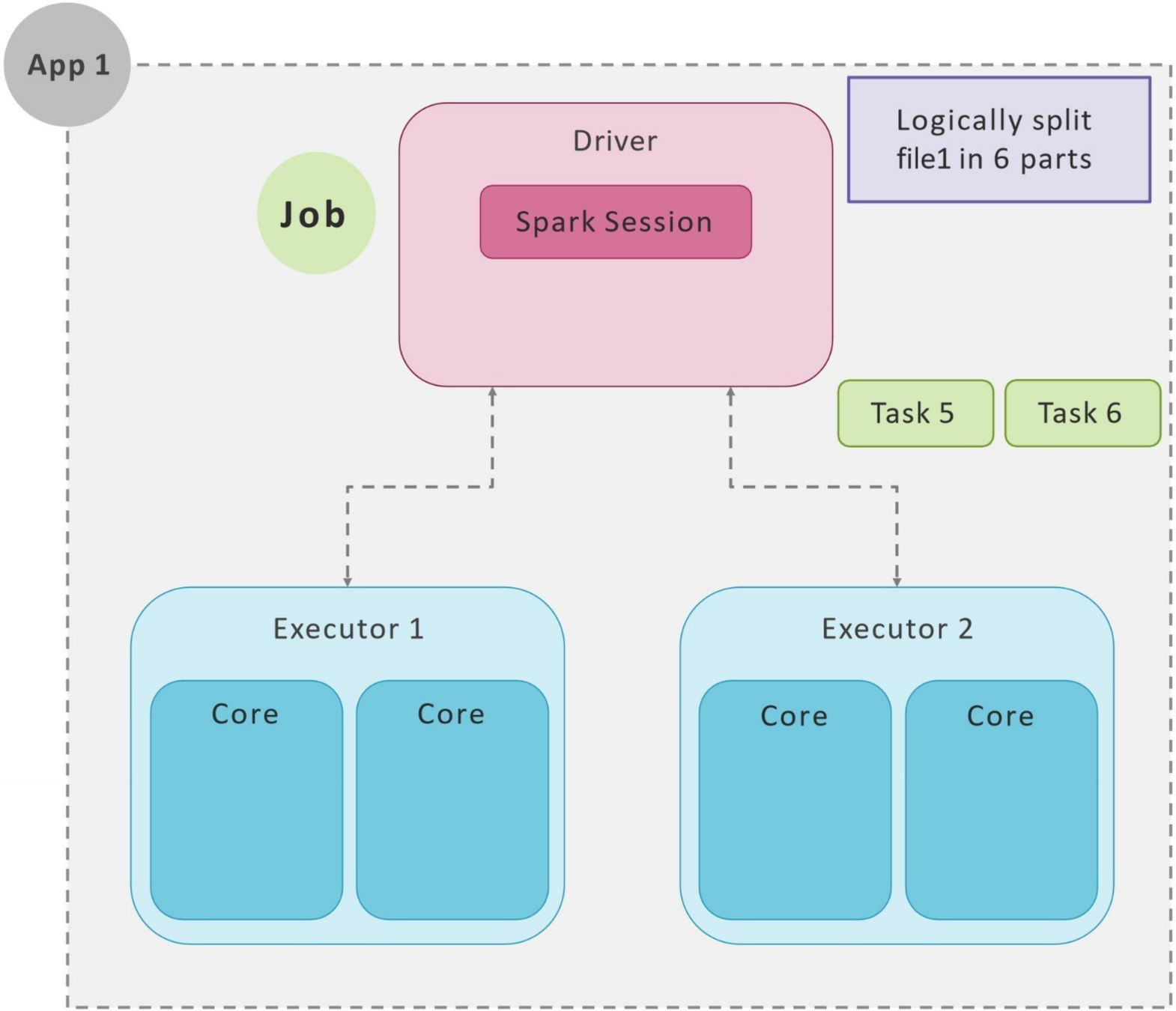
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



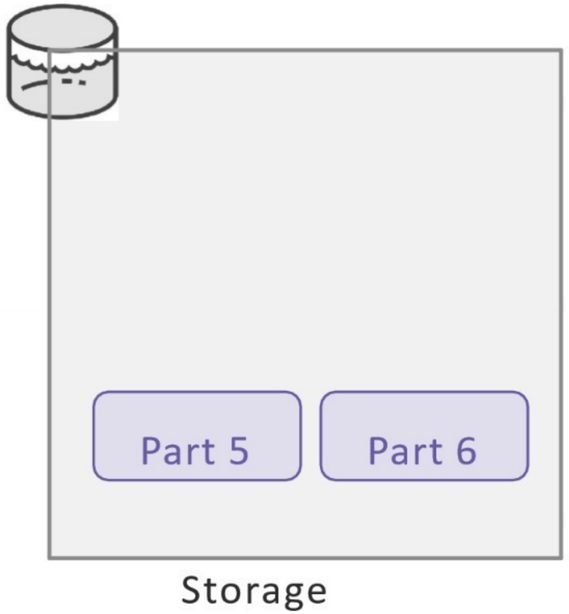


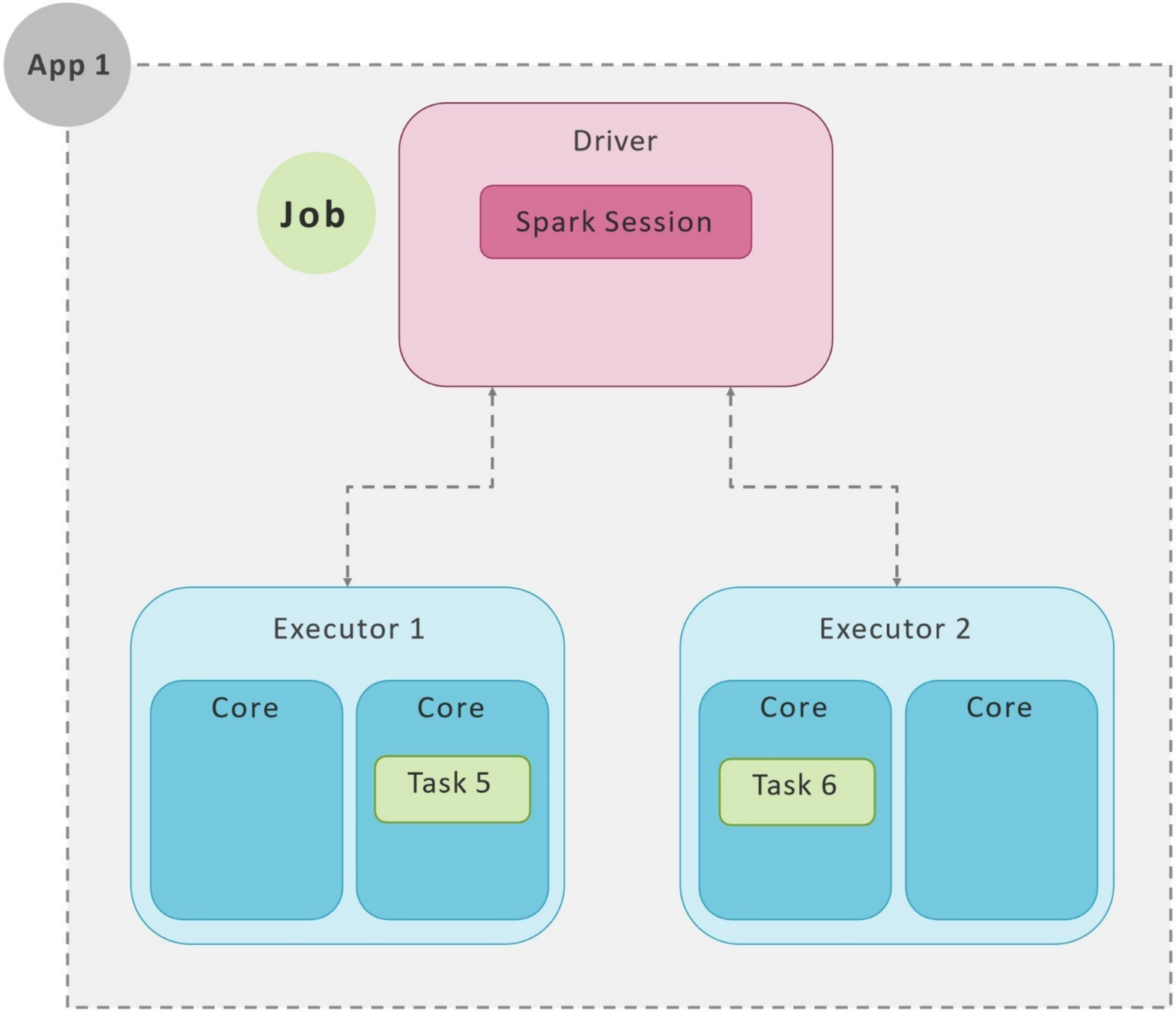
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage



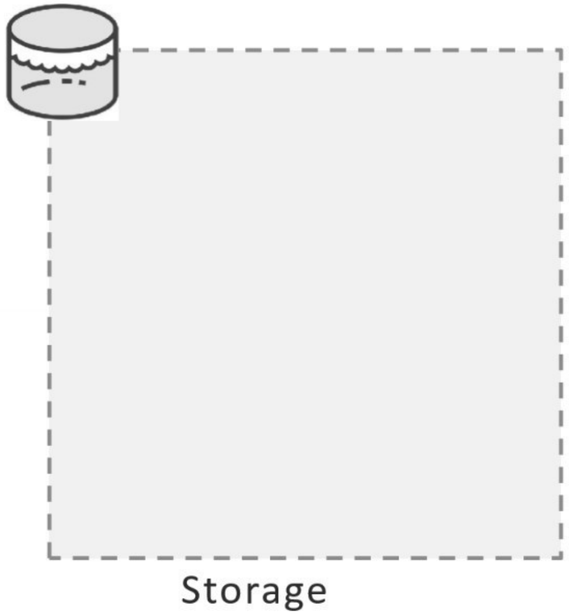


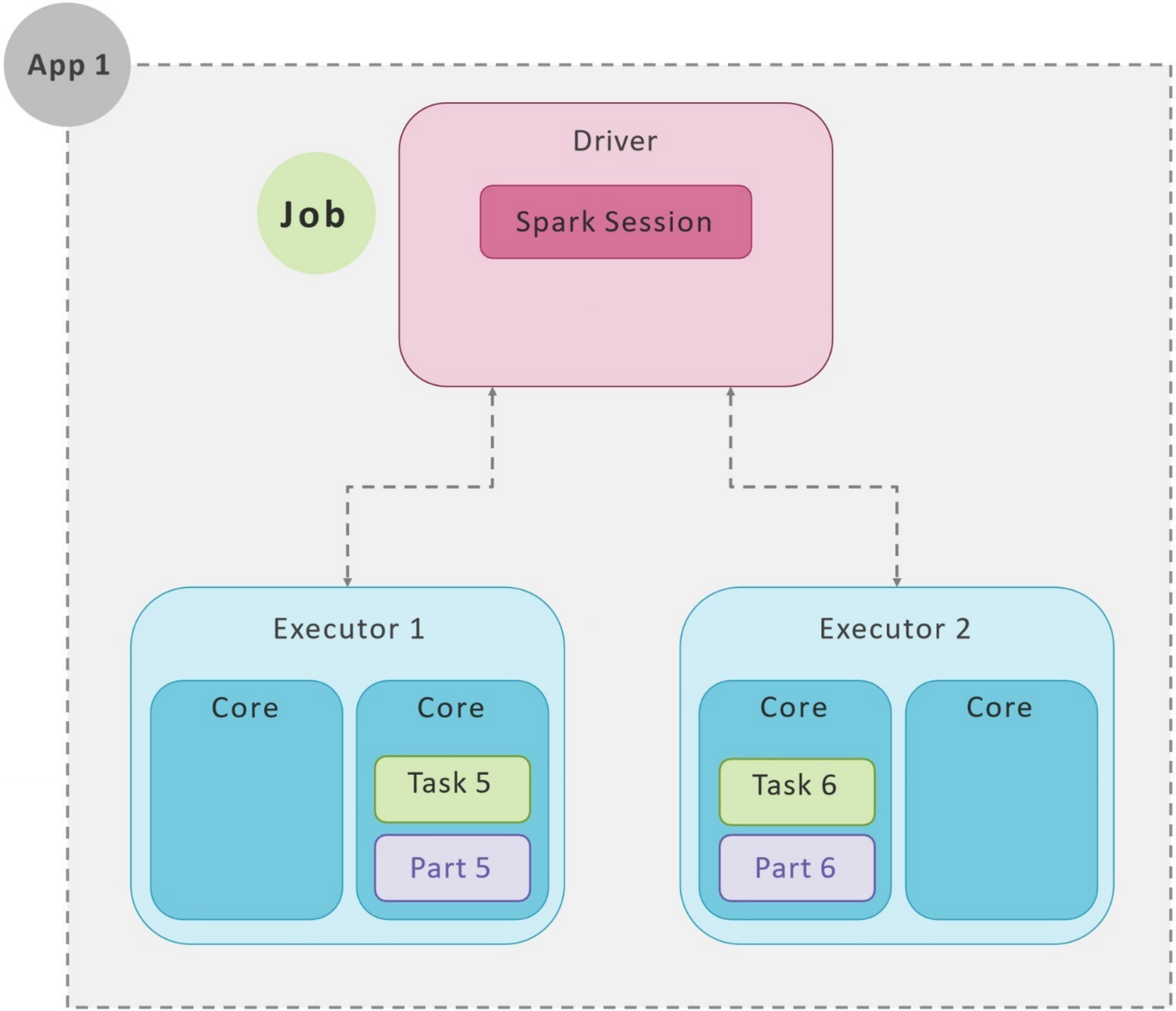
1. Read file1.csv from Storage
2. Apply transformations
3. Write processed data to Storage





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

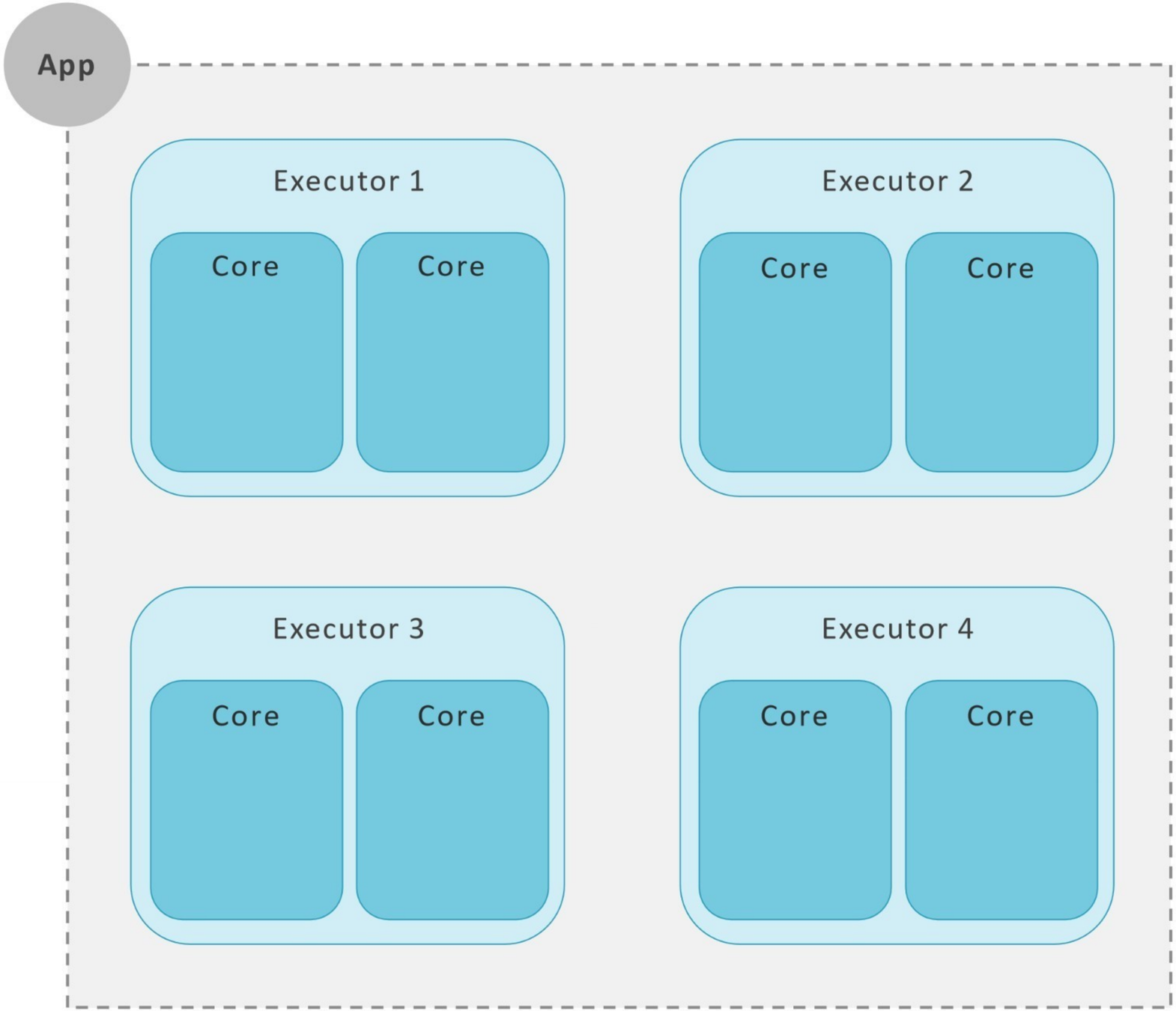
- Each core can execute only one task at a time
- Number of parallel tasks = Number of cores

Executor Size = 2 cores
14 GB RAM

Total Cores = 2 X 4
= 8 cores

Parallel Tasks = Total Cores

→ 8 tasks can execute in parallel



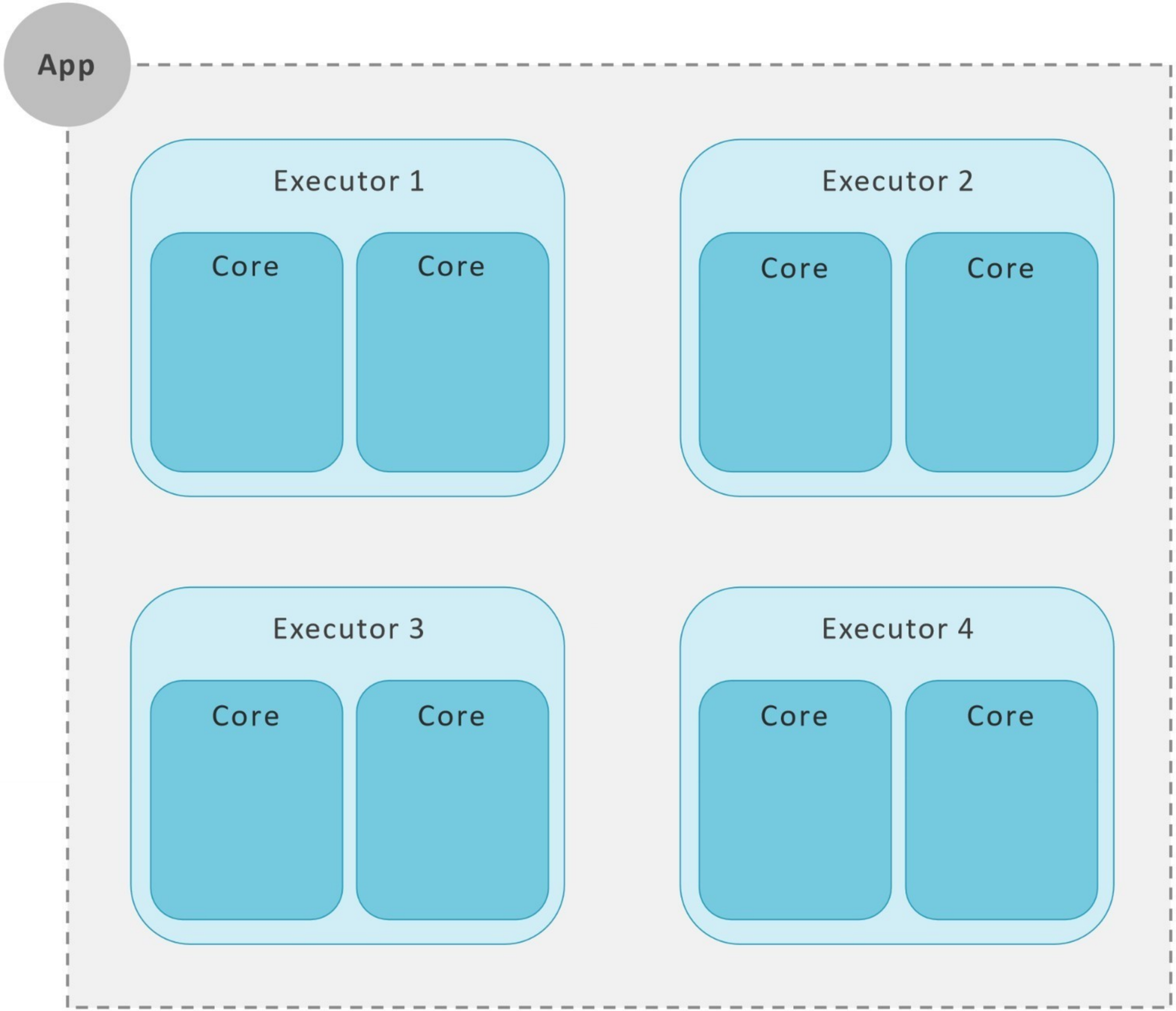
Partitions created by Driver = 16

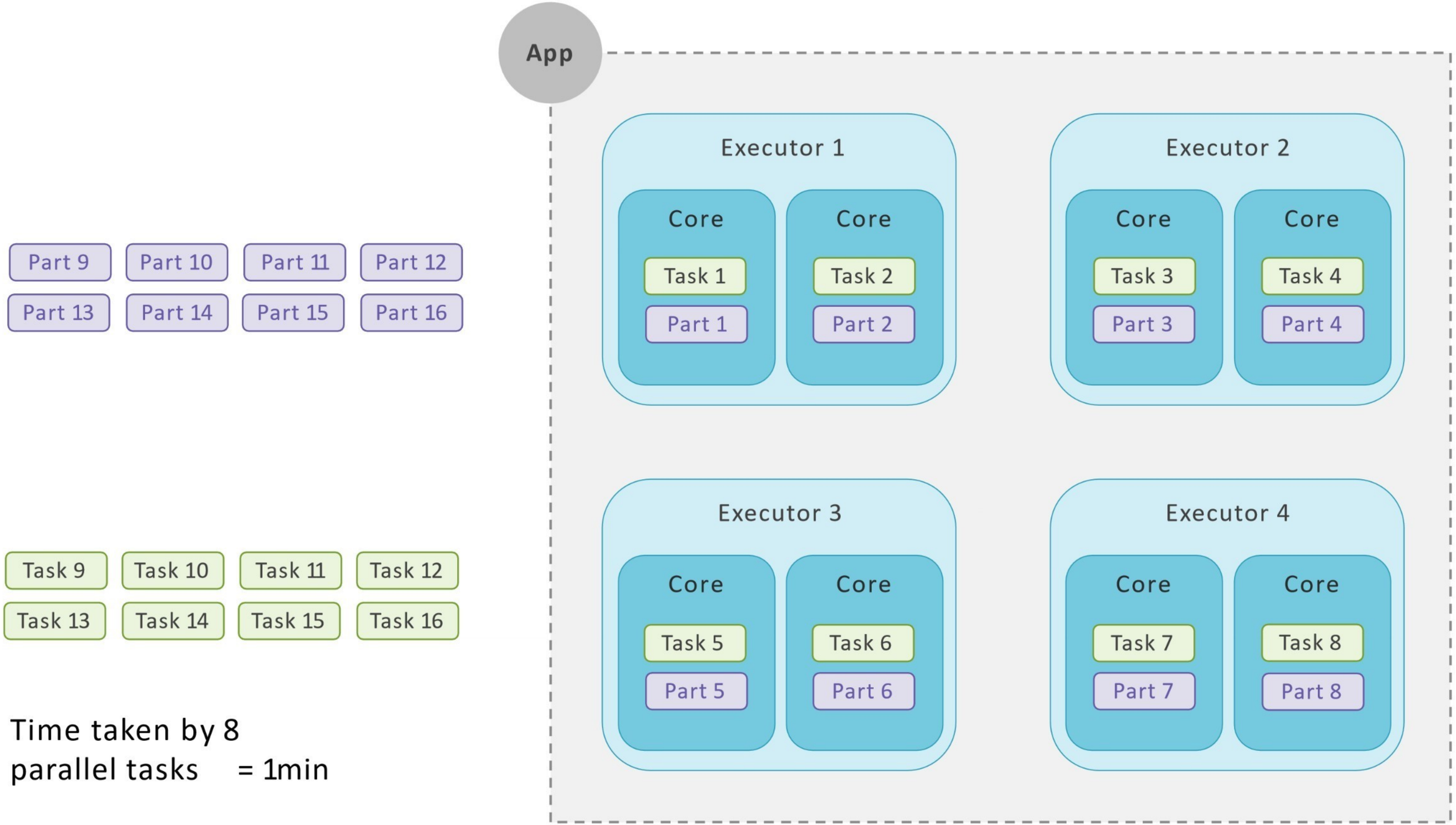
Part 1	Part 2	Part 3	Part 4
Part 5	Part 6	Part 7	Part 8
Part 9	Part 10	Part 11	Part 12
Part 13	Part 14	Part 15	Part 16

Total Tasks = Total Partitions

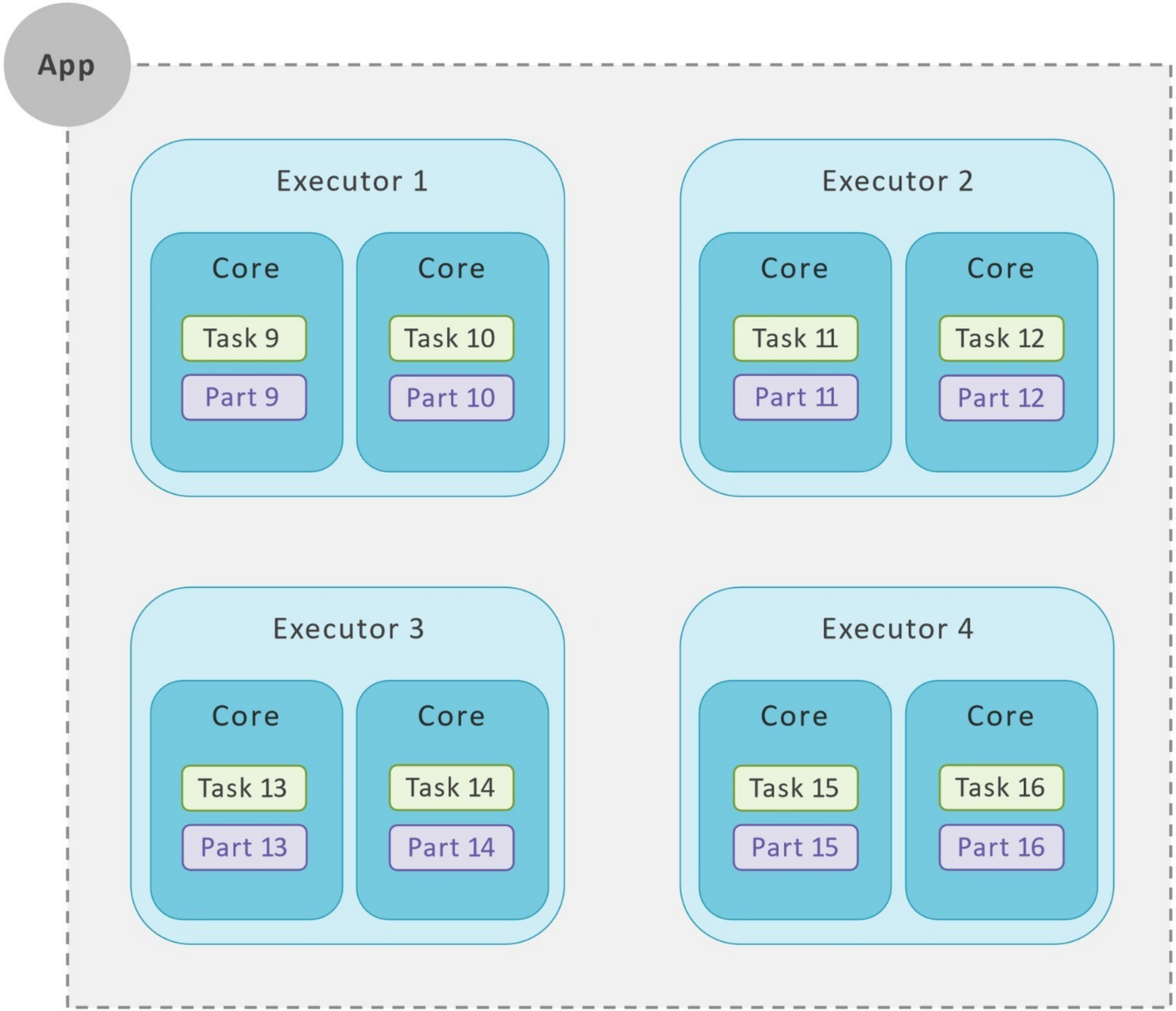
Task 1	Task 2	Task 3	Task 4
Task 5	Task 6	Task 7	Task 8
Task 9	Task 10	Task 11	Task 12
Task 13	Task 14	Task 15	Task 16

Time to complete task = 1min

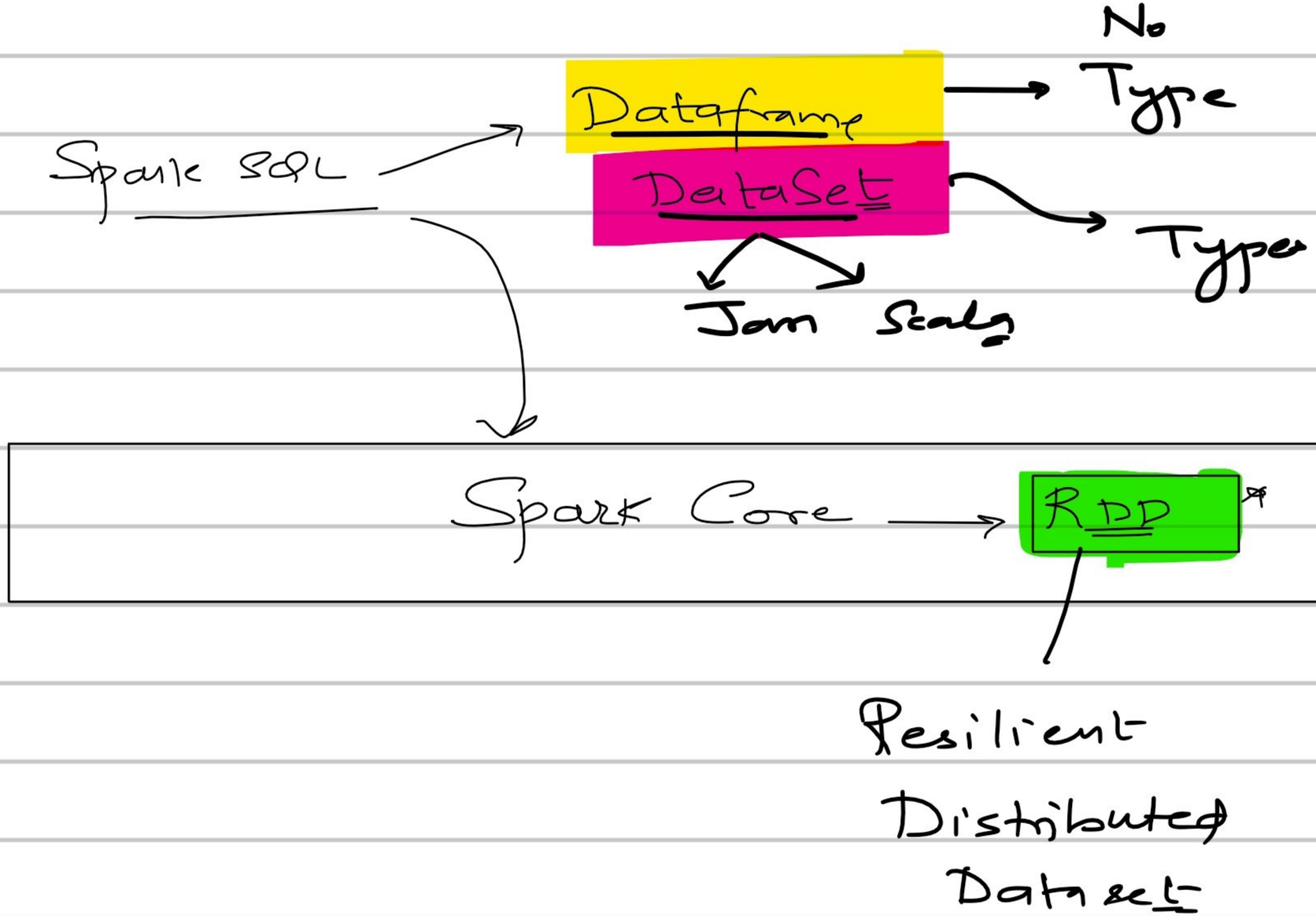



App
Executor 1
Core
Task 1
Part 1
Core
Task 2
Part 2
Executor 2
Core
Task 3
Part 3
Core
Task 4
Part 4
Executor 3
Core
Task 5
Part 5
Core
Task 6
Part 6
Executor 4
Core
Task 7
Part 7
Core
Task 8
Part 8
Part 9
Part 10
Part 11
Part 12
Part 13
Part 14
Part 15
Part 16
Task 9
Task 10
Task 11
Task 12
Task 13
Task 14
Task 15
Task 16
Time taken by 8
parallel tasks = 1min

Total time taken
by 16 tasks = 2 mins



Spark APIs – RDDs, DataFrames & Datasets


No Type
Dataframe
Spark SQL
DataSet
Type
Java
Scala
Spark Core
RDD
Resilient
Distributed
Dataset

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

Introduced with Spark 1.3

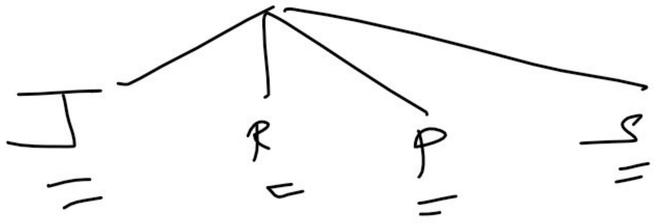
Based on RDDs

Collection of data, but in tabular format

No compile-time safety

Spark applies optimizations to code

Supported in all languages



Datasets

Introduced with Spark 1.6

Based on RDDs

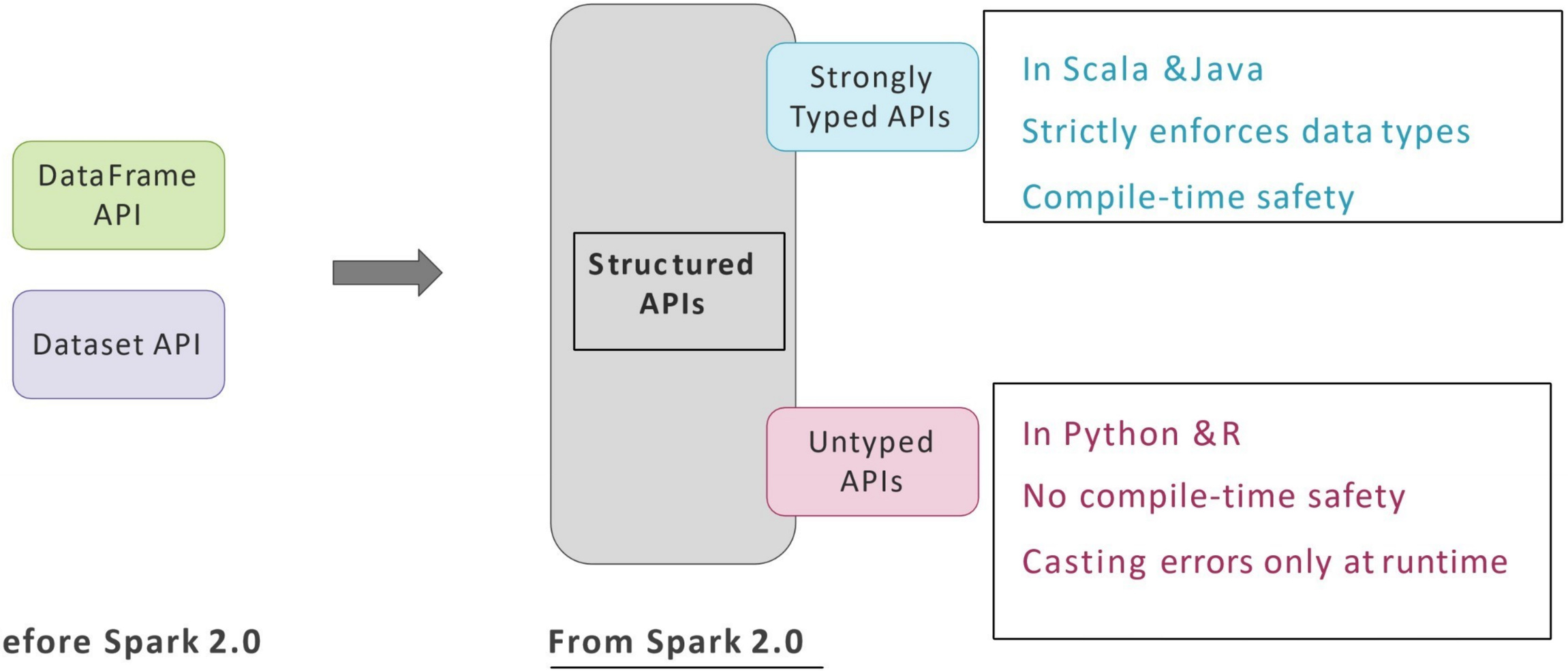
Combination of RDDs and DataFrames

Provides compile-time safety

Spark applies optimizations to code

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

Setting up Spark Environment

Understanding Spark Environments

Spark Development Options

Command line / Spark Shell

Notebooks

- Jupyter, Zeppelin
- Can run locally or in cloud platforms like Azure HDInsight & Google Dataproc

IDEs to build projects

- IntelliJ, PyCharm, VS Code
- Run code locally, or build & run in Spark cluster

Spark Submit command to run jobs

Cloud platform notebooks

- Offered by Databricks, Azure Synapse Analytics etc.

