Working with RDDs – Resilient Distributed Datasets

Mohit Batra
Founder, Crystal Talks

linkedin.com/in/mohitbatra

Overview

Understand RDDs

Create RDDs

Work with Pair RDDs

Apply operations on RDDs

Use narrow transformations

Use wide transformations and data shuffling

Spark application concepts

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

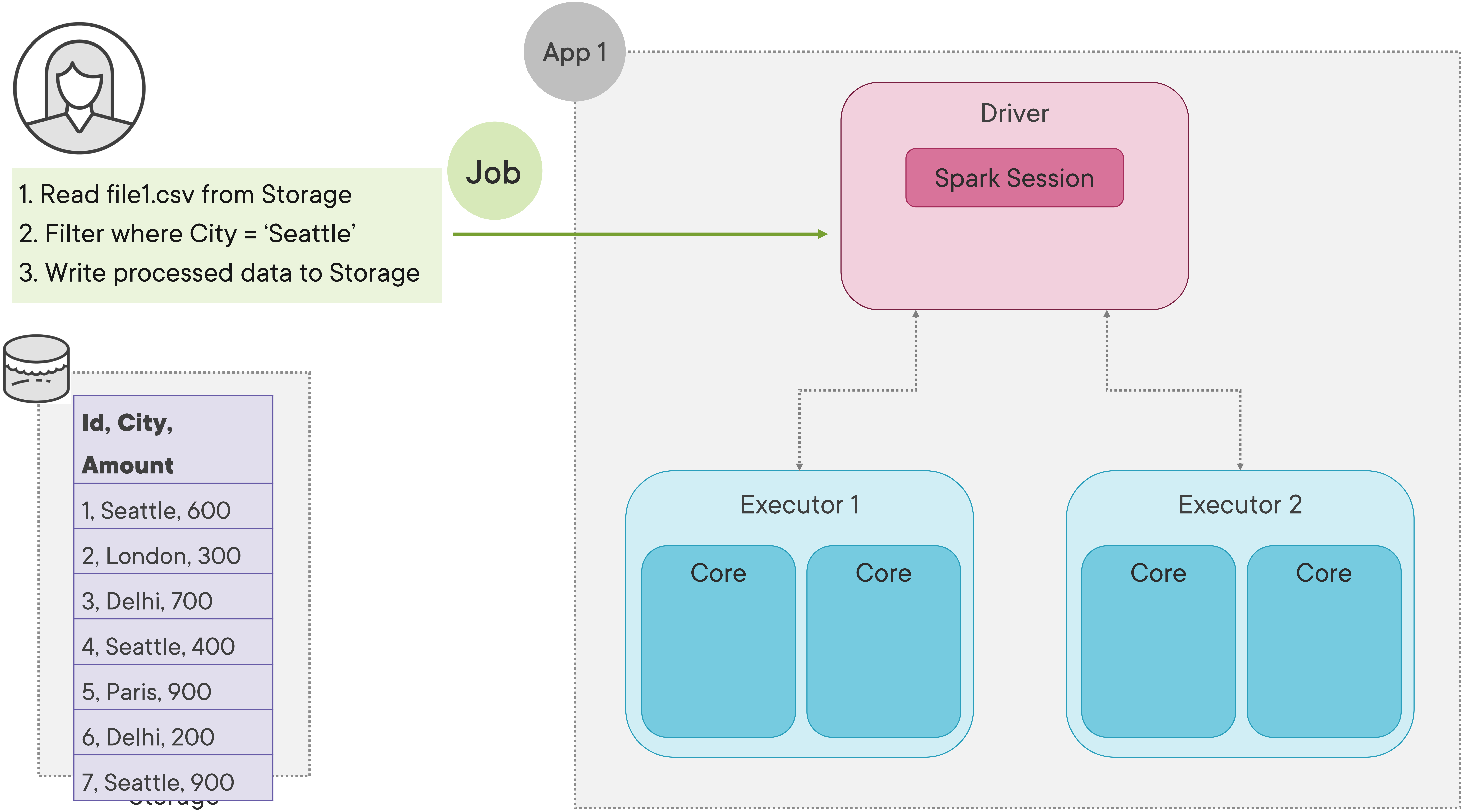
RDD in Spark

In-memory objects

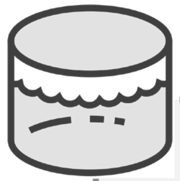
Do not have schema

Distributed collection of elements

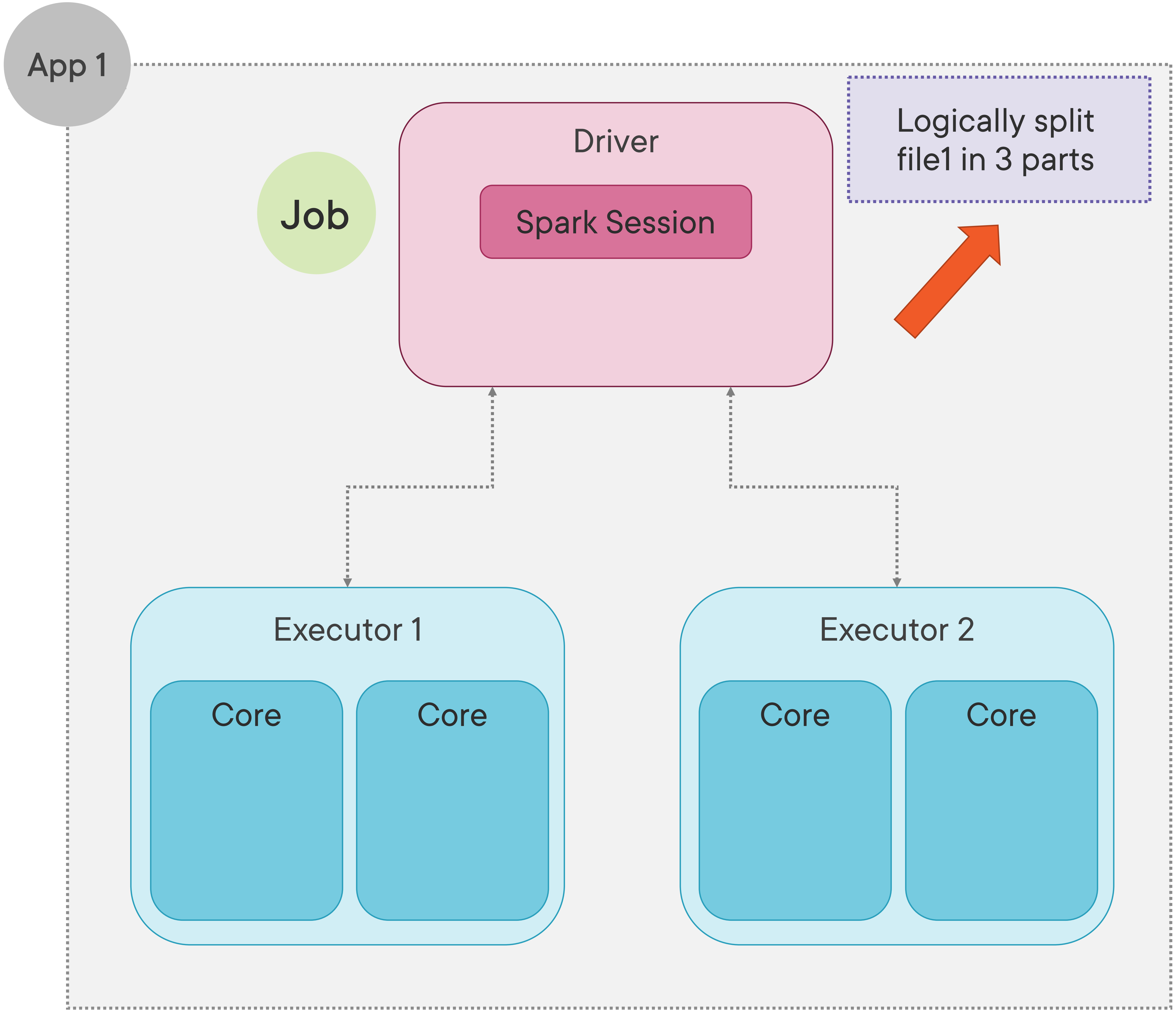
All processing in Spark happens on RDDs


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

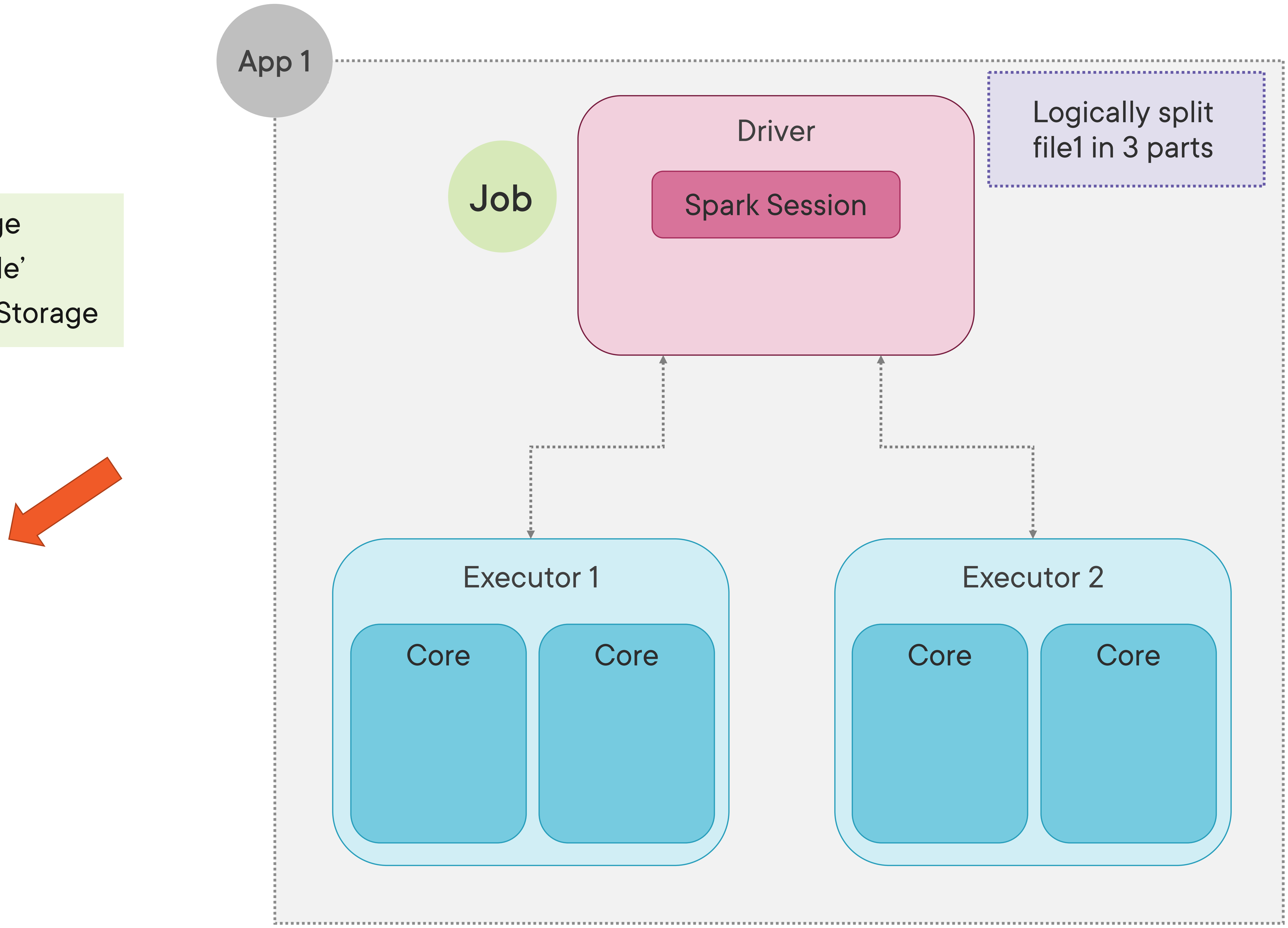
1. Read file1.csv from Storage
2. Filter where City = ‘Seattle’
3. Write processed data to Storage