Cluster Managers

Manages & allocates resources to applications on the cluster

Local

Standalone

YARN

Mesos

Kubernetes

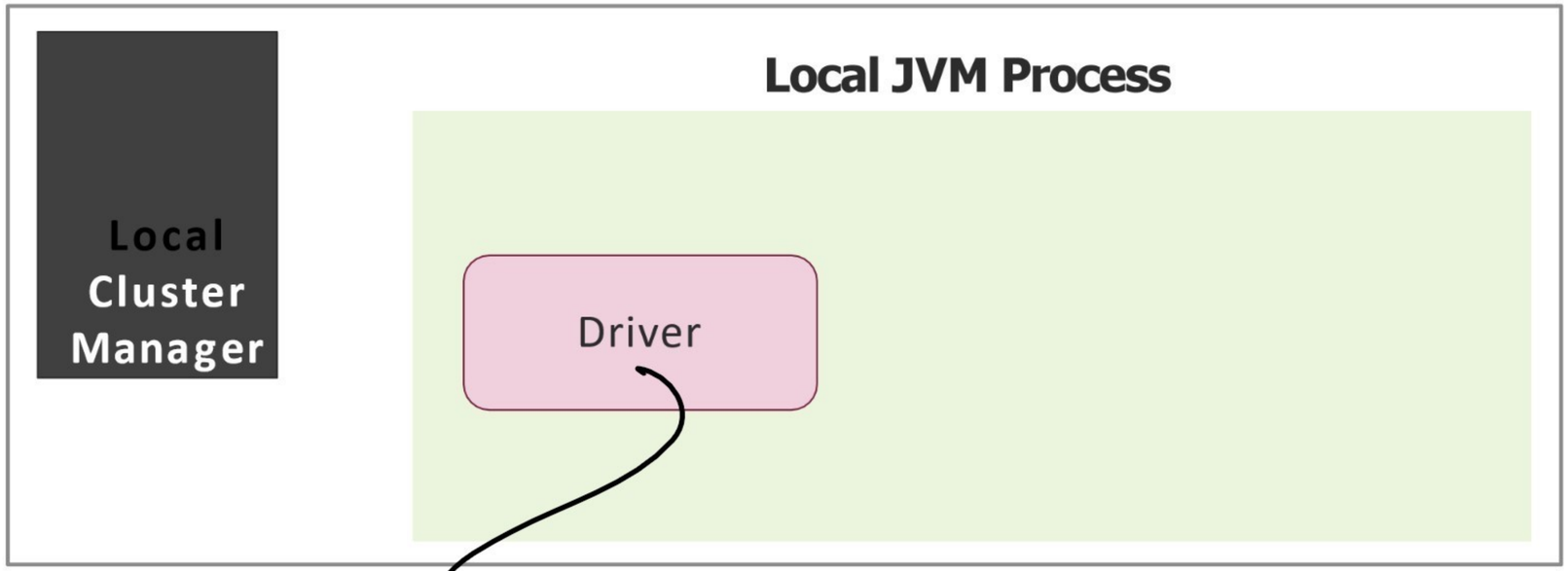
Application Execution Modes

Local Mode

Client Mode

Cluster Mode

Local Execution Mode



Driver = Executor

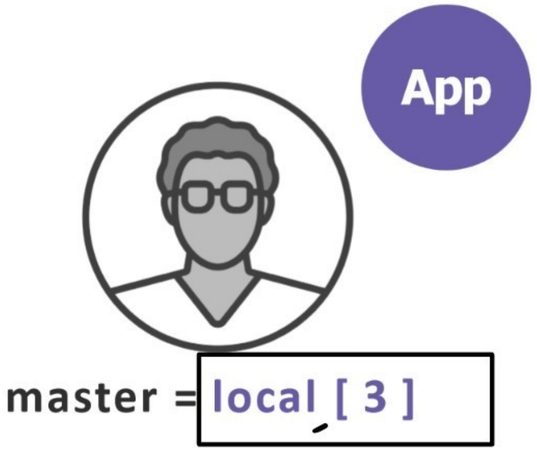
Single thread is used

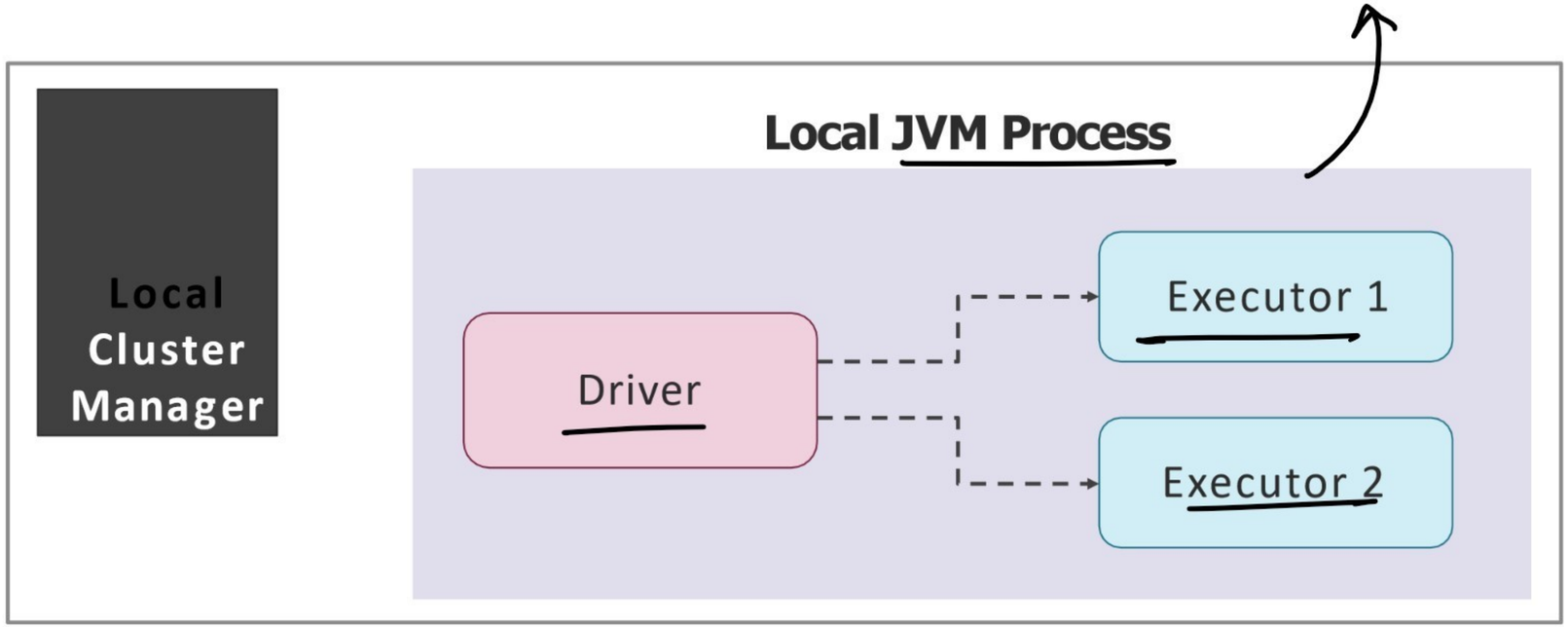
Driver runs all the jobs

No executors are created

Local Execution Mode

Core





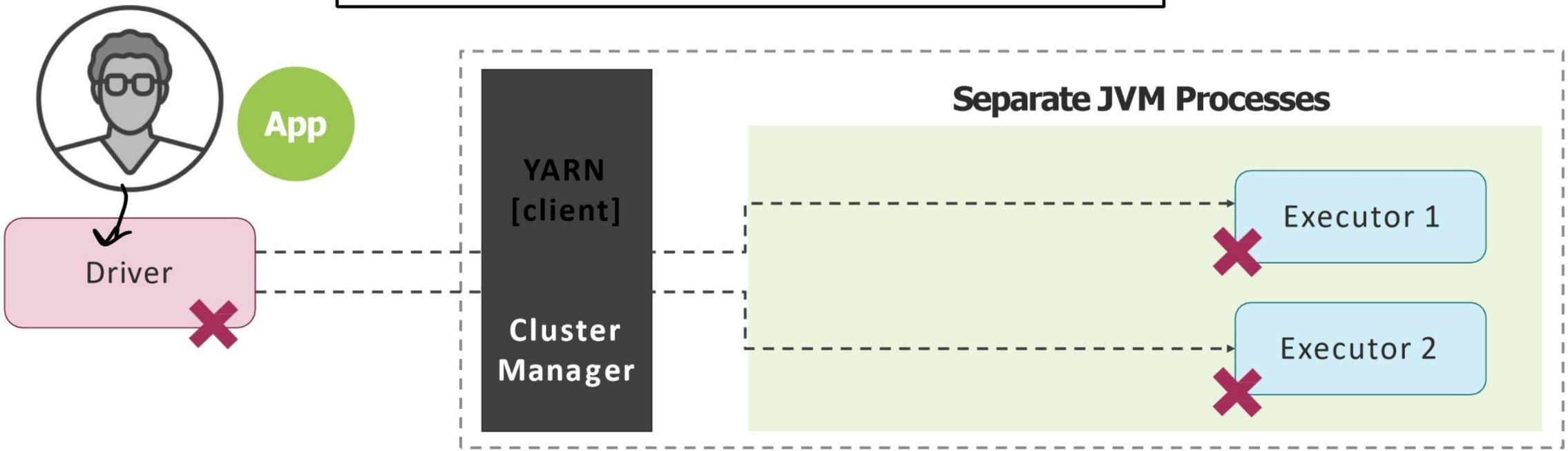
Three threads are created in same JVM

Executors run tasks

Simulation of production cluster environment

Execution Mode	How it works?	Cluster Managers	Development Tools	Purpose
Local	Driver & executors on same JVM	Local	Spark Shell Notebooks IDEs	Run locally for dev/testing

Client Execution Mode

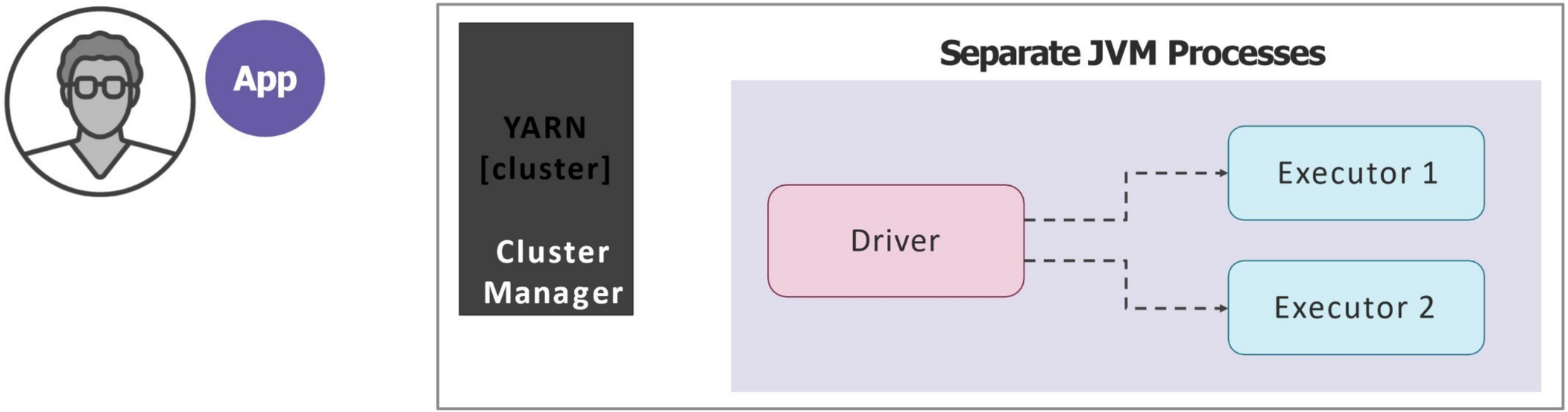


Driver runs on client machine

Each executor runs as a separate JVM process in cluster

Execution Mode	How it works?	Cluster Managers	Development Tools	Purpose
Local	Driver &executors on same JVM	Local	Spark Shell Notebooks IDEs	Run locally for dev/testing
Client	Driver on client, Executors on cluster	YARN [client] Standalone	Spark Shell Notebooks Spark-submit	Interactive querying, Data analysis

Cluster Execution Mode



Driver & executors run on cluster

Each process consumes resources on cluster

Used to run production workloads

Execution Mode	How it works?	Cluster Managers	Development Tools	Purpose
Local	Driver &executors on same JVM	Local	Spark Shell Notebooks IDEs	Run locally for dev/testing
Client	Driver on client, Executors on cluster	YARN [client] Standalone	Spark Shell Notebooks Spark-submit	Interactive querying, Data analysis
Cluster	Driver &executors in separate JVMs on cluster	YARN [cluster] Standalone Kubernetes Mesos	Spark-submit Cloud notebooks	Long running jobs in production

Let’s Install Spark

Spark 3+ Minimum Requirements

Windows, Linux, MacOS

Java v8 (8u92+) / v11 / v17

Scala 2.12 and above 2.13

Python 3.6 and above

Unzip utility like 7-Zip

Winutils.exe *(for Windows only)*

hadoop.dll

Option 4:
Running Jobs with Spark Submit

Anaconda Navigator

File Help

ANACONDA.NAVIGATOR

Connect

Home

Environments

Learning

Community

Documentation

Anaconda Blog

Search Environments

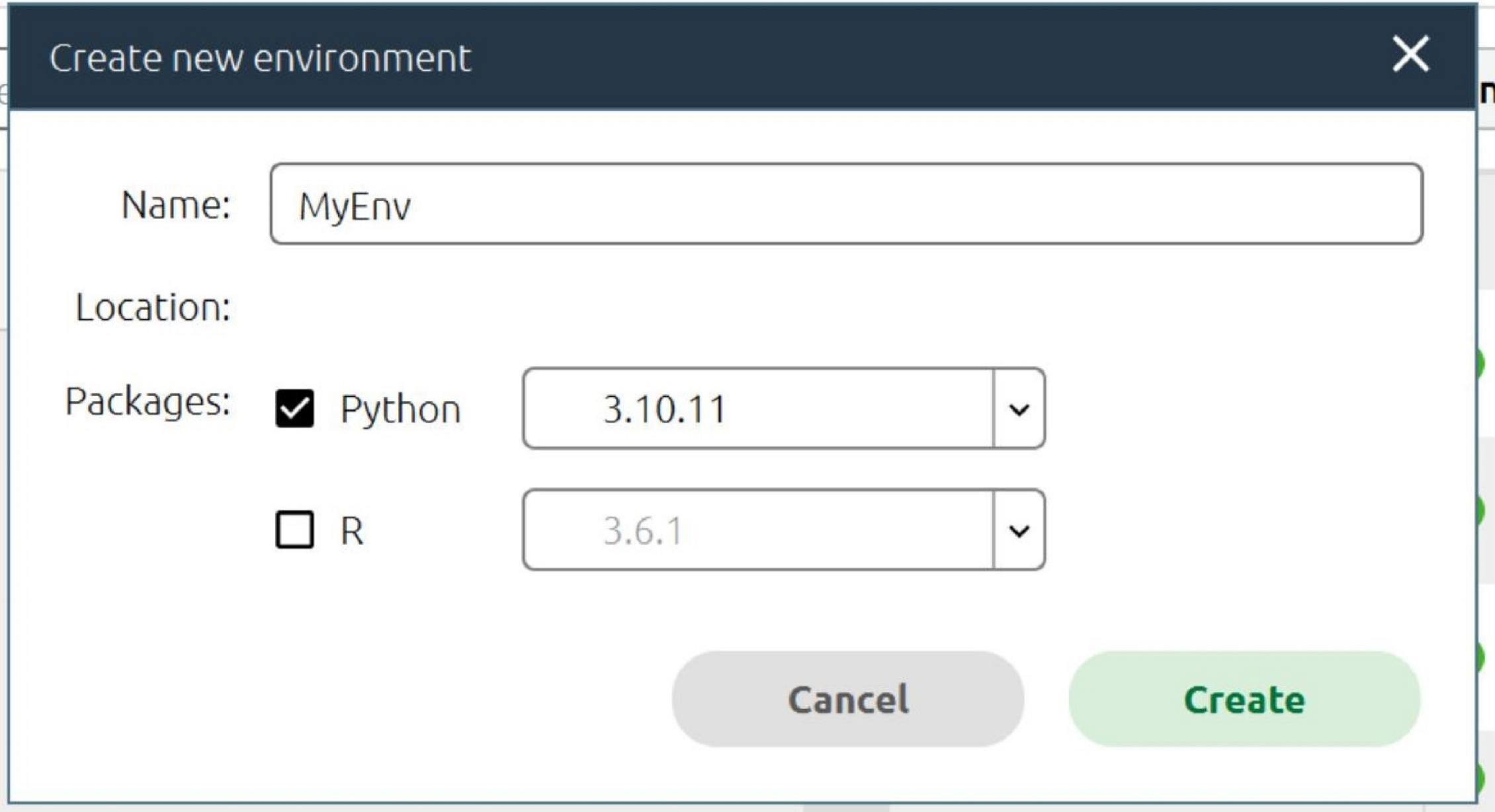
base (root)

Create Clone Import Backup Remove

Installed Channels Update index... Search Pac

Name	T	Description	Ver
alabaster		Configurable, python 2+3 compatible sphinx theme.	
anaconda-client		Anaconda.org command line client library	
anaconda-project		Tool for encapsulating, running, and reproducing data science projects	
anyio		High level compatibility layer for ...	
appdirs		A small python module for determining appropriate platform-specific dirs.	
argon2-cffi		The secure argon2 password hashing algorithm	

446 packages available


Create new environment
Name:
MyEnv
Location:
Packages:
Python
3.10.11
R
3.6.1
Cancel
Create

Anaconda Navigator

File Help

ANACONDA.NAVIGATOR

Connect

Home

Environments

Learning

Community

Documentation

Anaconda Blog

Search Environments

base (root)

MyEnv

Open Terminal

Open with Python

Open with IPython

Open with Jupyter Notebook

Create Clone Import Backup Remove

Installed Channels Update index... Search Pac

Name	T	Description	Ver
		High-quality data compressor	
		Certificates for use with other packages.	
		A portable foreign function interface library	
openssl		Openssl is an open-source implem…	
pip		Pypa recommended tool for installing python packages	
python		General purpose programming language	

15 packages available

ENG 06:38 AM

C:\windows\system32\cmd.e›

```
(MyEnv) C:\Users\VICTUS>conda install -c conda-forge findspark pyhive delta-spark jupyter
```

Spark-Submit

Utility to submit a Spark Application / Job to a cluster

Code can be in any language

- Scala, Python or Java

Can be submitted to any supported Cluster Manager

- Local, Standalone, YARN, Mesos, Kubernetes

Provide configuration and dependencies

Both Spark Shell & Spark-Submit are command line

- Spark Shell is typically used for interactive querying
- Spark-Submit is typically used to submit long running jobs

```
==========================================================
Steps to create multi-node cluster on local machine
==========================================================

A. In Command Prompt 1 - Spark Master

A.1. Create Spark master
      => spark-class org.apache.spark.deploy.master.Master --host <machine name or localhost>

      - Example => spark-class org.apache.spark.deploy.master.Master --host localhost

A.2. Copy Master URL and Spark UI URL
      - Typically master URL is at port 7077
      - Typically Spark UI is at port 4040
      - If any of these ports are in use, a different port might be used

-------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------------------------------------

B. In Command Prompt 2 - Worker Node 1

B.1. Create worker node 1
      => spark-class org.apache.spark.deploy.worker.Worker <master URL> --host <machine name or localhost>

      - Example => spark-class org.apache.spark.deploy.worker.Worker spark://localhost:7077 --host localhost

-----------------------------------------------------------------------------------------------------------

C. In Command Prompt 3 - Worker Node 2

C.1. Create worker node 2
      => spark-class org.apache.spark.deploy.worker.Worker <master URL> --host <machine name or localhost>

      - Example => spark-class org.apache.spark.deploy.worker.Worker spark://localhost:7077 --host localhost

-----------------------------------------------------------------------------------------------------------
```

Summary

Can work with Spark in various ways

- Several development options and cluster managers

Understood application execution modes

- Local mode – Single JVM to run driver & executors
- Client mode – Driver on client & executors on cluster
- Cluster mode – Driver & executors run on cluster

Spark can be installed on local machine or on cluster

Development Options

- Command Line can be used for dev/testing
- Jupyter for interactive analysis or development
- PyCharm IDE for project development
- Spark-submit to run long running production jobs

Web UI can be used for monitoring

Working with RDDs – Resilient Distributed Datasets

Understanding RDDs

Resilient Distributed Dataset (RDD) is the native Data Structure of Spark

Resilient Distributed Datasets

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

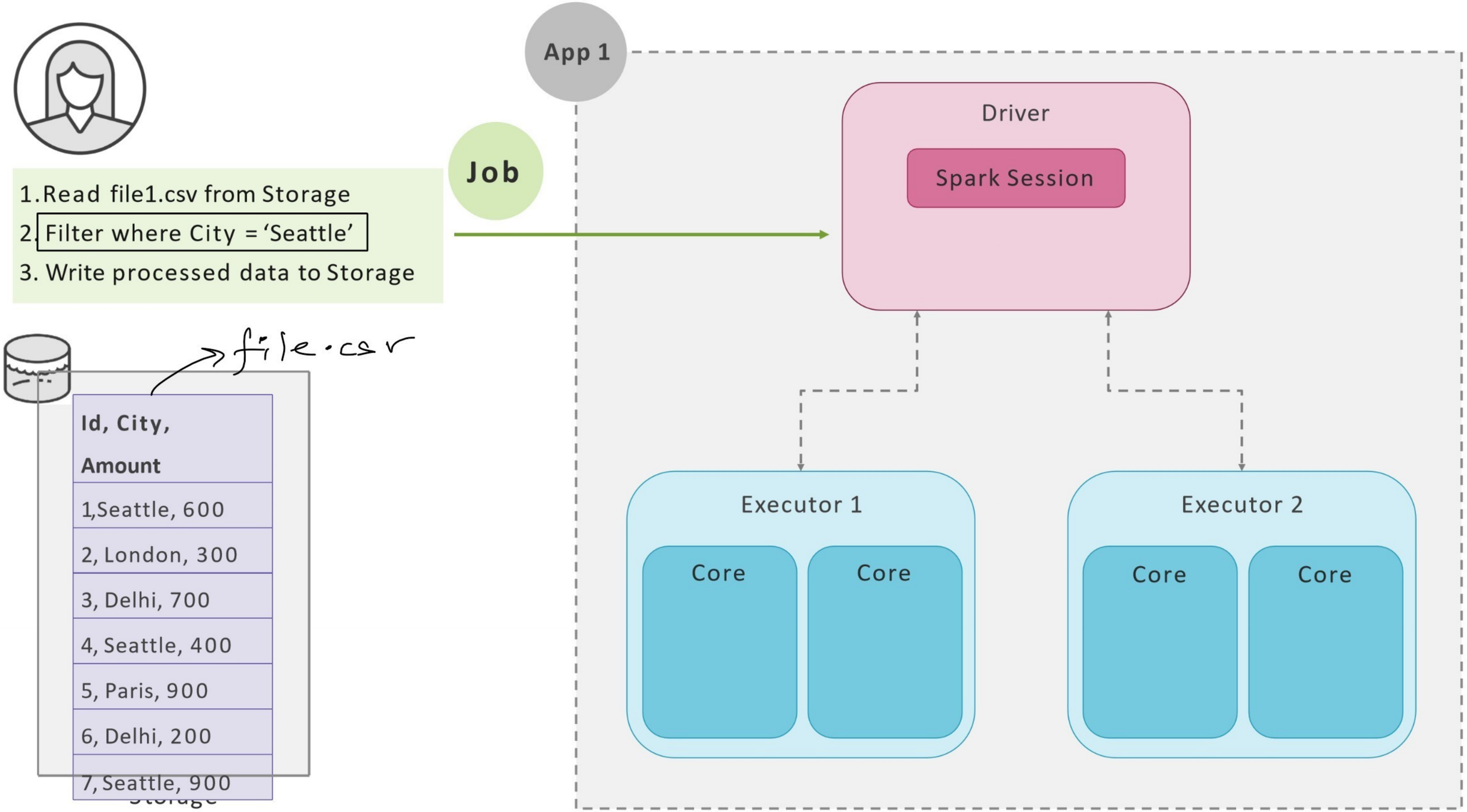
CSV File in Storage

Read File with
RDD API

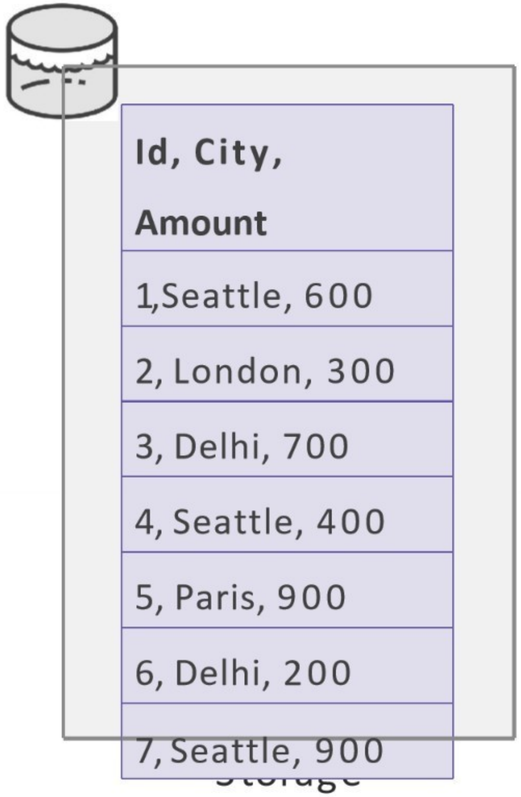
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900

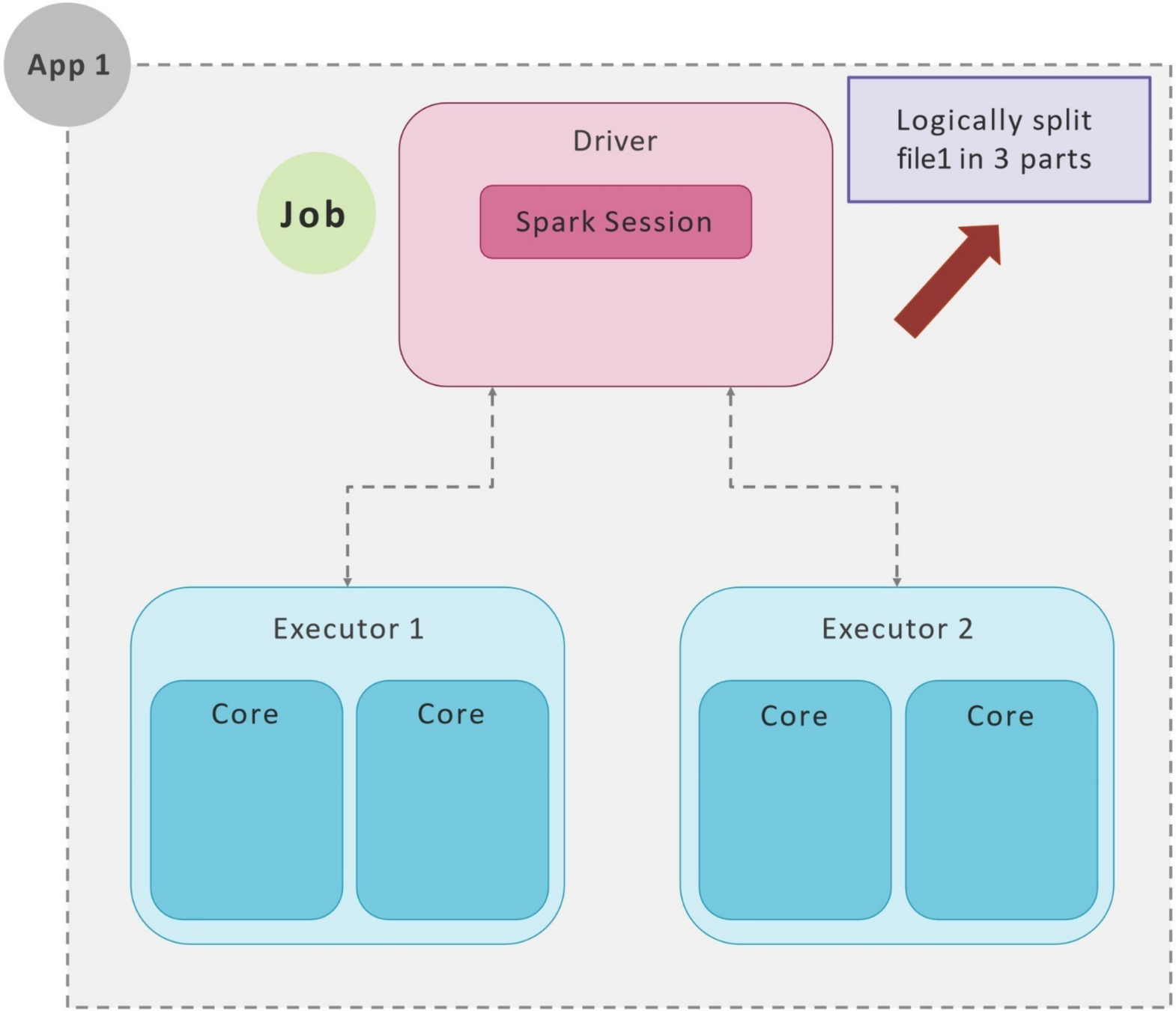
RDD in Spark

- In-memory objects
- Do not have schema
- Distributed collection of elements
- All processing in Spark happens on RDDs

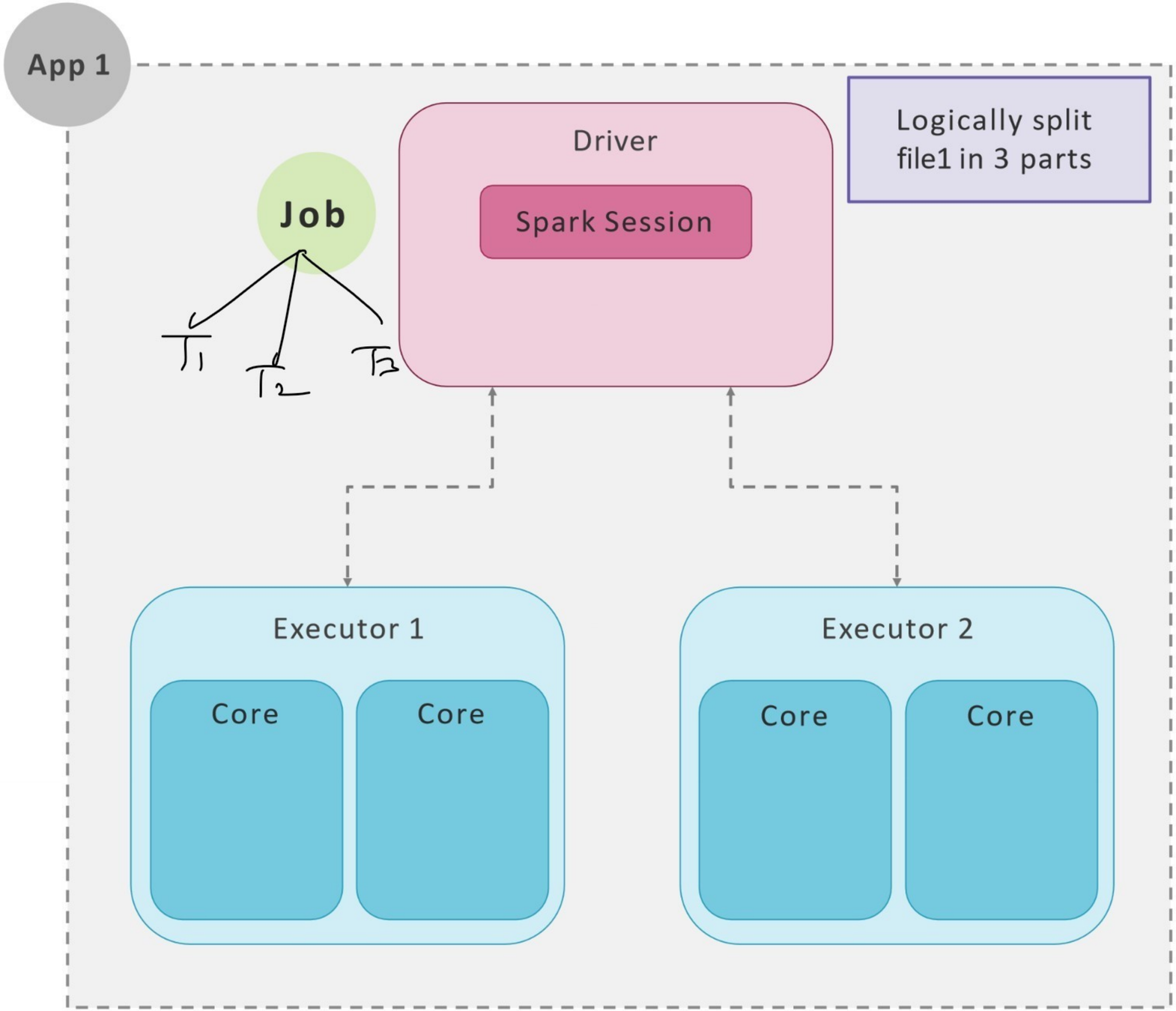

1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Job
App 1
Driver
Spark Session
Executor 1
Core
Core
Executor 2
Core
Core
file.csv
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Storage

1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage

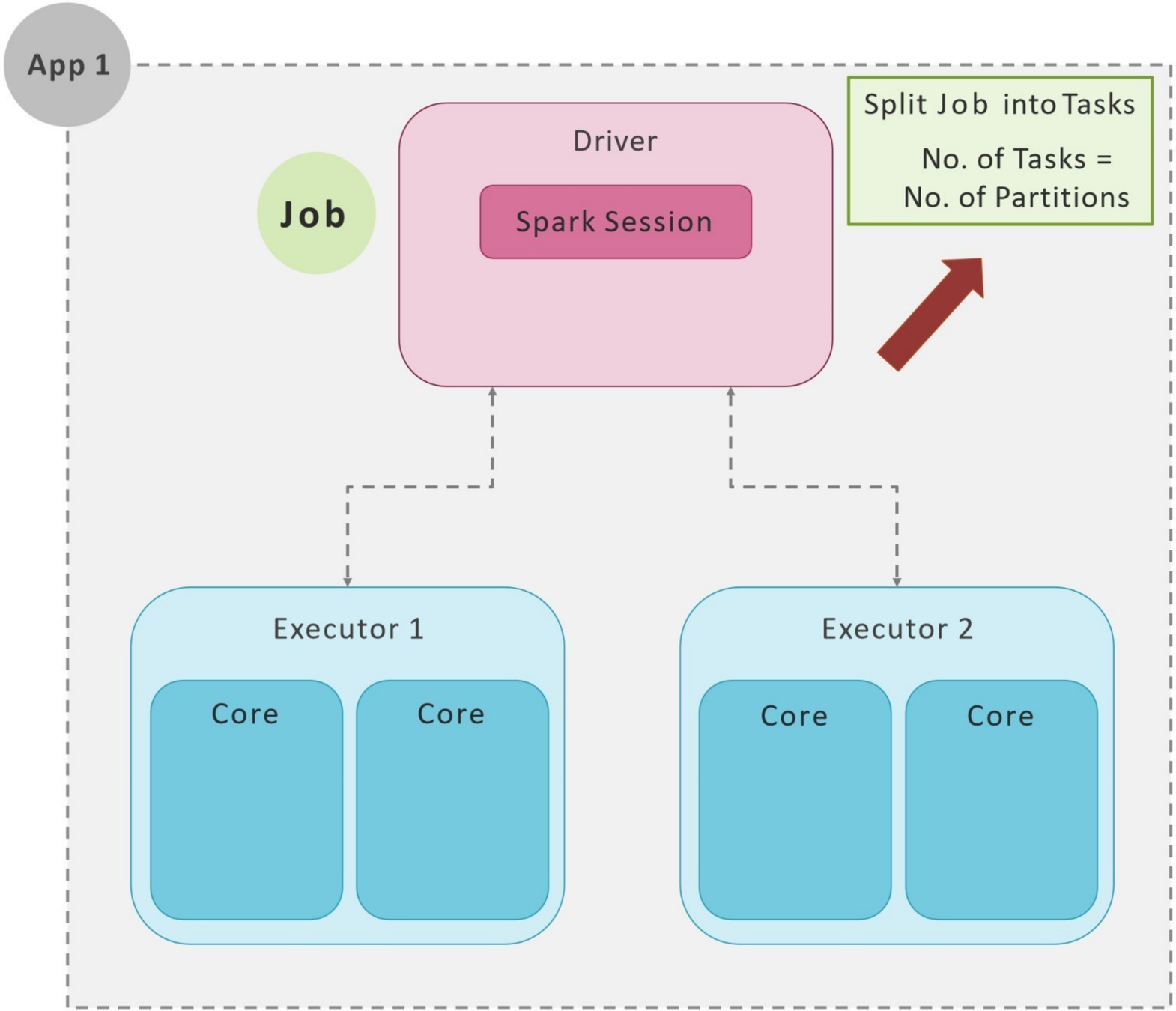




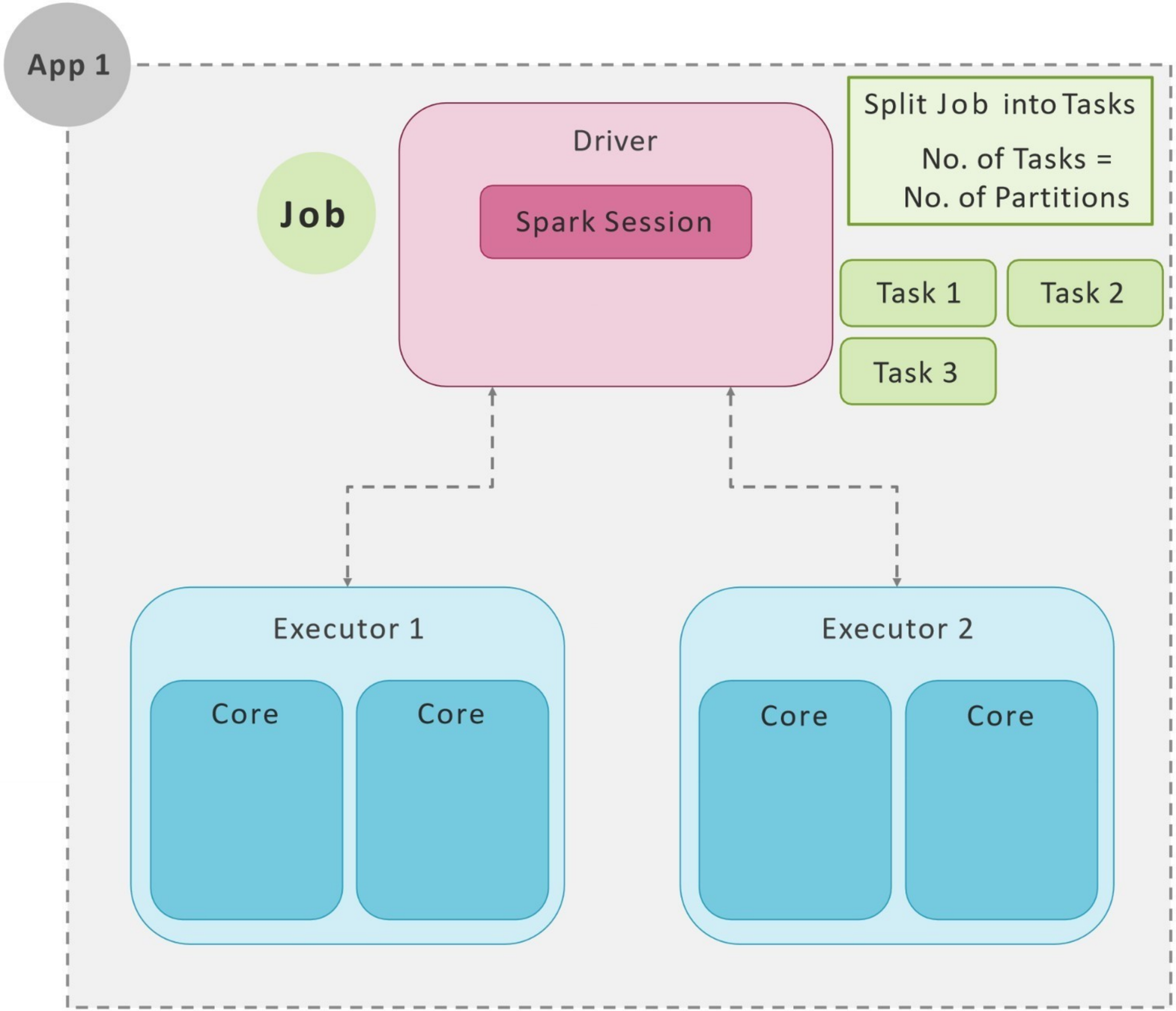
1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage

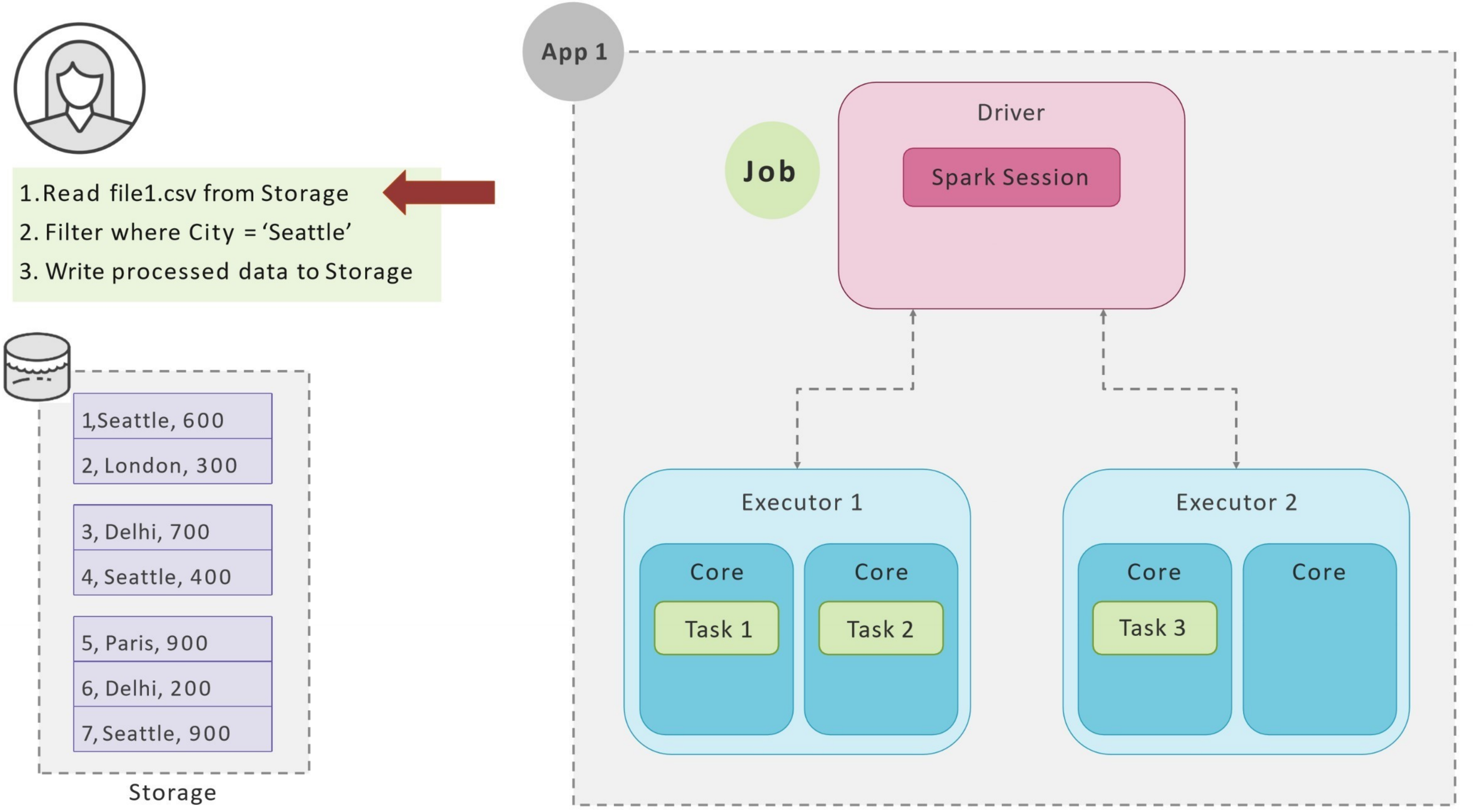


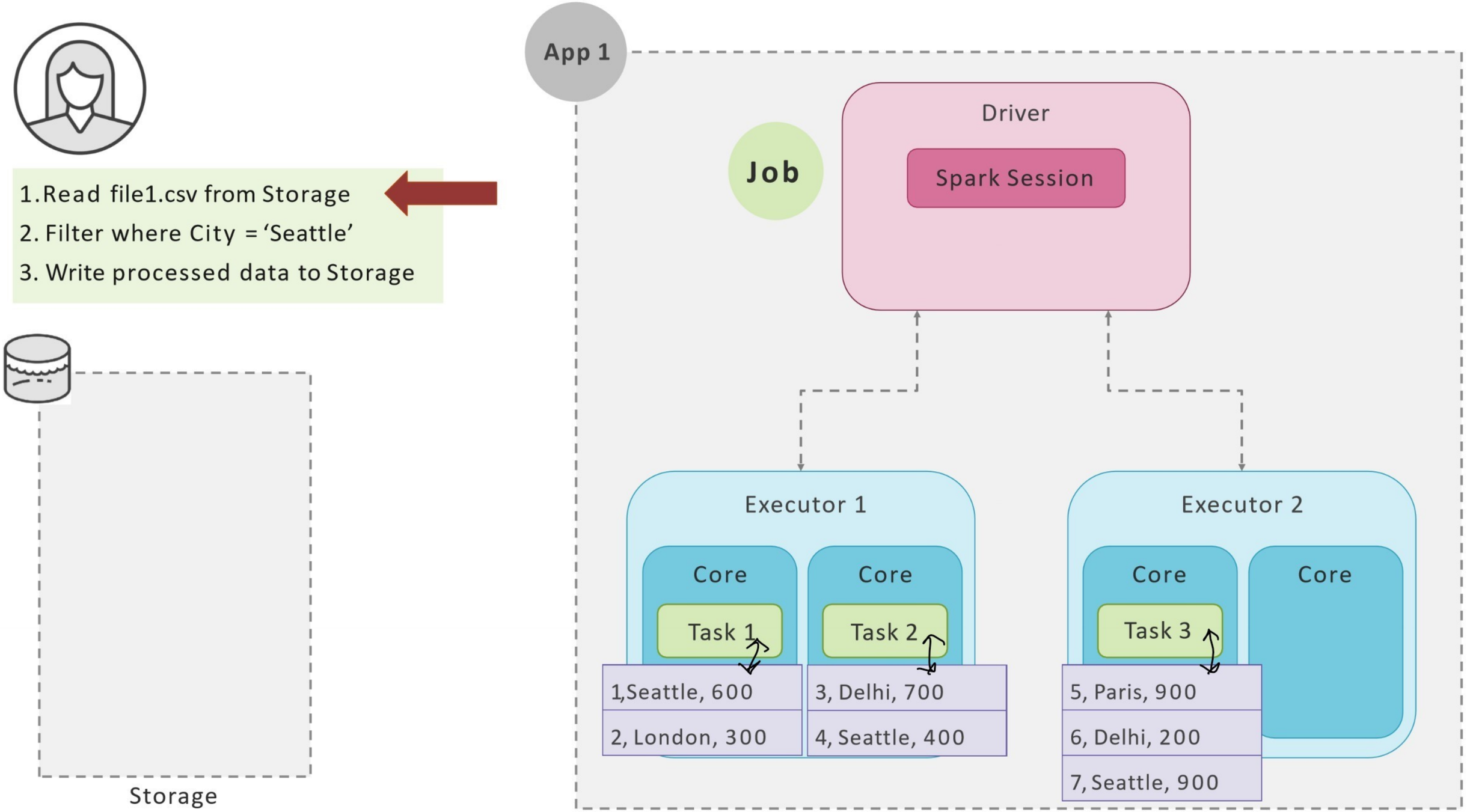
1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage

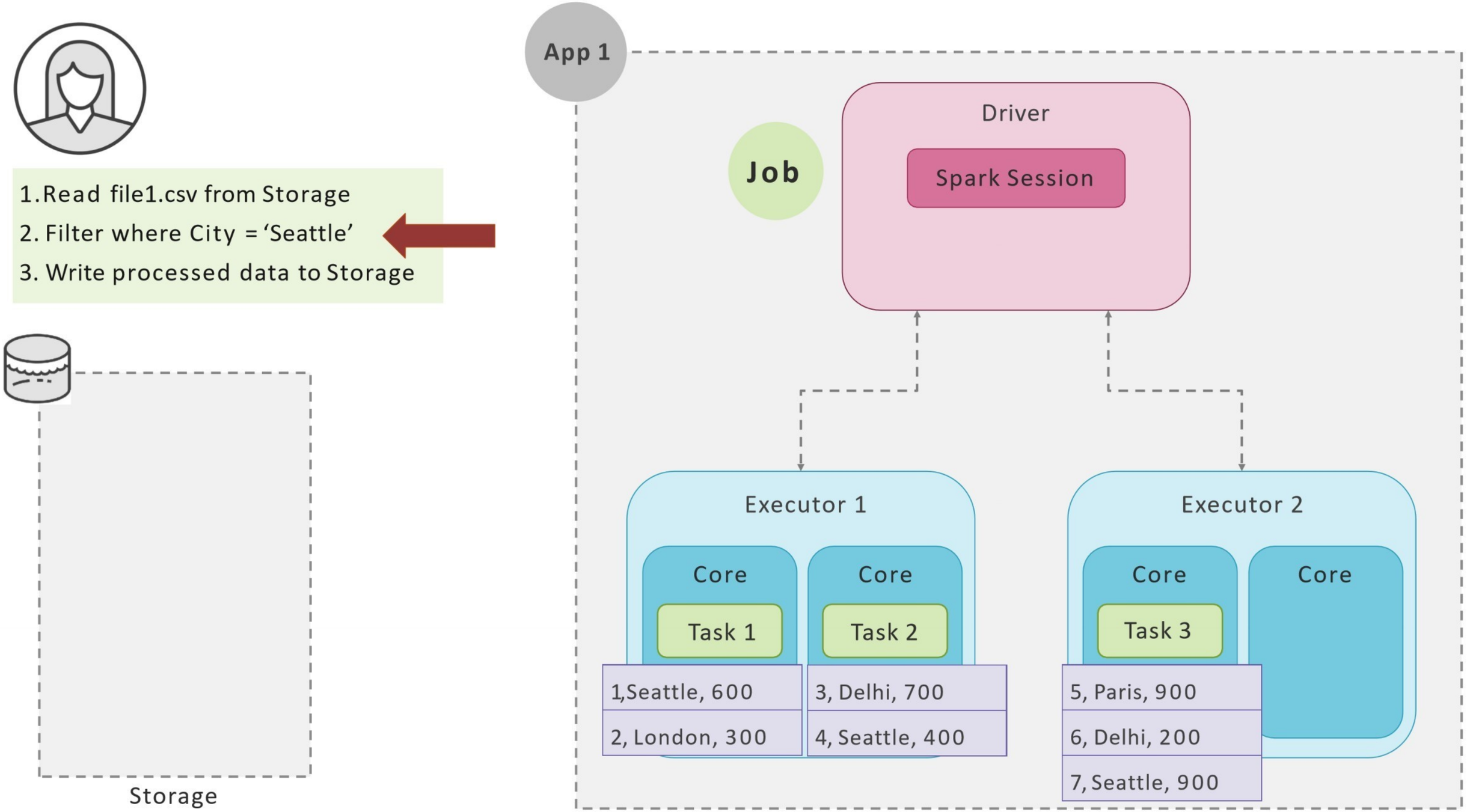


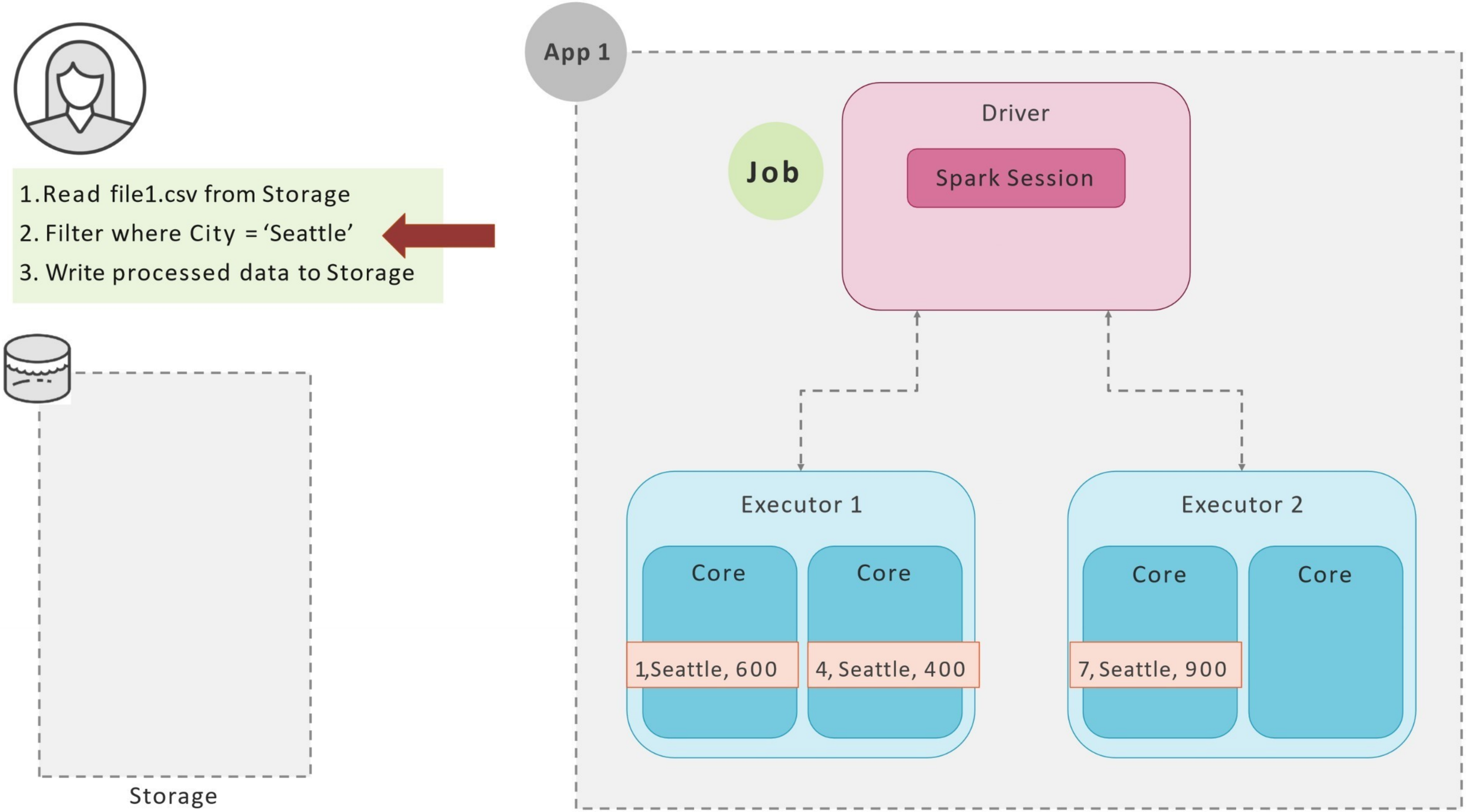
1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage

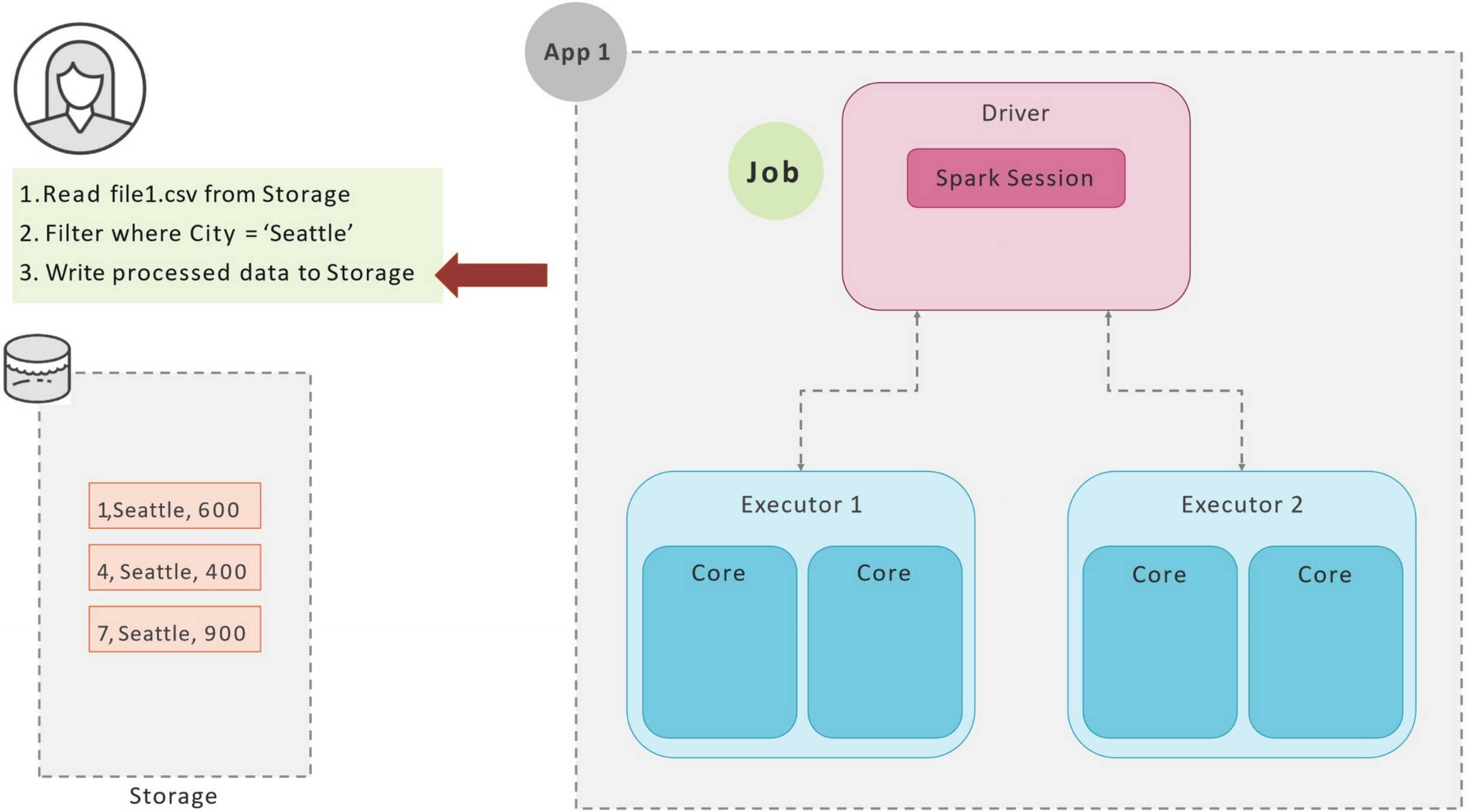



1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Core
Task 2
Executor 2
Core
Task 3
Core


1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Core
Task 1
Task 2
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
Executor 2
Core
Core
Task 3
5, Paris, 900
6, Delhi, 200
7, Seattle, 900


1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Core
Task 2
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
Executor 2
Core
Task 3
Core
5, Paris, 900
6, Delhi, 200
7, Seattle, 900


1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Core
1,Seattle, 600
4, Seattle, 400
Executor 2
Core
Core
7, Seattle, 900


App 1
Job
Driver
Spark Session
1.Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Executor 1
Core
Core
Executor 2
Core
Core
1,Seattle, 600
4, Seattle, 400
7, Seattle, 900
Storage

Features of RDDs

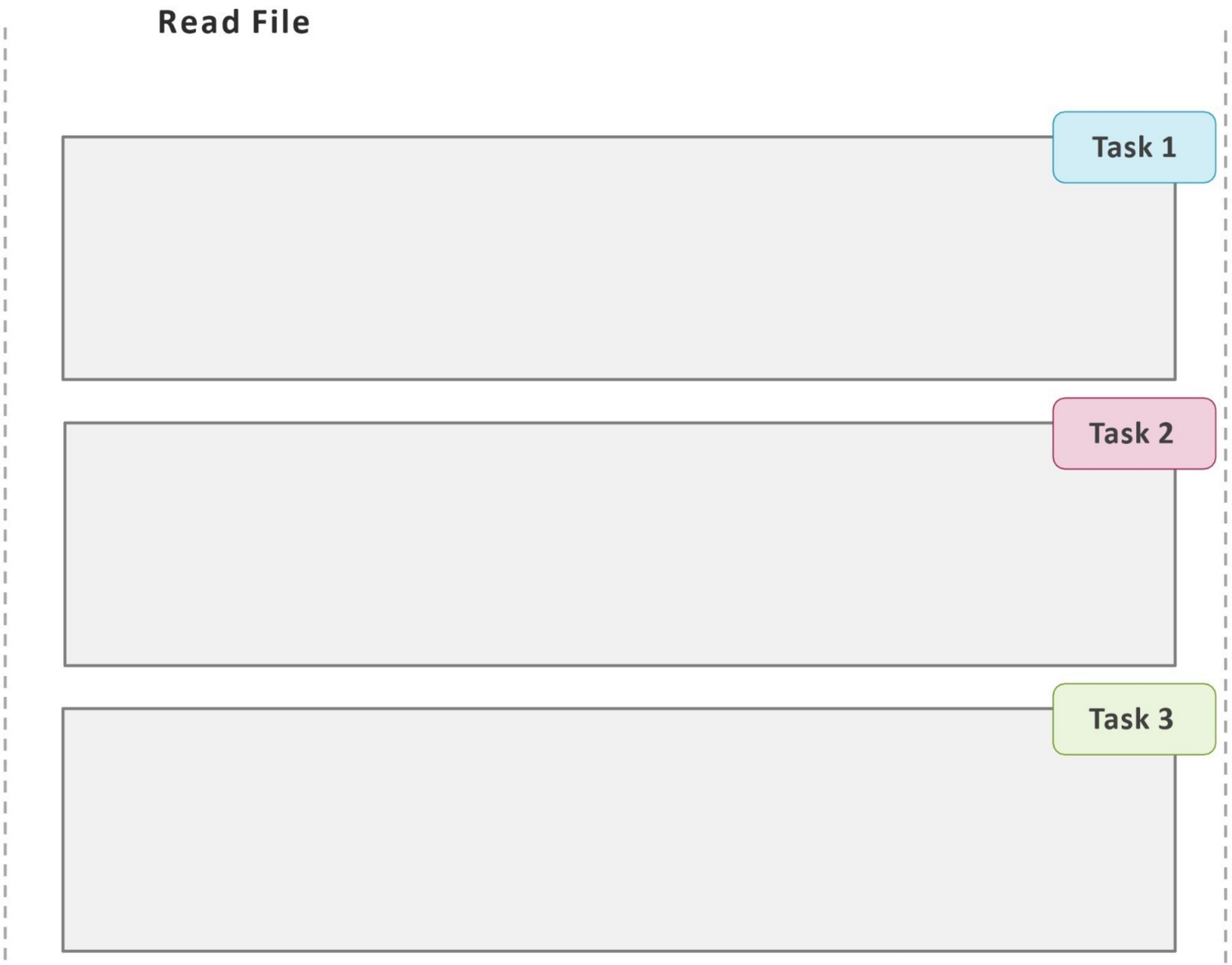
In-memory

Resides in the memory of cluster

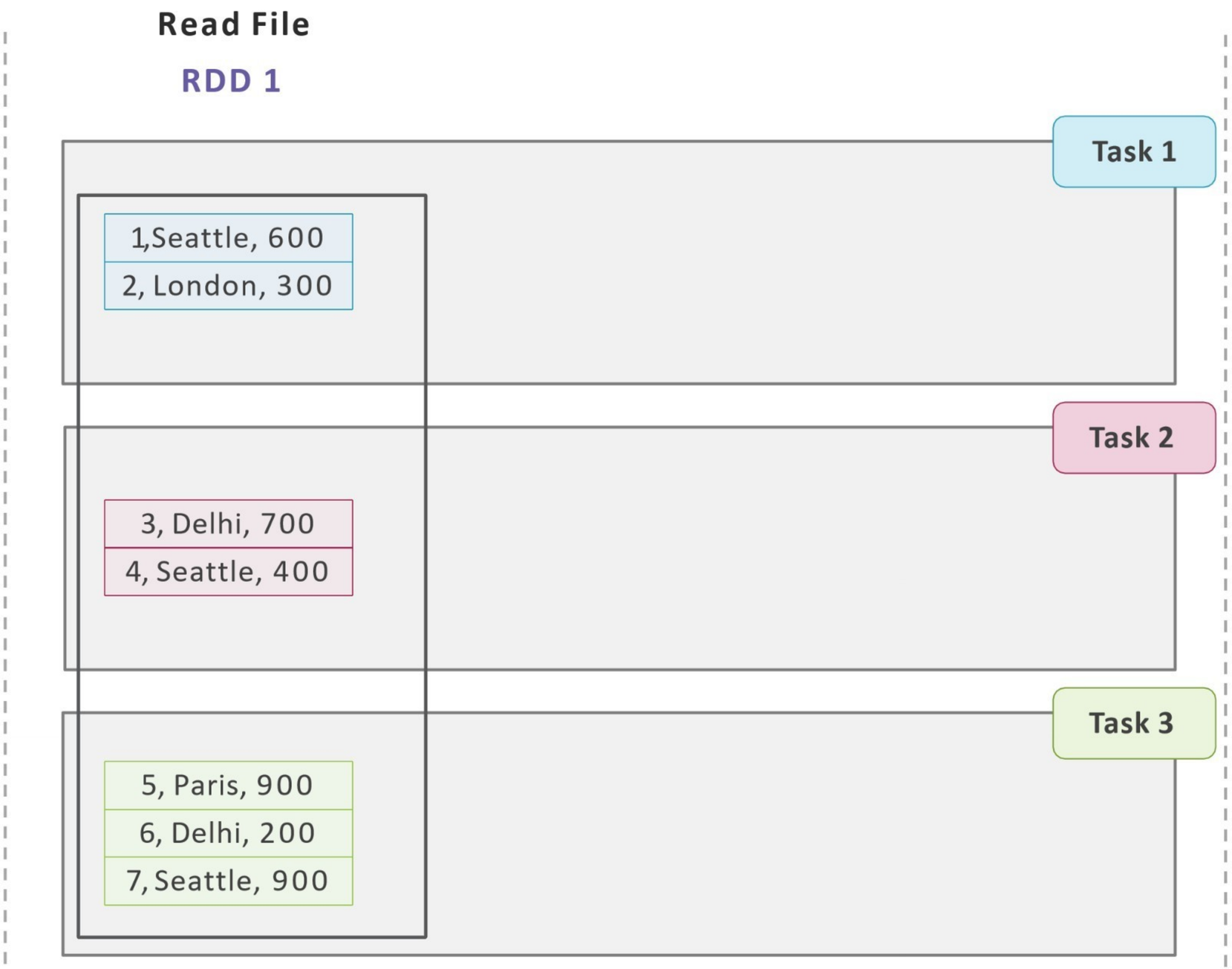
Partitioned

Split into partitions, processed by tasks

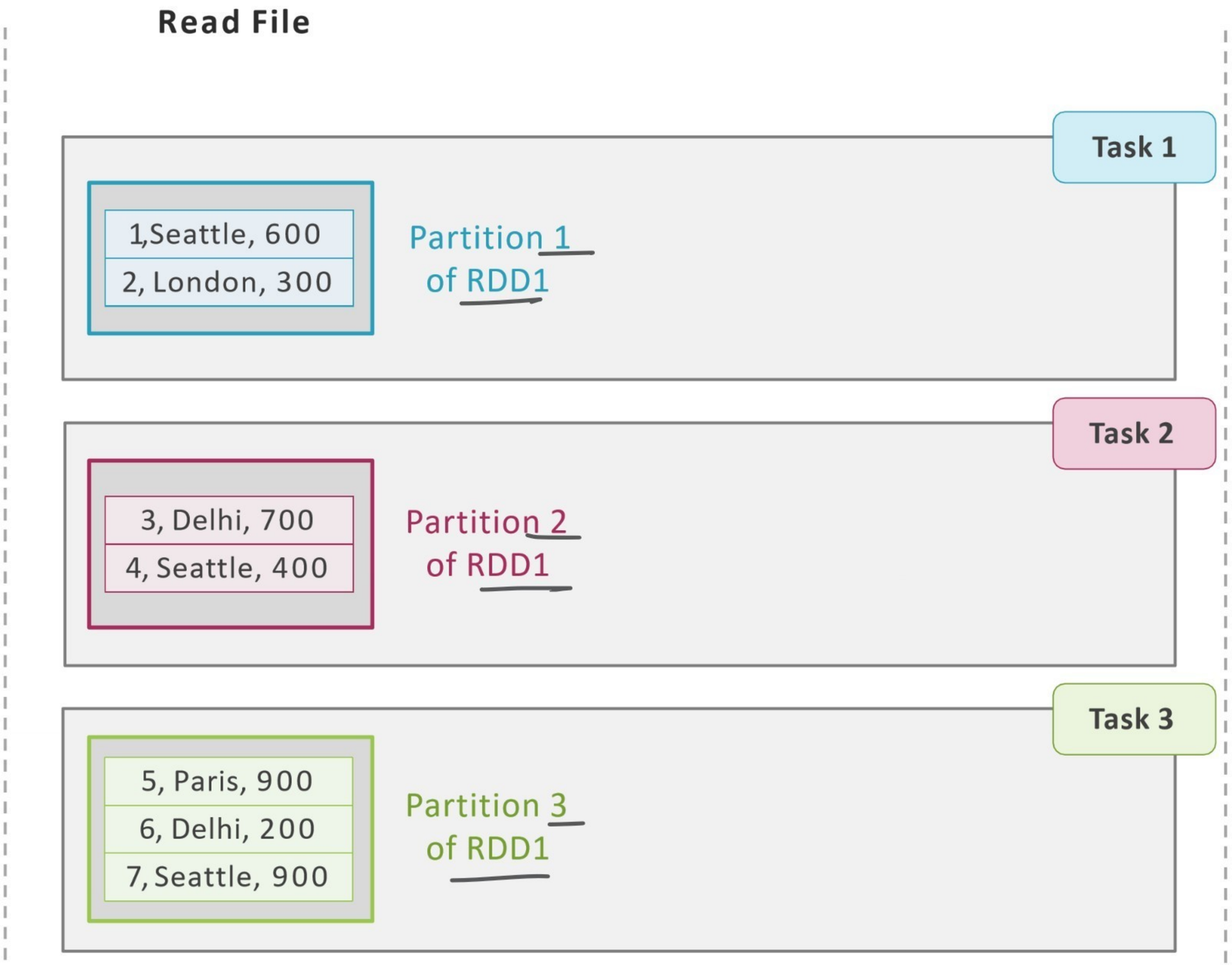
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900



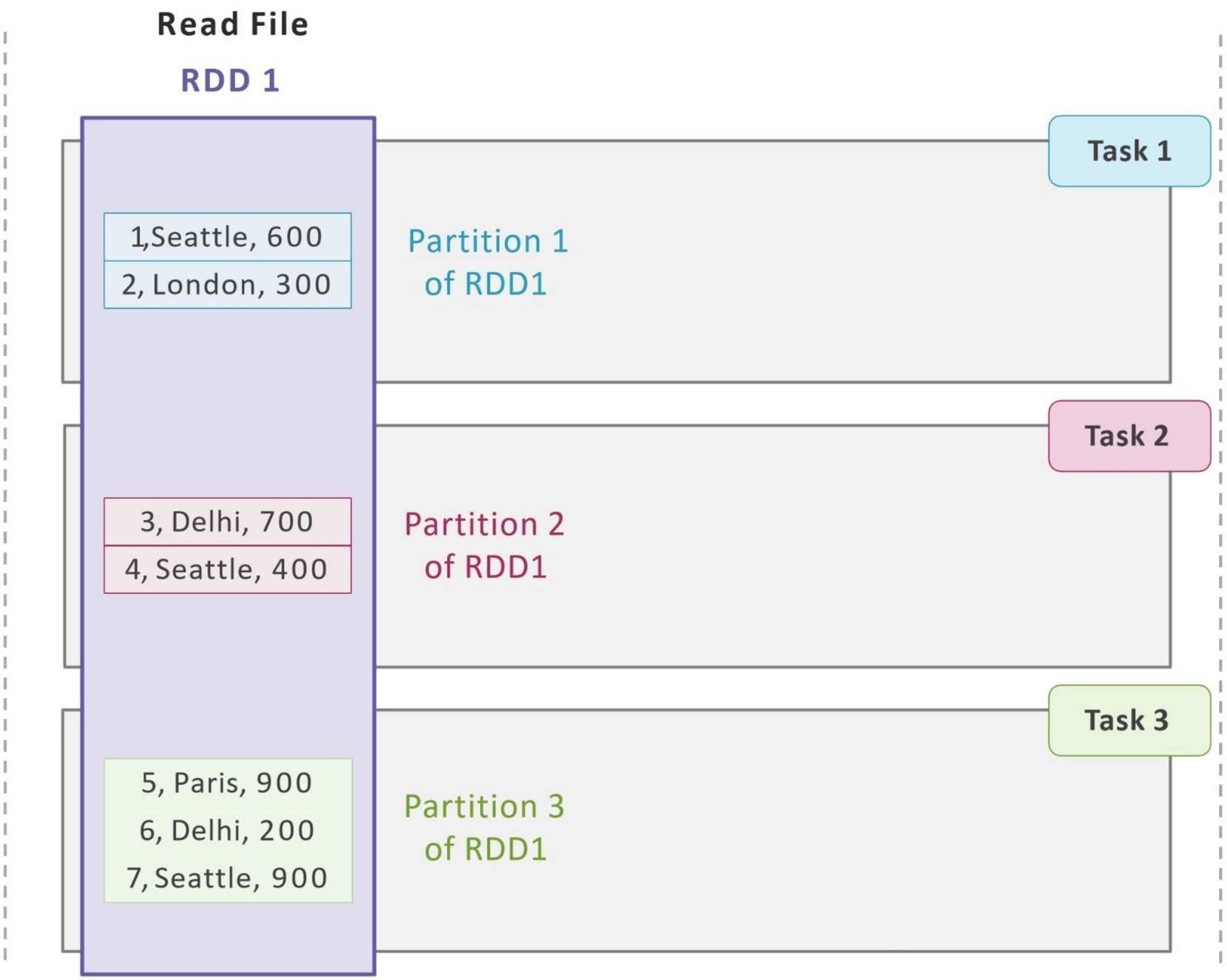
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900



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



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



Features of RDDs

In-memory

Resides in the memory of cluster

Partitioned

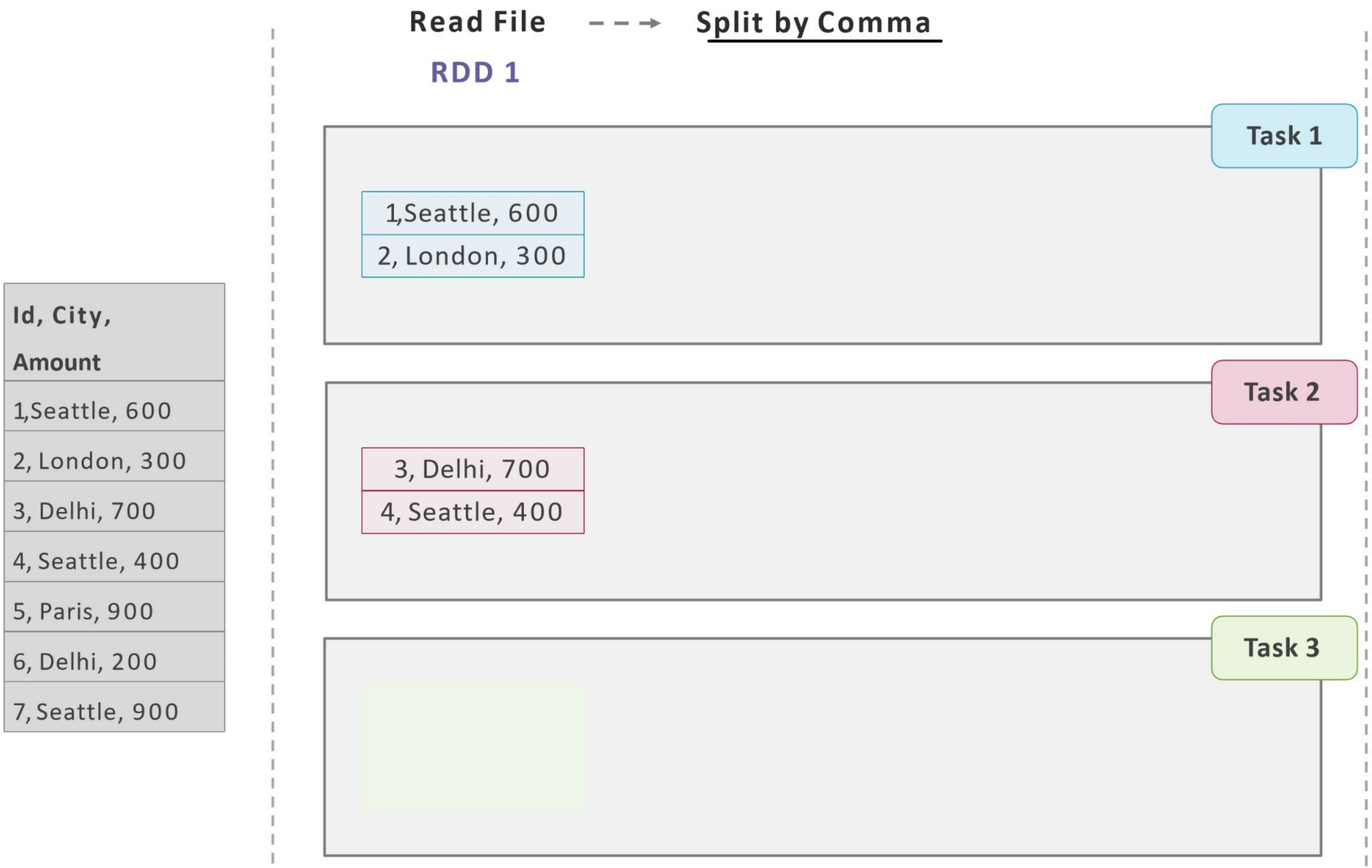
Split into partitions, processed by tasks

Read-only

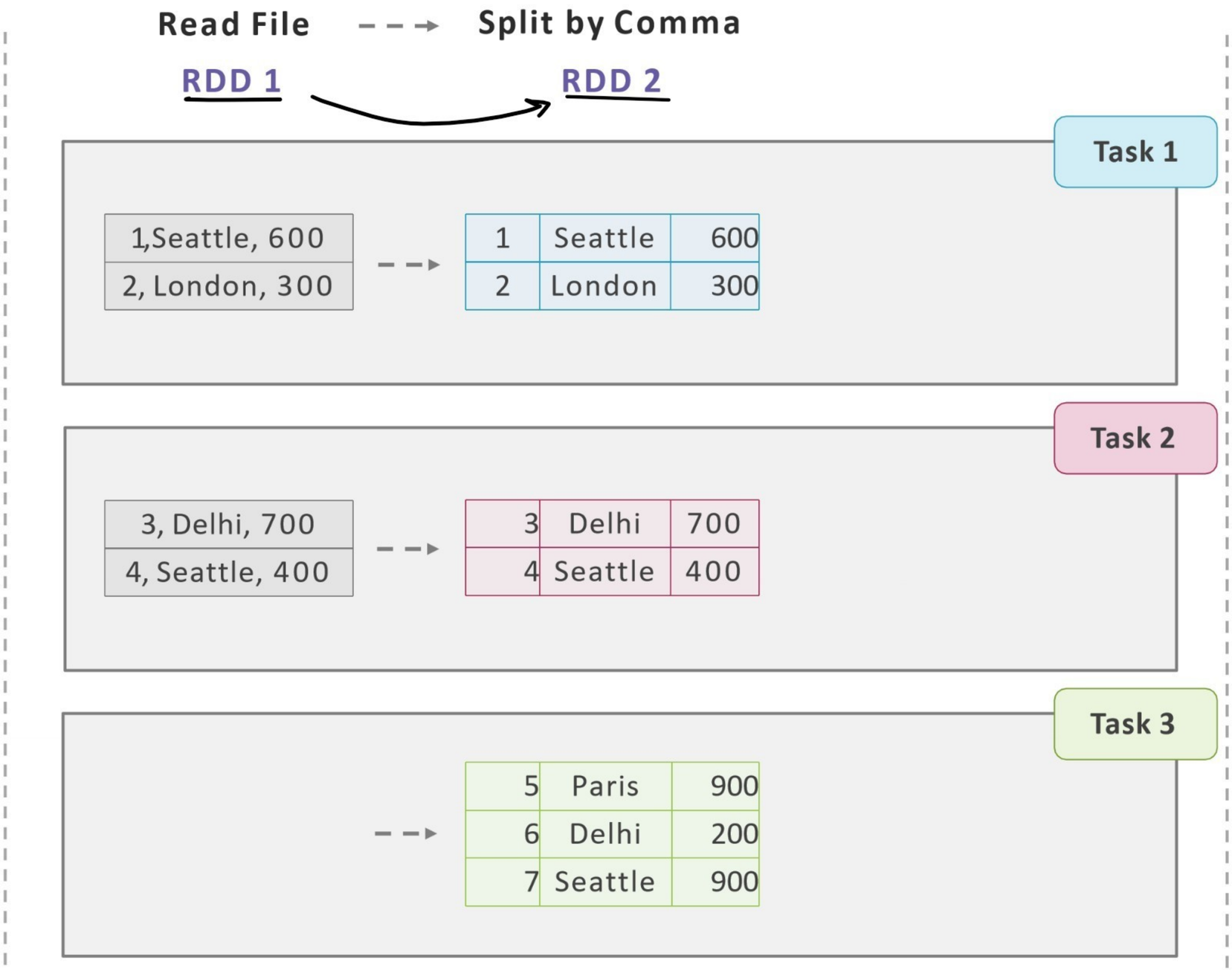
Transformed into another RDD or result

Transformation

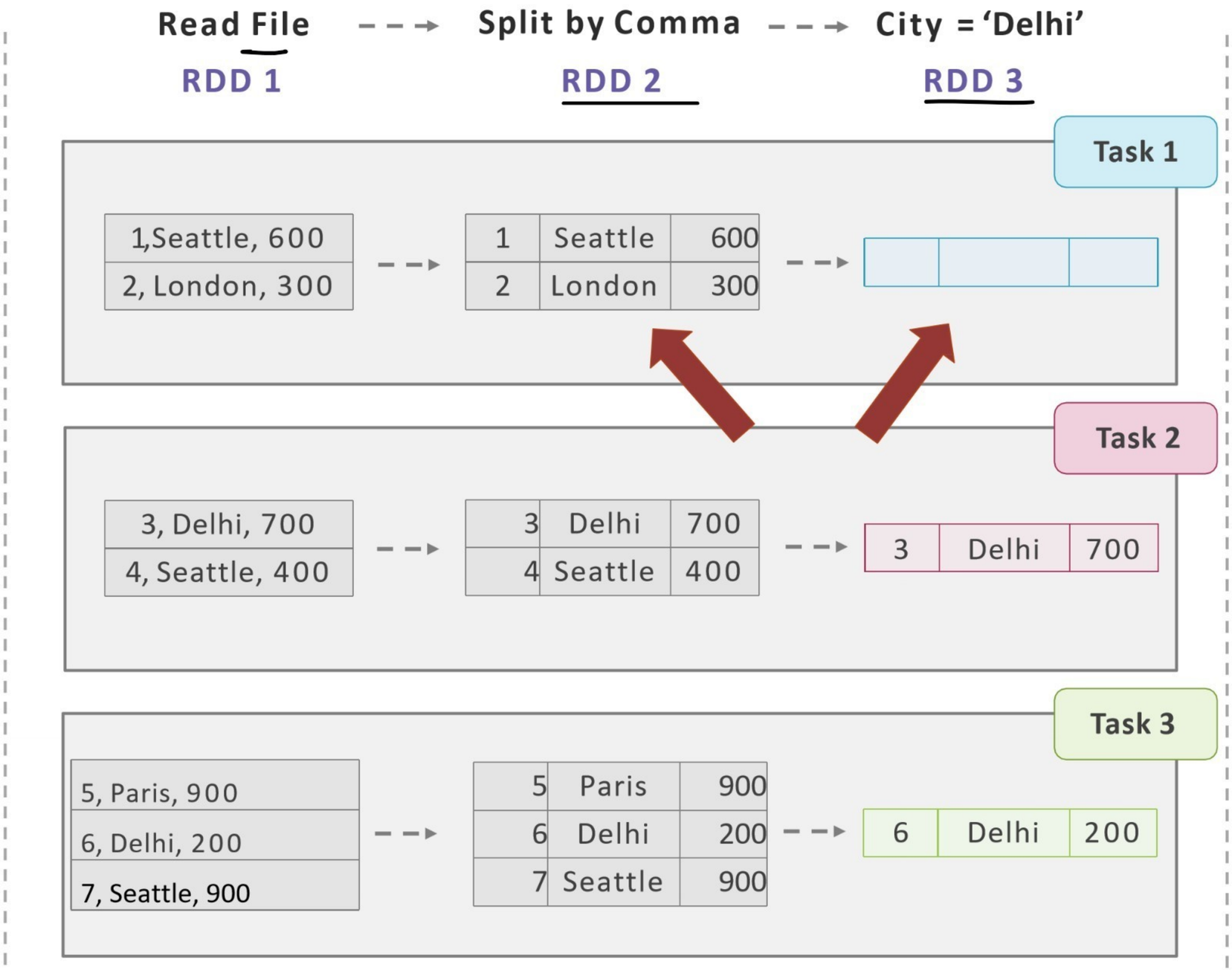
Action


Read File
Split by Comma
RDD 1
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Task 1
1,Seattle, 600
2, London, 300
Task 2
3, Delhi, 700
4, Seattle, 400
Task 3

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



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



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

Read File → **Split by Comma** → **City = 'Delhi'**

RDD 1 **RDD 2** **RDD 3**

Task 1

1,Seattle, 600
2, London, 300

→

1	Seattle	600
2	London	300

→

Task 2

3, Delhi, 700
4, Seattle, 400

→

3	Delhi	700
4	Seattle	400

→

3	Delhi	700

Task 3

→

5	Paris	900
6	Delhi	200
7	Seattle	900

→

6	Delhi	200

3	Delhi	700
6	Delhi	200

Features of RDDs

In-memory

Resides in the memory of cluster

Partitioned

Split into partitions, processed by tasks

Read-only

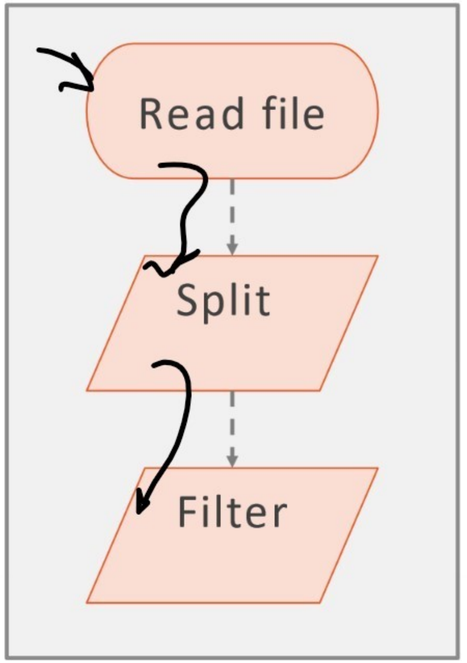
Transformed into another RDD or result

Resilient

Auto recover in case of a failure

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

1. Read file1.csv from Storage → T
2. Split data by comma → T
3. Filter where City='Delhi' → T
4. Write processed data to storage → A