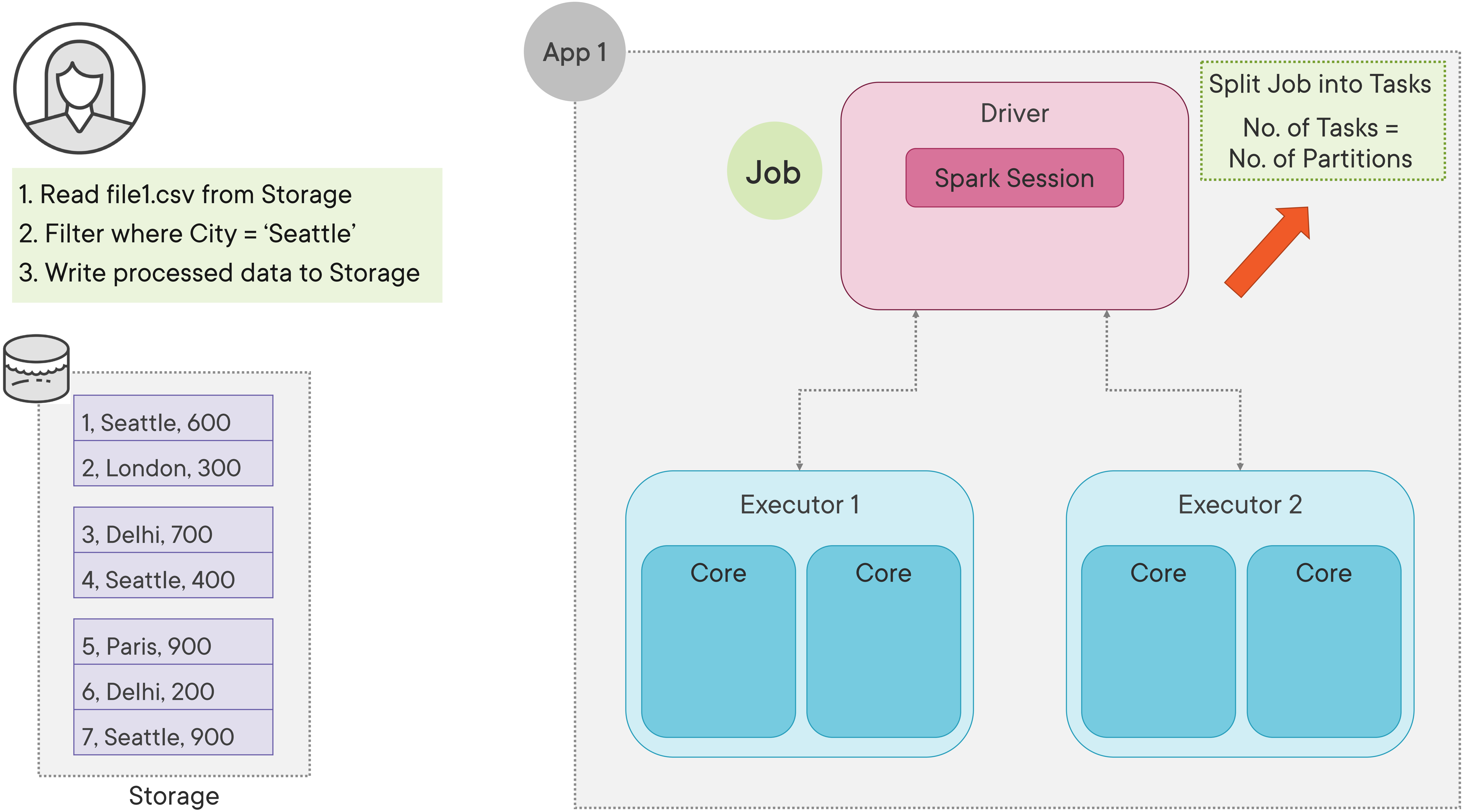


Id, City, Amount
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900

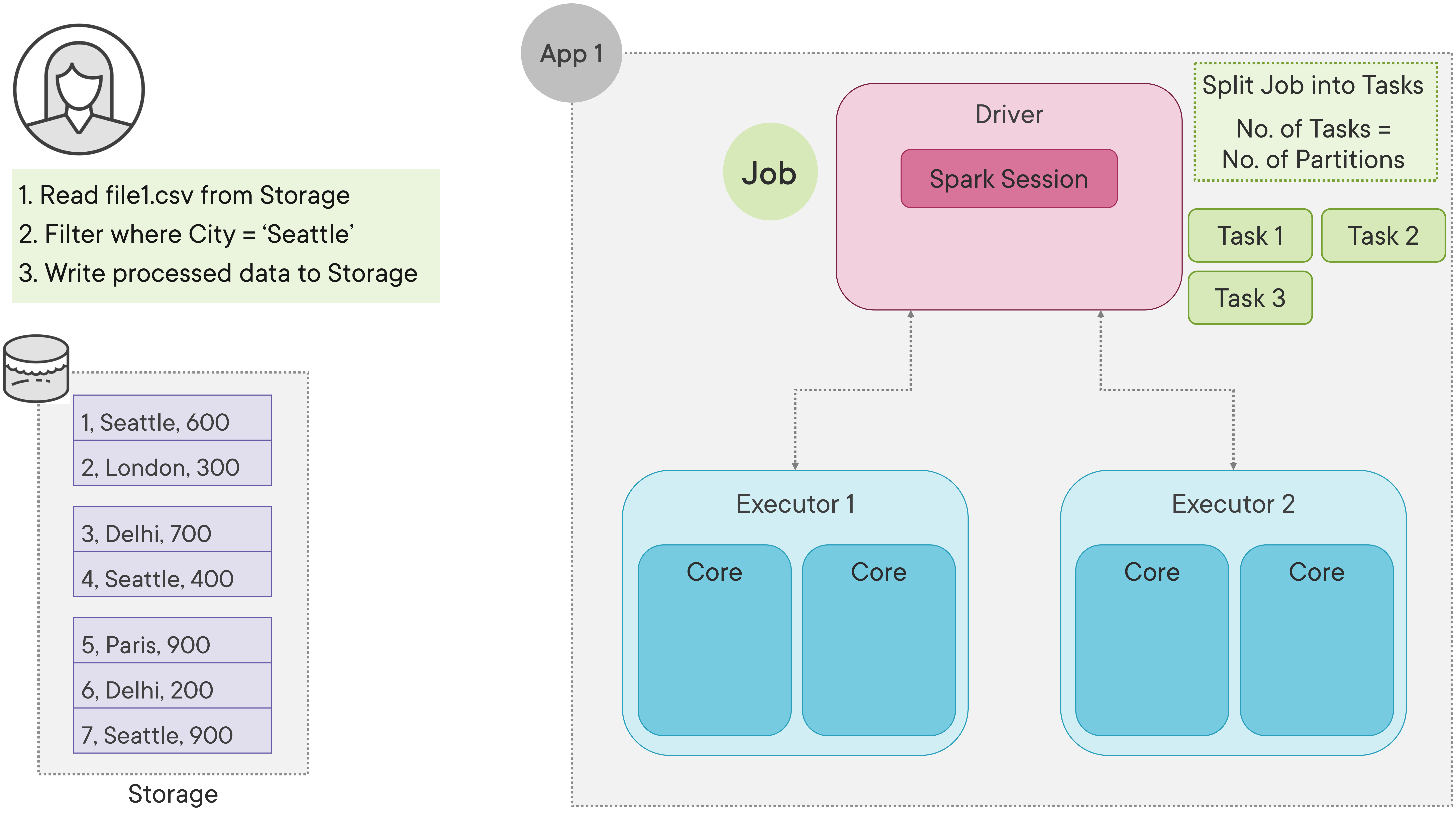


1. Read file1.csv from Storage
2. Filter where City = ‘Seattle’
3. Write processed data to Storage

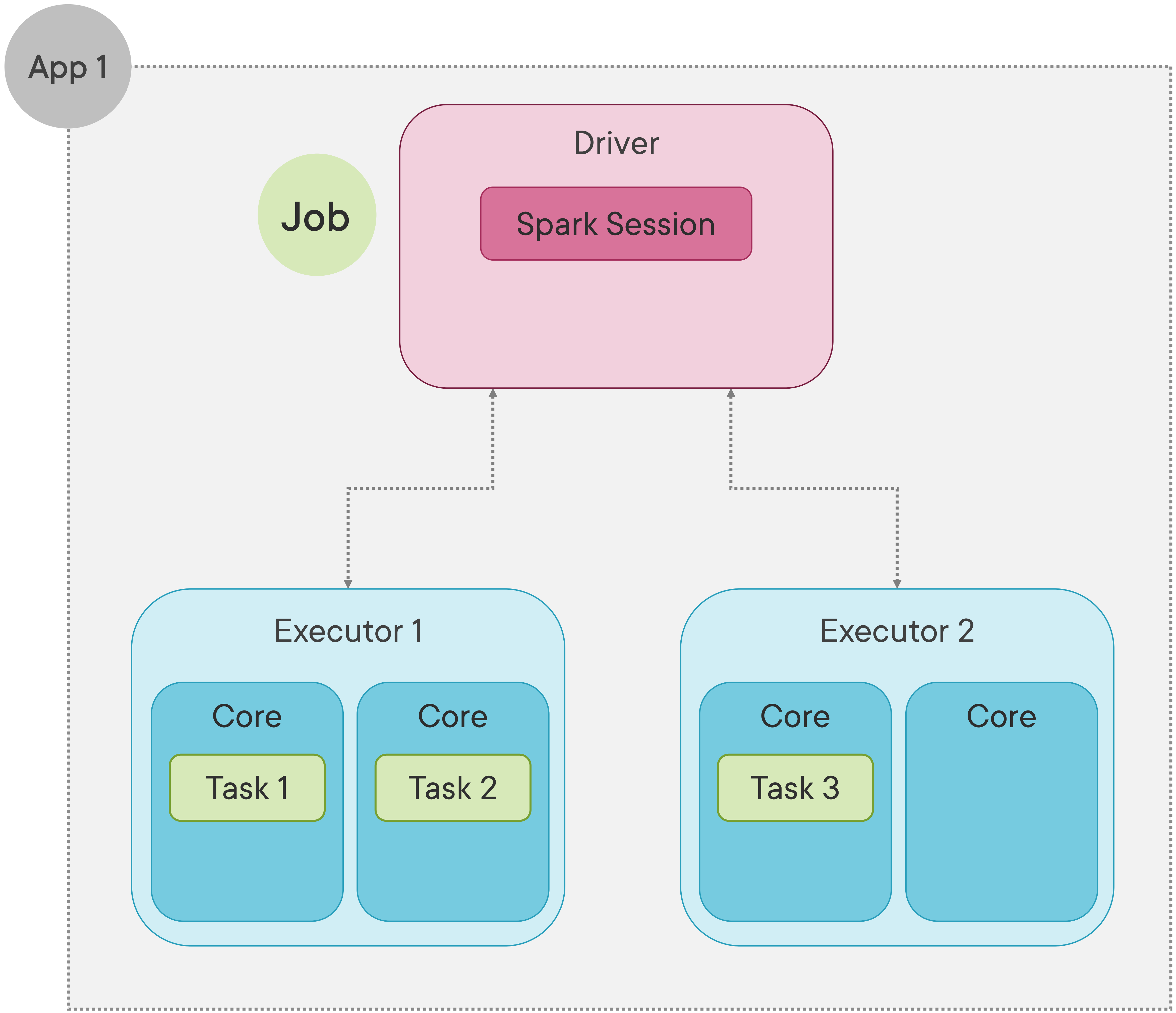


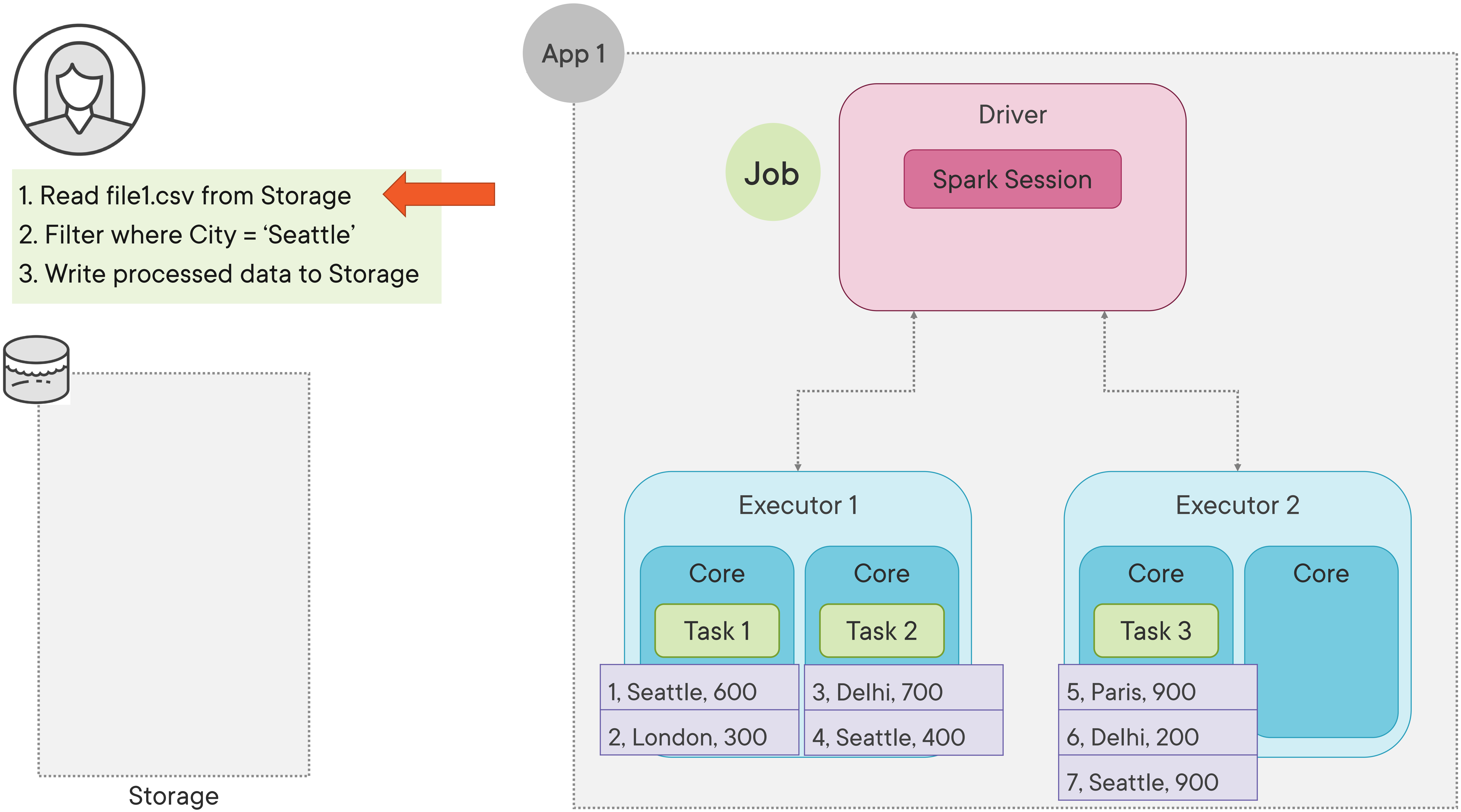


1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Storage
App 1
Job
Driver
Spark Session
Split Job into Tasks
No. of Tasks =
No. of Partitions
Executor 1
Core
Core
Executor 2
Core
Core

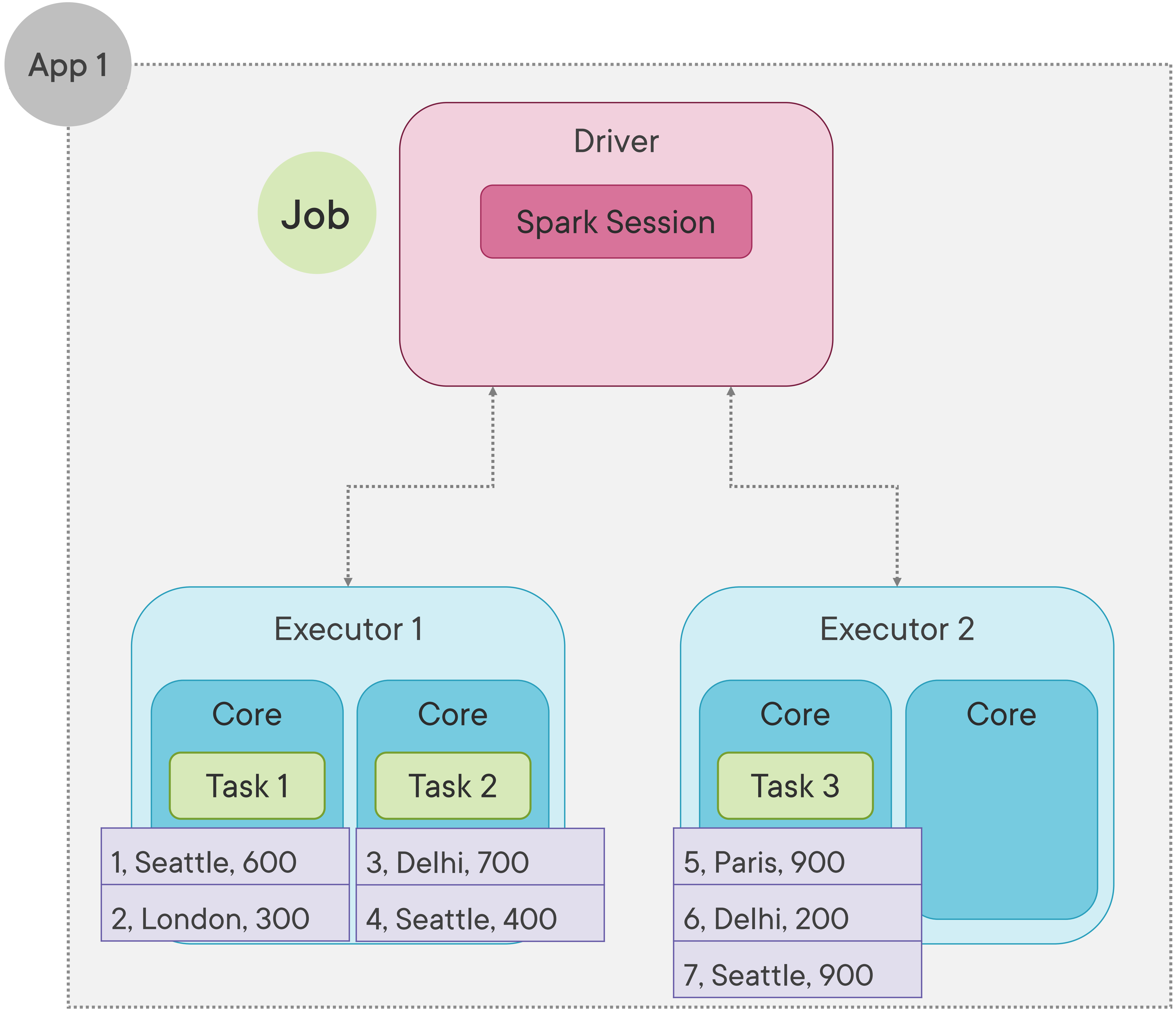

1. Read file1.csv from Storage
2. Filter where City = ‘Seattle’
3. Write processed data to Storage
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Storage
App 1
Job
Driver
Spark Session
Split Job into Tasks
No. of Tasks =
No. of Partitions
Task 1
Task 2
Task 3
Executor 1
Core
Core
Executor 2
Core
Core