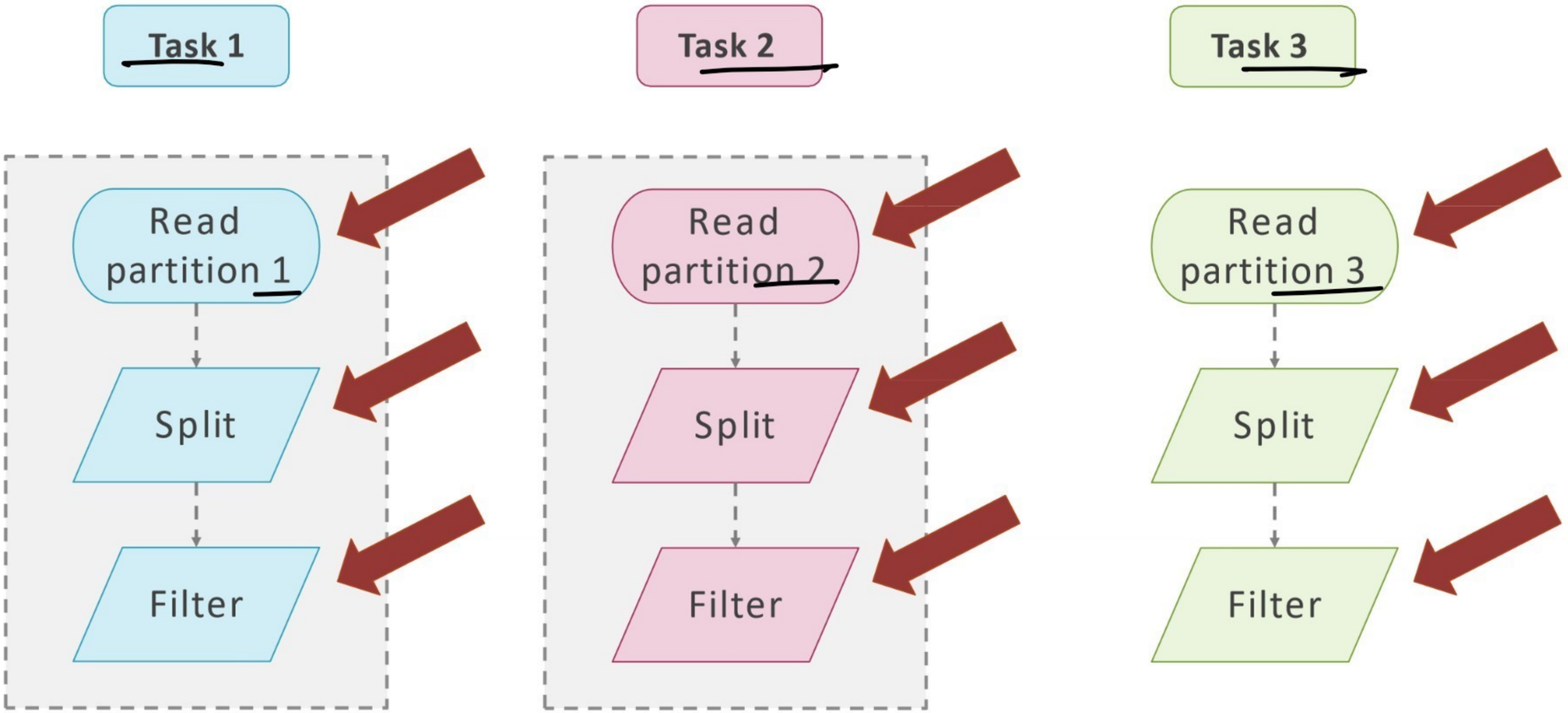
Directed Acyclic Graphs

Lineage Graph

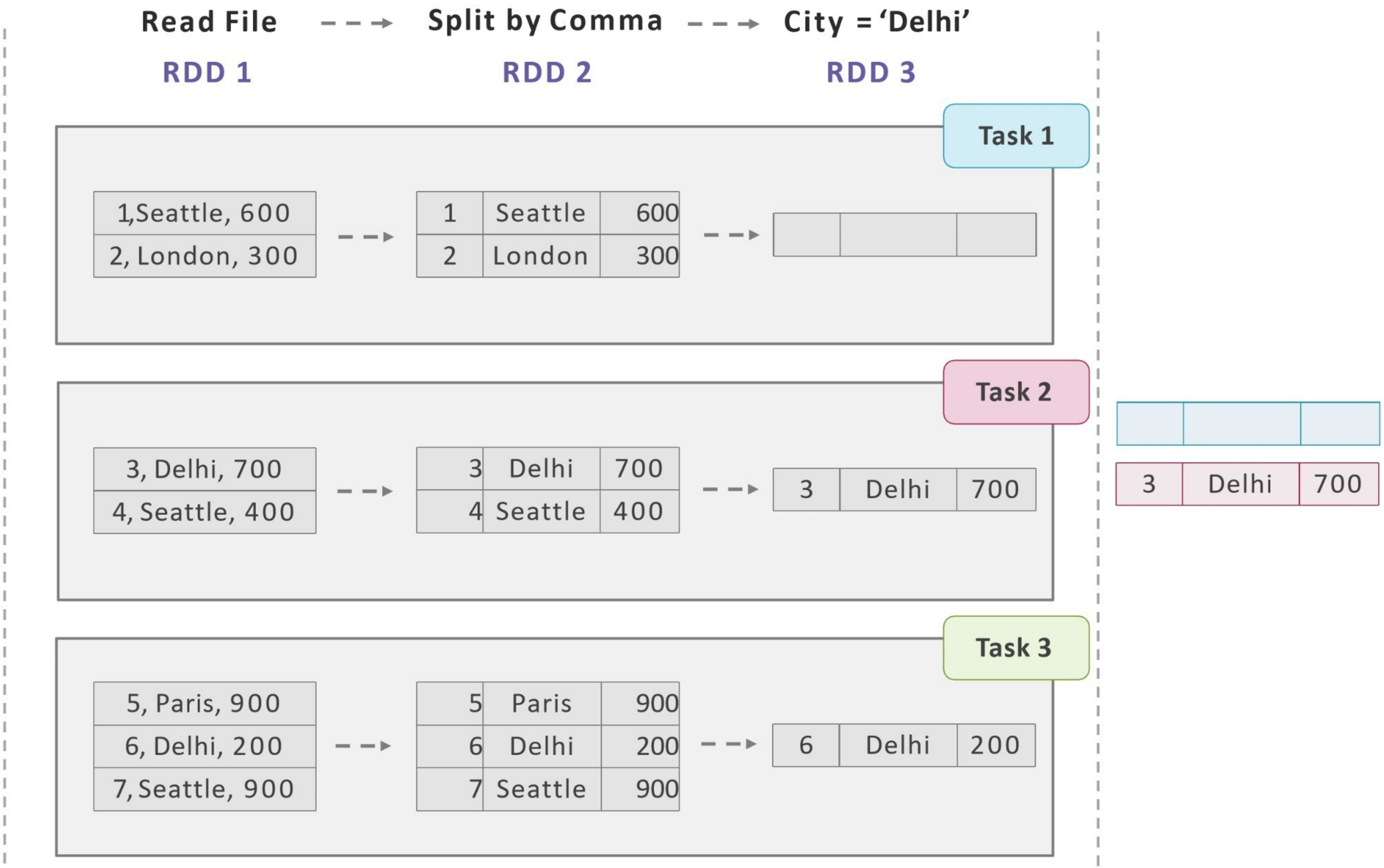
Defines steps that are used for creation of an RDD

1. Read file1.csv from Storage
2. Split data by comma
3. Filter where City='Delhi'
4. Write processed data to storage

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



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



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

Read File → **Split by Comma** → **City = 'Delhi'**

RDD 1 **RDD 2** **RDD 3**

Task 1

1,Seattle, 600	→	1	Seattle	600	→			
2, London, 300		2	London	300				

Task 2

3, Delhi, 700	→	3	Delhi	700	→	3	Delhi	700
4, Seattle, 400		4	Seattle	400				

3	Delhi	700

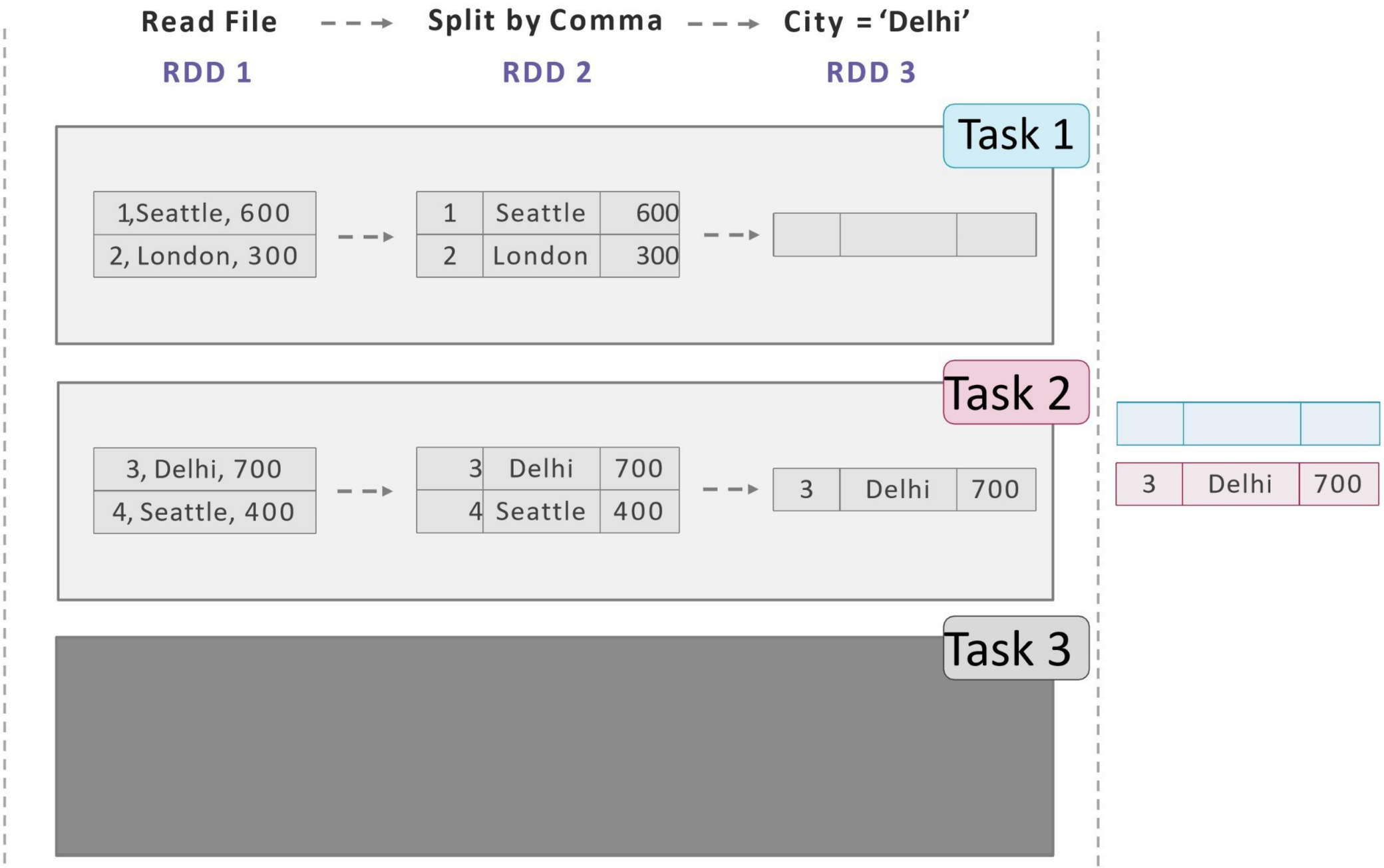
Task 3

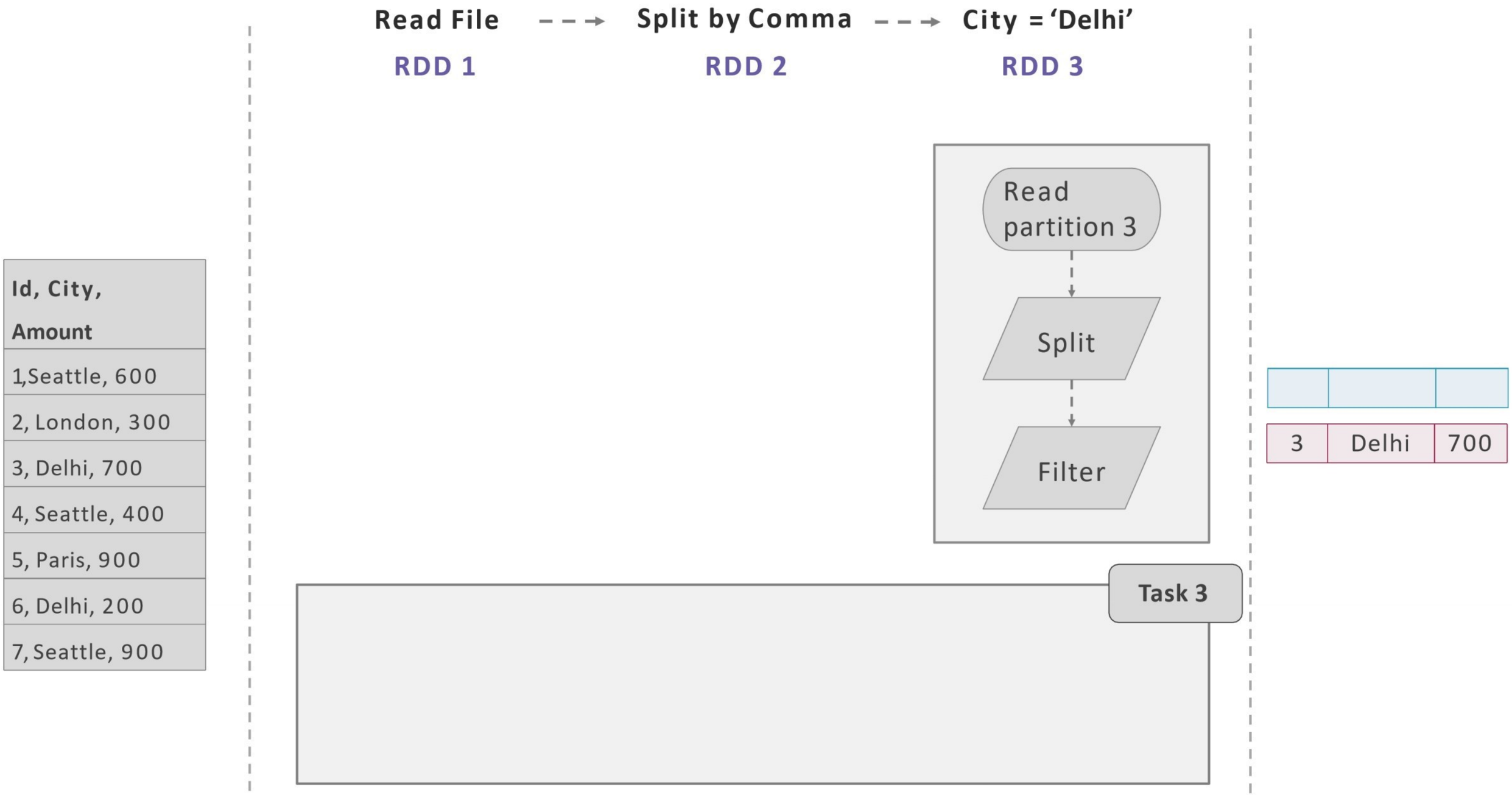
Let say Task 1 and 2 runs fine, but Task 3 fails

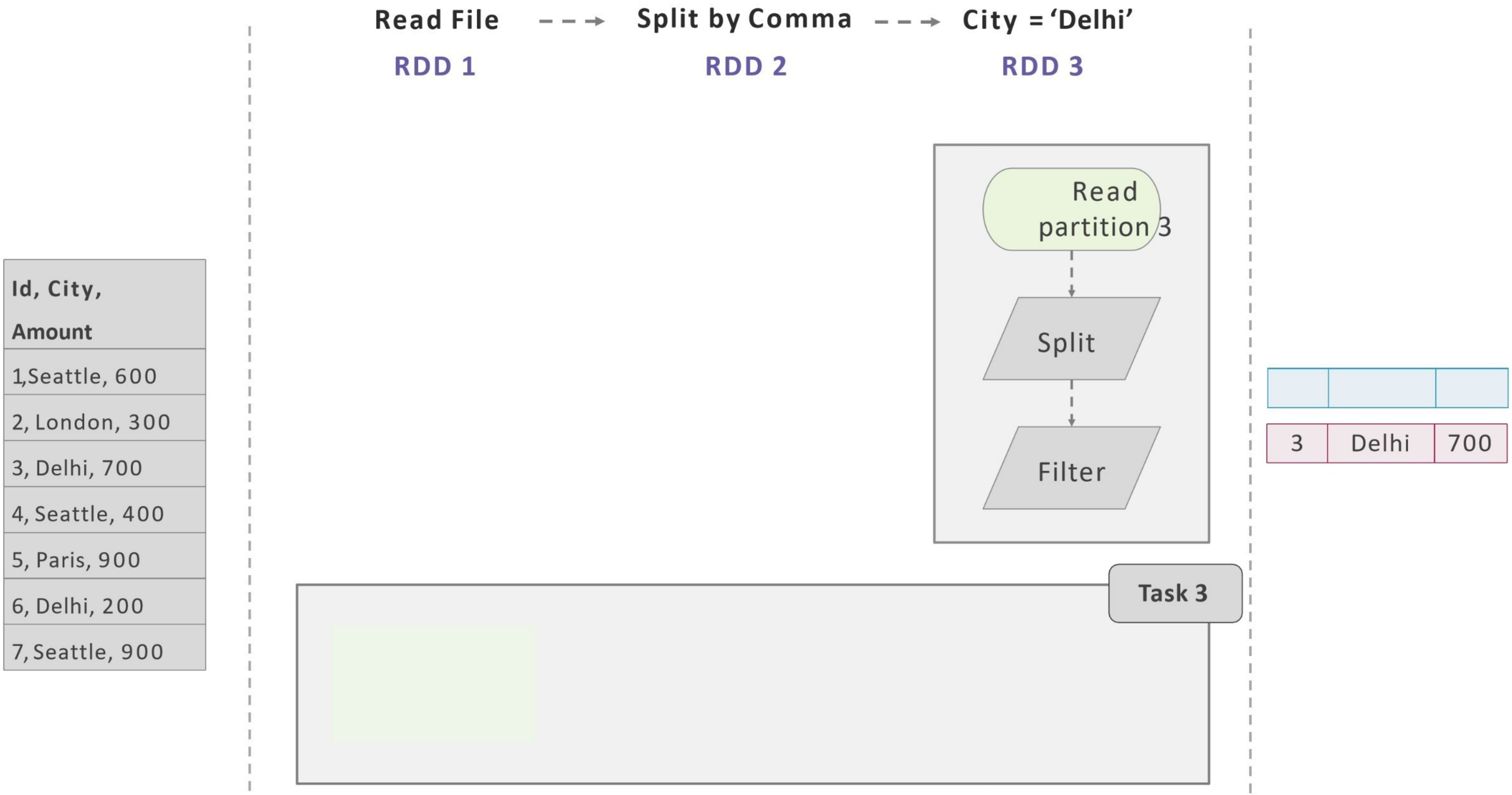
Spark will re-execute failed Tasks using their Lineage Graph

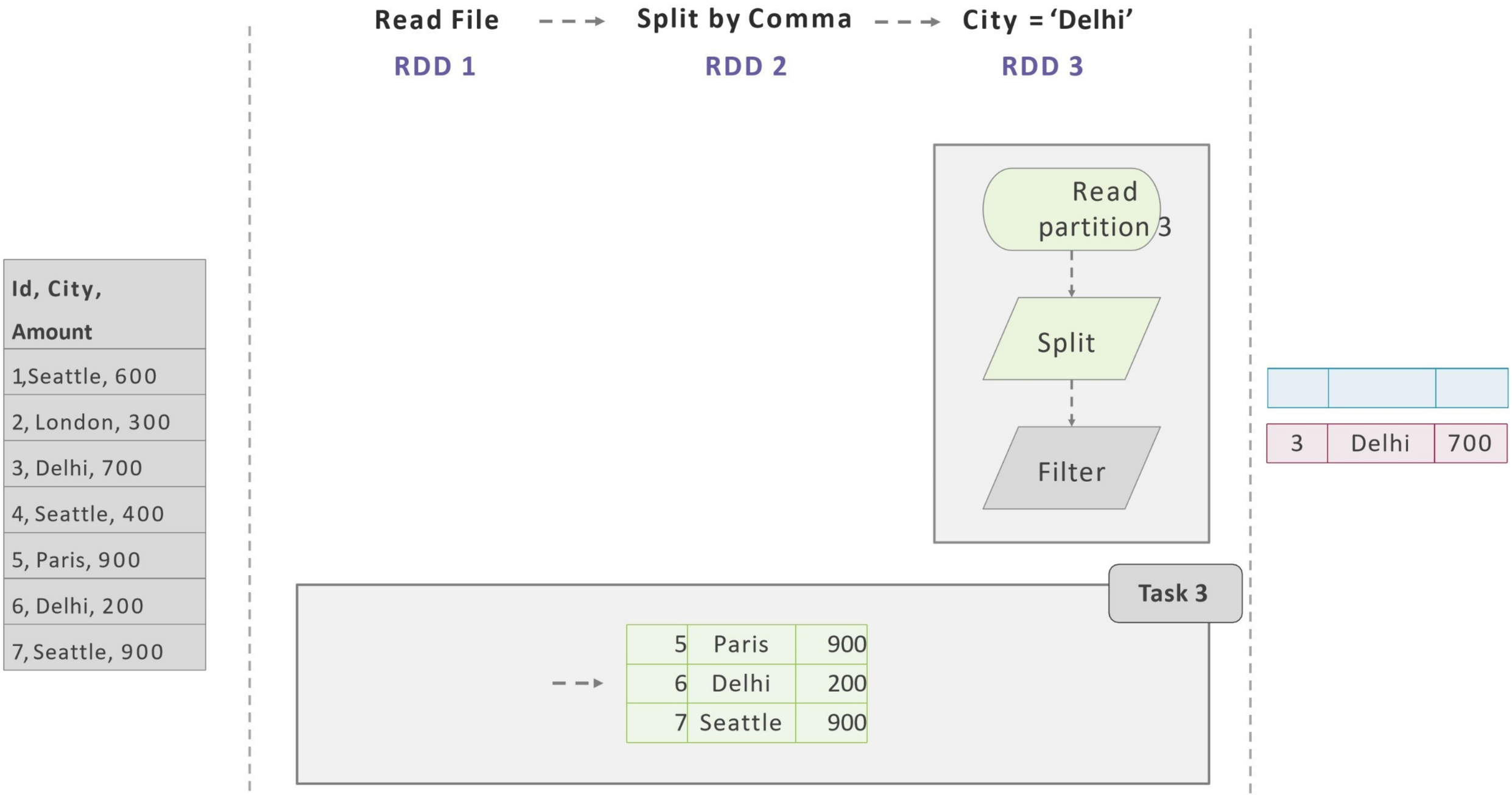
Only if the failure is transient, &not permanent (like divide by zero)

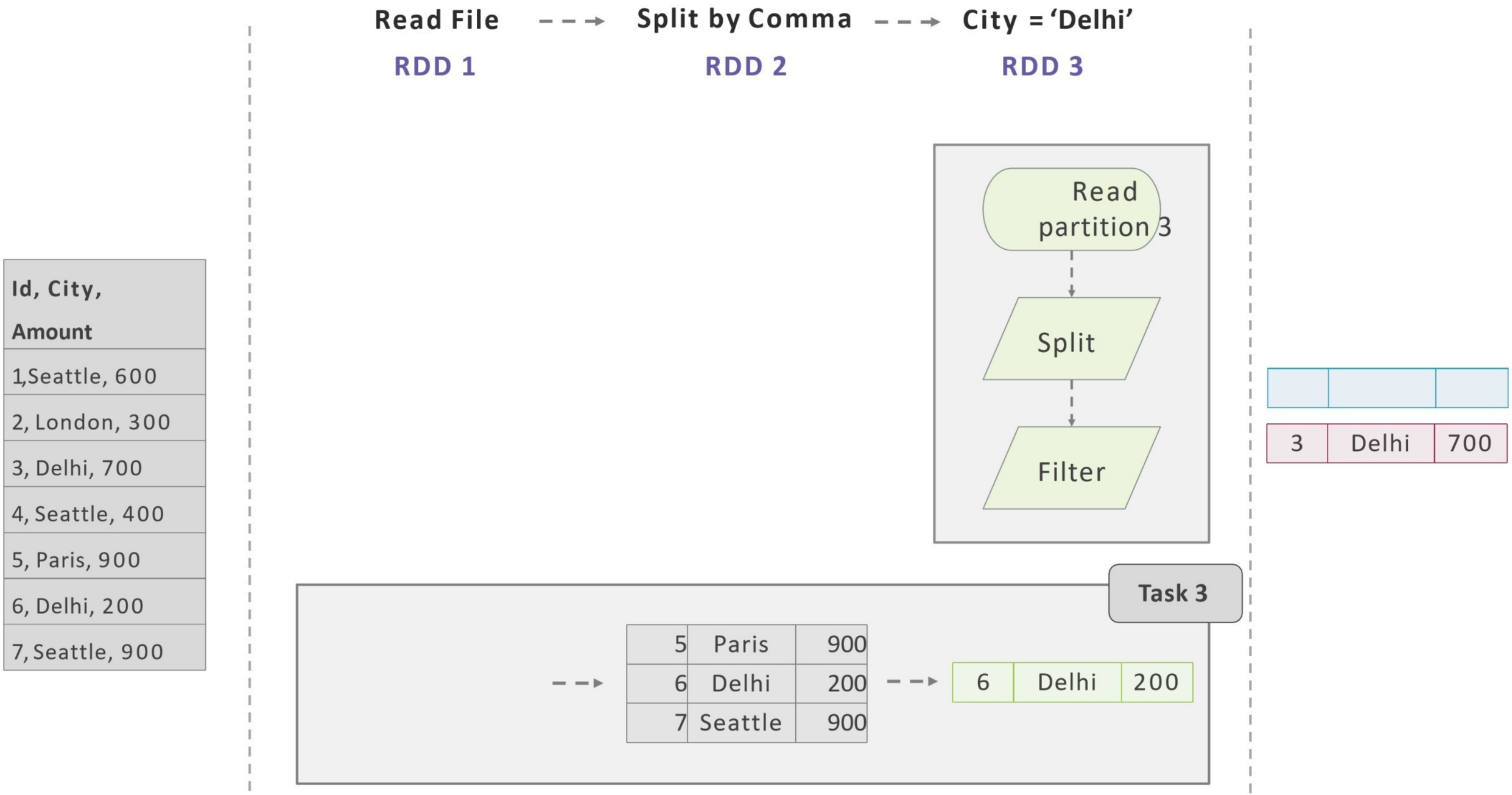
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900

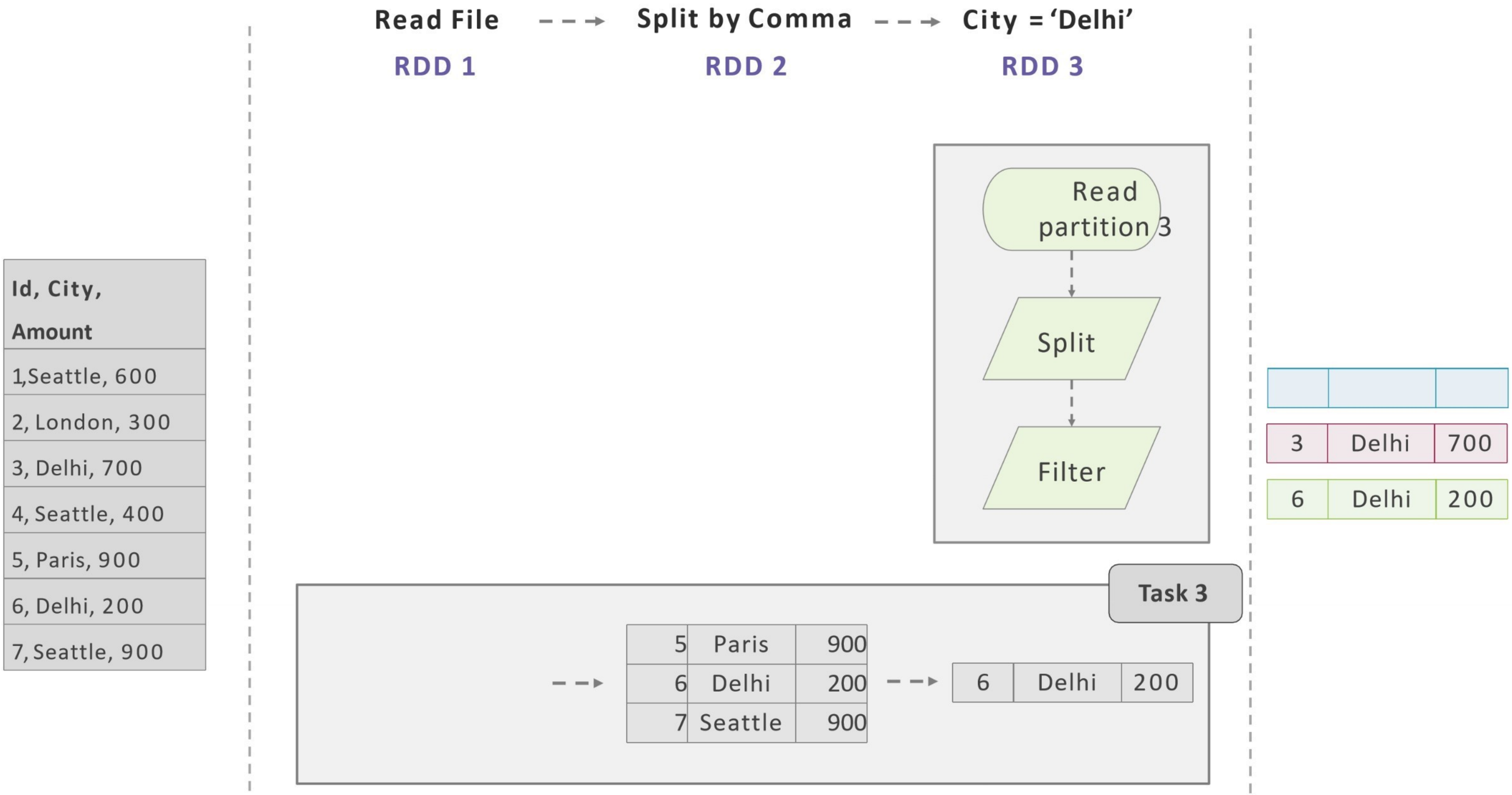



Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read partition 3
Split
Filter
Task 3
3
Delhi
700


Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read partition 3
Split
Filter
Task 3
3
Delhi
700


Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read partition 3
Split
Filter
Task 3
5 Paris 900
6 Delhi 200
7 Seattle 900
3 Delhi 700


Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read partition 3
Split
Filter
Task 3
5 Paris 900
6 Delhi 200
7 Seattle 900
6 Delhi 200
3 Delhi 700


Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Id, City, Amount
1,Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read partition 3
Split
Filter
Task 3
5 Paris 900
6 Delhi 200
7 Seattle 900
6 Delhi 200
3 Delhi 700
6 Delhi 200

Features of RDDs

In-memory

Resides in the memory of cluster

Partitioned

Split into partitions, processed by tasks

Read-only

Transformed into another RDD or result

Resilient

Auto recover in case of a failure

Creating RDDs

Options to Create RDD

Parallelize a Collection

Read a File

From another RDD

```python
import findspark
findspark.init()
findspark.find()
```

```
'C:\\Installation\\spark-3.3.2-bin-hadoop3'
```

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = (
            SparkSession
                .builder
                .appName("RDDApp")
                .master("local[4]")
                .getOrCreate()
        )
```

```python
spark
```

Create RDD - Option 1: Parallelize

+ Code + Markdown

```
# Create a variable for SparkContext

sc = spark.sparkContext
```

```
# Create RDD using parallelize

numbersRdd = sc.parallelize( [1, 2, 3, 4, 5] )
```

[6]

```
# Check RDD partitions

numbersRdd.getNumPartitions()
```

```
4
```

[7]

```
# Get result from RDD

output = numbersRdd.collect()

print(output)
```

```
[1, 2, 3, 4, 5]
```

[8]

```
# Get any 2 records from RDD

numbersRdd.take(2)
```

```
[1, 2]
```

```python
# Get first record from RDD
# first() returns an element, while take(1) returns an array with one element

numbersRdd.first()
```

```
1
```

```python
# Create RDD with complex types

employeesRdd = sc.parallelize(
                                [
                                    [ 1, "Neha",  10000 ],
                                    [ 2, "Steve", 20000 ],
                                    [ 3, "Kari",  30000 ],
                                    [ 4, "Ivan",  40000 ],
                                    [ 5, "Mohit", 50000 ]
                                ]
                              )
```

```python
# Get first record from RDD

employeesRdd.first()
```

```
[1, 'Neha', 10000]
```

Create RDD - Option 2: Read a File

```
# Read TaxiZones.csv file and create RDD

taxiZonesRdd = sc.textFile("C:\SparkCourse\DataFiles\Raw\TaxiZones.csv")
```

12]

```
# Get 10 records

taxiZonesRdd.take(10)
```

13]

```
['1,EWR,Newark Airport,EWR',
 '2,Queens,Jamaica Bay,Boro Zone',
 '3,Bronx,Allerton/Pelham Gardens,Boro Zone',
 '4,Manhattan,Alphabet City,Yellow Zone',
 '5,Staten Island,Arden Heights,Boro Zone',
 '6,Staten Island,Arrochar/Fort Wadsworth,Boro Zone',
 '7,Queens,Astoria,Boro Zone',
 '8,Queens,Astoria Park,Boro Zone',
 '9,Queens,Auburndale,Boro Zone',
 '10,Queens,Baisley Park,Boro Zone']
```

```
# Check RDD partitions

taxiZonesRdd.getNumPartitions()
```

14]

2

```
# Read TaxiZones.csv file and create RDD (with 4 partitions)

taxiZonesRdd = sc.textFile("C:\SparkCourse\DataFiles\Raw\TaxiZones.csv", 4)

taxiZonesRdd.getNumPartitions()
```

15]

4

Create RDD - Option 3: Convert another RDD

```
# Apply map operation to split each record by comma, and get array of items

taxiZonesWithColsRdd = (
                        taxiZonesRdd

                            .map( lambda zone: zone.split(",") )
                    )

taxiZonesWithColsRdd.take(5)
```

6]

```
[['1', 'EWR', 'Newark Airport', 'EWR'],
 ['2', 'Queens', 'Jamaica Bay', 'Boro Zone'],
 ['3', 'Bronx', 'Allerton/Pelham Gardens', 'Boro Zone'],
 ['4', 'Manhattan', 'Alphabet City', 'Yellow Zone'],
 ['5', 'Staten Island', 'Arden Heights', 'Boro Zone']]
```

```python
# Apply filter operation
# Get data where Borough is 'Manhattan' and Zone begins with 'central'

filteredZonesRdd = (
                taxiZonesWithColsRdd

                    .filter( lambda zoneRow: zoneRow[1] == "Manhattan"

                                    and zoneRow[2].lower().startswith("central")
                                )
            )

filteredZonesRdd.take(5)
```

```
[['41', 'Manhattan', 'Central Harlem', 'Boro Zone'],
 ['42', 'Manhattan', 'Central Harlem North', 'Boro Zone'],
 ['43', 'Manhattan', 'Central Park', 'Yellow Zone']]
```

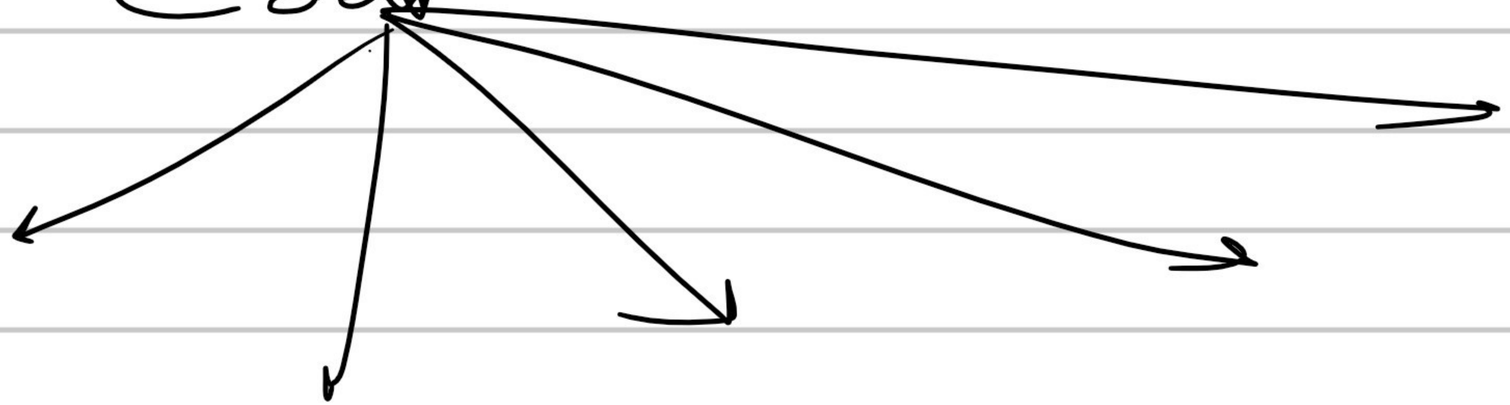
```python
# Get list of Zones with even LocationId

evenZoneIds = (
                    taxiZonesRdd

                        .filter( lambda zoneRow: int(zoneRow[0]) % 2 == 0 )
                )

evenZoneIds.take(5)
```

```
['2,Queens,Jamaica Bay,Boro Zone',
 '4,Manhattan,Alphabet City,Yellow Zone',
 '6,Staten Island,Arrochar/Fort Wadsworth,Boro Zone',
 '8,Queens,Astoria Park,Boro Zone',
 '20,Bronx,Belmont,Boro Zone']
```

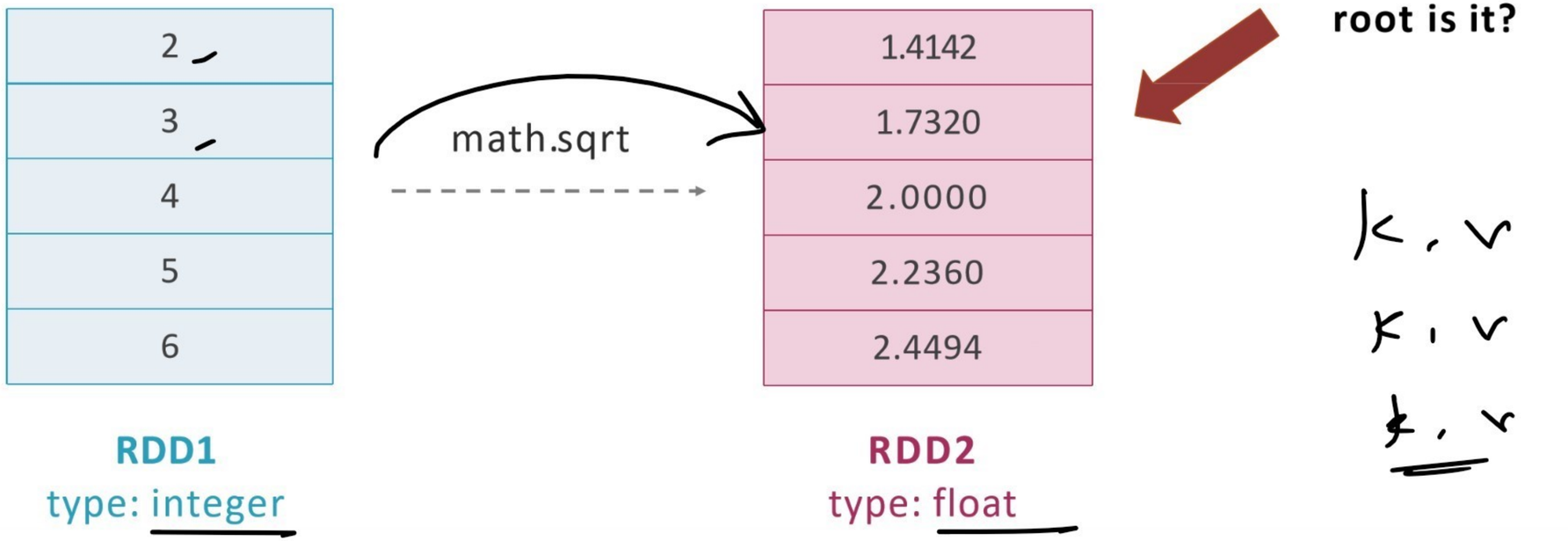
Working with Pair RDDs

Spark → Scala

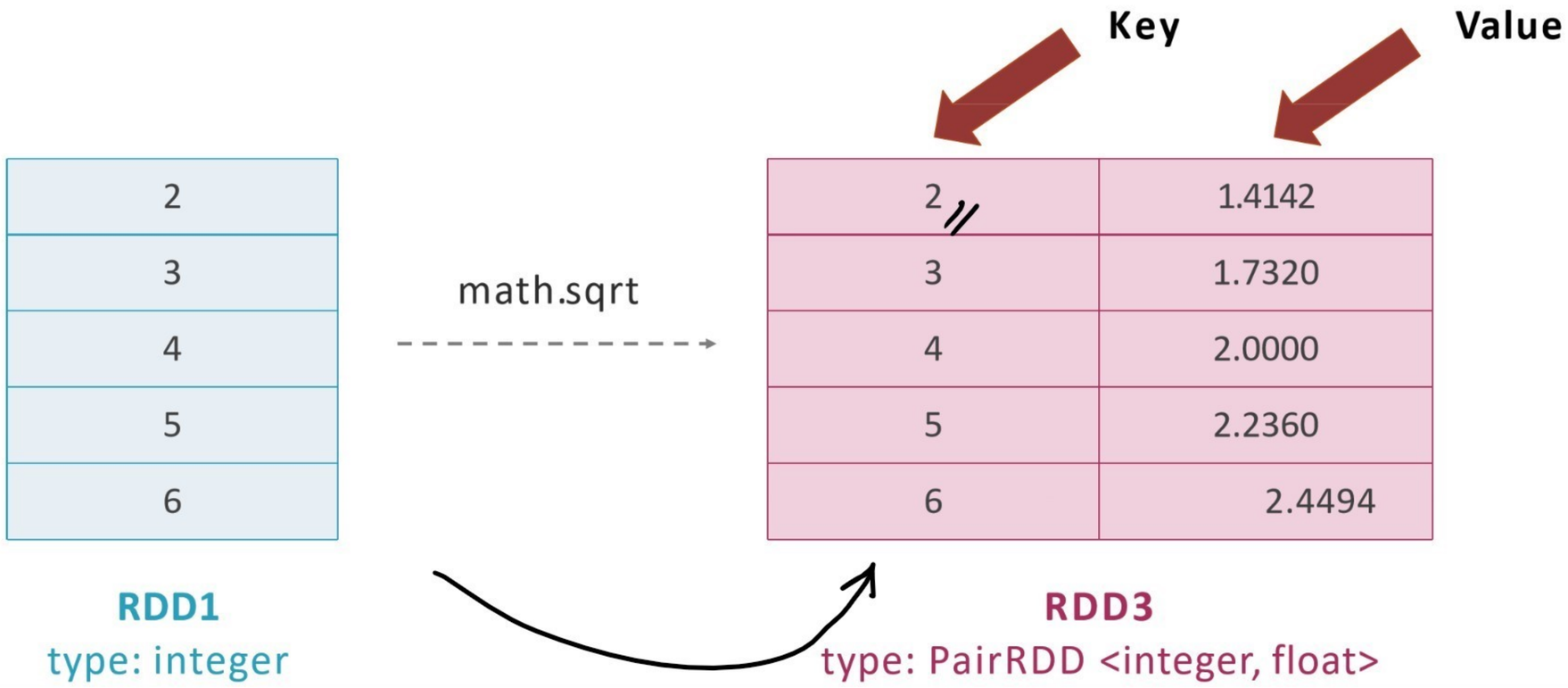
dev



Pair RDDs


2
3
4
5
6
math.sqrt
1.4142
1.7320
2.0000
2.2360
2.4494
Whose square root is it?
RDD1
type: integer
RDD2
type: float

Pair RDDs


Key
Value
2
3
4
5
6
math.sqrt
2
1.4142
3
1.7320
4
2.0000
5
2.2360
6
2.4494
RDD1
type: integer
RDD3
type: PairRDD <integer, float>

Pair RDDs

Key
(CustomerID)

Value
(Customer Info)

1	['Mohit', '20000', 'Hyderabad']
2	['Ivan', '23000', 'Seattle']
3	['Kerri', '27000', 'London']
4	['Andrew', '18000', 'Paris']
5	['Neha', '32000', 'Delhi']

Customer Data – PairRDD <integer, array>

Pair RDDs

1	['Mohit', '20000', 'Hyderabad']
2	['Ivan', '23000', 'Seattle']
3	['Kerri', '27000', 'London']
4	['Andrew', '18000', 'Paris']
5	['Neha', '32000', 'Delhi']

RDDs with Key-Value pairs

- Two items are linked together
- Key is identifier. Value is corresponding data

Key need not be unique

Spark has special operations for Pair RDDs

- reducebyKey, sortByKey, countByKey etc.
- Some operations are costly

Pair RDDs