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

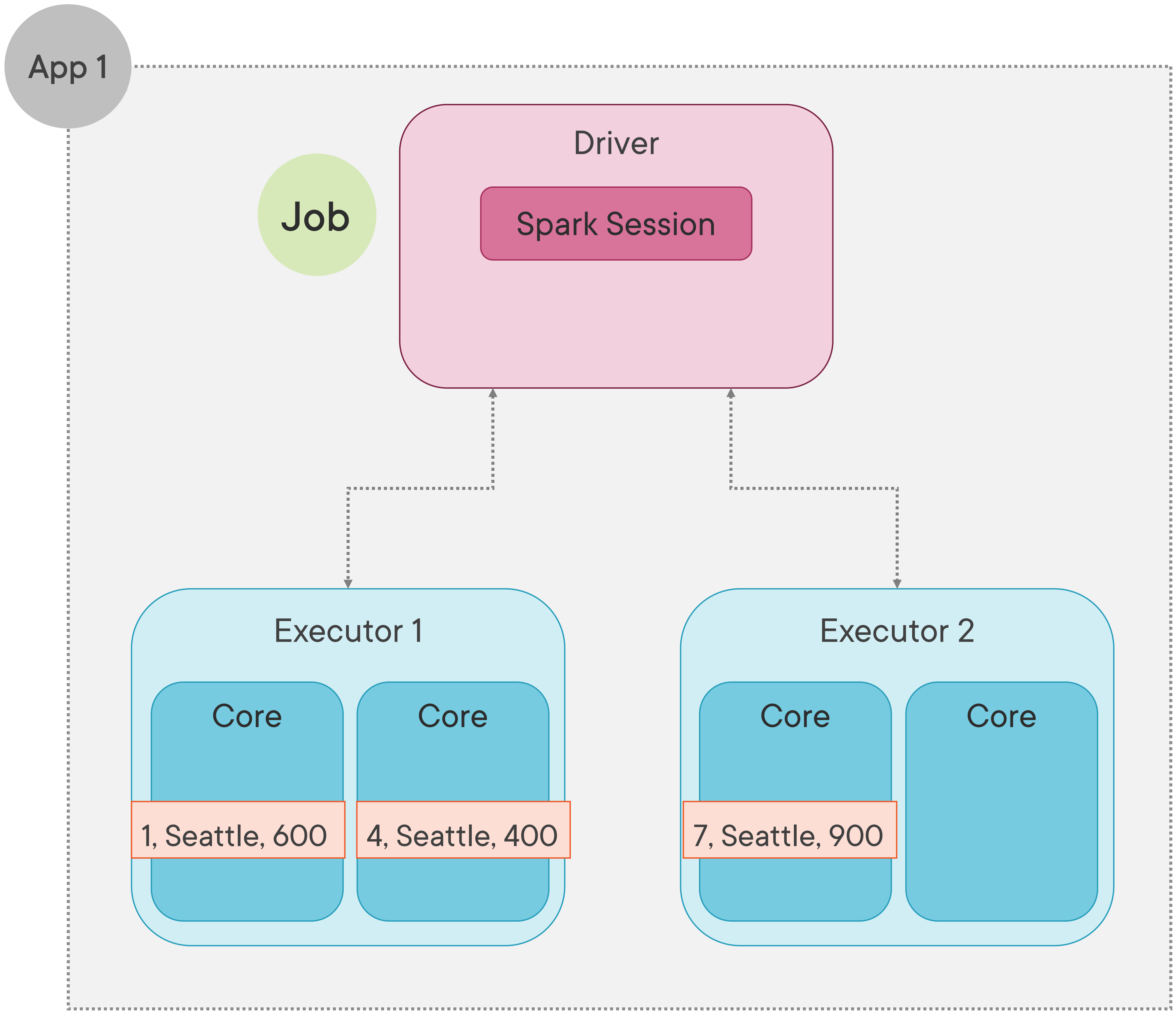


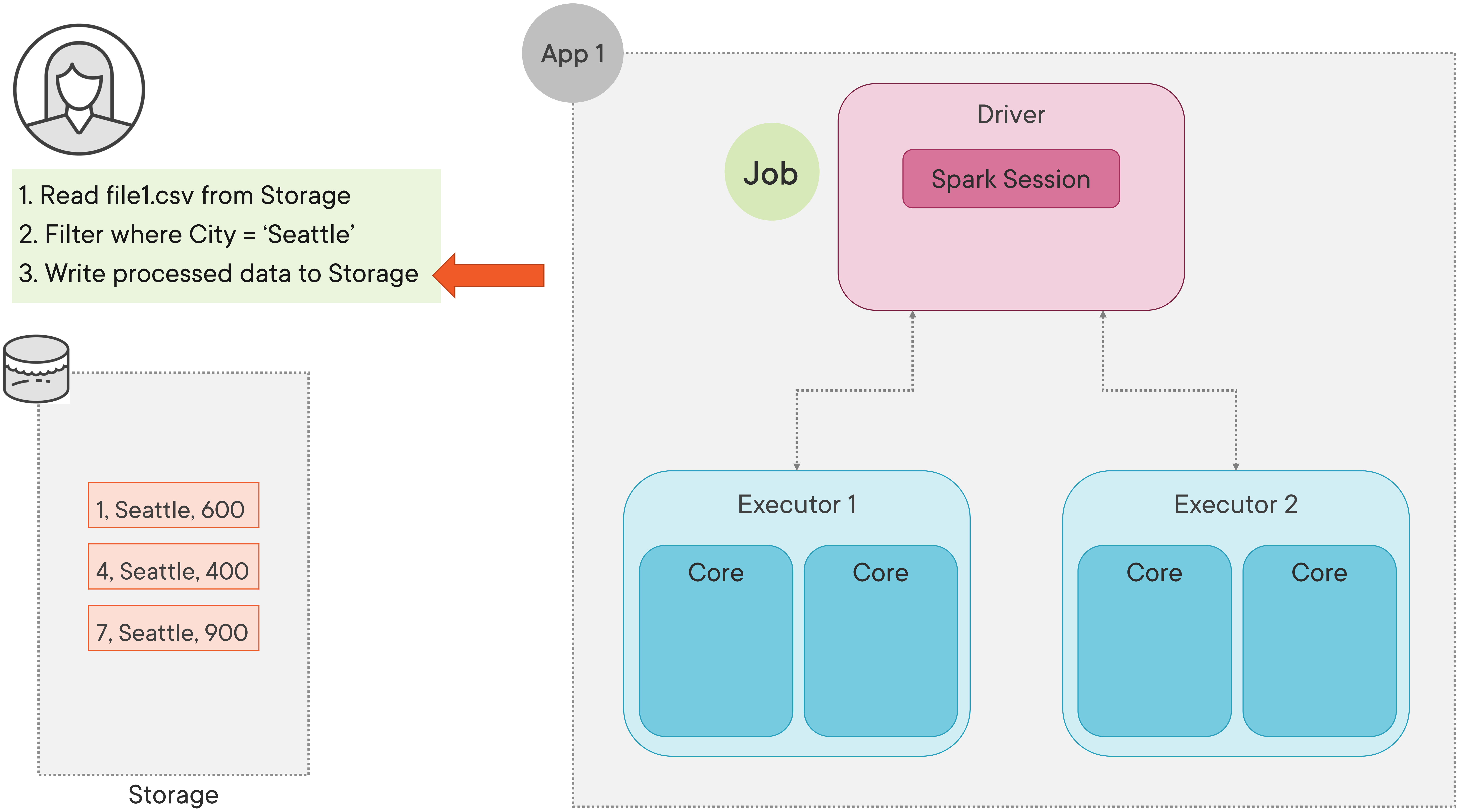

1. Read file1.csv from Storage
2. Filter where City = 'Seattle'
3. Write processed data to Storage
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Task 1
Core
Task 2
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
Executor 2
Core
Task 3
Core
5, Paris, 900
6, Delhi, 200
7, Seattle,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

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



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
4, Seattle, 400
7, Seattle, 900
Storage
App 1
Job
Driver
Spark Session
Executor 1
Core
Core
Executor 2
Core
Core