```
import math

numbersRDD = sc.parallelize( [2, 3, 4, 5, 6] )

# Create Pair RDD
numsWithSquareRootRdd = (
                    numbersRDD

                        .map(lambda num: (
                                                num,
                                                math.sqrt(num)
                                            )
                        )
                )

numsWithSquareRootRdd.collect()
```

```
[(2, 1.4142135623730951),
 (3, 1.7320508075688772),
 (4, 2.0),
 (5, 2.23606797749979),
 (6, 2.449489742783178)]
```

Exercise 1

Use taxiZonesWithColsRdd and create a Pair RDD (LocationId as Key, Zone Details as Value)

```python
# Exercise 1 Answer

taxiZonesPairRdd = (
                    taxiZonesWithColsRdd

                        .map(lambda zoneRow: (
                                                zoneRow[0],
                                                zoneRow
                                              )
                             )
                   )

taxiZonesPairRdd.take(10)
```

reduce()

Calculate sum of Amount by City

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

CSV File

Create

Pair RDD

Seattle	600
Seattle	300
Delhi	700
Seattle	400

Delhi	900
Delhi	200
Seattle	900

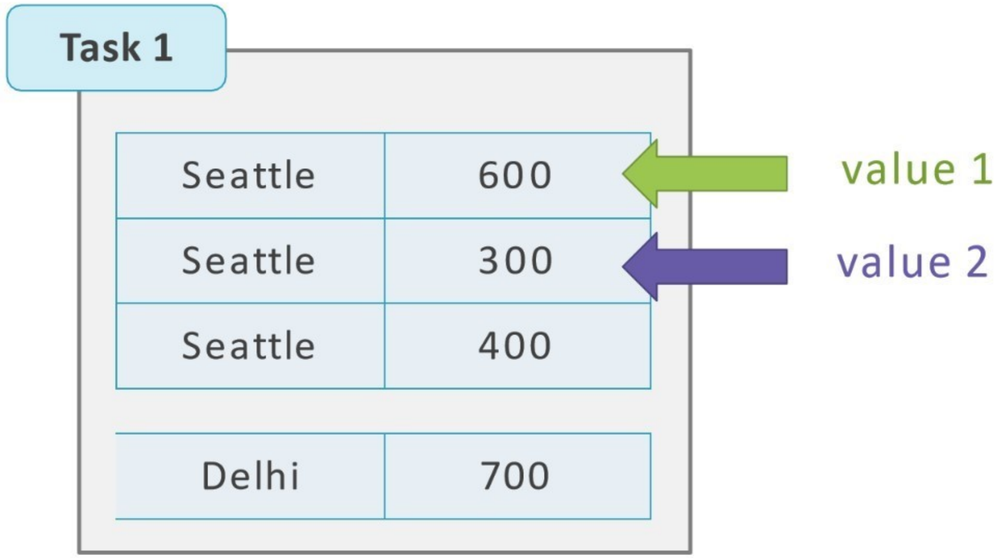
PairRDD <string, integer>

```
cityPairRDD
   .reduceByKey (lambda value1, value2: value1 + value2)
```

Calculate sum of Amount by City

Seattle	600
Seattle	300
Delhi	700
Seattle	400

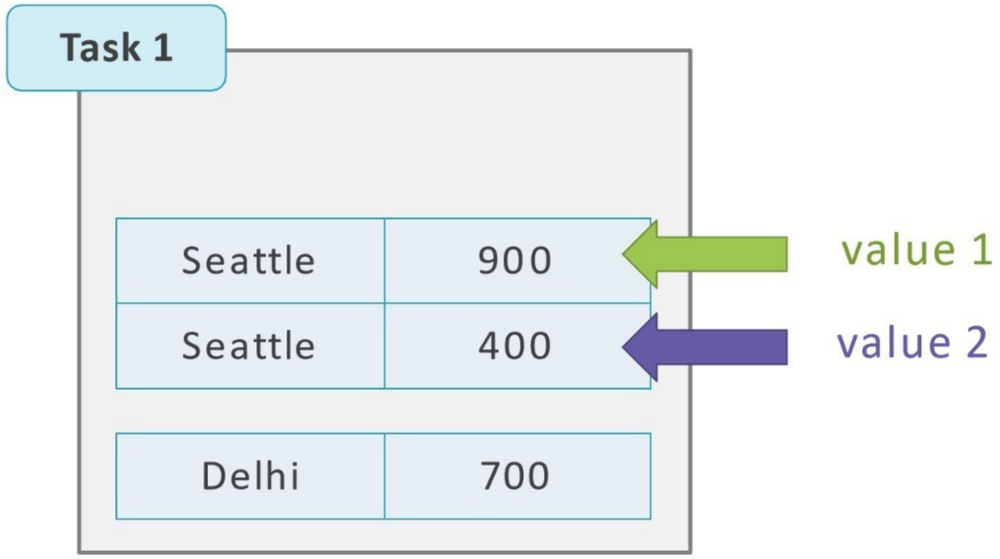
Delhi	900
Delhi	200
Seattle	900



Lambda Function

```
value1, value2: value1 + value2
```

Calculate sum of Amount by City



Delhi	900
Delhi	200
Seattle	900

Lambda Function

```
value1, value2: value1 + value2
```

Calculate sum of Amount by City

Task 1

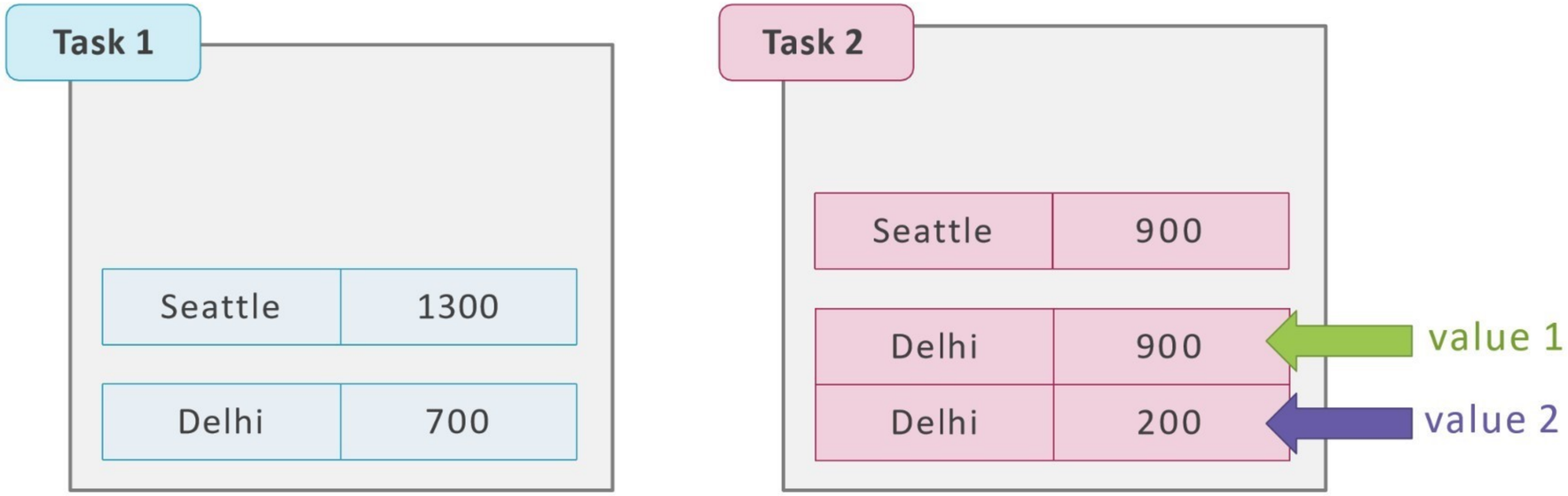
Seattle	1300
Delhi	700

Delhi	900
Delhi	200
Seattle	900

Lambda Function

```
value1, value2: value1 + value2
```

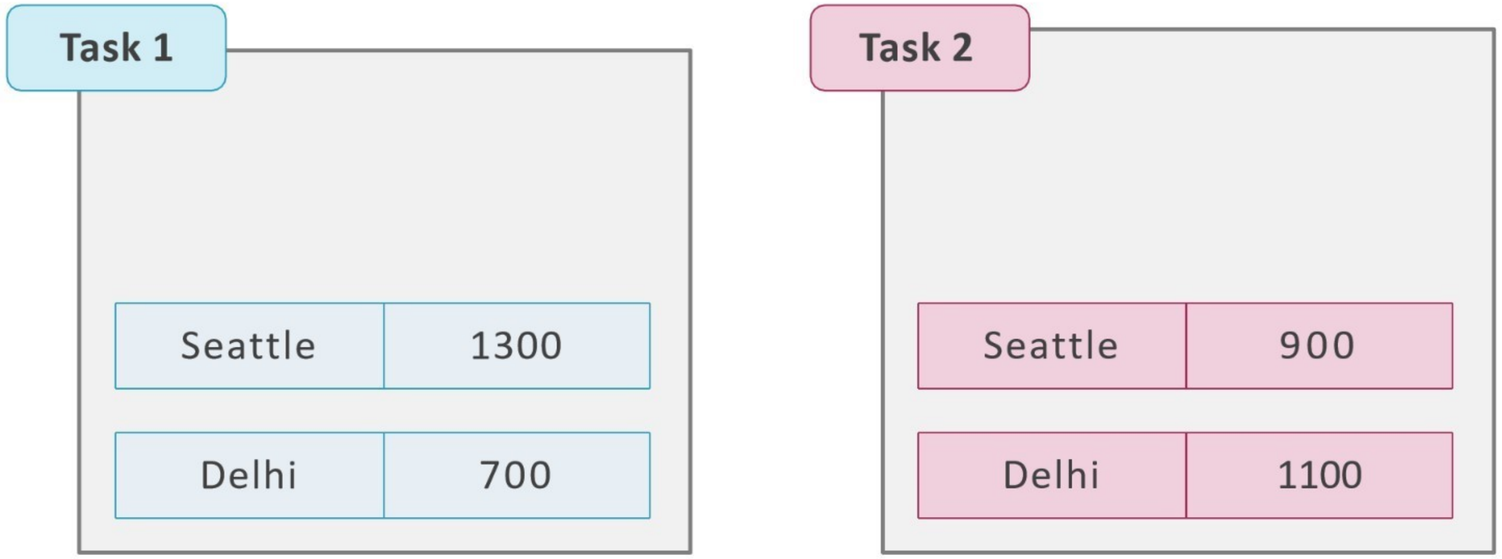
Calculate sum of Amount by City



Lambda Function

```
value1, value2: value1 + value2
```

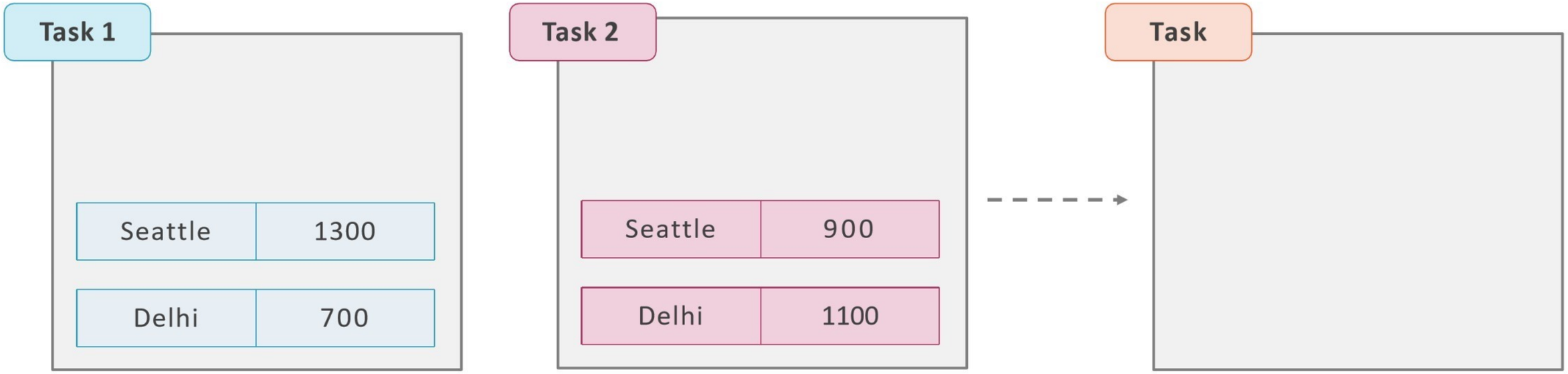
Calculate sum of Amount by City



Lambda Function

```
value1, value2: value1 + value2
```

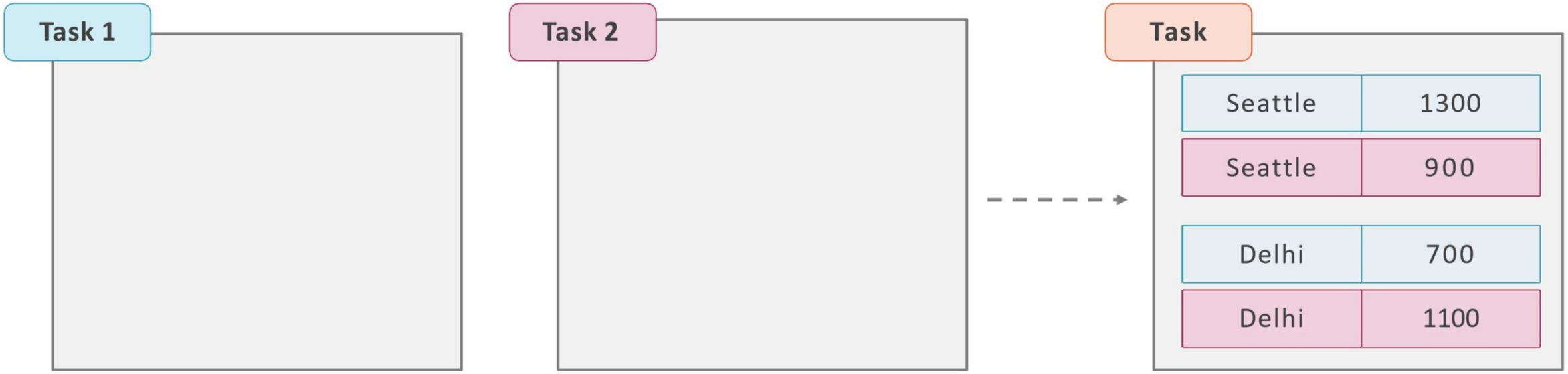
Calculate sum of Amount by City



Lambda Function

```
value1, value2: value1 + value2
```

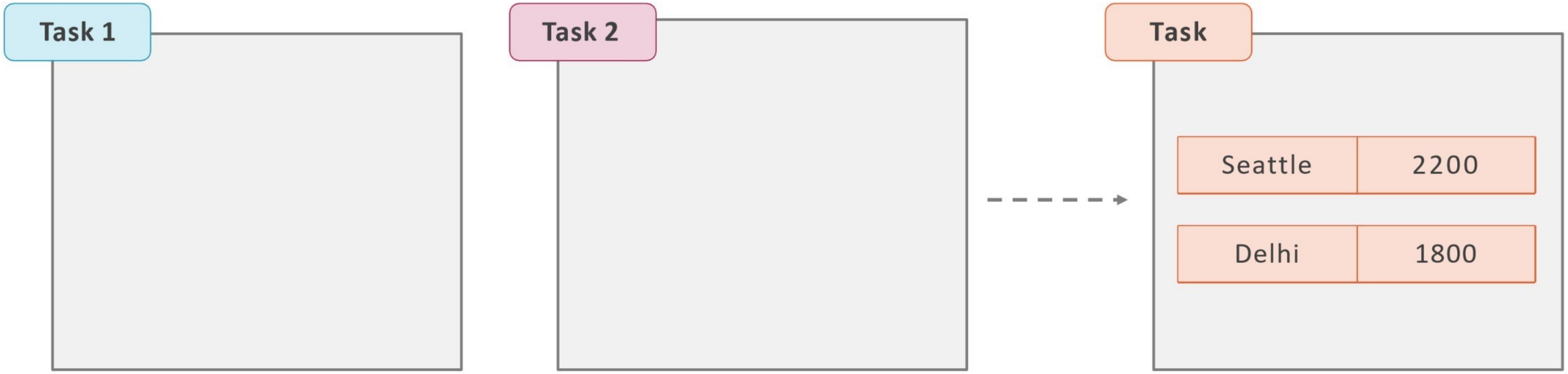
Calculate sum of Amount by City



Lambda Function

```
value1, value2: value1 + value2
```

Calculate sum of Amount by City



Lambda Function

```
value1, value2: value1 + value2
```

Calculate count of records for each Borough

+ Code + Markdown

```
# Create Pair RDD (LocationId as Key, Value as 1)

taxiZonesPairRdd = (
                    taxiZonesWithColsRdd

                        .map(lambda zoneRow: (
                                                zoneRow[1],   # Key   - Borough
                                                1             # Value - 1
                                              )
                             )
                   )

taxiZonesPairRdd.take(10)
```

20]

```
[('EWR', 1),
 ('Queens', 1),
 ('Bronx', 1),
 ('Manhattan', 1),
 ('Staten Island', 1),
 ('Staten Island', 1),
```

```python
# Calculate count of records for each Borough

boroughCountRdd = (
                    taxiZonesPairRdd

                        .reduceByKey( lambda value1, value2: value1 + value2 )
                    )

boroughCountRdd.collect()
```

```
[('Bronx', 43),
 ('Staten Island', 20),
 ('EWR', 1),
 ('Manhattan', 69),
 ('Brooklyn', 61),
 ('Unknown', 2),
 ('Queens', 69)]
```

Let’s explore more operations of Pair RDDs

```python
# Sort the data by Key

(
    taxiZonesPairRdd

        .sortByKey()

        .collect()
)
```

```
[('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
 ('Bronx', 1),
```

```python
# Only return the keys

(
    taxiZonesPairRdd

        .keys()

        .take(10)
)
```

```
['EWR',
 'Queens',
 'Bronx',
 'Manhattan',
 'Staten Island',
 'Staten Island',
 'Queens',
 'Queens',
 'Queens',
 'Queens']
```

```python
# Only return the values

(
    taxiZonesPairRdd

        .values()

        .take(10)
)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```python
# Return distinct pairs

(
    taxiZonesPairRdd

        .distinct()

        .collect()
)
```

```
[('Bronx', 1),
 ('Staten Island', 1),
 ('Queens', 1),
 ('EWR', 1),
 ('Manhattan', 1),
 ('Brooklyn', 1),
```

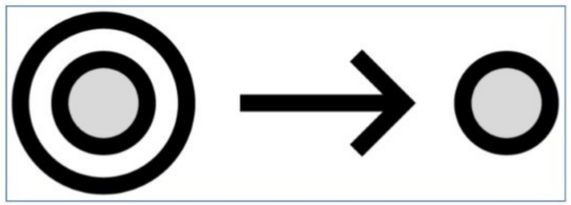
Applying Operations on RDDs

RDD Operations

Transformation

Action

Transformation Operation



Function that produces new RDD from existing RDDs

Transformation operations help in building a Lineage Graph

Examples:

- Read a file
- Convert sales amount from INR to USD
- Filter records with sales amount greater than 1000

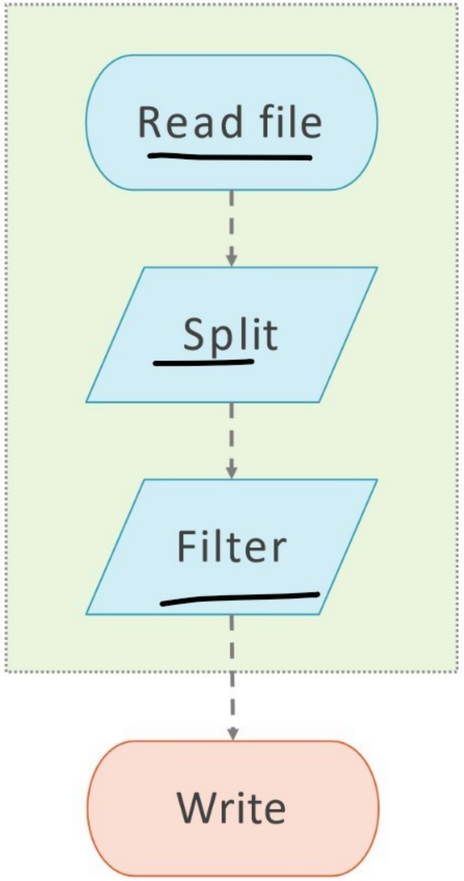
Transformations are Lazy operations, which are only executed when an Action operation is applied

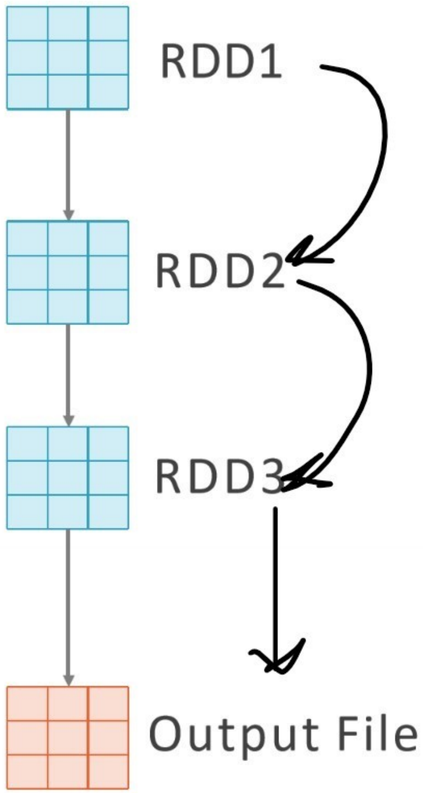
1. Read file1.csv from Storage → T
2. Split data by comma → T
3. Filter where City='Delhi' → T
4. Write processed data to storage → A

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

Lineage Graph

Built using transformation operations

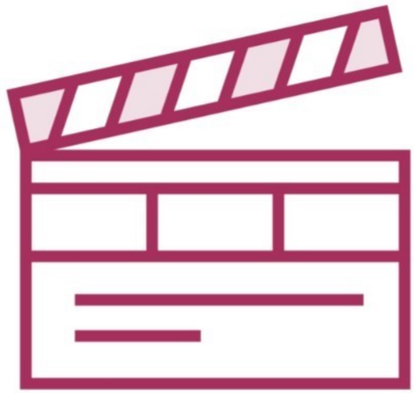




Action operation

collect()
take()
count()

Action Operation



Returns result of RDD computations

Triggers execution of transformations using Lineage Graph

Examples:

- Write the output to storage
- Print the output of operations
- Display the count

In [1]:
```
import findspark
findspark.init()
findspark.find()
```

Out[1]:
```
'E:\\Local d\\Zartab\\Apache Spark Installation\\spark-3.3.2-bin-hadoop3'
```

In [2]:
```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = (
            SparkSession
                .builder
                .appName("RDDOps")
                .master("local[4]")
                .getOrCreate()
        )
```

In [5]:
```
spark

sc=spark.sparkContext
```

In [6]:
```
# Create RDD using parallelize

numbersRDD= sc.parallelize([1,2,3,4,5,6])
```

In [7]:
```
spark
```

Out[7]:

SparkSession - in-memory

SparkContext

Spark UI

Version

`v3.3.2`

Master

`local[4]`

Spark 3.3.2 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame

Spark Jobs (?)

User: VICTUS
Total Uptime: 16 s
Scheduling Mode: FIFO

▸ Event Timeline