Features of RDDs

In-memory

Resides in the memory of cluster

Partitioned

Split into partitions, processed by tasks

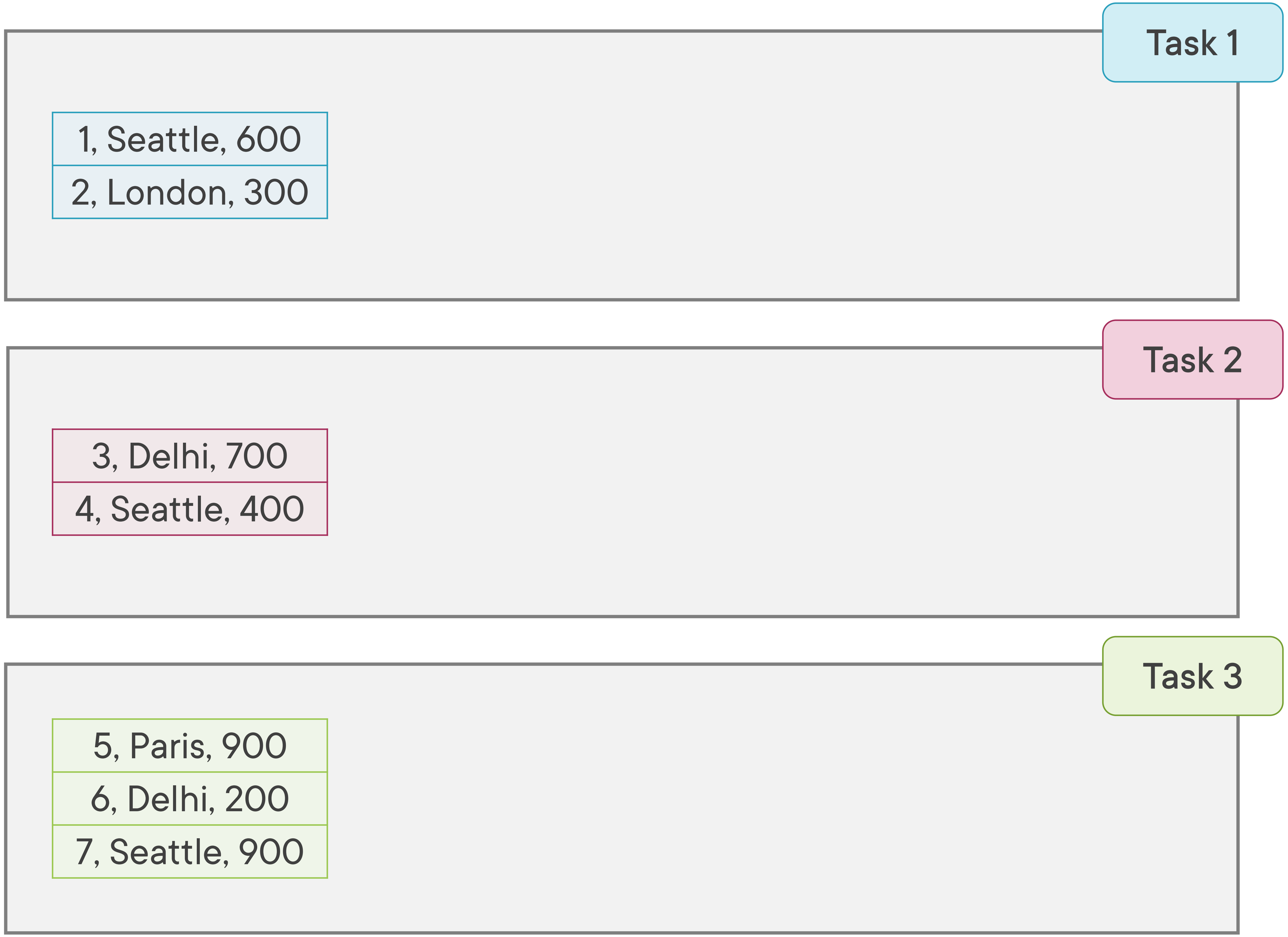
Read File

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

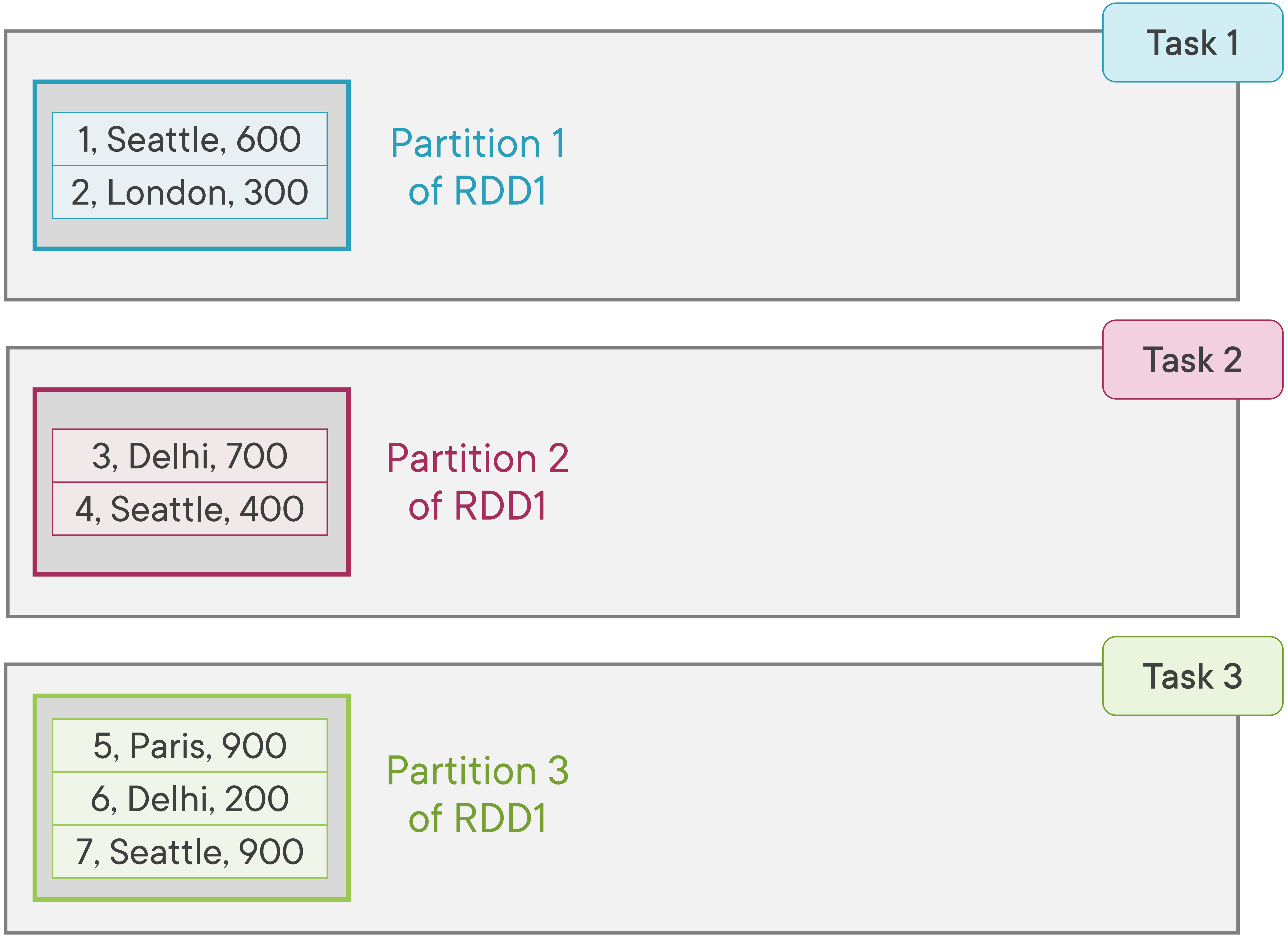
Read File

RDD 1

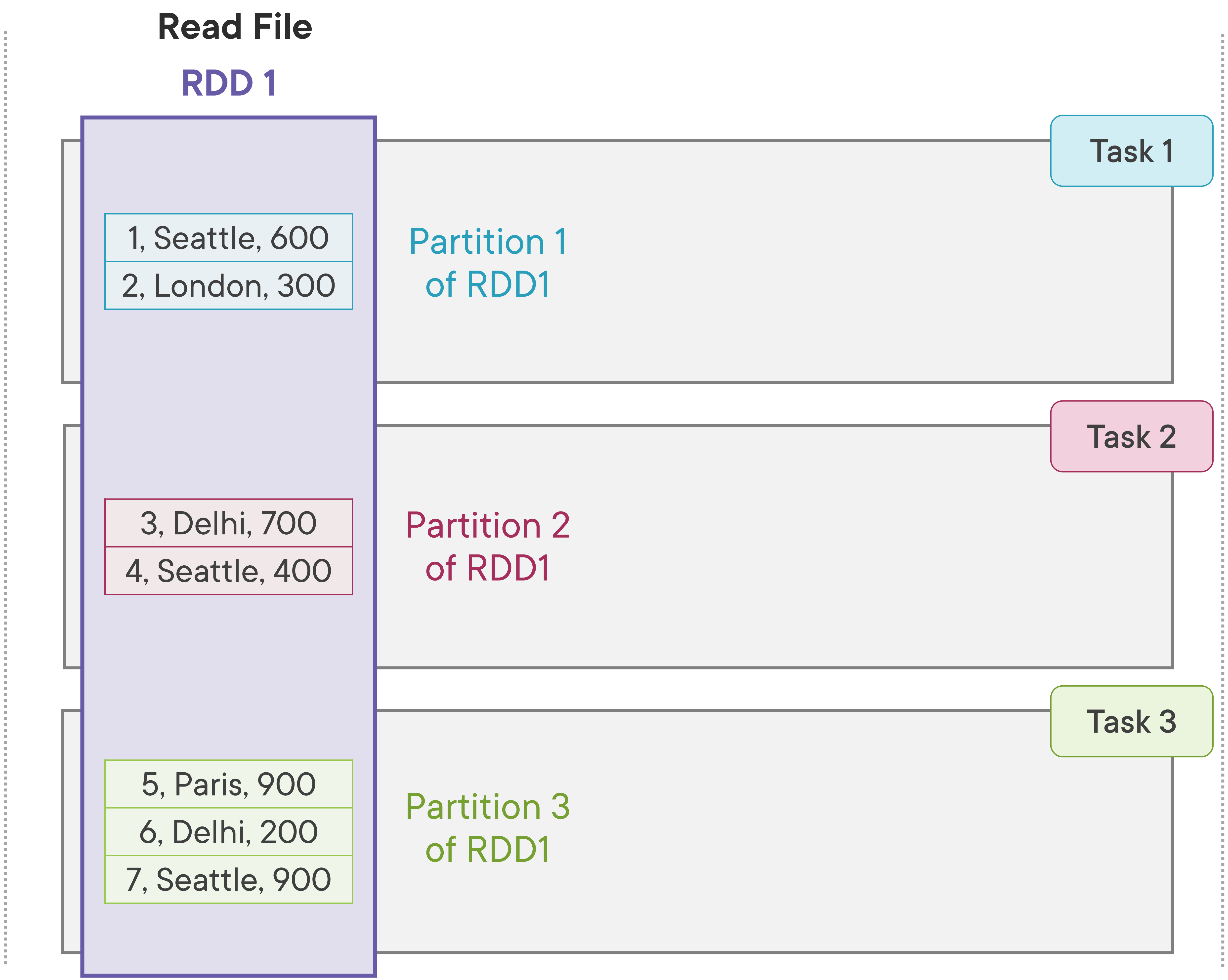


Read File

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

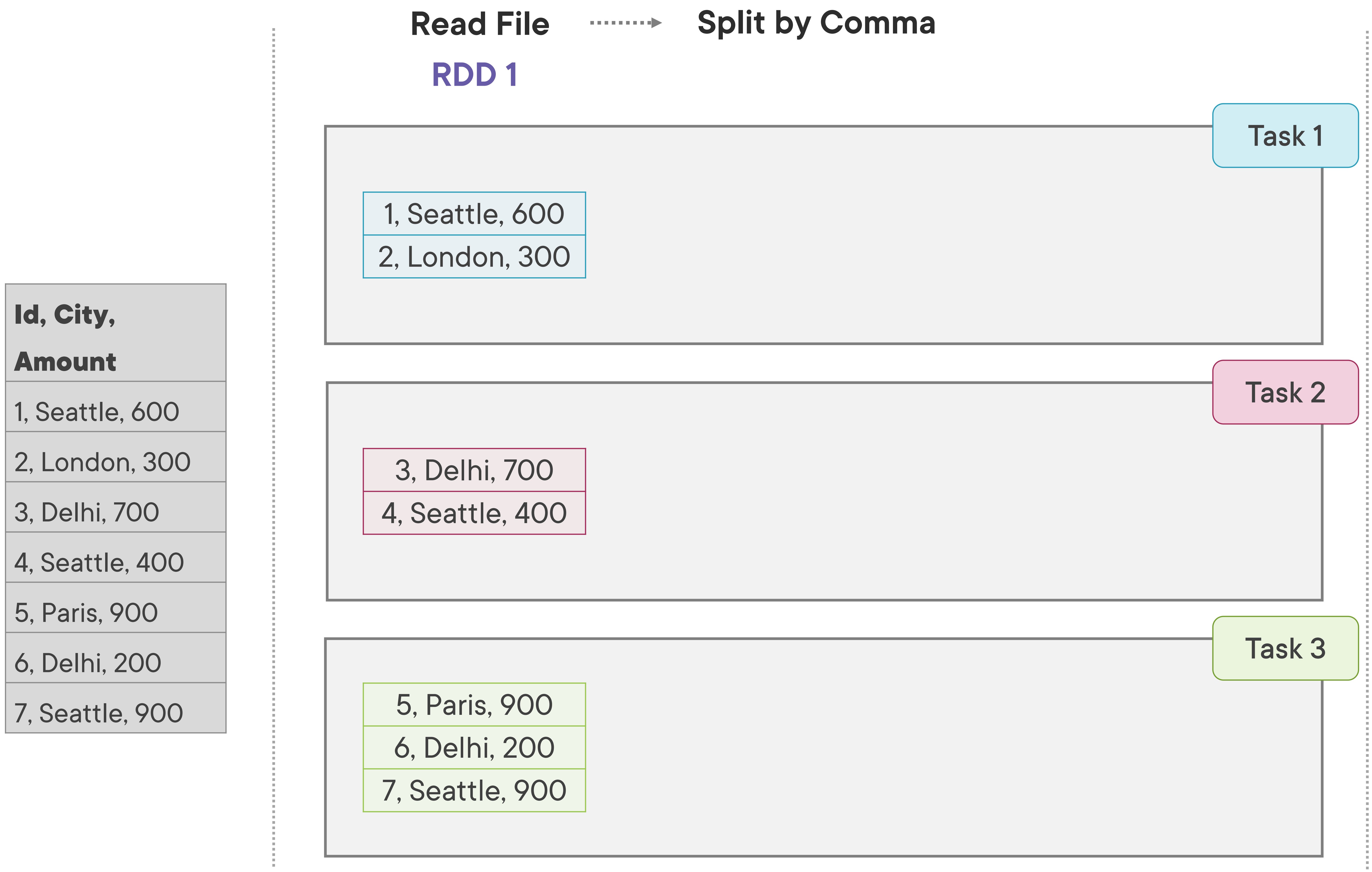
Resides in the memory of cluster

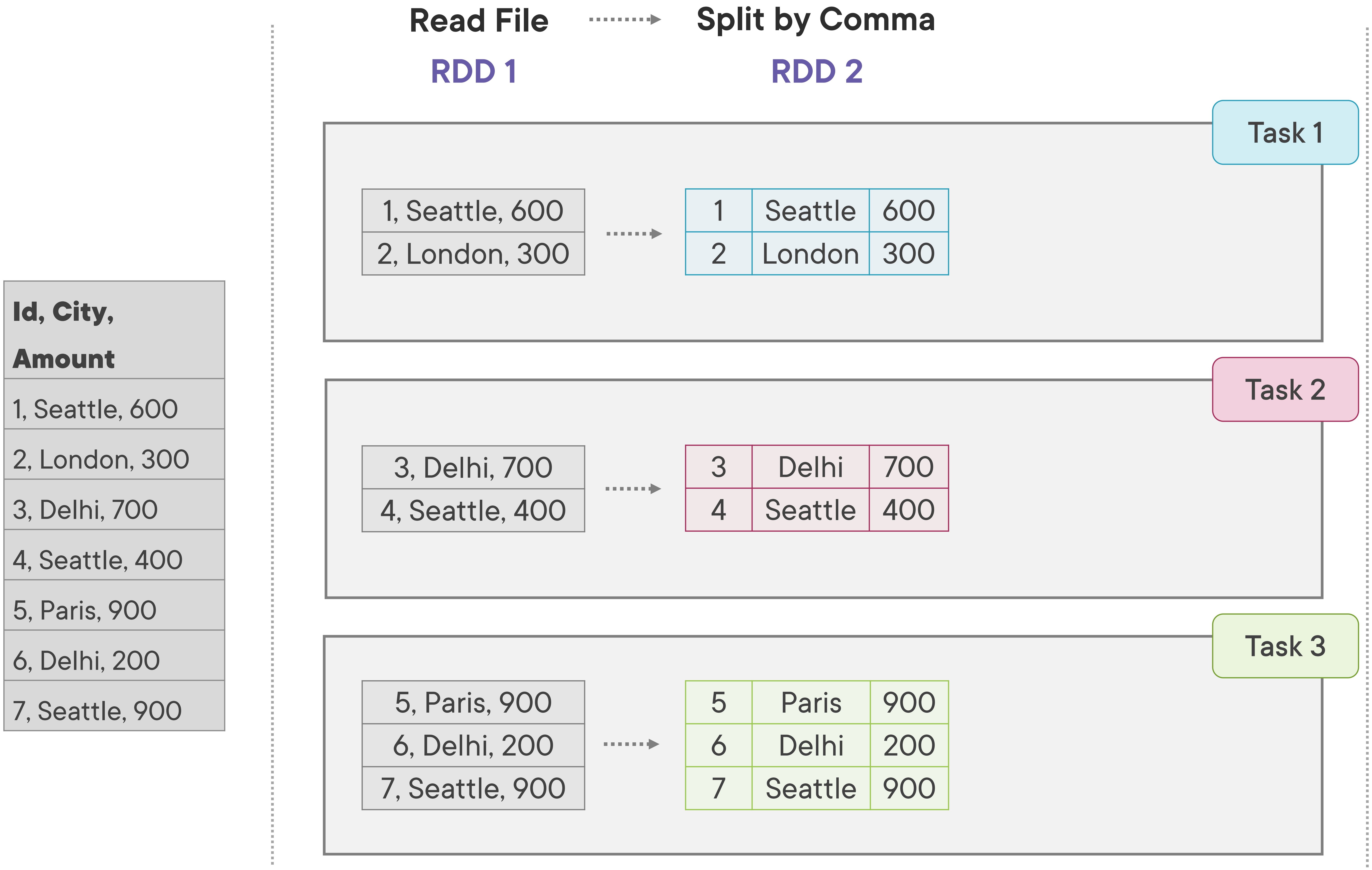
Partitioned

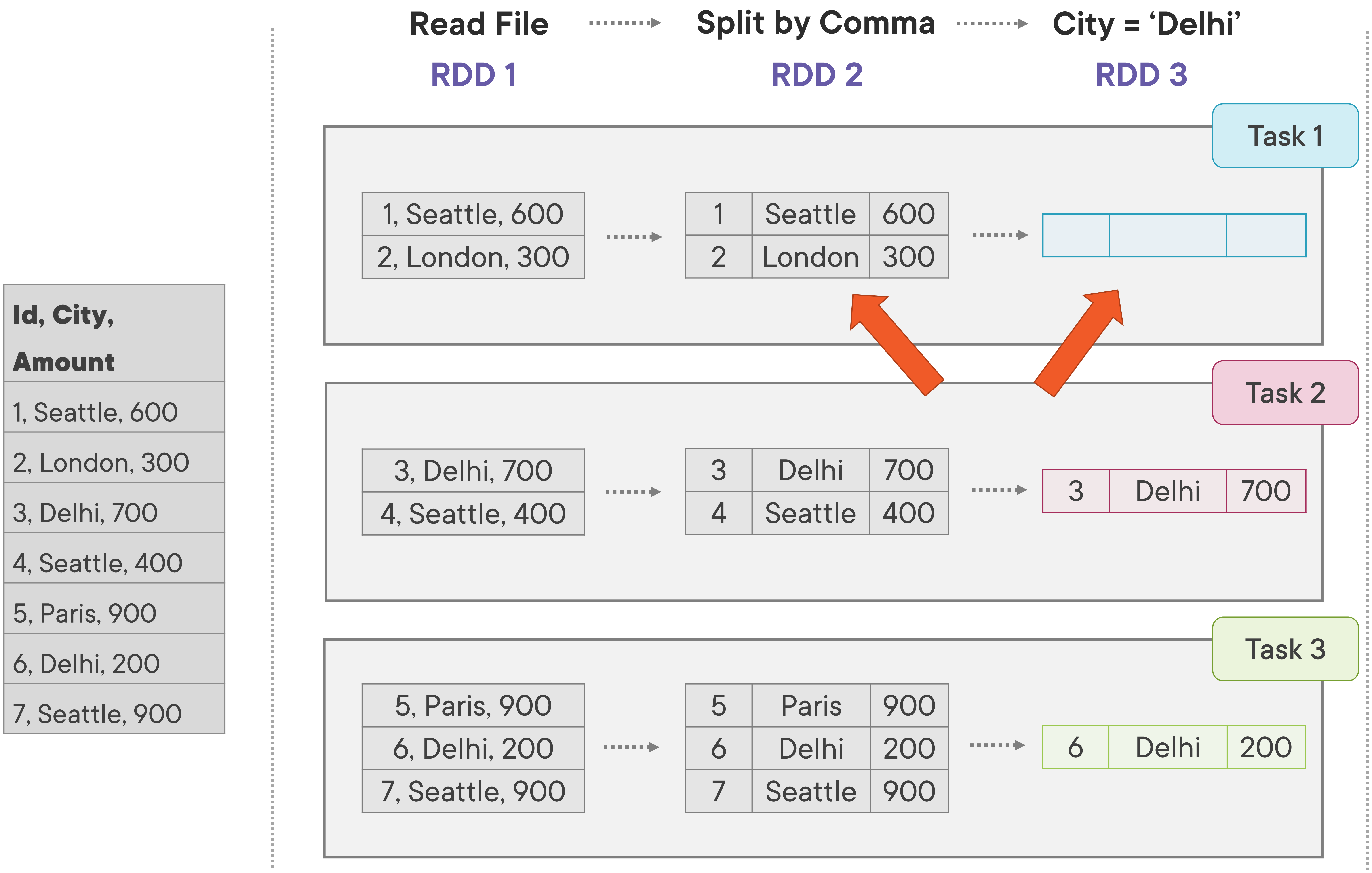
Split into partitions, processed by tasks

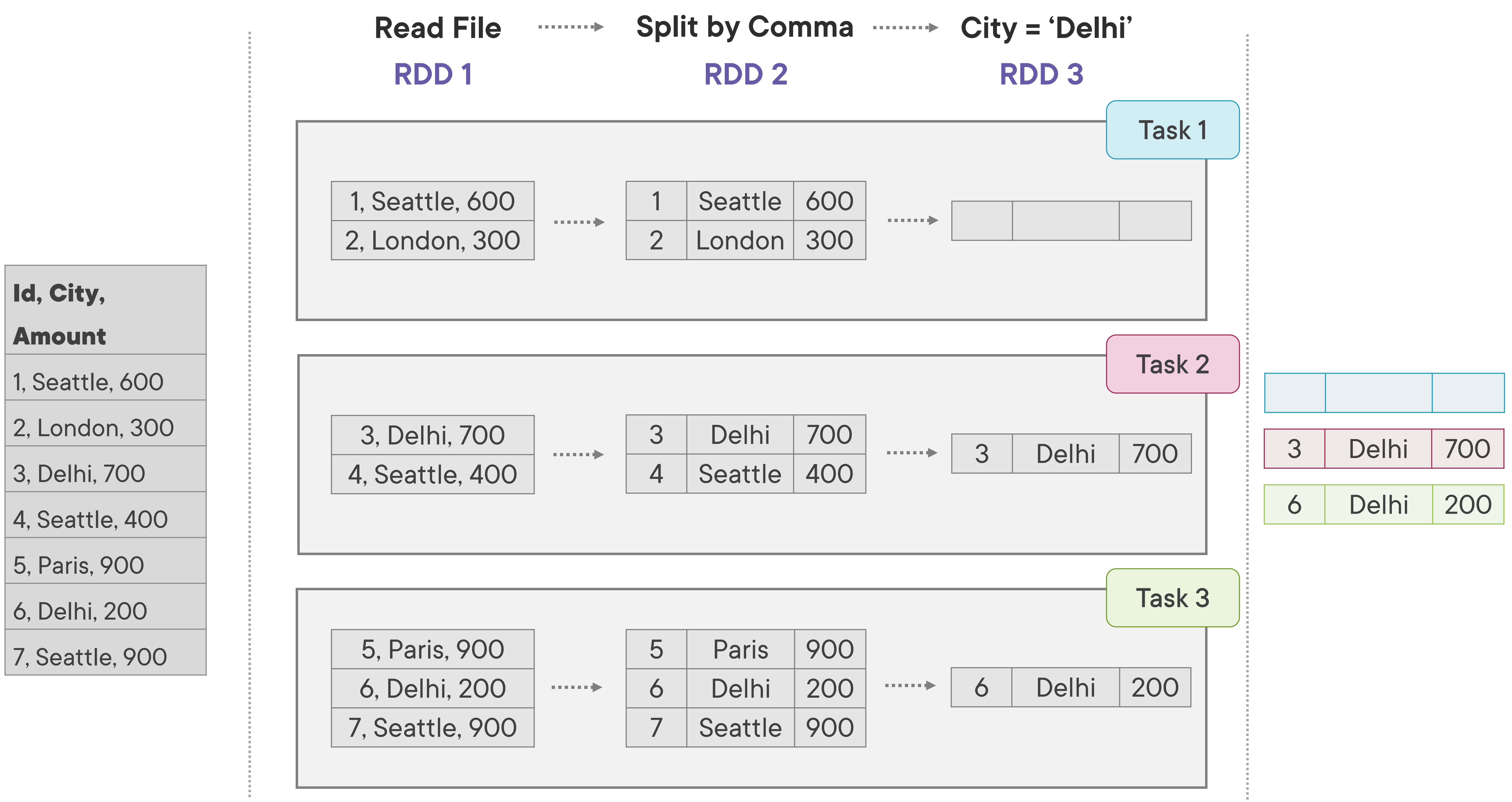
Read-only

Transformed into another RDD or result


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
5, Paris, 900
6, Delhi, 200
7, Seattle, 900


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



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
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
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