```
In [8]: numbersRDD.collect()
Out[8]: [1, 2, 3, 4, 5, 6]
```

```
In [ ]:
```

Spark 3.3.2 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | **RDDOps** application UI

Spark Jobs (?)

User: VICTUS
Total Uptime: 1.4 min
Scheduling Mode: FIFO
Completed Jobs: 1

▸ Event Timeline

▾ Completed Jobs (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

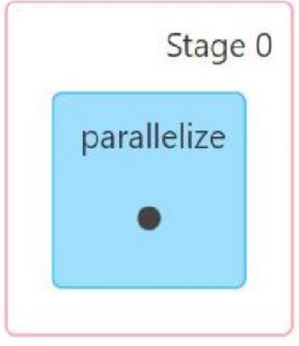
Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_24236\2661653226.py:1 collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_24236\2661653226.py:1	2023/05/18 06:58:17	0.6 s	1/1	4/4

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Details for Job 0

Status: SUCCEEDED
Submitted: 2023/05/18 06:58:17
Duration: 0.6 s
Completed Stages: 1

Event Timeline
DAG Visualization



Completed Stages (1)

Page: 1

1 Pages. Jump to 1 . Show 100

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output

Using Narrow Transformations

Transformation Operations

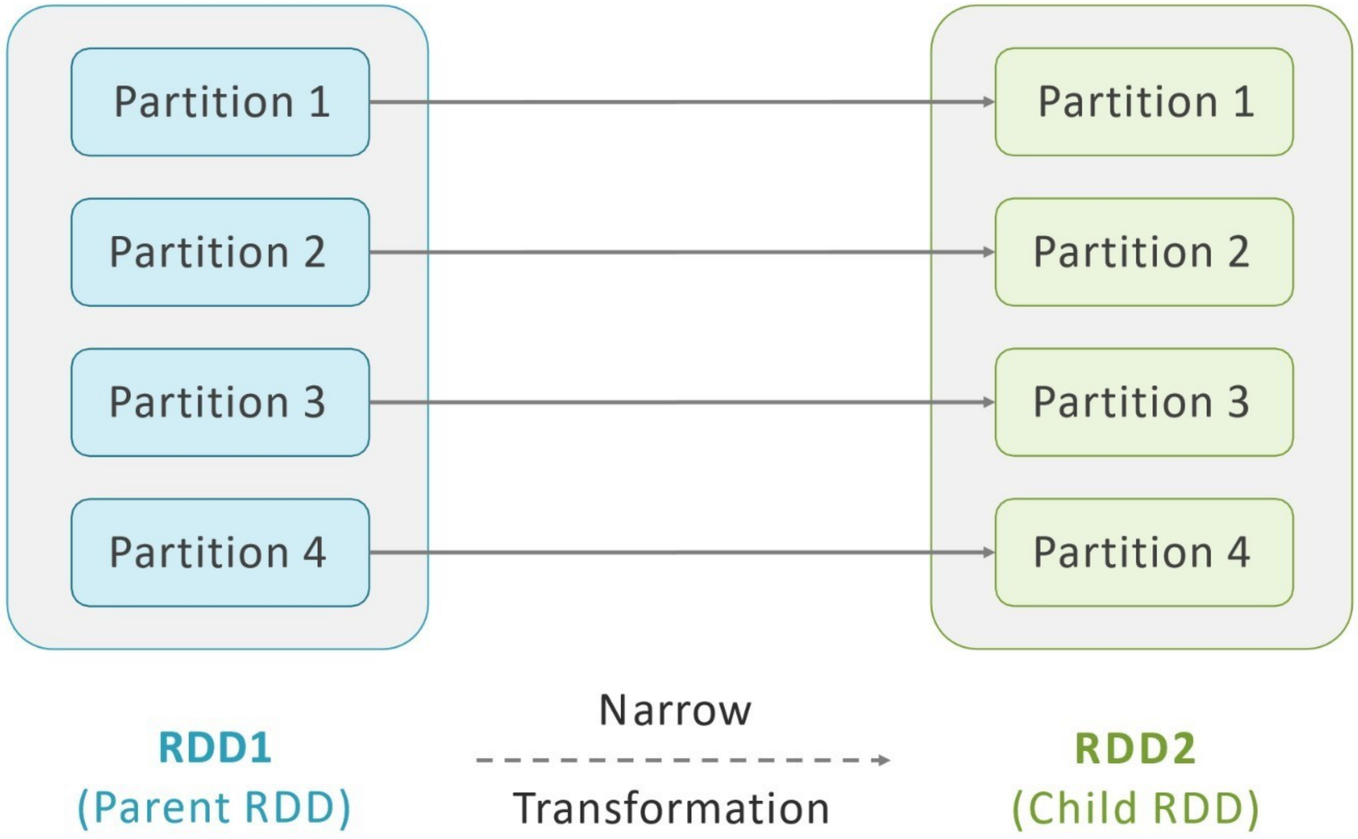
Narrow Dependency Transformation

Wide Dependency Transformation

A transformation where each input partition is used at-most once to produce output partitions

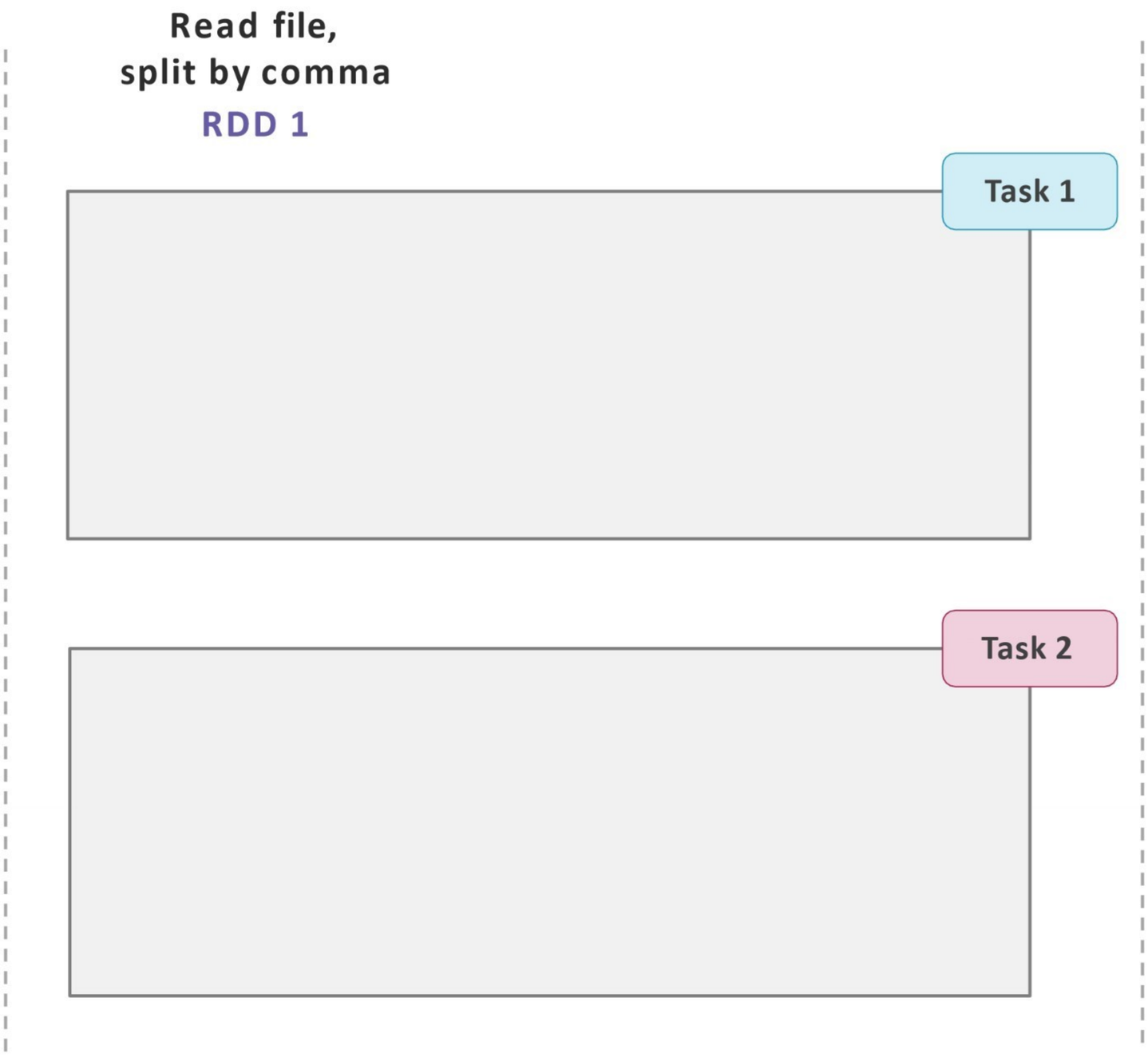
Narrow Transformations

Each input partition is used **at-most once** to produce output partitions



Filter & Map operations are examples of this

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



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

Read file, split by comma

RDD 1

→

Filter City = 'Seattle'

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

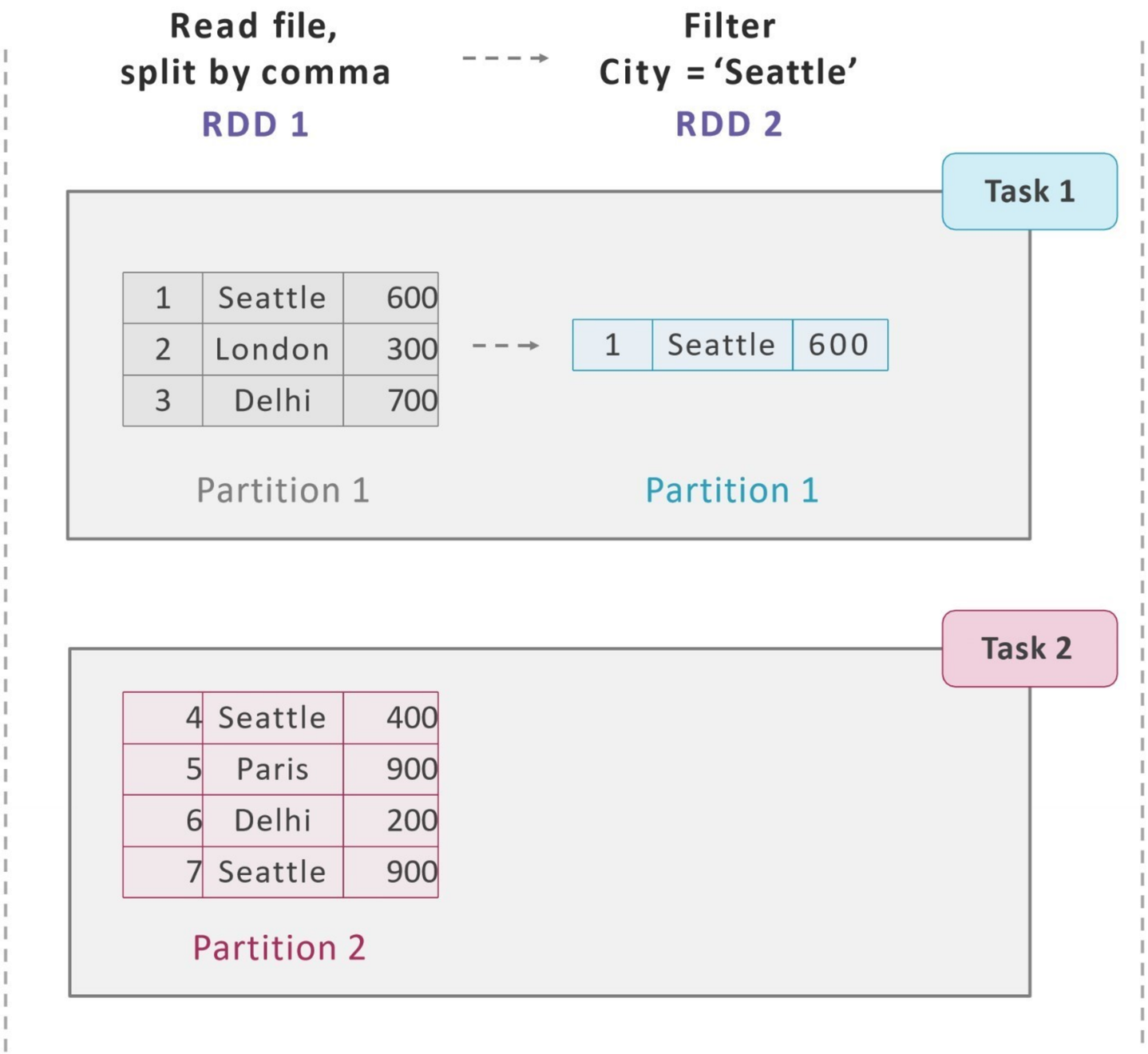
Partition 1

Task 2

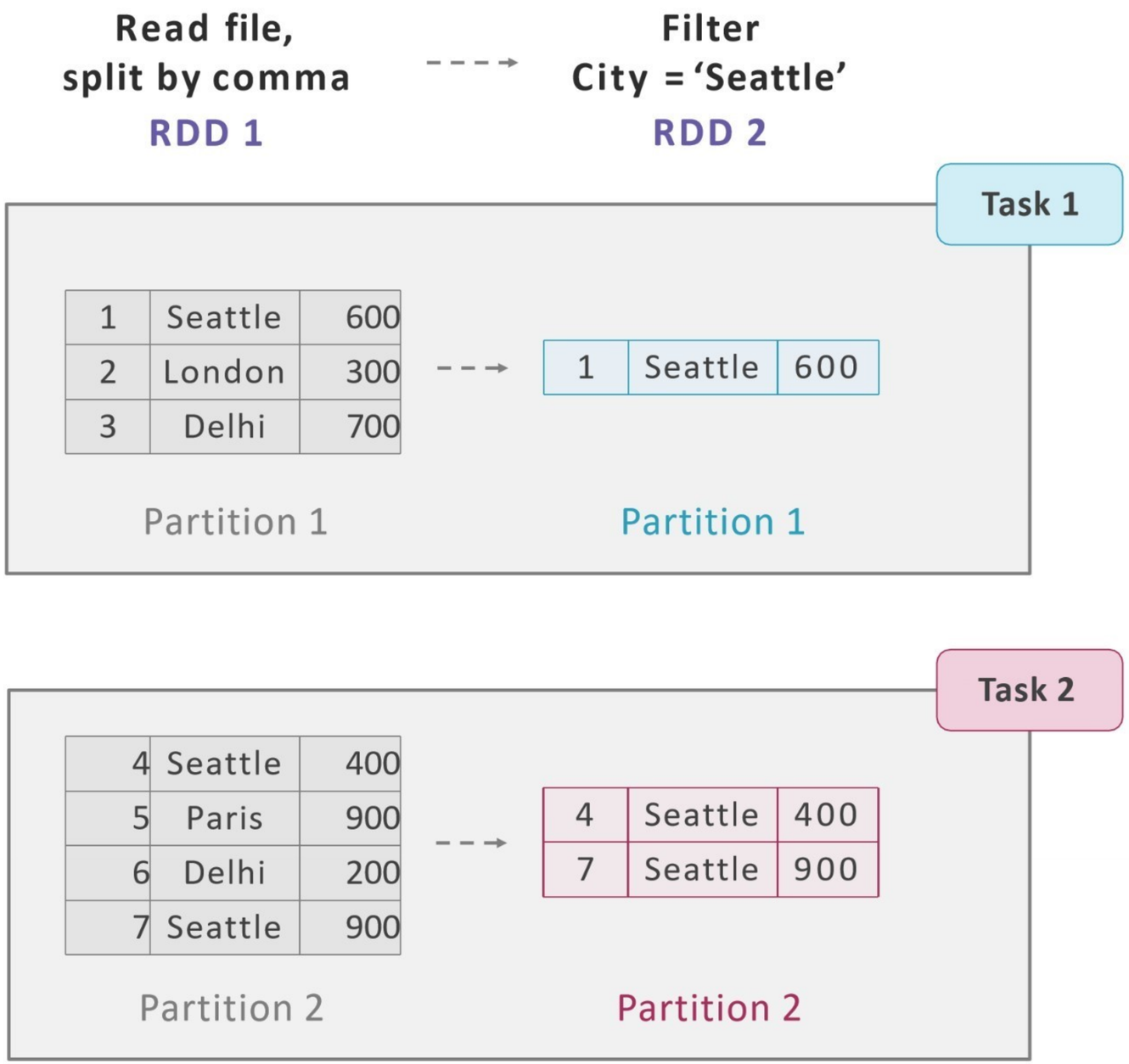
4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

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



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



Filter is a Narrow Transformation

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

Read file, split by comma

RDD 1

→

Filter City = 'Seattle'

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

→

1	Seattle	600

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

→

4	Seattle	400
7	Seattle	900

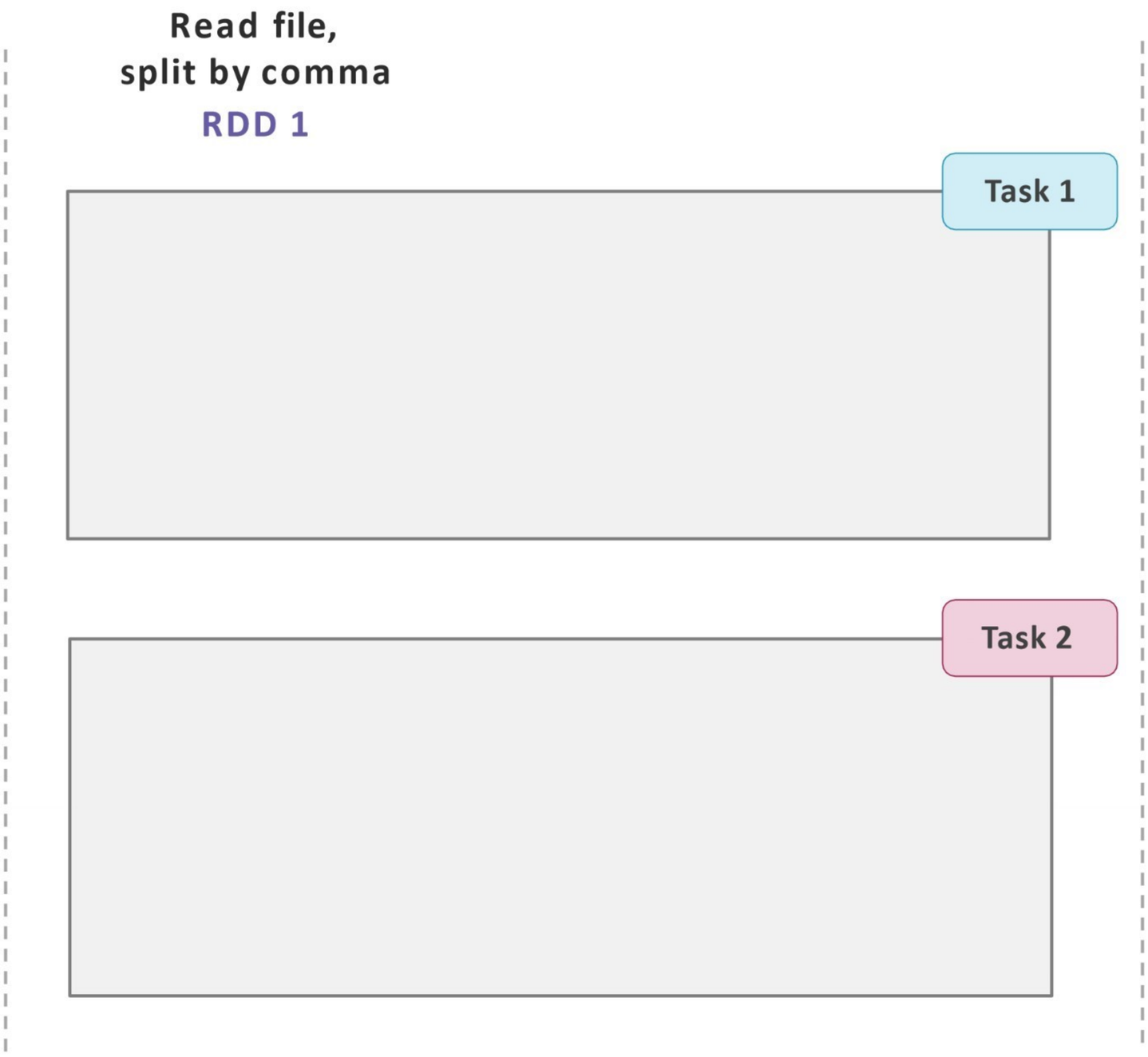
Partition 2

Filter is a Narrow Transformation

1	Seattle	600

4	Seattle	400
7	Seattle	900

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



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

Read file, split by comma

RDD 1

→

Use map() to derive annual amount

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

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

Read file, split by comma

RDD 1

Use map() to derive annual amount

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

1	Seattle	600	7200
2	London	300	3600
3	Delhi	700	8400

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

4	Seattle	400	4800
5	Paris	900	10800
6	Delhi	200	2400
7	Seattle	900	10800

Partition 2

Map is a Narrow Transformation

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

Read file, split by comma

RDD 1

Use map() to derive annual amount

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

1	Seattle	600	7200
2	London	300	3600
3	Delhi	700	8400

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

4	Seattle	400	4800
5	Paris	900	10800
6	Delhi	200	2400
7	Seattle	900	10800

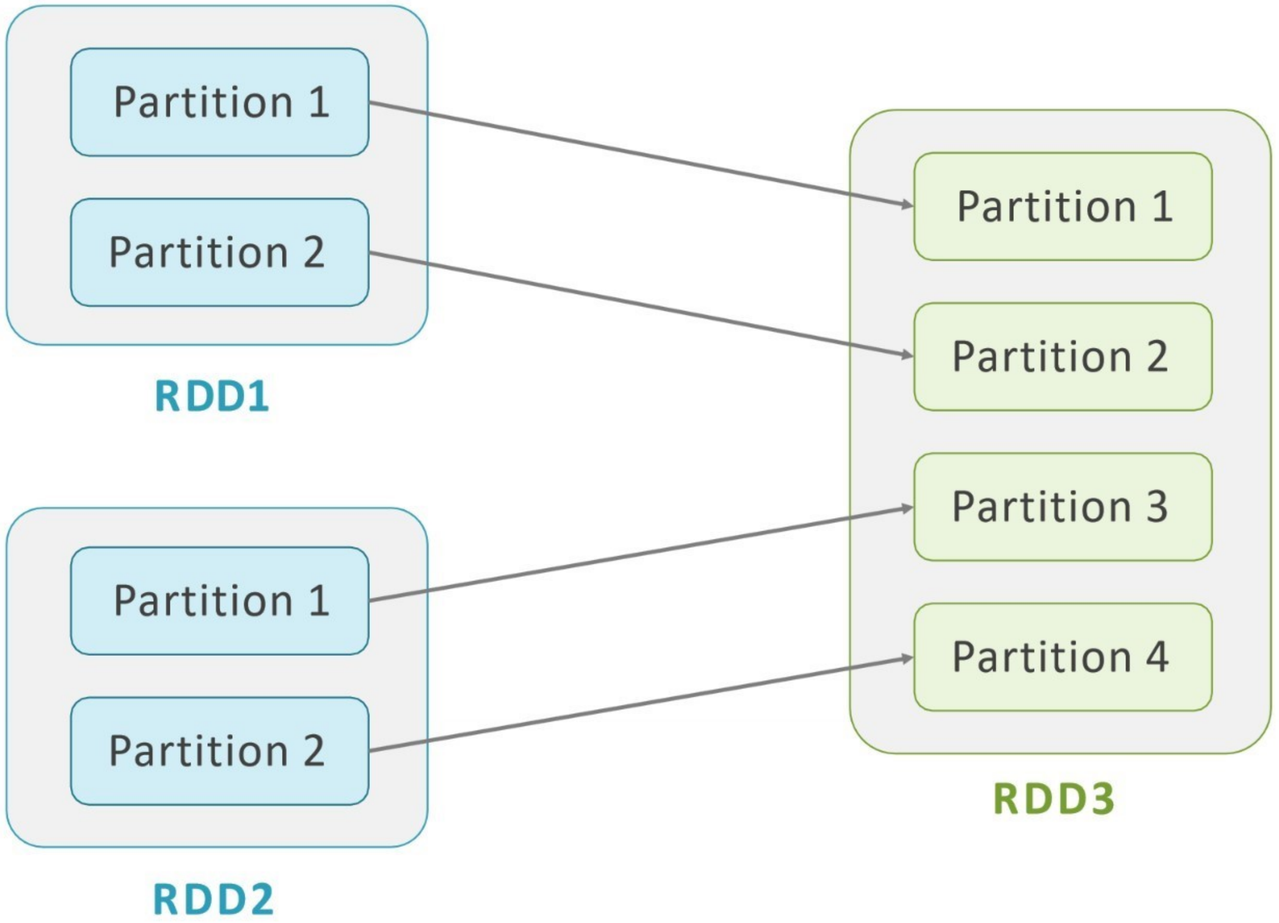
Partition 2

Map is a Narrow Transformation

1	Seattle	600	7200
2	London	300	3600
3	Delhi	700	8400

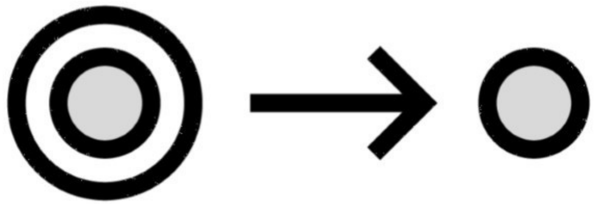
4	Seattle	400	4800
5	Paris	900	10800
6	Delhi	200	2400
7	Seattle	900	10800

Each input partition is used **at-most once** to produce output partitions



Union operation is an example of this

Narrow Transformation



Extremely fast

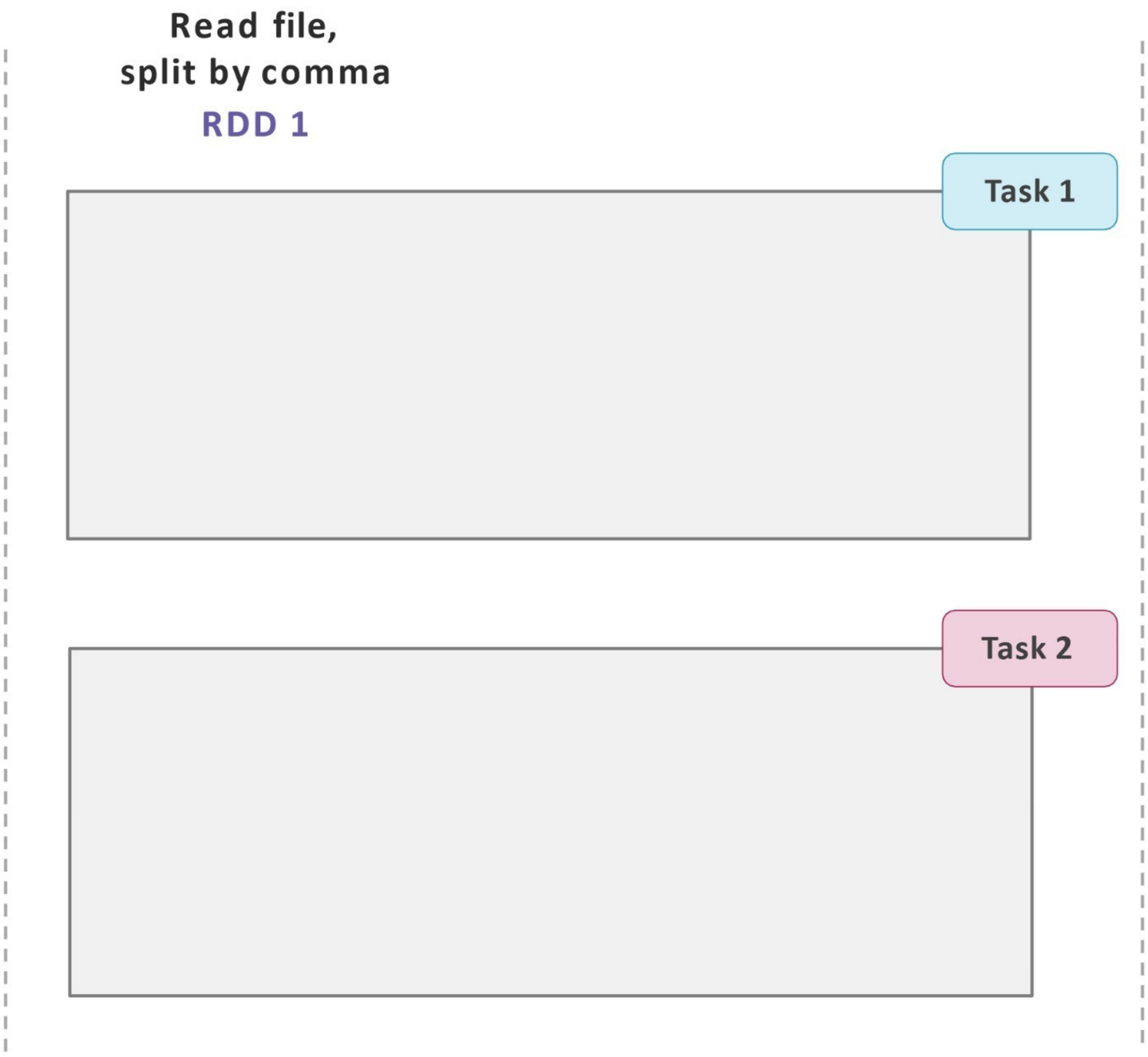
No data movement between partitions / No shuffling

Examples

- Filter, Map, FlatMap, MapPartition, Sample, Union etc.

Wide Transformations and Data Shuffling

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



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

Read file, split by comma

RDD 1

→

Group by City, calculate count

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

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

Read file, split by comma

RDD 1

→

Group by City, calculate count

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

→

Seattle	1
London	1
Delhi	1

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

→

Seattle	2
Paris	1
Delhi	1

Partition 2

?

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

Read file, split by comma

RDD 1

→

Group by City, calculate count

RDD 2

Task 1

1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

→

Seattle	1
London	1
Delhi	1

Partition 1

Task 2

4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

Partition 2

→

Seattle	2
Paris	1
Delhi	1

Partition 2

Incorrect Output!

Data is not completely grouped

Seattle	1
London	1
Delhi	1

Seattle	2
Paris	1
Delhi	1

Transformation Operations

Narrow Dependency Transformation

Wide Dependency Transformation

Wide Transformation is a two-step process and requires shuffling of data

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

Read file, get City &Amount

RDD 1

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Read file,
get City &Amount

RDD 1

Group by City, calculate sum of Amount

RDD 2

Step 1– Group: Shuffle / Exchange

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Shuffle Write

Read file,
get City &Amount
RDD 1

Group by City, calculate sum of Amount
RDD 2

Step 1– Group: Shuffle / Exchange

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Shuffle Write

Read file, get City &Amount

RDD 1

Group by City, calculate sum of Amount

RDD 2

Step 1– Group: Shuffle / Exchange

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Seattle	1300
Paris	900
Delhi	200

Shuffle Block 2

Shuffle Write → Shuffle Read

Read file, get City &Amount

RDD 1

Group by City, calculate sum of Amount

RDD 2

Step 1– Group: Shuffle / Exchange

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Seattle	1300
Paris	900
Delhi	200

Shuffle Block 2

Seattle	600
Seattle	1300

Shuffle Write → Shuffle Read

Read file, get City &Amount

RDD 1

Group by City, calculate sum of Amount

RDD 2

Step 1– Group: Shuffle / Exchange

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Seattle	1300
Paris	900
Delhi	200

Shuffle Block 2

Seattle	600
Seattle	1300

London	300

Shuffle Partition 1

Shuffle Write → Shuffle Read

Read file, get City &Amount

RDD 1

Group by City, calculate sum of Amount

RDD 2

Step 1– Group: Shuffle / Exchange

Step 2 – Sum

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Seattle	1300
Paris	900
Delhi	200

Shuffle Block 2

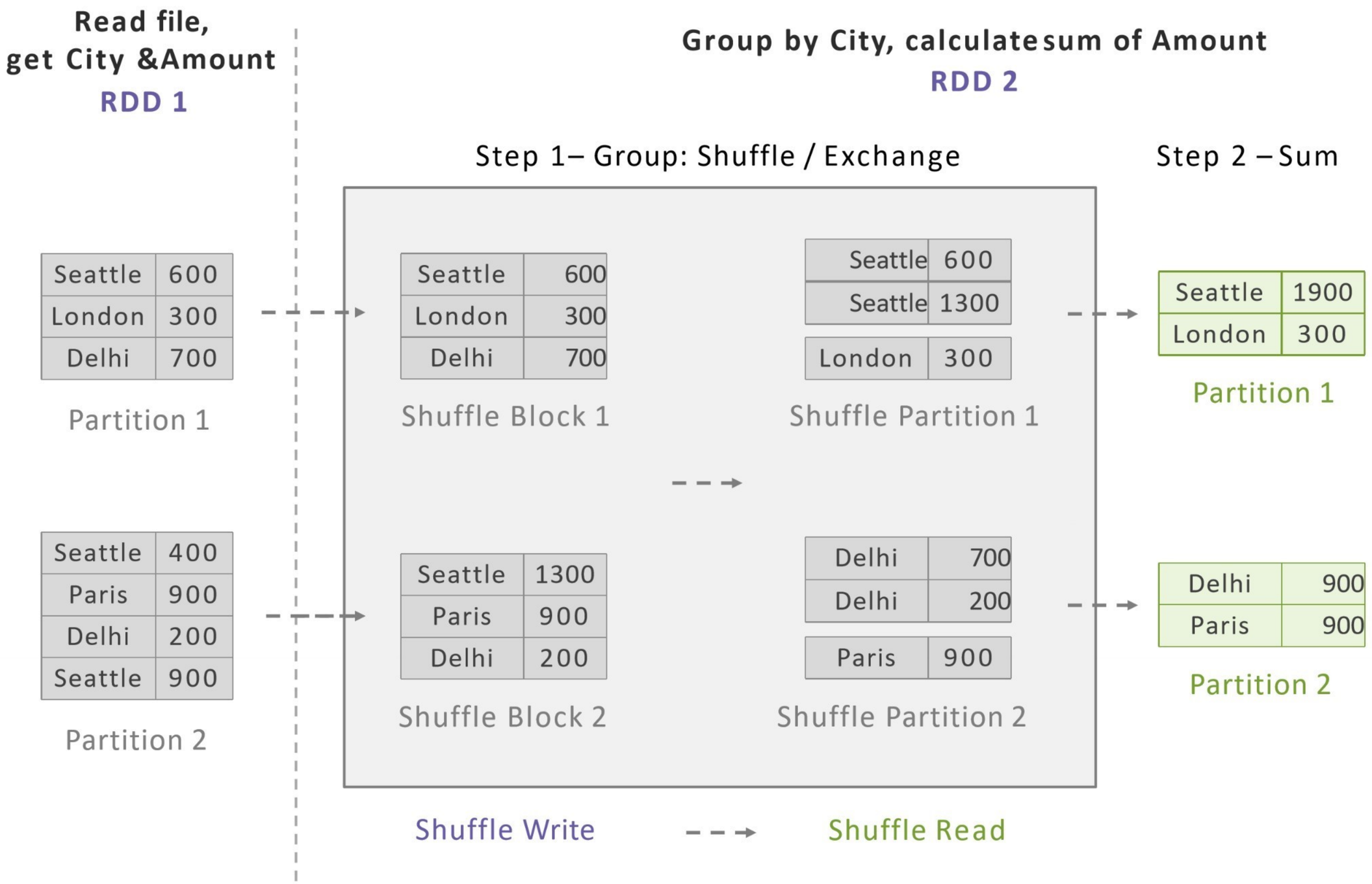
Seattle	600
Seattle	1300
London	300

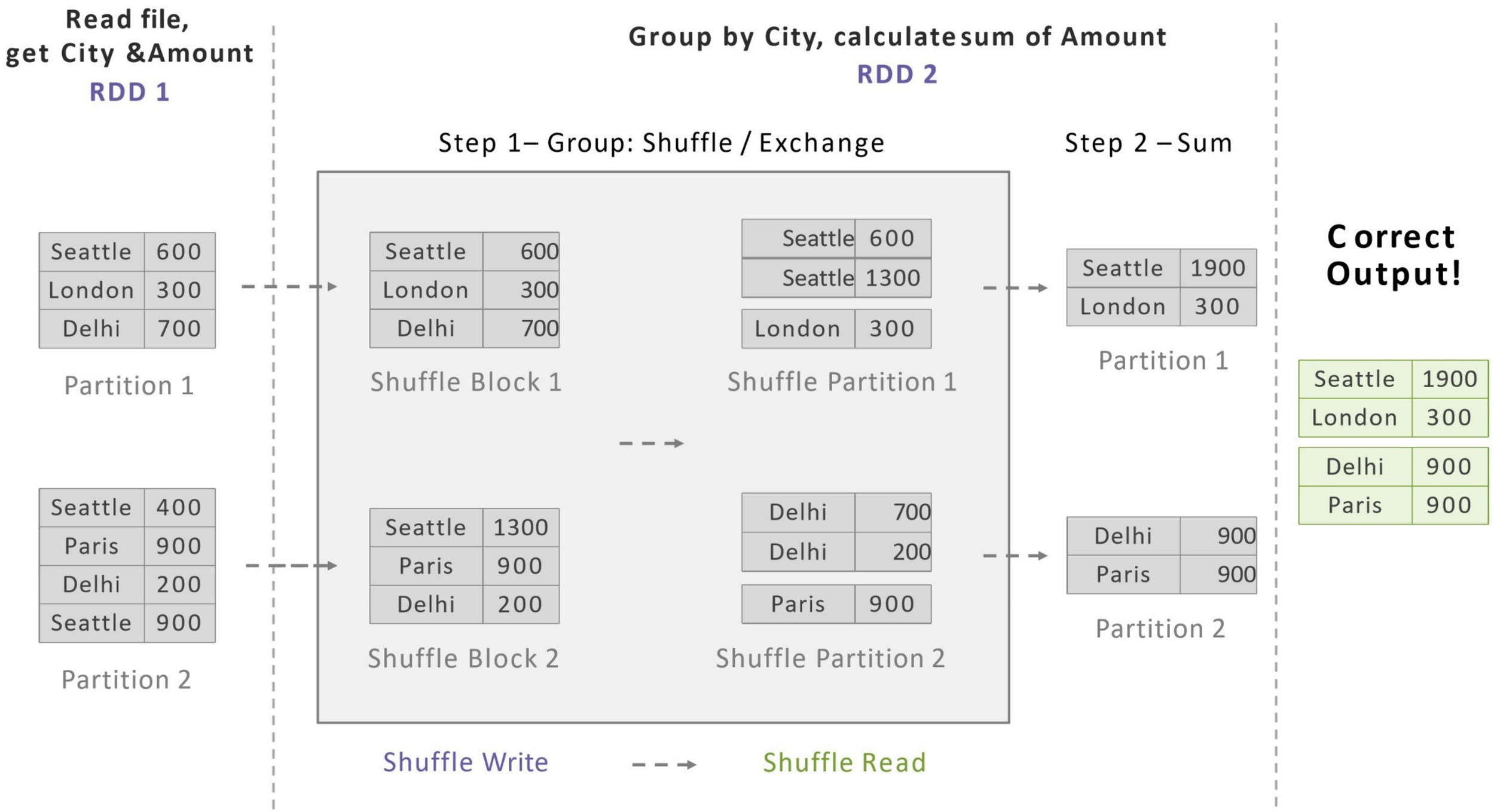
Shuffle Partition 1

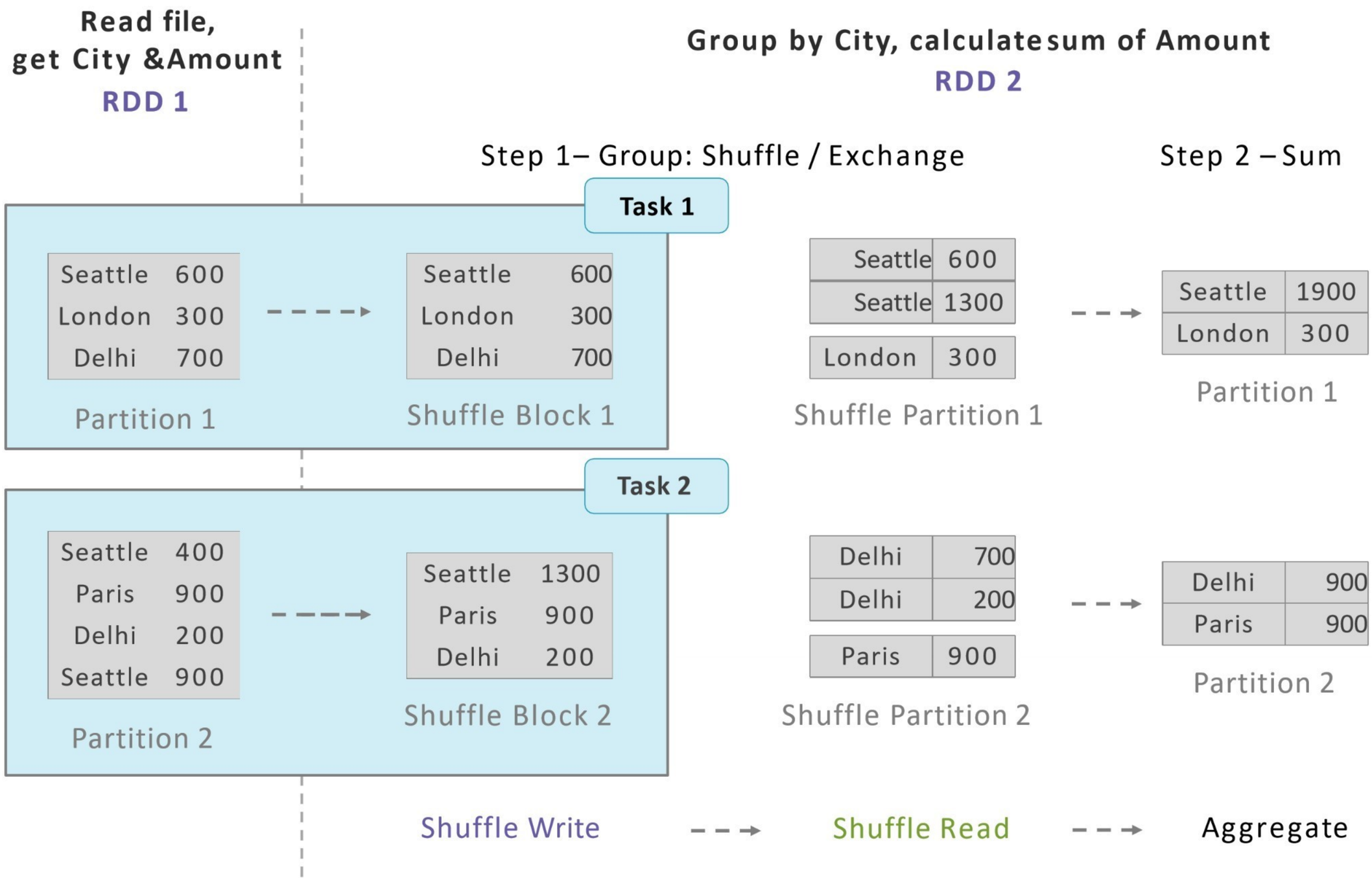
Delhi	700
Delhi	200
Paris	900

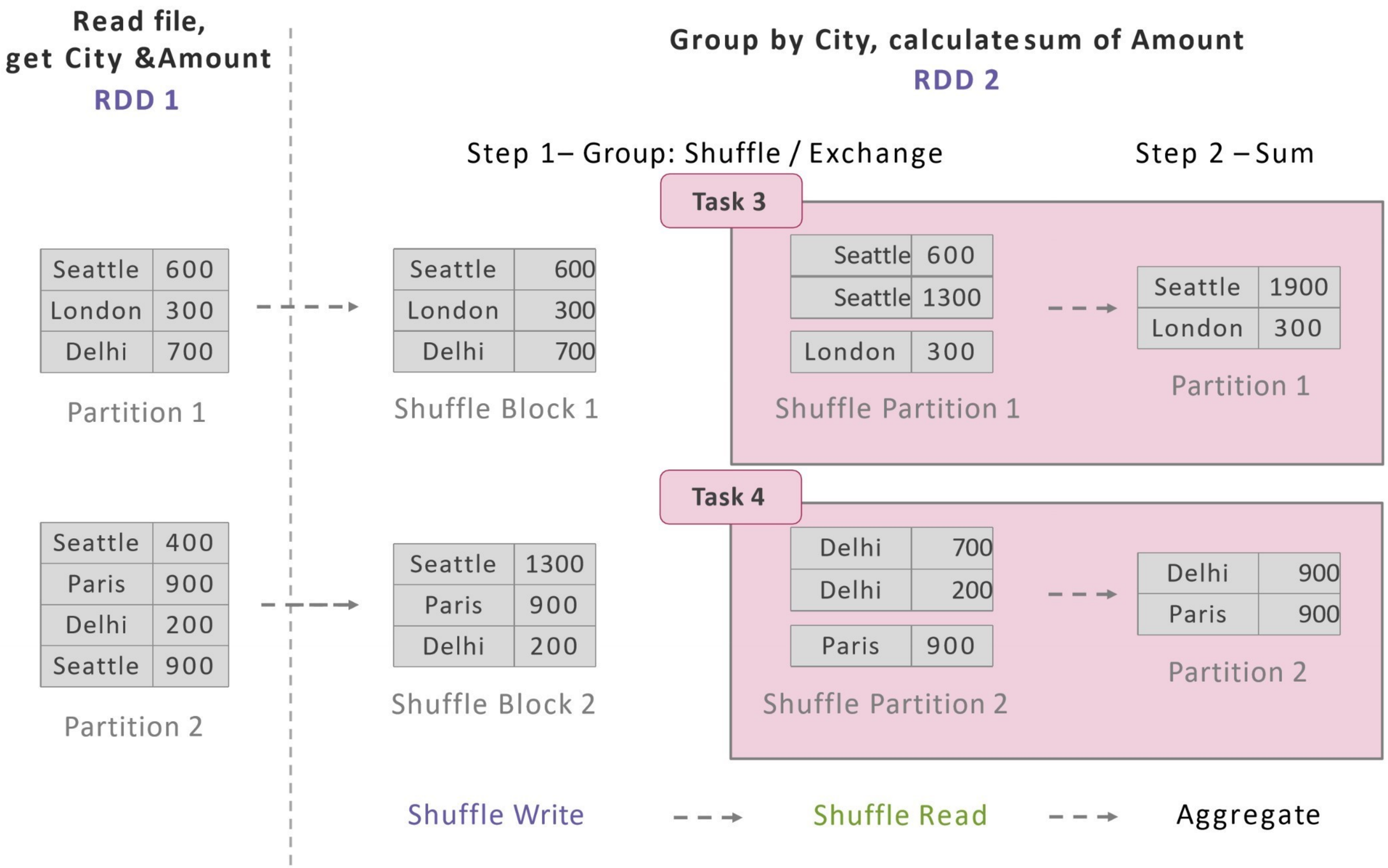
Shuffle Partition 2

Shuffle Write → Shuffle Read


Read file,
get City &Amount
RDD 1
Group by City, calculate sum of Amount
RDD 2
Step 1– Group: Shuffle / Exchange
Step 2 – Sum
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle 600
London 300
Delhi 700
Shuffle Block 1
Seattle 1300
Paris 900
Delhi 200
Shuffle Block 2
Seattle 600
Seattle 1300
London 300
Shuffle Partition 1
Delhi 700
Delhi 200
Paris 900
Shuffle Partition 2
Seattle 1900
London 300
Partition 1
Delhi 900
Paris 900
Partition 2
Shuffle Write
Shuffle Read

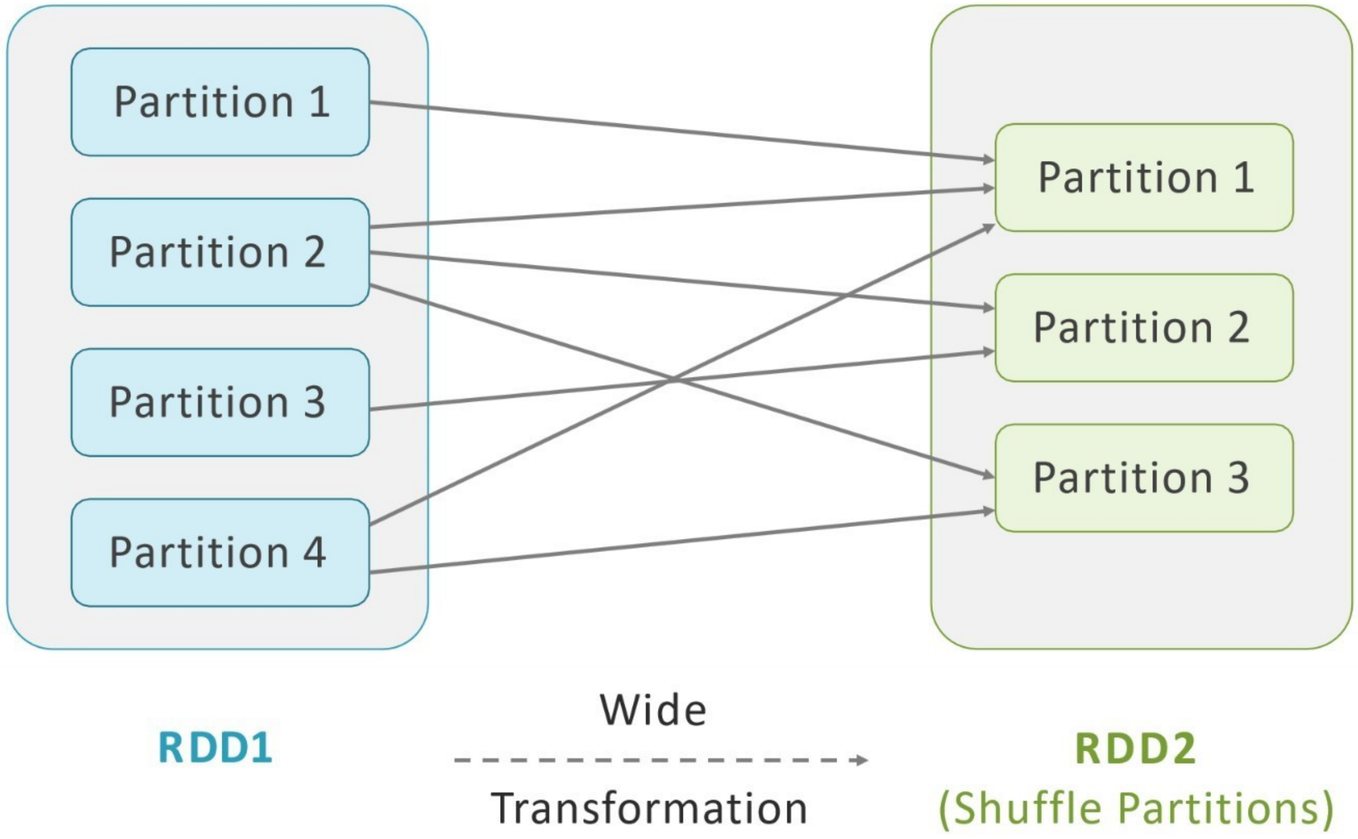

Read file,
get City &Amount
RDD 1
Group by City, calculate sum of Amount
RDD 2
Step 1– Group: Shuffle / Exchange
Step 2 – Sum
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle 600
London 300
Delhi 700
Shuffle Block 1
Seattle 1300
Paris 900
Delhi 200
Shuffle Block 2
Seattle 600
Seattle 1300
London 300
Shuffle Partition 1
Delhi 700
Delhi 200
Paris 900
Shuffle Partition 2
Seattle 1900
London 300
Partition 1
Delhi 900
Paris 900
Partition 2
Correct Output!
Seattle 1900
London 300
Delhi 900
Paris 900
Shuffle Write
Shuffle Read


Read file,
get City &Amount
RDD 1
Group by City, calculatesum of Amount
RDD 2
Step 1– Group: Shuffle / Exchange
Step 2 – Sum
Task 1
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 600
London 300
Delhi 700
Shuffle Block 1
Seattle 600
Seattle 1300
London 300
Shuffle Partition 1
Seattle 1900
London 300
Partition 1
Task 2
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle 1300
Paris 900
Delhi 200
Shuffle Block 2
Delhi 700
Delhi 200
Paris 900
Shuffle Partition 2
Delhi 900
Paris 900
Partition 2
Shuffle Write
Shuffle Read
Aggregate


Read file,
get City &Amount
RDD 1
Group by City, calculate sum of Amount
RDD 2
Step 1– Group: Shuffle / Exchange
Step 2 – Sum
Task 3
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 600
London 300
Delhi 700
Shuffle Block 1
Seattle 600
Seattle 1300
London 300
Shuffle Partition 1
Seattle 1900
London 300
Partition 1
Task 4
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle 1300
Paris 900
Delhi 200
Shuffle Block 2
Delhi 700
Delhi 200
Paris 900
Shuffle Partition 2
Delhi 900
Paris 900
Partition 2
Shuffle Write
Shuffle Read
Aggregate

In Wide Transformation, one input partition might be used multiple times to produce output partitions

One input partition might be used multiple times to produce output partitions



ReduceByKey & Distinct operations are examples of this

Read file,
get City &Amount
RDD 1

Get distinct cities
RDD 2

Step 1– Group: Shuffle / Exchange

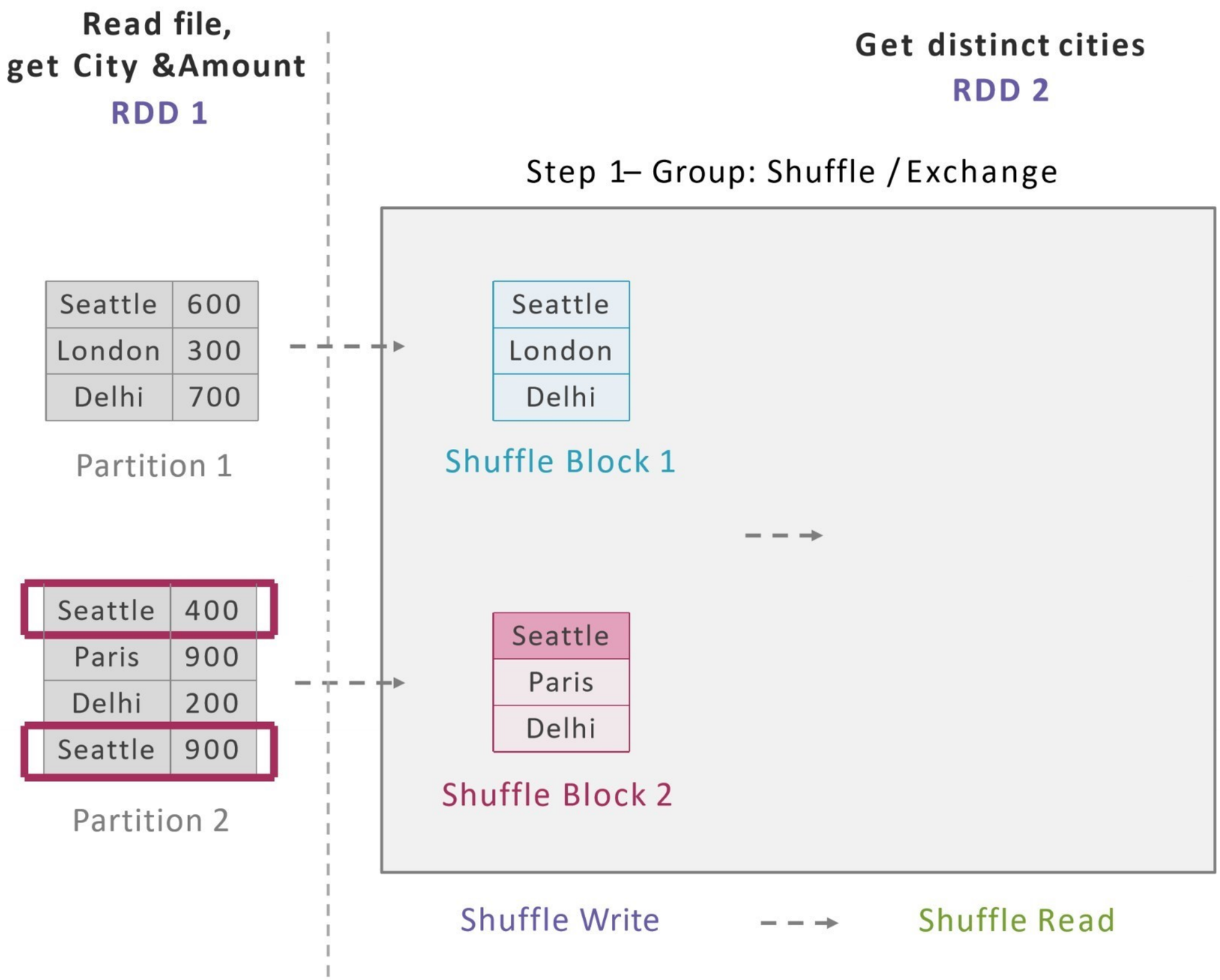
Seattle	600
London	300
Delhi	700

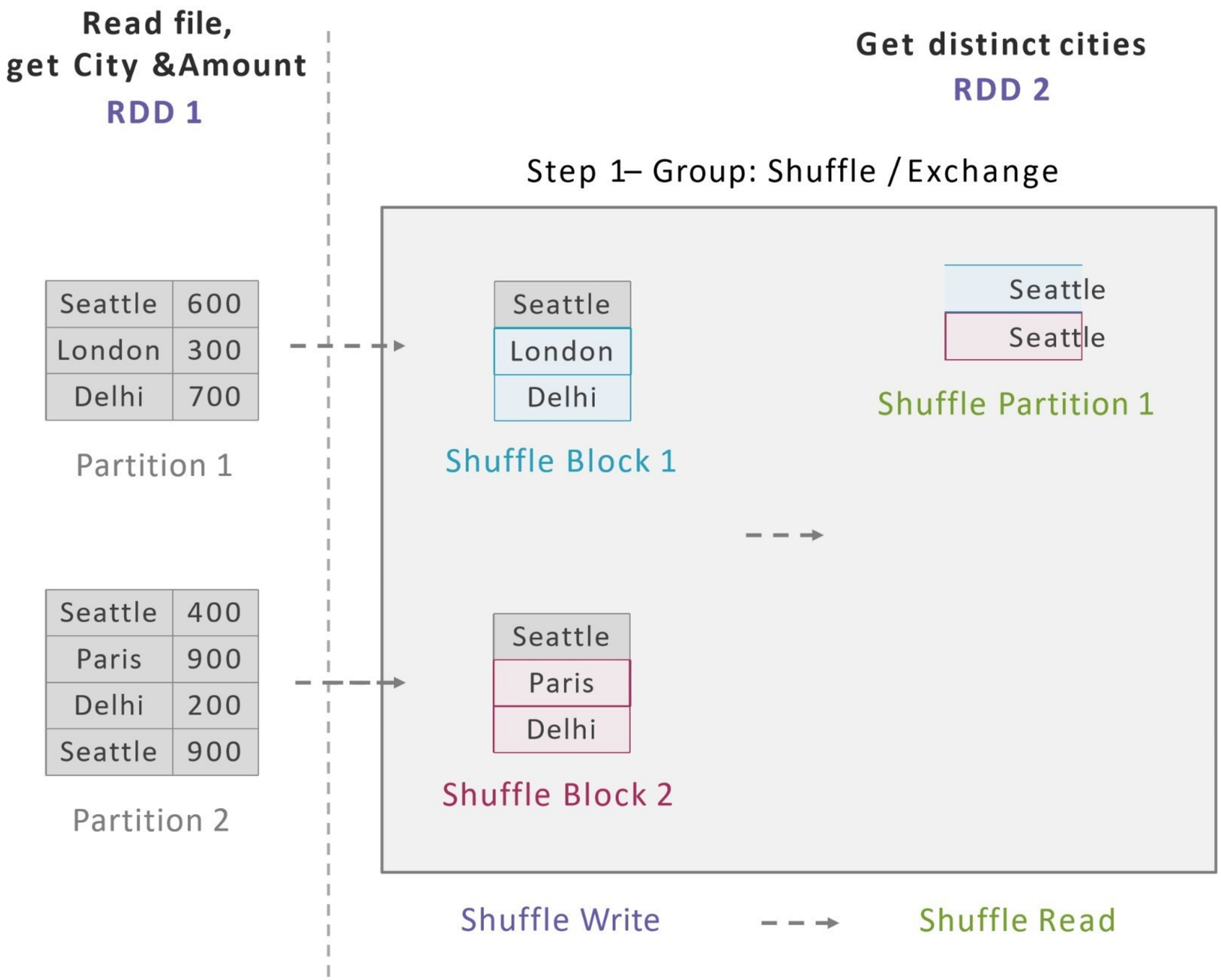
Partition 1

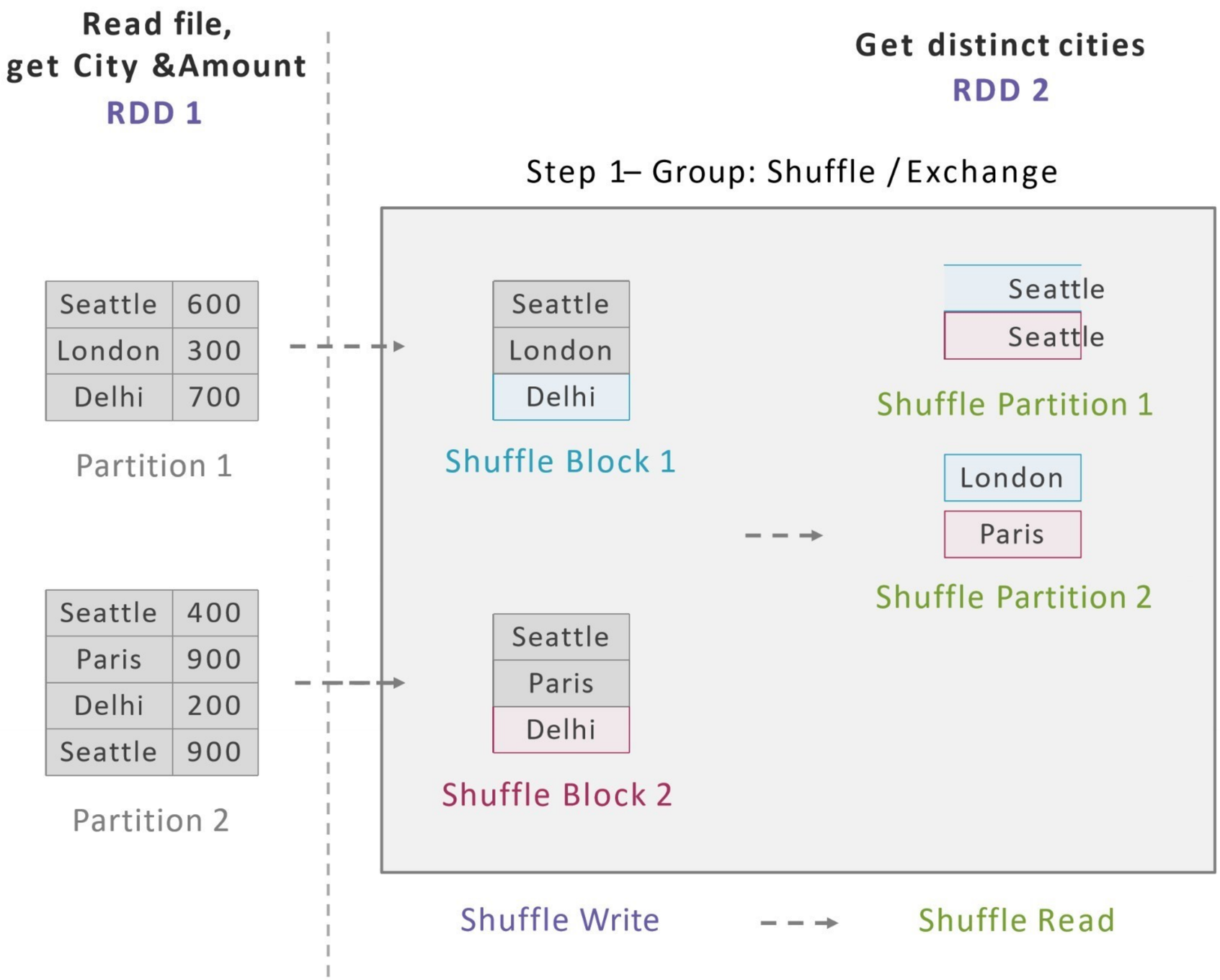
Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Shuffle Write


Read file,
get City &Amount
RDD 1
Get distinct cities
RDD 2
Step 1– Group: Shuffle / Exchange
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Shuffle Write
Shuffle Read


Read file,
get City &Amount
RDD 1
Get distinct cities
RDD 2
Step 1– Group: Shuffle / Exchange
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle Partition 1
Shuffle Write
Shuffle Read


Read file,
get City &Amount
RDD 1
Get distinct cities
RDD 2
Step 1– Group: Shuffle / Exchange
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle Partition 1
London
Paris
Shuffle Partition 2
Shuffle Write
Shuffle Read

Read file, get City &Amount

RDD 1

Get distinct cities

RDD 2

Step 1– Group: Shuffle / Exchange

Step 2 – ~~Sum~~ Distinct

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Seattle
London
Delhi

Shuffle Block 1

Seattle
Paris
Delhi

Shuffle Block 2

Seattle

Seattle

Shuffle Partition 1

London

Paris

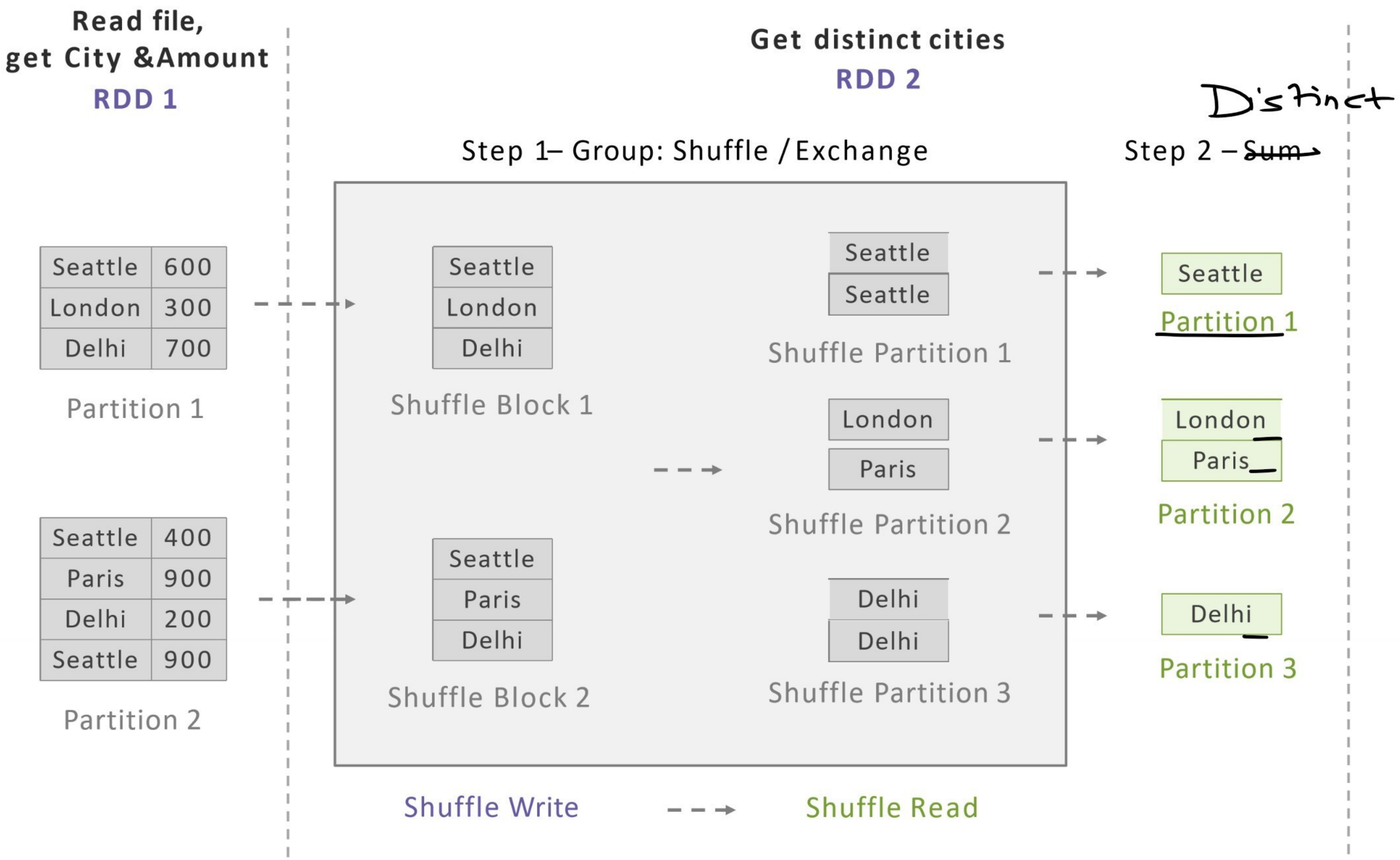
Shuffle Partition 2

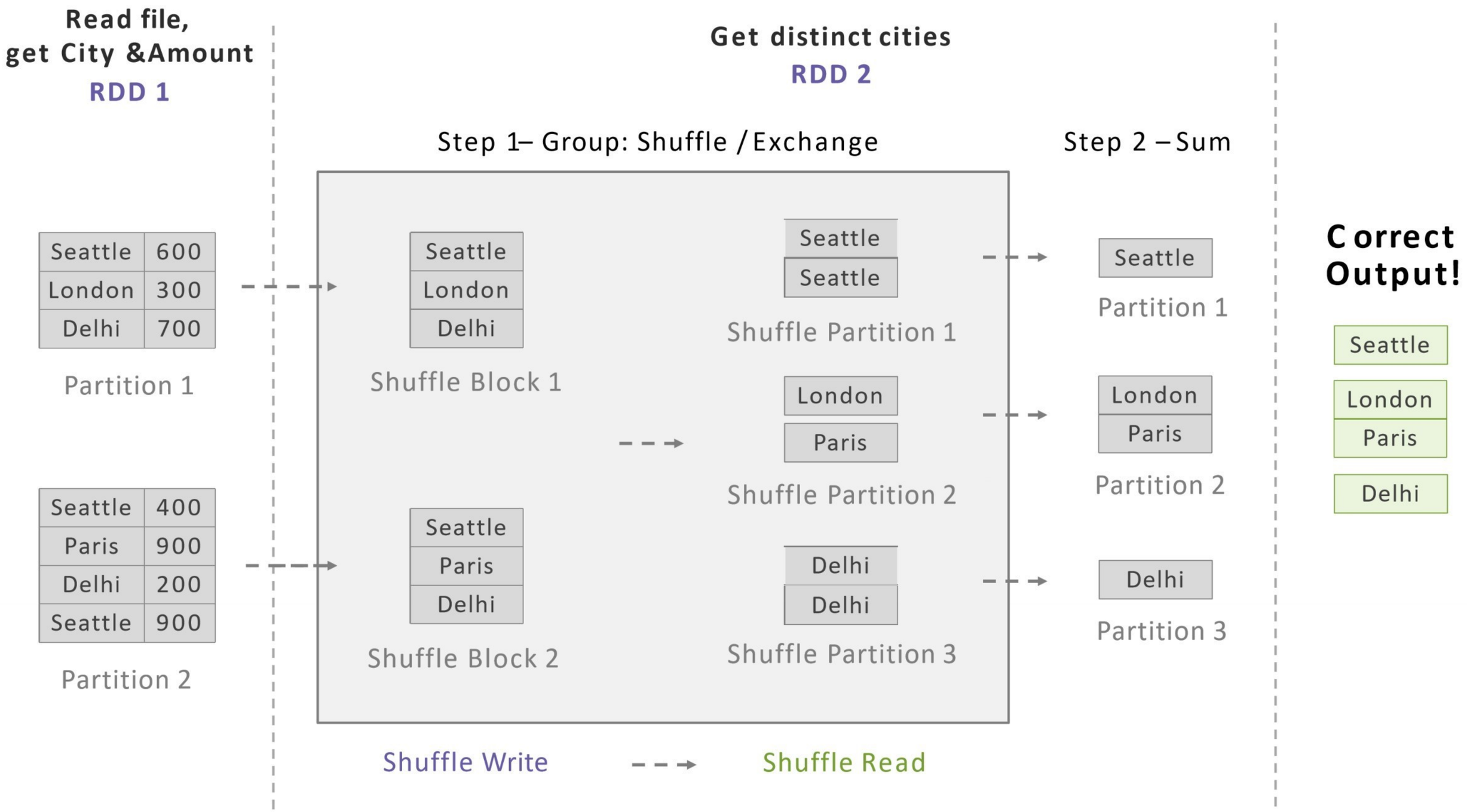
Delhi

Delhi

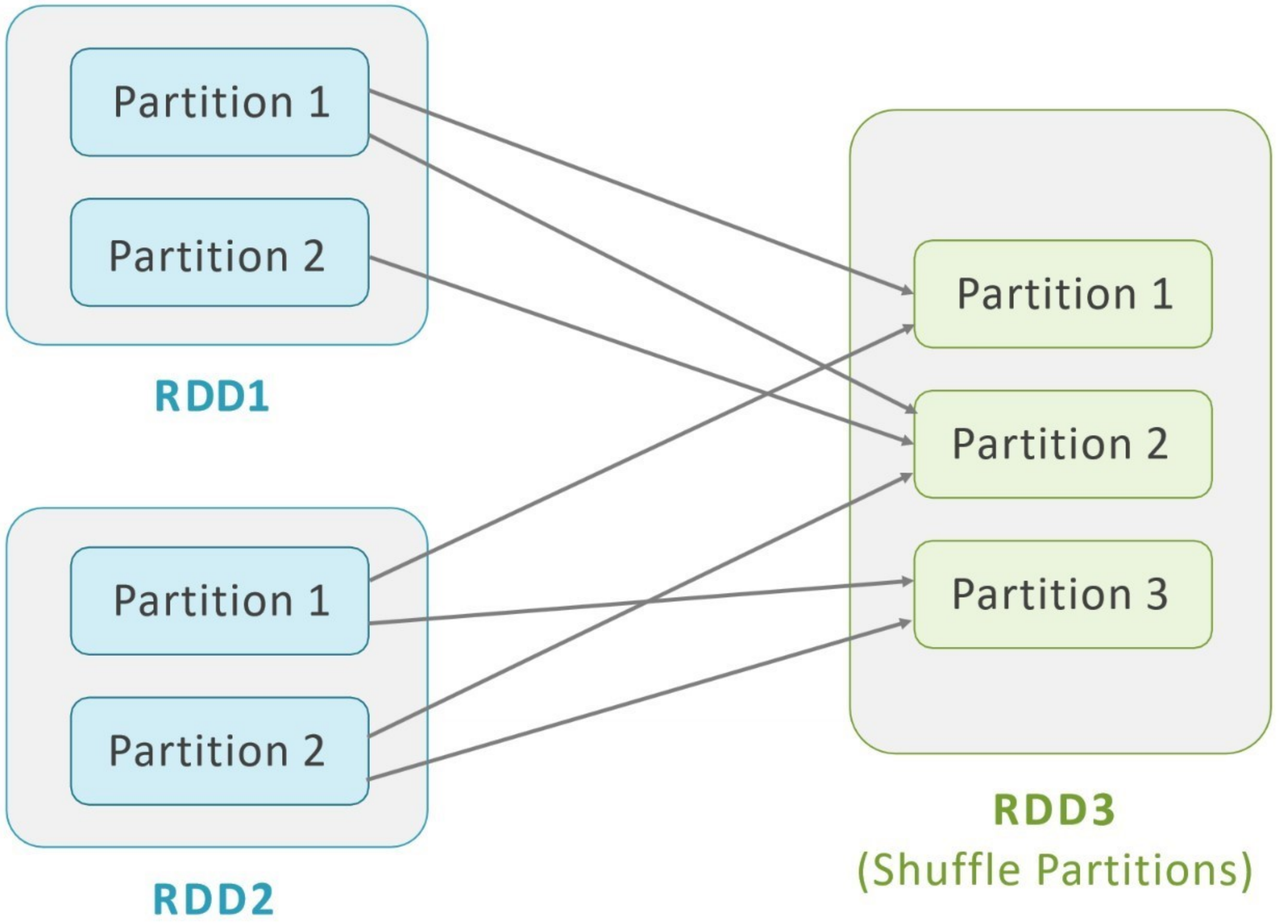
Shuffle Partition 3

Shuffle Write → Shuffle Read


Read file,
get City &Amount
RDD 1
Get distinct cities
RDD 2
Distinct
Step 1– Group: Shuffle / Exchange
Step 2 – ~~Sum~~
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle Partition 1
London
Paris
Shuffle Partition 2
Delhi
Delhi
Shuffle Partition 3
Seattle
Partition 1
London
Paris
Partition 2
Delhi
Partition 3
Shuffle Write
Shuffle Read

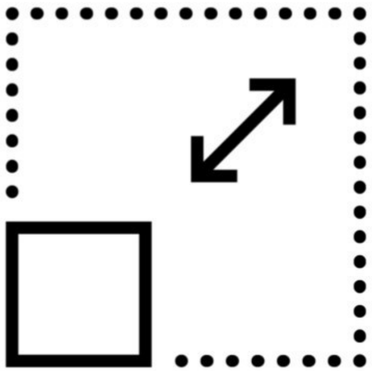

Read file,
get City &Amount
RDD 1
Get distinct cities
RDD 2
Step 1– Group: Shuffle / Exchange
Step 2 – Sum
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle Partition 1
London
Paris
Shuffle Partition 2
Delhi
Delhi
Shuffle Partition 3
Seattle
Partition 1
London
Paris
Partition 2
Delhi
Partition 3
Correct Output!
Seattle
London
Paris
Delhi
Shuffle Write
Shuffle Read

One input partition might be used multiple times to produce output partitions



Join operation is an example of this

Wide Transformation



Requires shuffling of data between partitions

Expensive operation

Shuffle Read can only start when all data is written out as Shuffle Blocks

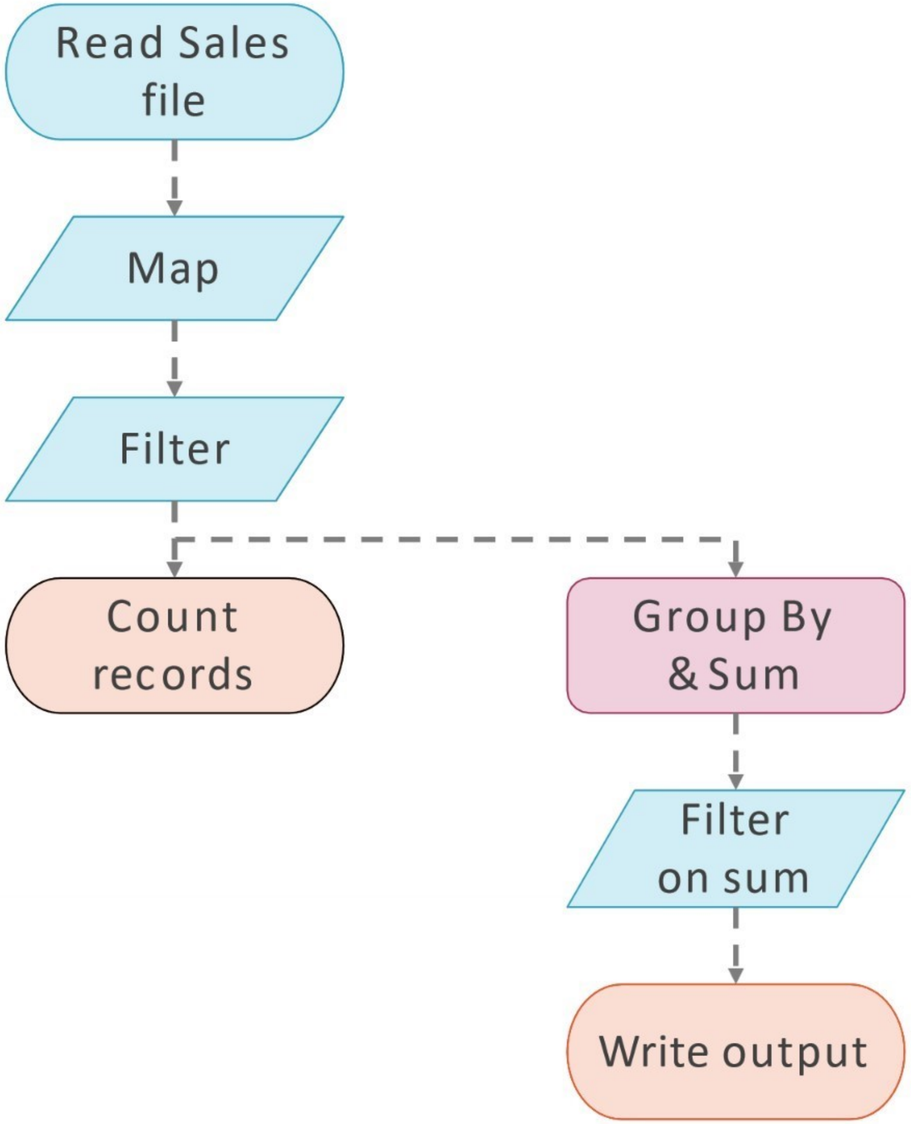
Number of shuffle partitions can be different than parent RDD partitions

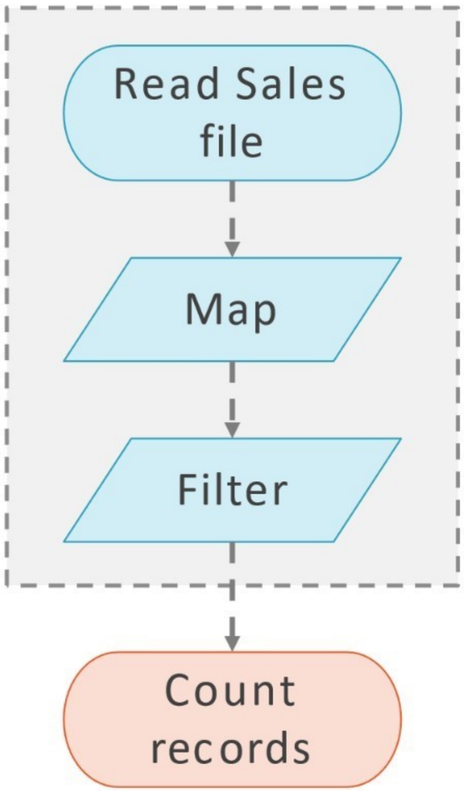
Examples

- reduceByKey, aggregateByKey, distinct, join, intersection etc.

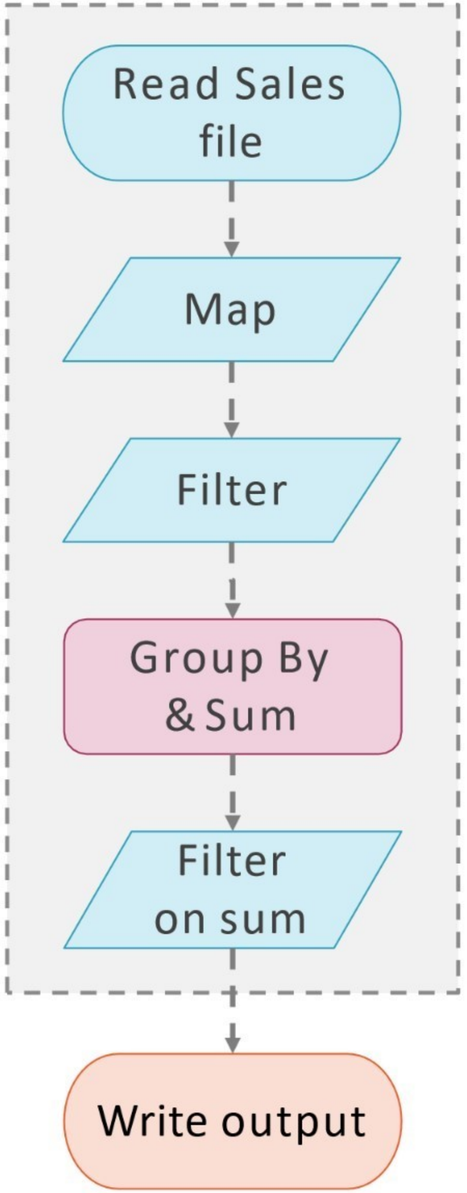
Spark Application Concepts: Jobs, Stages & Tasks

1. Read Sales csv file from Storage
2. Split Sales data by comma & define schema
3. Filter Sales where City = 'Delhi' → T
4. Count number of Sales → A
5. Group by Product, calculate sum of Amount
6. Filter grouped data where TotalAmount > '10000'
7. Write processed data to storage



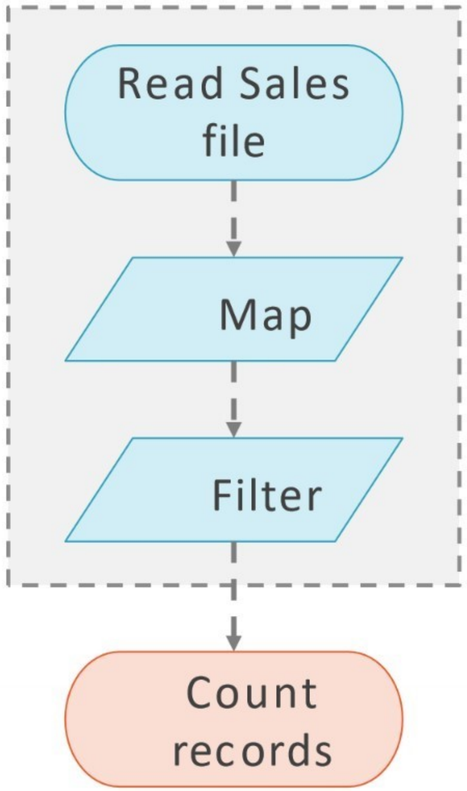


Job 1

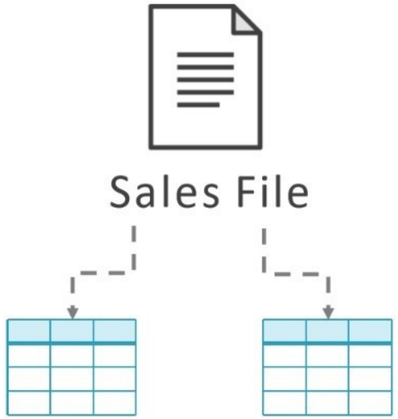


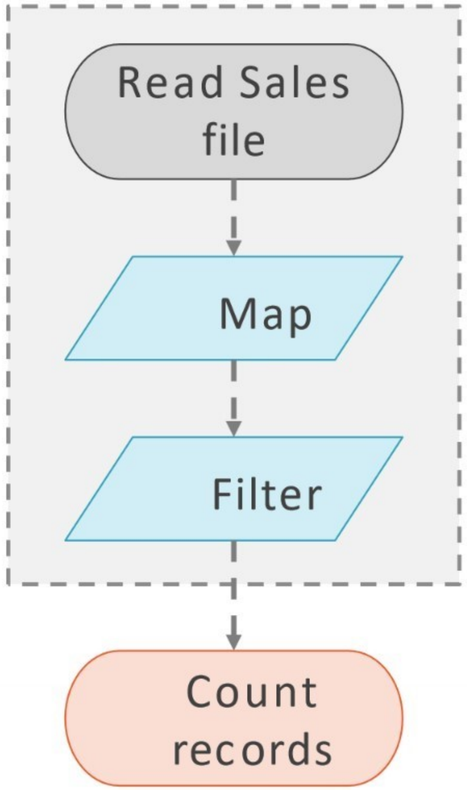
Job 2

No of Jobs
=
Action Operations

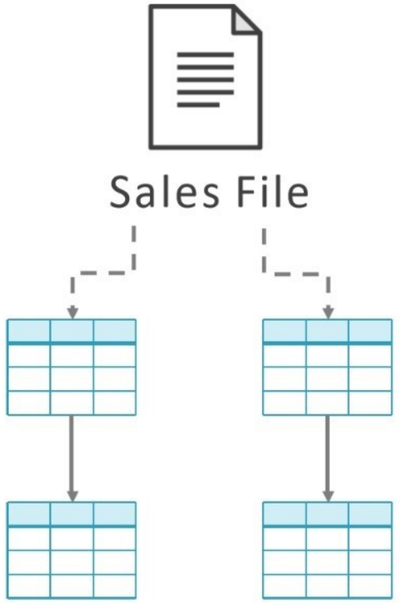


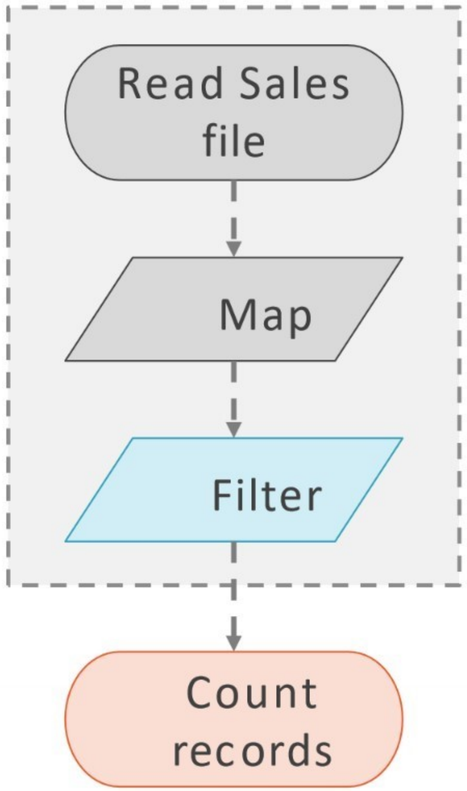
Job 1



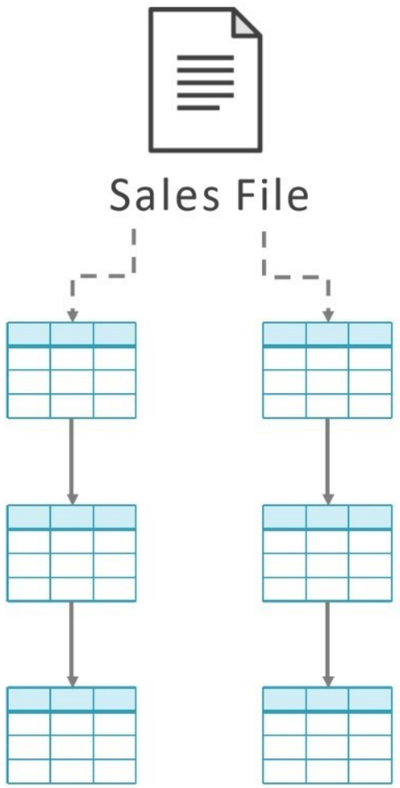


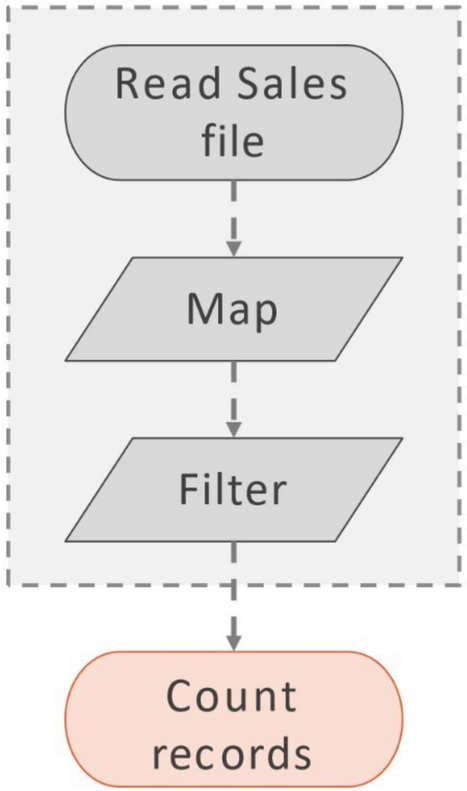
Job 1



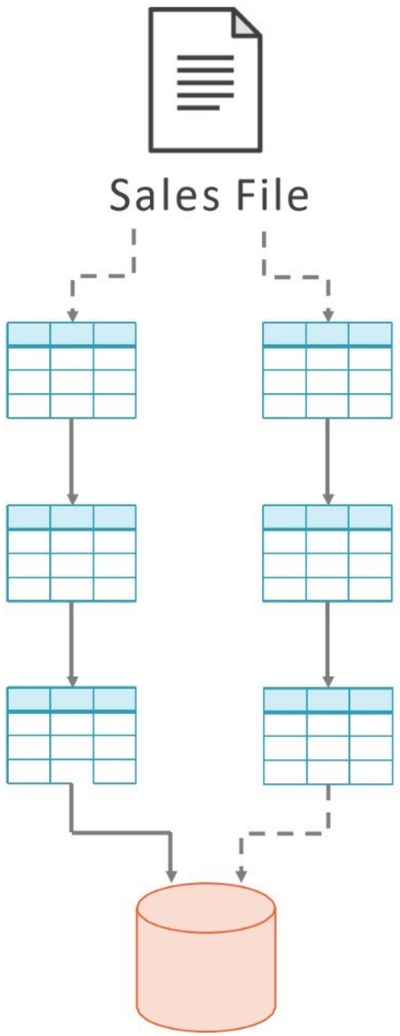


Job 1



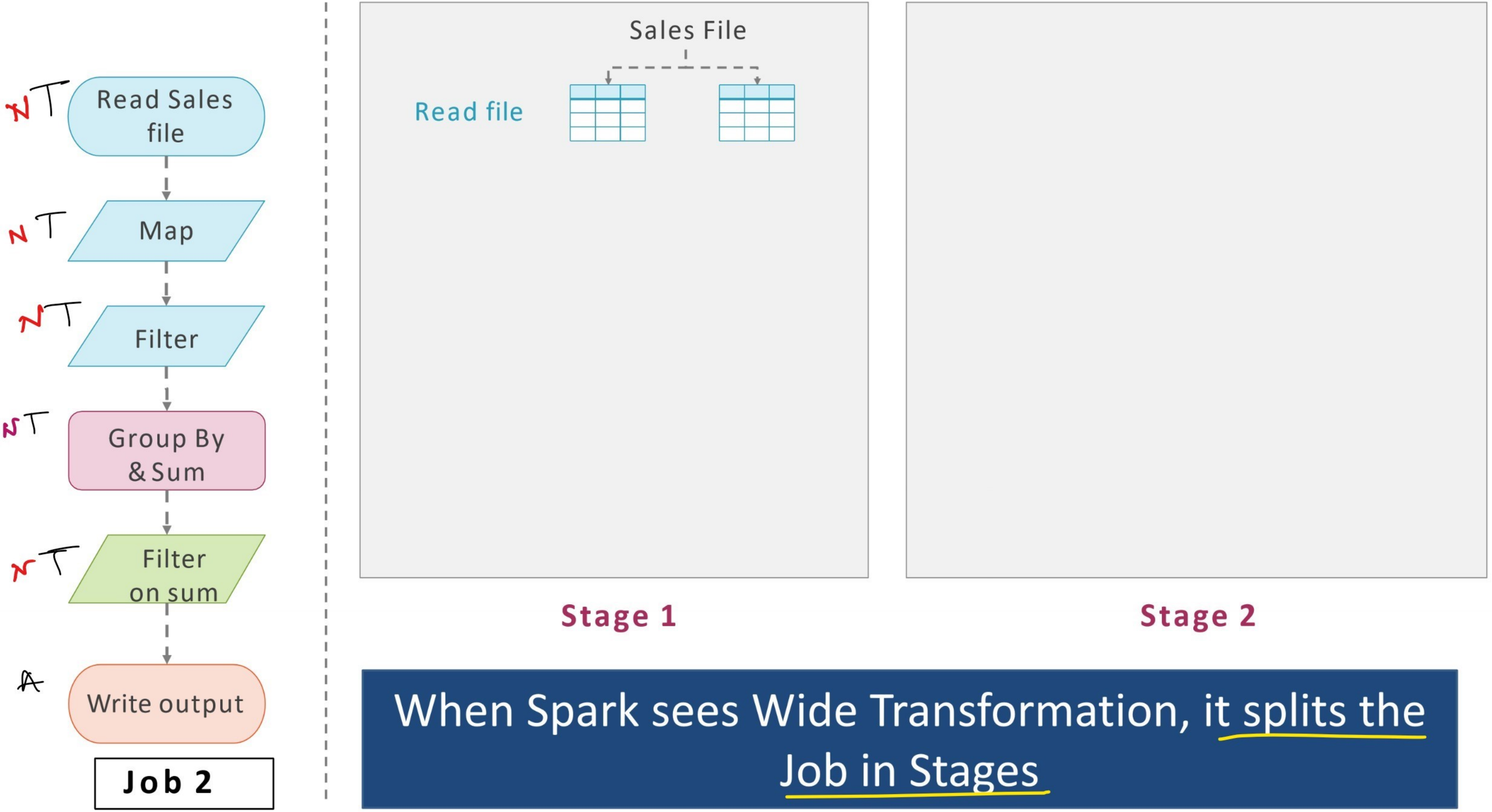


Job 1

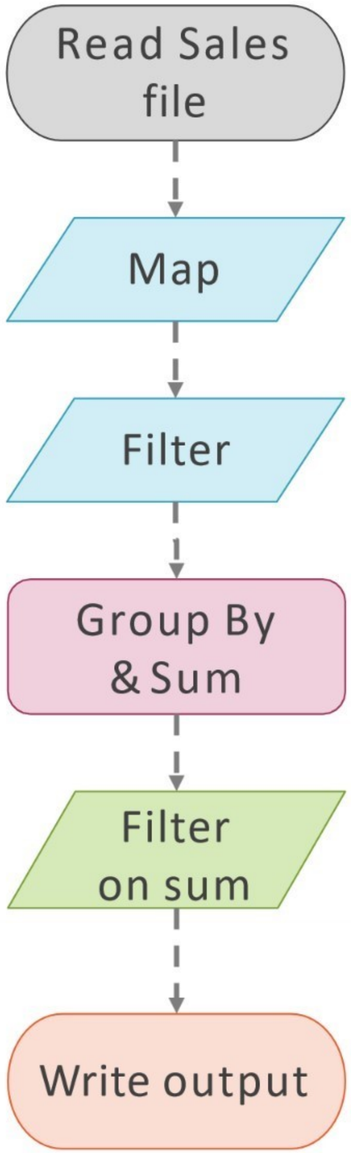


Narrow Transformation

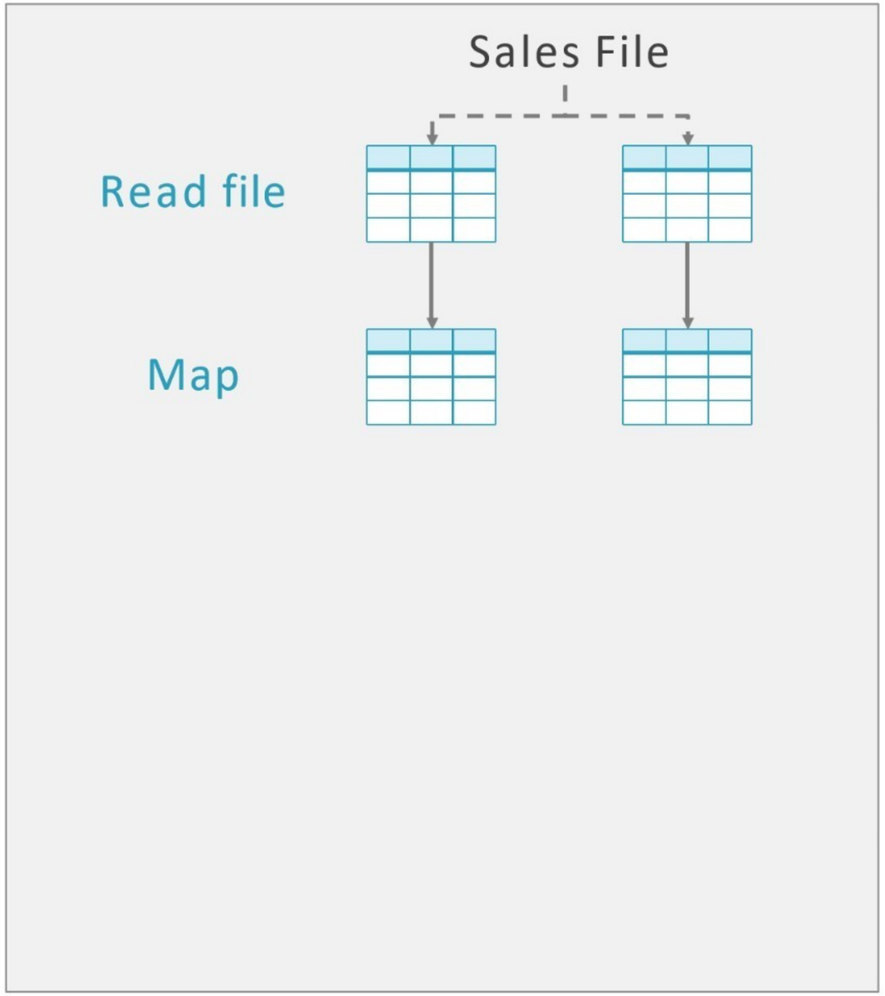
Number of output partitions are typically the same as input partitions



When Spark sees Wide Transformation, it splits the Job in Stages

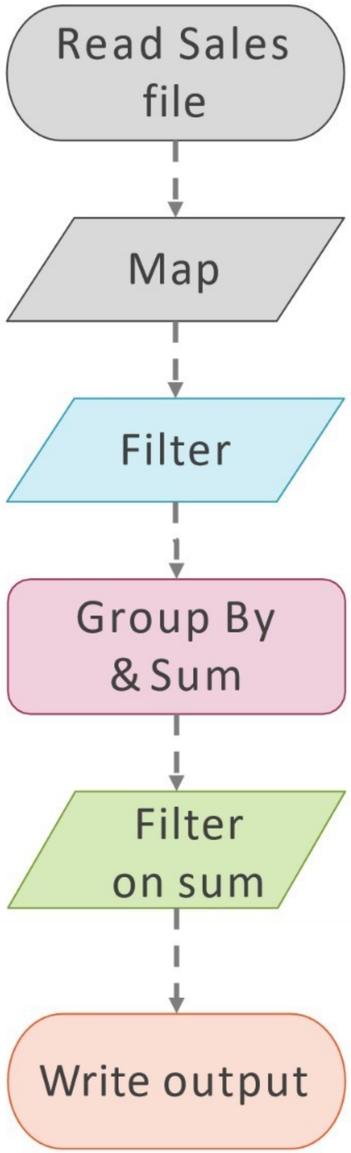


Job 2

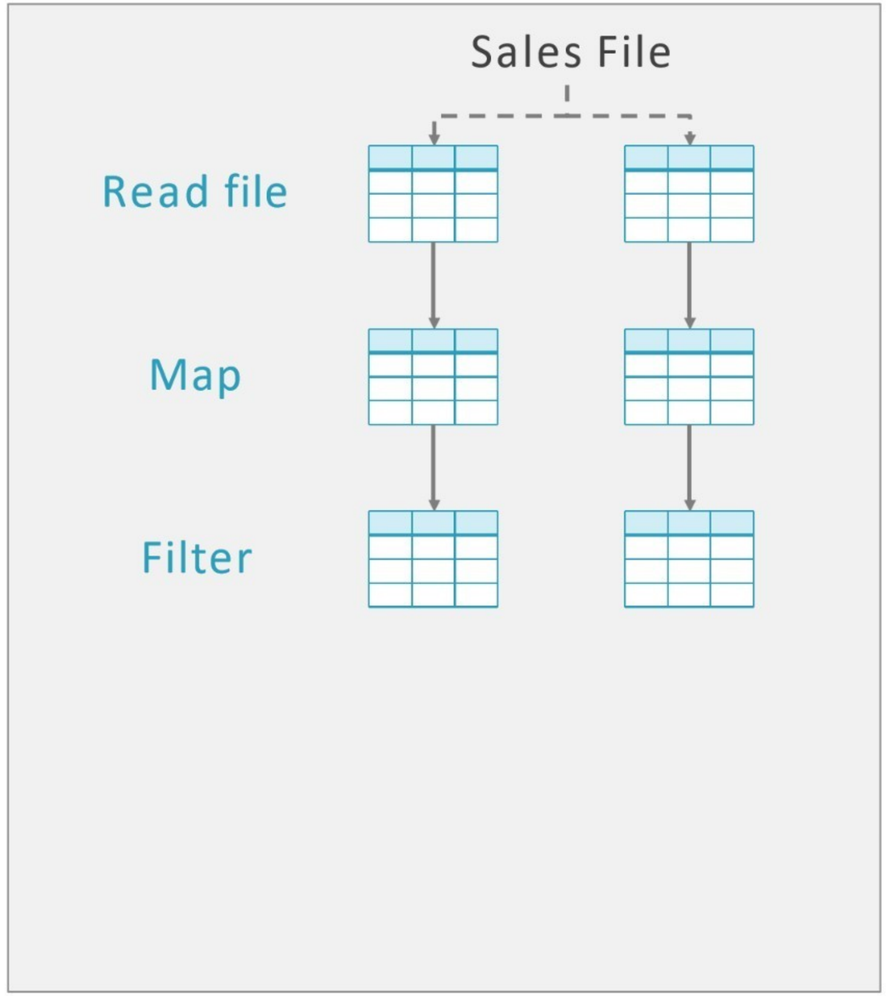


Stage 1

Stage 2

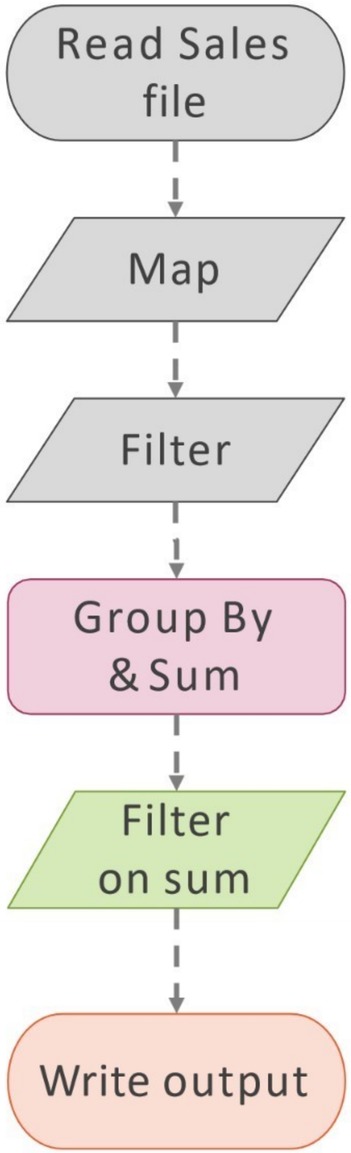


Job 2

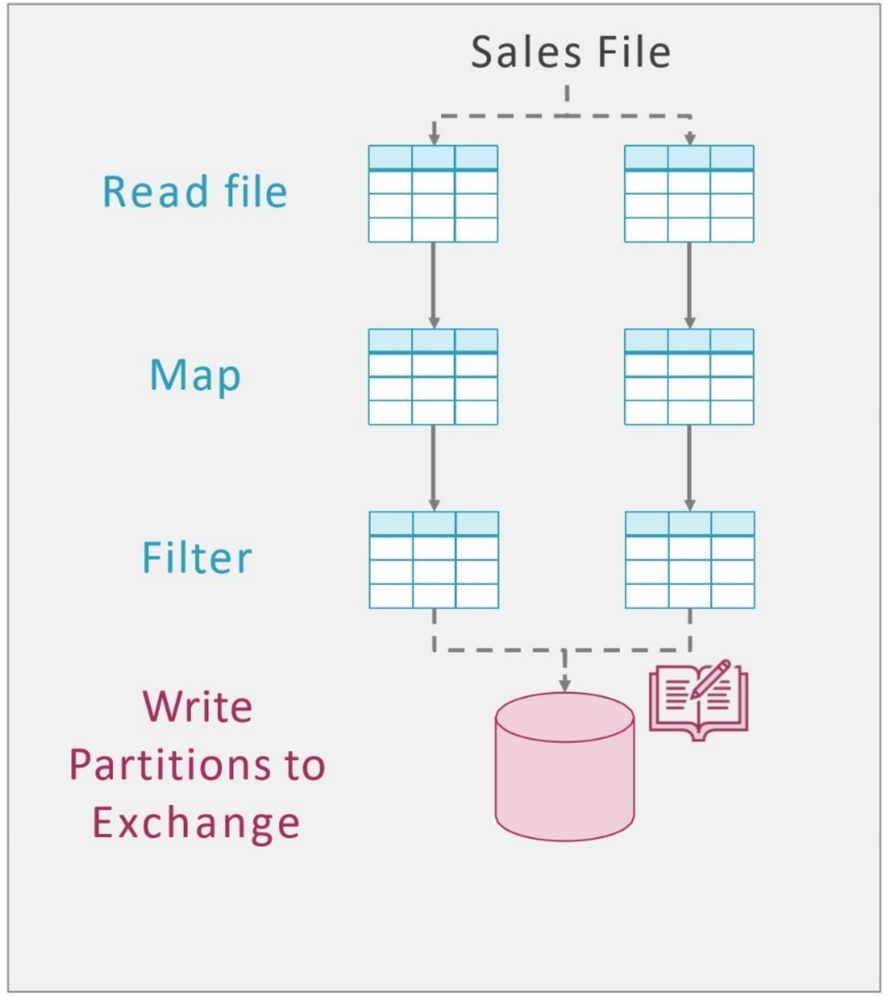


Stage 1

Stage 2

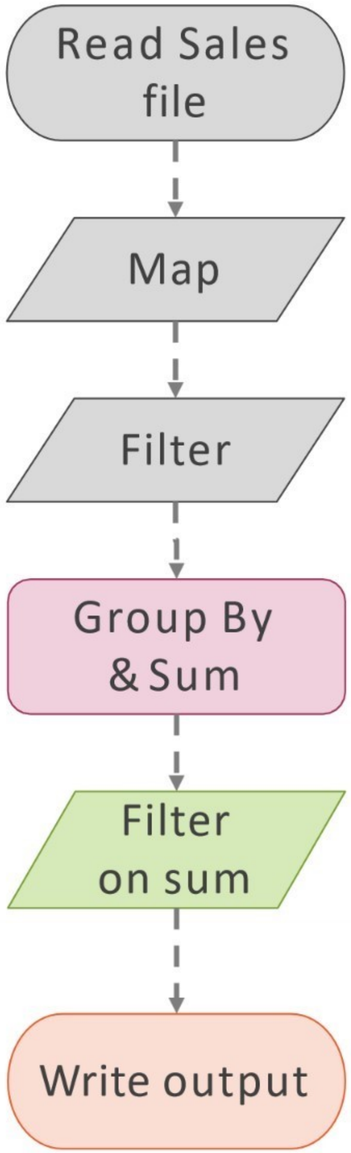


Job 2

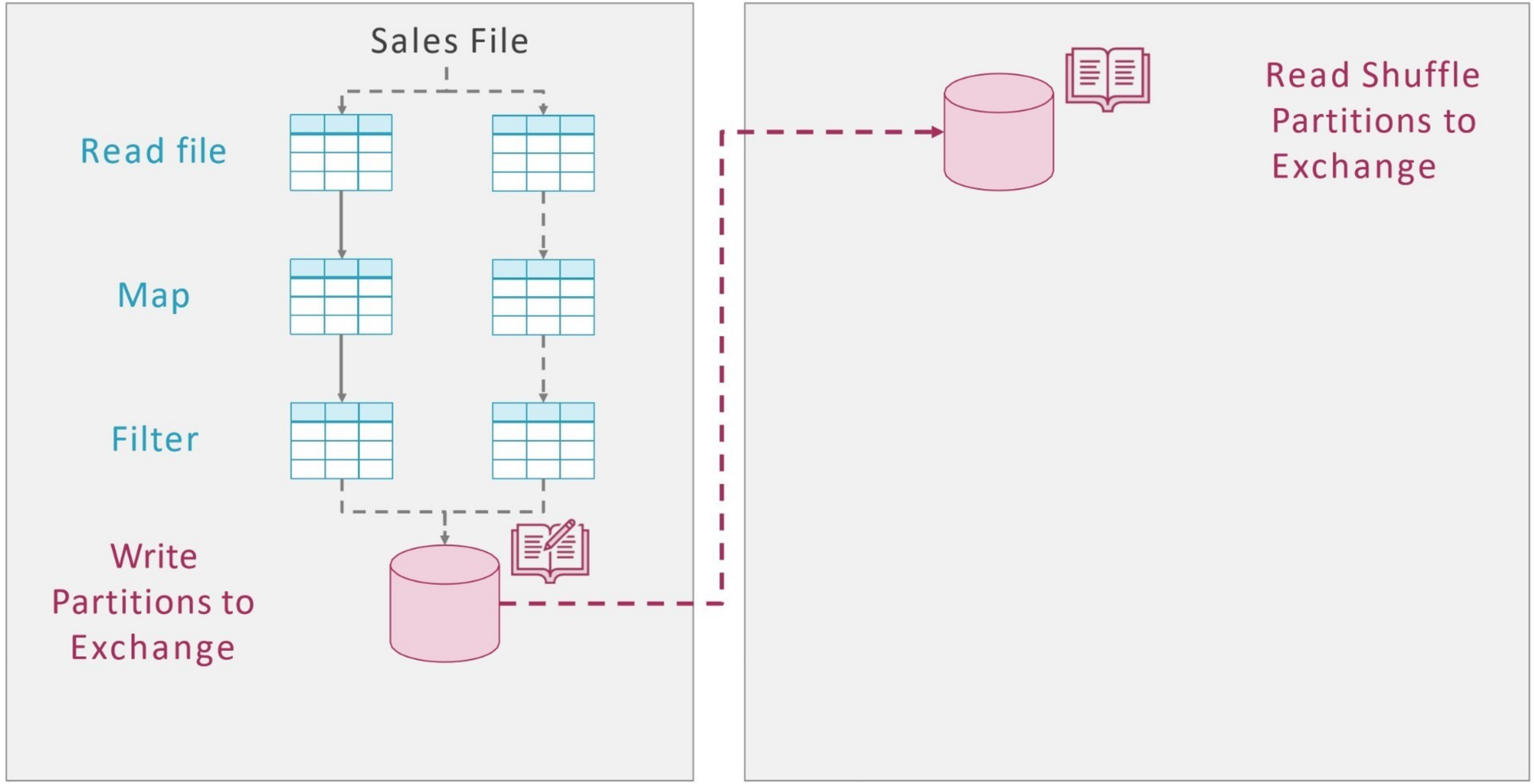


Stage 1

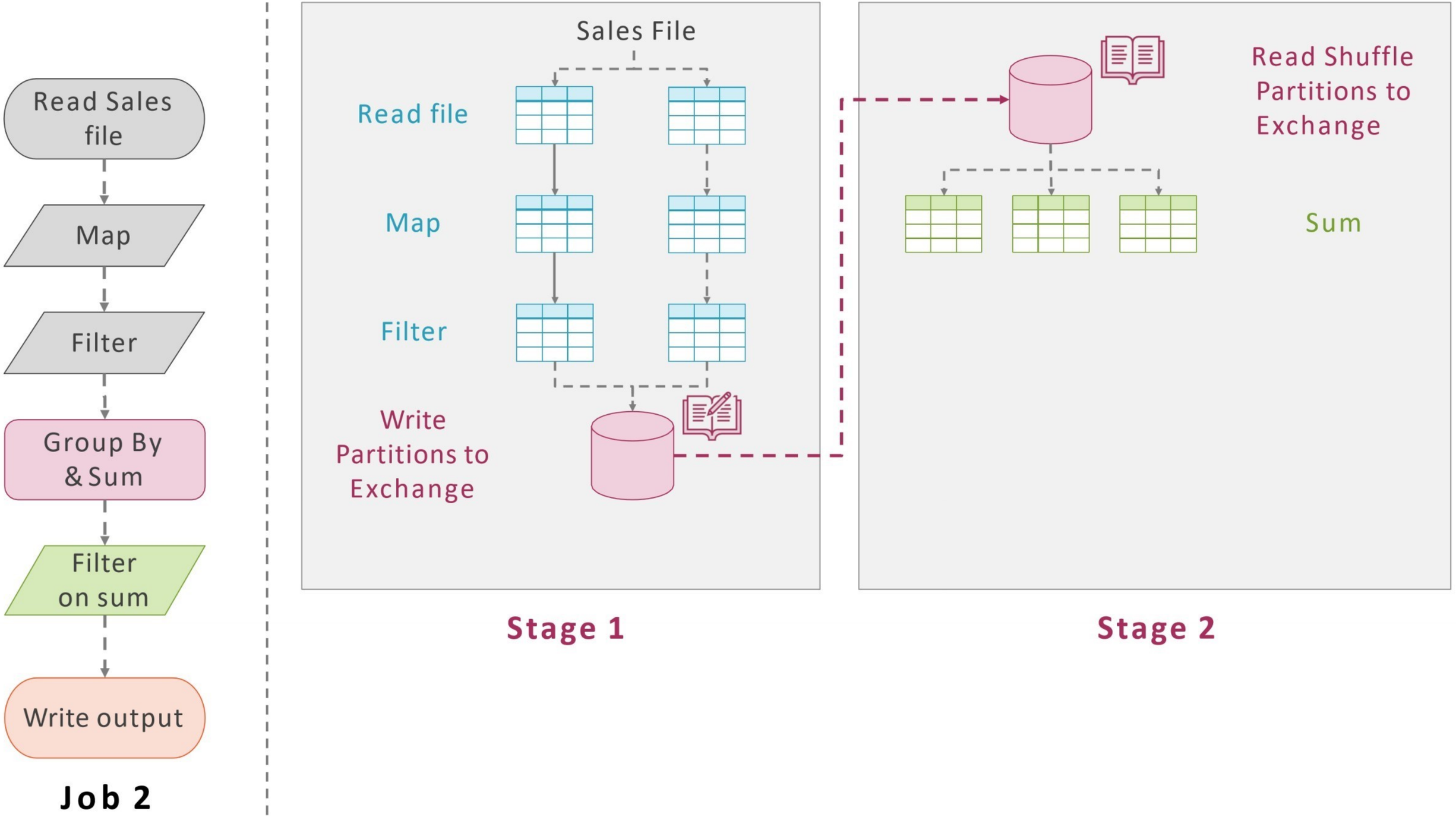
Stage 2

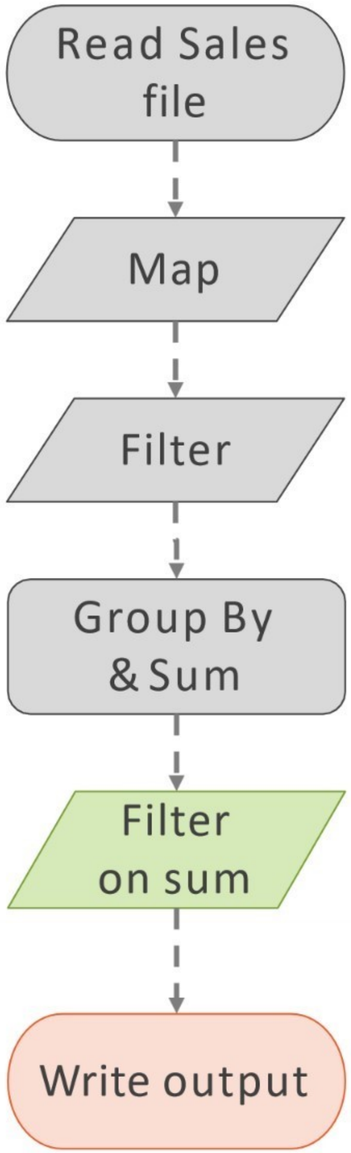


Job 2

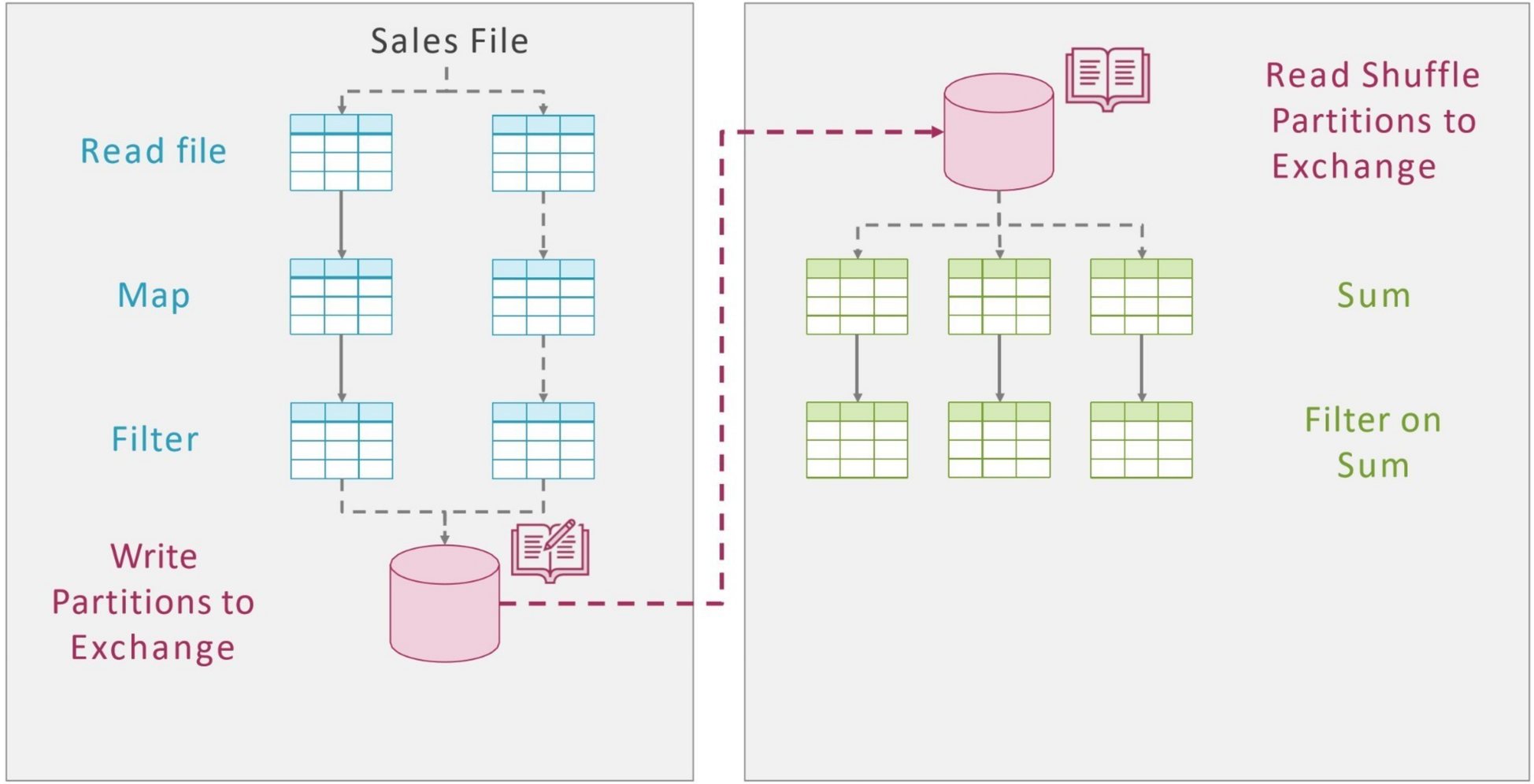


Stage 1 **Stage 2**

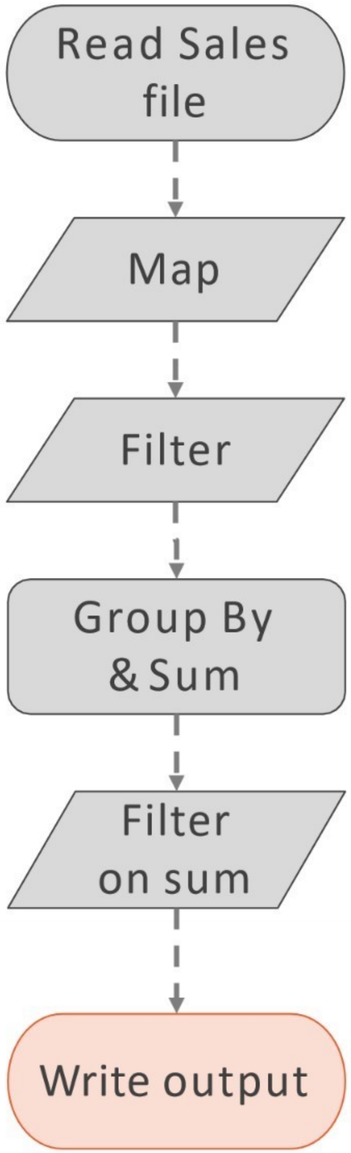

Read Sales file
Map
Filter
Group By & Sum
Filter on sum
Write output
Job 2
Sales File
Read file
Map
Filter
Write Partitions to Exchange
Stage 1
Read Shuffle Partitions to Exchange
Sum
Stage 2



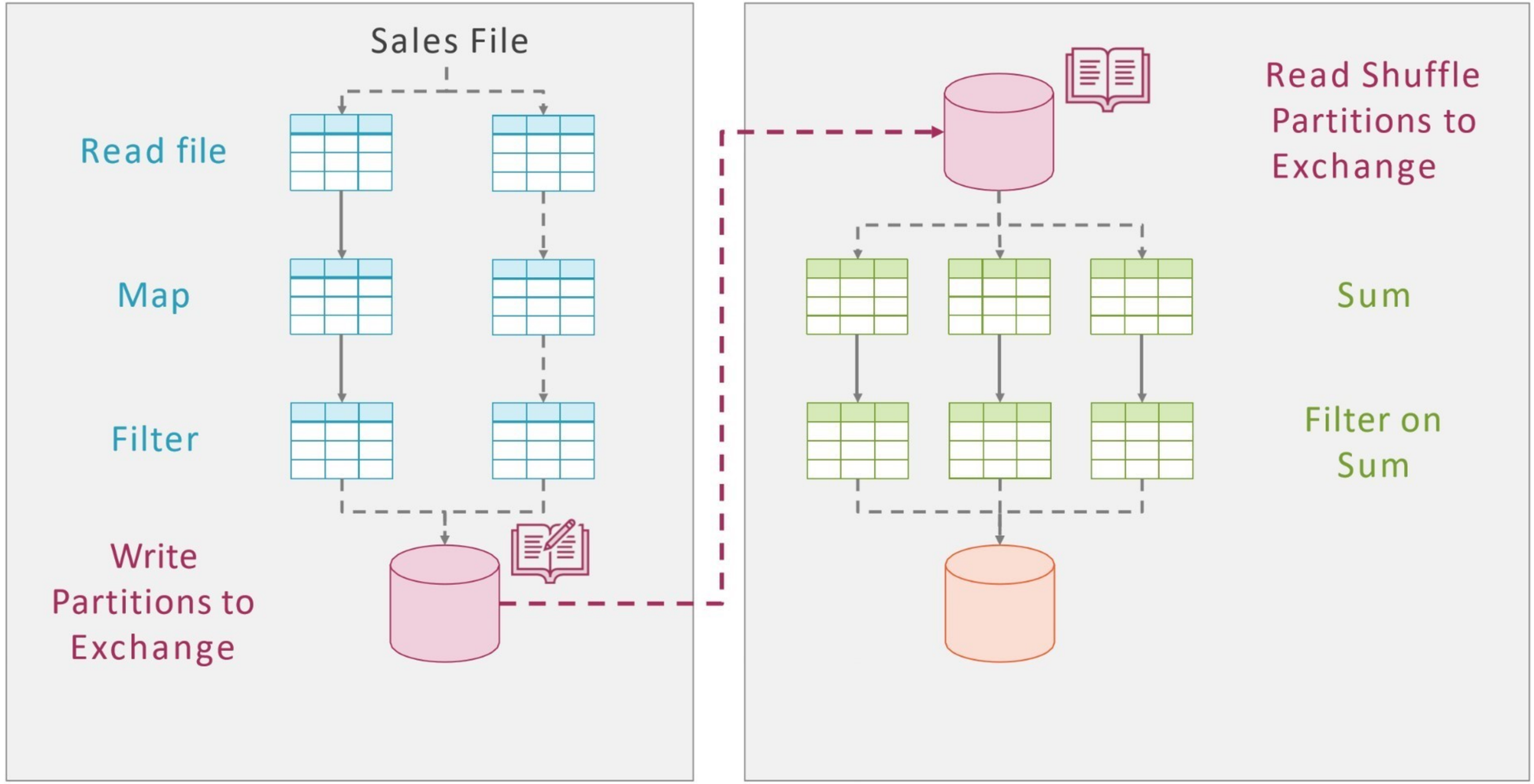
Job 2



Stage 1 **Stage 2**



Job 2



Stage 1 **Stage 2**

Spark Execution Components

Spark Application is a set of resources

- Contains driver and executor processes

Multiple jobs can run in an Application

- Number of jobs = Action operations applied

Each job is divided into Stages

- Number of stages = Wide transformations + 1

Stages are typically executed in sequence

- When one stage finishes, then only next can start
- Exceptions – Join where 2 datasets can be read parallelly as separate stages

Each Stage has its own set of Tasks

- Number of tasks = Number of partitions
- Number of parallel tasks = Number of cores

File Edit View Insert Cell Kernel Widgets Help

Run Markdown

In [1]:
```
import findspark
findspark.init()
findspark.find()
```

Out[1]:
```
'E:\\Local d\\Zartab\\Apache Spark Installation\\spark-3.3.2-bin-hadoop3'
```

In [2]:
```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = (
            SparkSession
                .builder
                .appName("SparkConceptsApp")
                .master("local[4]")
                .getOrCreate()
        )

sc = spark.sparkContext

spark
```

Out[2]:

SparkSession - in-memory

SparkContext

Spark UI

Identify Entities

1. Narrow Transformations
2. Wide Transformations
3. Actions
4. Jobs
5. Stages
6. Tasks
7. Partitions

1. Read CSV File

Identify transformations, actions, jobs, stages, partitions, tasks

In [5]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)

taxiZonesRdd.collect()
```

Out[5]:
```
['1,EWR,Newark Airport,EWR',
 '2,Queens,Jamaica Bay,Boro Zone',
 '3,Bronx,Allerton/Pelham Gardens,Boro Zone',
 '4,Manhattan,Alphabet City,Yellow Zone',
 '5,Staten Island,Arden Heights,Boro Zone',
 '6,Staten Island,Arrochar/Fort Wadsworth,Boro Zone',
 '7,Queens,Astoria,Boro Zone',
 '8,Queens,Astoria Park,Boro Zone',
 '9,Queens,Auburndale,Boro Zone',
 '10,Queens,Baisley Park,Boro Zone',
 '11,Brooklyn,Bath Beach,Boro Zone',
 '12,Manhattan,Battery Park,Yellow Zone',
 '13,Manhattan,Battery Park City,Yellow Zone',
 '14,Brooklyn,Bay Ridge,Boro Zone',
 '15,Queens,Bay Terrace/Fort Totten,Boro Zone'
```

1. Read CSV File

Identify transformations, actions, jobs, stages, partitions, tasks

In [5]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)

taxiZonesRdd.collect()
```

Out[5]:
```
['1,EWR,Newark Airport,EWR',
 '2,Queens,Jamaica Bay,Boro Zone',
 '3,Bronx,Allerton/Pelham Gardens,Boro Zone',
 '4,Manhattan,Alphabet City,Yellow Zone',
 '5,Staten Island,Arden Heights,Boro Zone',
 '6,Staten Island,Arrochar/Fort Wadsworth,Boro Zone',
 '7,Queens,Astoria,Boro Zone',
 '8,Queens,Astoria Park,Boro Zone',
 '9,Queens,Auburndale,Boro Zone',
 '10,Queens,Baisley Park,Boro Zone',
 '11,Brooklyn,Bath Beach,Boro Zone',
 '12,Manhattan,Battery Park,Yellow Zone',
 '13,Manhattan,Battery Park City,Yellow Zone',
 '14,Brooklyn,Bay Ridge,Boro Zone',
 '15,Queens,Bay Terrace/Fort Totten,Boro Zone'
```

Transformation: 1
Action : 1
Jobs: 1
Stages: 1

Spark 3.3.2 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | **SparkConceptsApp** application UI

Spark Jobs (?)

User: VICTUS
Total Uptime: 4.5 min
Scheduling Mode: FIFO
Completed Jobs: 1

▸ Event Timeline

▾ Completed Jobs (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1901593258.py:4 collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1901593258.py:4	2023/05/18 07:22:07	0.5 s	1/1	4/4

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

2. Additional step: Split by comma

Identify transformations, actions, jobs, stages, partitions, tasks

In [6]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


taxiZonesWithColsRdd.collect()
```

Out[6]:
```
[['1', 'EWR', 'Newark Airport', 'EWR'],
 ['2', 'Queens', 'Jamaica Bay', 'Boro Zone'],
 ['3', 'Bronx', 'Allerton/Pelham Gardens', 'Boro Zone'],
 ['4', 'Manhattan', 'Alphabet City', 'Yellow Zone'],
 ['5', 'Staten Island', 'Arden Heights', 'Boro Zone'],
 ['6', 'Staten Island', 'Arrochar/Fort Wadsworth', 'Boro Zone'],
 ['7', 'Queens', 'Astoria', 'Boro Zone'],
 ['8', 'Queens', 'Astoria Park', 'Boro Zone'],
 ['9', 'Queens', 'Auburndale', 'Boro Zone'],
 ['10', 'Queens', 'Baisley Park', 'Boro Zone'],
 ['11', 'Brooklyn', 'Bath Beach', 'Boro Zone'],
```

2. Additional step: Split by comma

Identify transformations, actions, jobs, stages, partitions, tasks

In [6]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


taxiZonesWithColsRdd.collect()
```

Out[6]:
```
[['1', 'EWR', 'Newark Airport', 'EWR'],
 ['2', 'Queens', 'Jamaica Bay', 'Boro Zone'],
 ['3', 'Bronx', 'Allerton/Pelham Gardens', 'Boro Zone'],
 ['4', 'Manhattan', 'Alphabet City', 'Yellow Zone'],
 ['5', 'Staten Island', 'Arden Heights', 'Boro Zone'],
 ['6', 'Staten Island', 'Arrochar/Fort Wadsworth', 'Boro Zone'],
 ['7', 'Queens', 'Astoria', 'Boro Zone'],
 ['8', 'Queens', 'Astoria Park', 'Boro Zone'],
 ['9', 'Queens', 'Auburndale', 'Boro Zone'],
 ['10', 'Queens', 'Baisley Park', 'Boro Zone'],
 ['11', 'Brooklyn', 'Bath Beach', 'Boro Zone'],
```

Action : 1
Jobs: 1
Transformation: (2 Narrow)
Stages: 1

Spark 3.3.2 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | **SparkConceptsApp** application UI

Spark Jobs (?)

User: VICTUS
Total Uptime: 8.2 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

▾ Completed Jobs (2)

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1379414820.py:9 collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1379414820.py:9	2023/05/18 07:26:02	2 s	1/1	4/4
0	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1901593258.py:4 collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\1901593258.py:4	2023/05/18 07:22:07	0.5 s	1/1	4/4

In [7]:
```
# Check partitions after reading file & after map operation

print ("After reading file = " + str( taxiZonesRdd.getNumPartitions() ))

print ("After applying map = " + str( taxiZonesWithColsRdd.getNumPartitions() ))
```

```
After reading file = 4
After applying map = 4
```

3. Additional step: Create Pair RDD

Identify transformations, actions, jobs, stages, partitions, tasks

In [8]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


# Map Operation - Create Pair RDD with <Borough, 1>
taxiZonesPairRdd = taxiZonesWithColsRdd.map( lambda zoneRow: (zoneRow[1], 1) )


taxiZonesPairRdd.count()
```

Out[8]: 265

3. Additional step: Create Pair RDD

Identify transformations, actions, jobs, stages, partitions, tasks

In [8]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


# Map Operation - Create Pair RDD with <Borough, 1>
taxiZonesPairRdd = taxiZonesWithColsRdd.map( lambda zoneRow: (zoneRow[1], 1) )


taxiZonesPairRdd.count()
```

Out[8]: 265

Action : ?
Jobs: ?
Transformation: ?
Stages: ?

Spark Jobs (?)

User: VICTUS
Total Uptime: 11 min
Scheduling Mode: FIFO
Completed Jobs: 3

▸ Event Timeline

▾ Completed Jobs (3)

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\2789000522.py:13 count at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\2789000522.py:13	2023/05/18 07:30:02	2 s	1/1	4/4

4. Additional step: Find distinct records

Identify transformations, actions, jobs, stages, partitions, tasks

In [9]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


# Map Operation - Create Pair RDD with <Borough, 1>
taxiZonesPairRdd = taxiZonesWithColsRdd.map( lambda zoneRow: (zoneRow[1], 1) )


# Distinct Operation - Find distinct items
distinctZonesRdd = taxiZonesPairRdd.distinct()


distinctZonesRdd.collect()
```

Out[9]:
```
[('Bronx', 1),
 ('Staten Island', 1),
```

4. Additional step: Find distinct records

Identify transformations, actions, jobs, stages, partitions, tasks

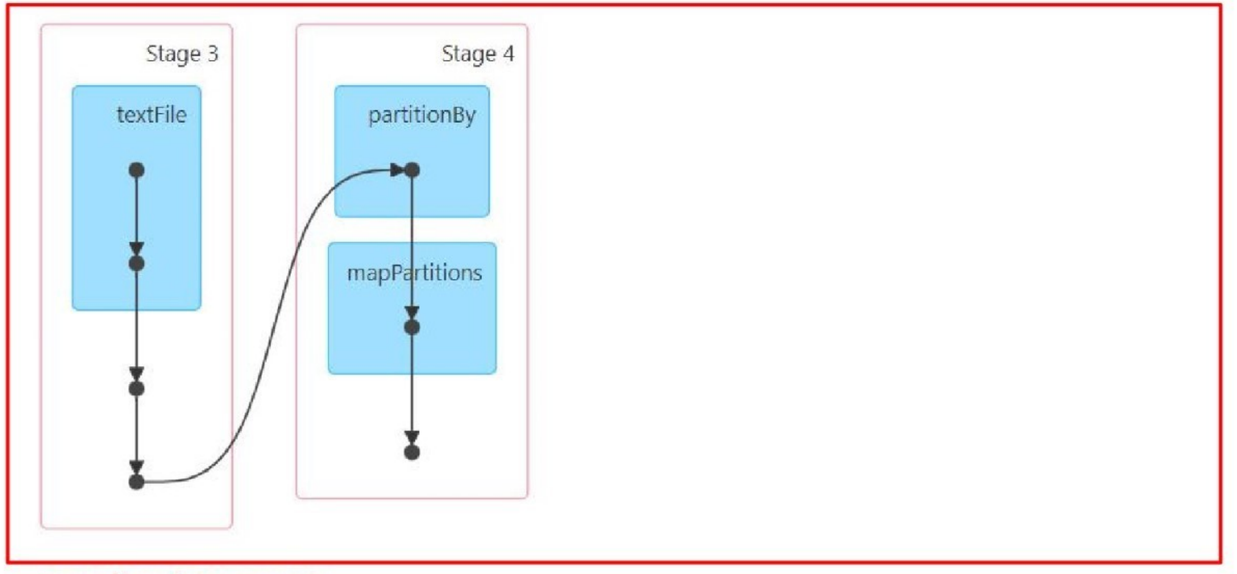
In [9]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


# Map Operation - Create Pair RDD with <Borough, 1>
taxiZonesPairRdd = taxiZonesWithColsRdd.map( lambda zoneRow: (zoneRow[1], 1) )


# Distinct Operation - Find distinct items
distinctZonesRdd = taxiZonesPairRdd.distinct()


distinctZonesRdd.collect()
```

Out[9]:
```
[('Bronx', 1),
 ('Staten Island', 1),
```

Action : 2
Jobs: 2
Transformation:
3 narrow
1 wide
Stages: 2

Submitted: 2023/05/18 07:33:09
Duration: 4 s
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization



▾ Completed Stages (2)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\3401495829.py:17	+details	2023/05/18 07:33:11	2 s	4/4			1123.0 B	
3	distinct at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\3401495829.py:14	+details	2023/05/18 07:33:09	2 s	4/4	26.1 KiB			1123.0 B

5. Additional steps:

a) Group by Borough

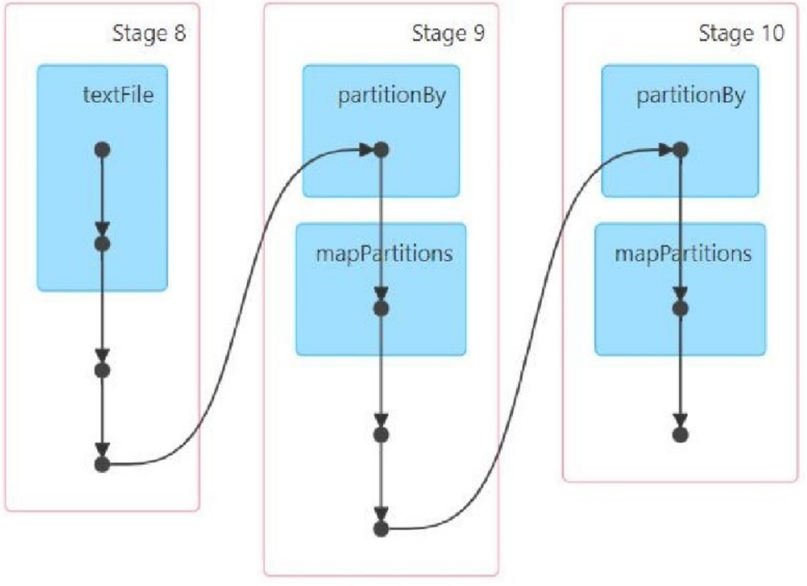
b.) Get rows with Borough count > 10

Identify transformations, actions, jobs, stages, partitions, tasks

In [11]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


# Map Operation - Split data by comma
taxiZonesWithColsRdd = taxiZonesRdd.map( lambda zone: zone.split(",") )


# Map Operation - Create Pair RDD with <Borough, 1>
taxiZonesPairRdd = taxiZonesWithColsRdd.map( lambda zoneRow: (zoneRow[1], 1) )


# Distinct Operation - Find distinct items
distinctZonesRdd = taxiZonesPairRdd.distinct()


# ReduceByKey Operation - Group by Borough and calculate count
boroughCountRdd = distinctZonesRdd.reduceByKey( lambda value1, value2: value1 + value2 )


# Filter Operation - Get rows with Borough count > 10
filteredZonesRdd = boroughCountRdd.filter( lambda row: row[1] > 10 )


filteredZonesRdd.collect()
```

Completed Stages: 3

▸ Event Timeline
▾ DAG Visualization



▾ Completed Stages (3)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
10	collect at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\3211385598.py:25	+details	2023/05/18 07:39:47	2 s	4/4			279.0 B	
9	reduceByKey at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\3211385598.py:18	+details	2023/05/18 07:39:45	2 s	4/4			1123.0 B	279.0 B
8	distinct at C:\Users\VICTUS\AppData\Local\Temp\ipykernel_13836\3211385598.py:14	+details	2023/05/18 07:39:43	2 s	4/4	26.1 KiB			1123.0 B

Run first and take methods

Identify transformations, actions, jobs, stages, partitions, tasks

In [12]:
```
# Read File - TaxiZones.csv
taxiZonesRdd = sc.textFile("E:\SparkData\Raw\TaxiZones.csv", 4)


print ( taxiZonesPairRdd.first() )

print ( taxiZonesPairRdd.take(2) )
```

```
('EWR', 1)
[('EWR', 1), ('Queens', 1)]
```

Spark 3.3.2 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | **SparkConceptsApp** application UI

Spark Jobs (?)

User: VICTUS
Total Uptime: 24 min
Scheduling Mode: FIFO
Completed Jobs: 10

▸ Event Timeline

▾ Completed Jobs (10)

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	runJob at PythonRDD.scala:166 runJob at PythonRDD.scala:166	2023/05/18 07:43:53	0.5 s	1/1	1/1
8	runJob at PythonRDD.scala:166 runJob at PythonRDD.scala:166	2023/05/18 07:43:52	0.5 s	1/1	1/1

Summary

RDD is the native data structure of Spark

- In-memory, Partitioned, Read-only & Resilient

RDD can be created in many ways

- Parallelize a collection, read a file or convert an RDD

Pair RDD is an RDD with Key-Value pairs

Apply transformation and action operations on RDDs

- Transformation gives new RDD. Action gives result
- Transformations are lazy operations

Transformations are of two types

- Narrow – Each input partition is used at-most once
- Wide – An input partition may be used multiple times

Data shuffling is a two-step process

Application → Jobs → Stages → Tasks