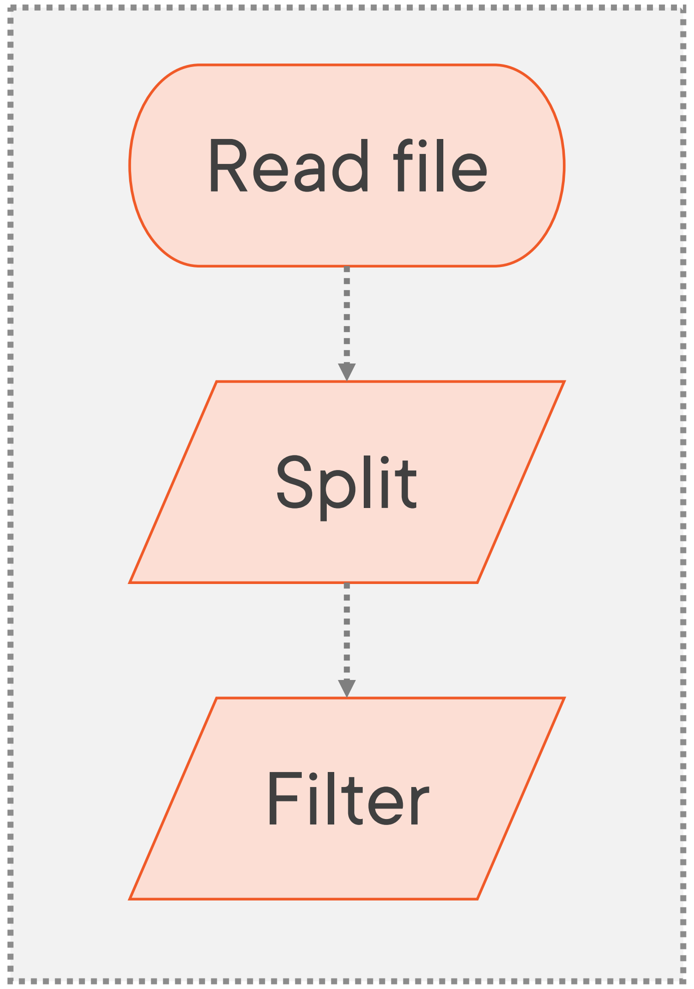
Transformed into another RDD or result

Resilient

Auto recover in case of a failure

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

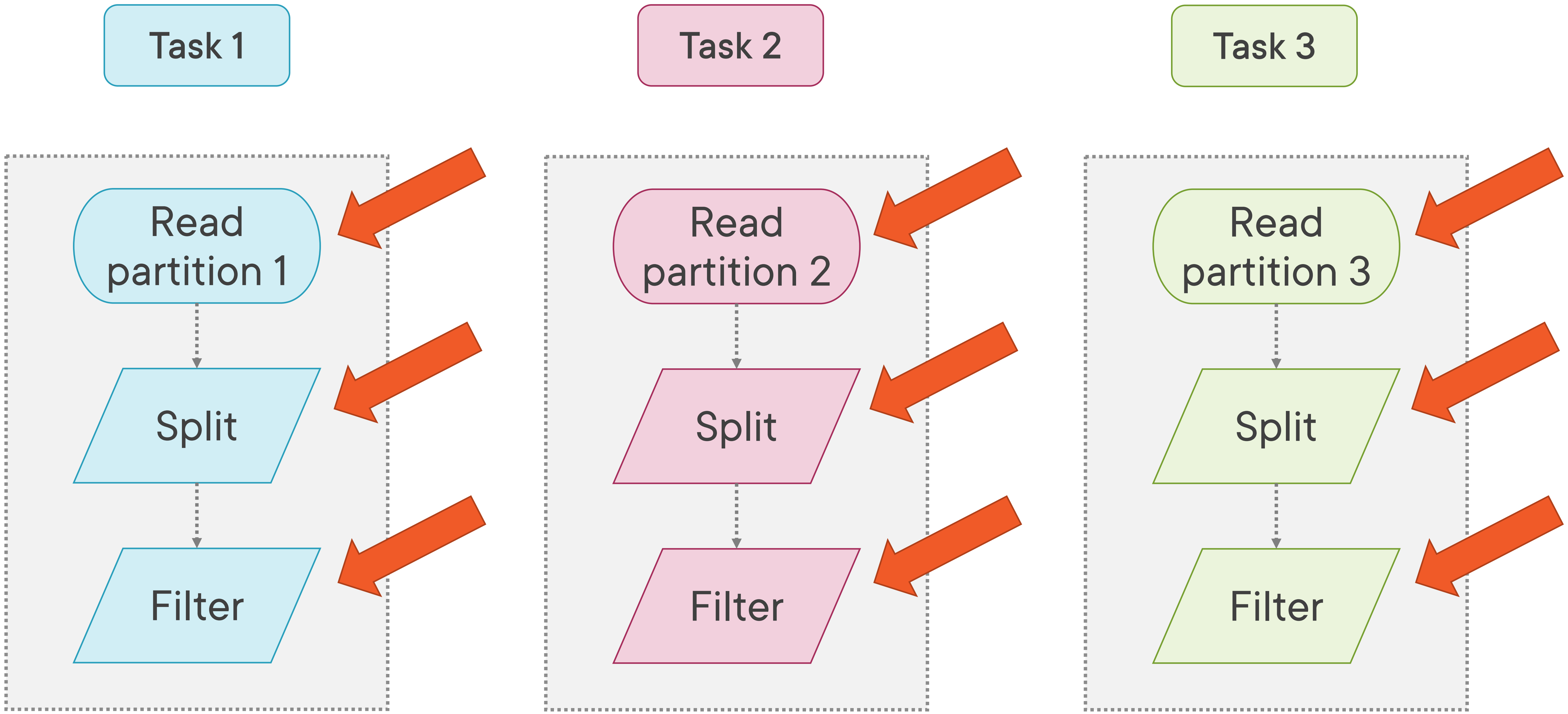


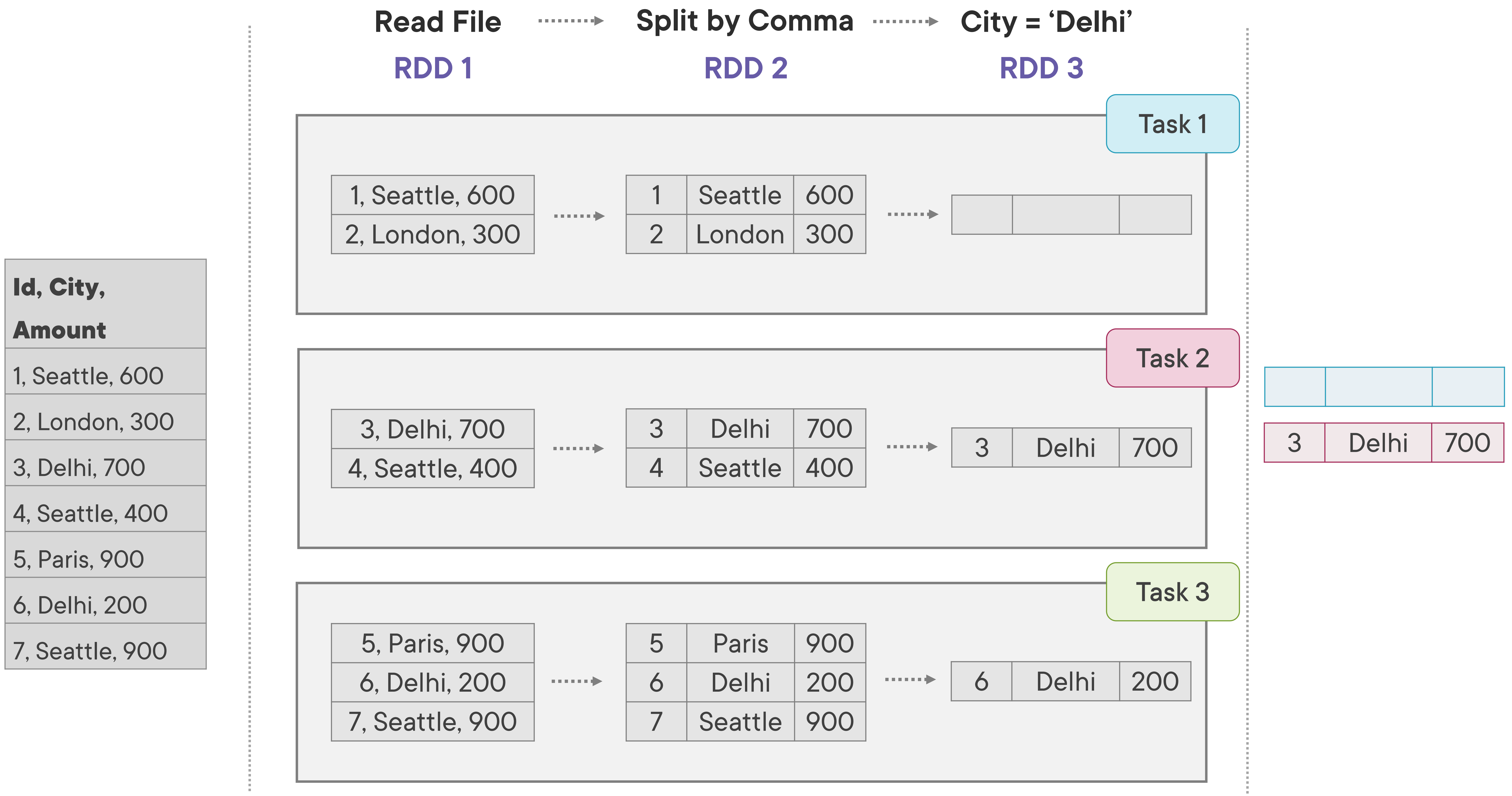
Lineage Graph

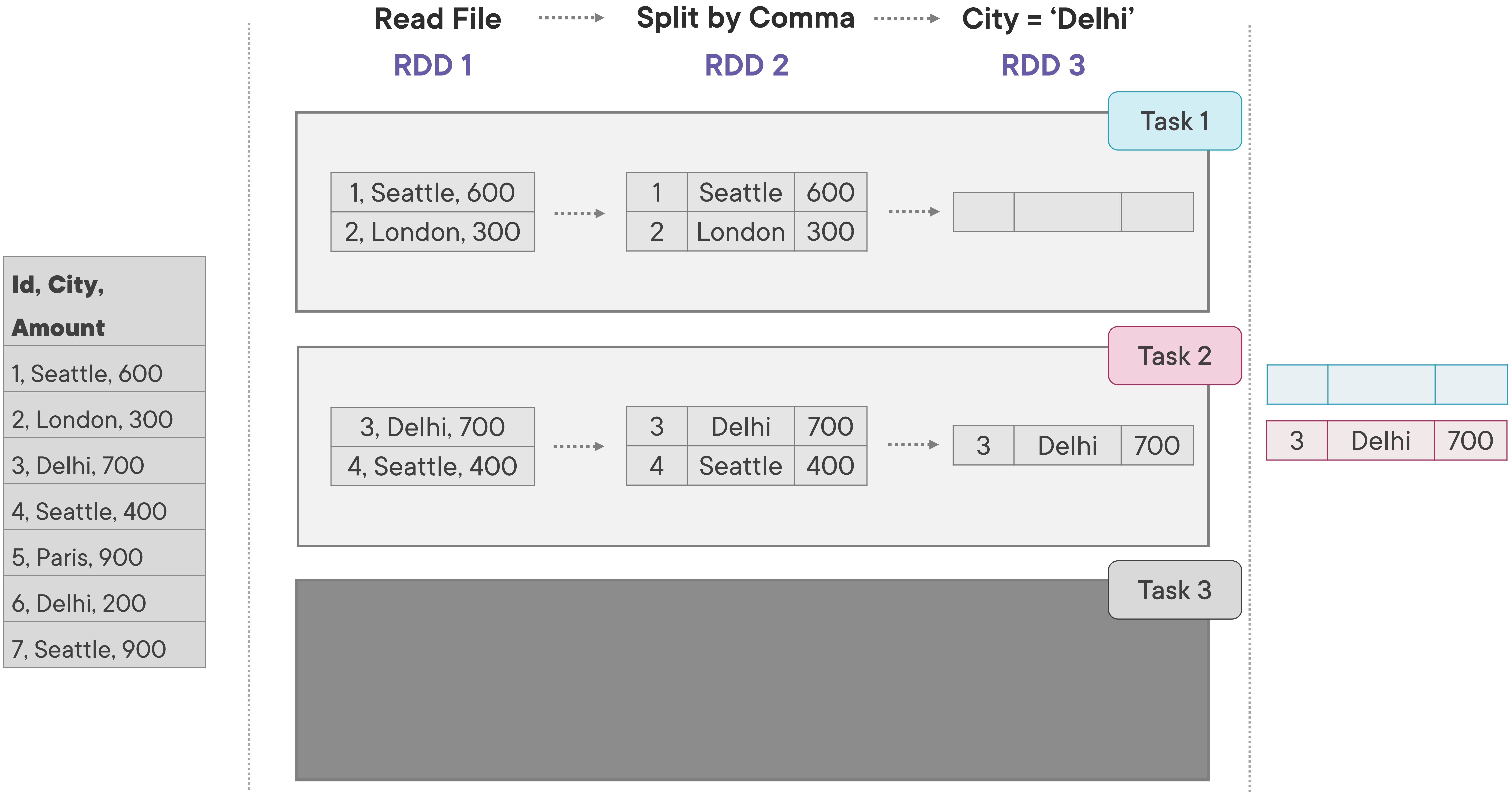
Defines steps that are used for creation of an RDD

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
2. Split data by comma
3. Filter where City='Delhi'
4. Write processed data to storage



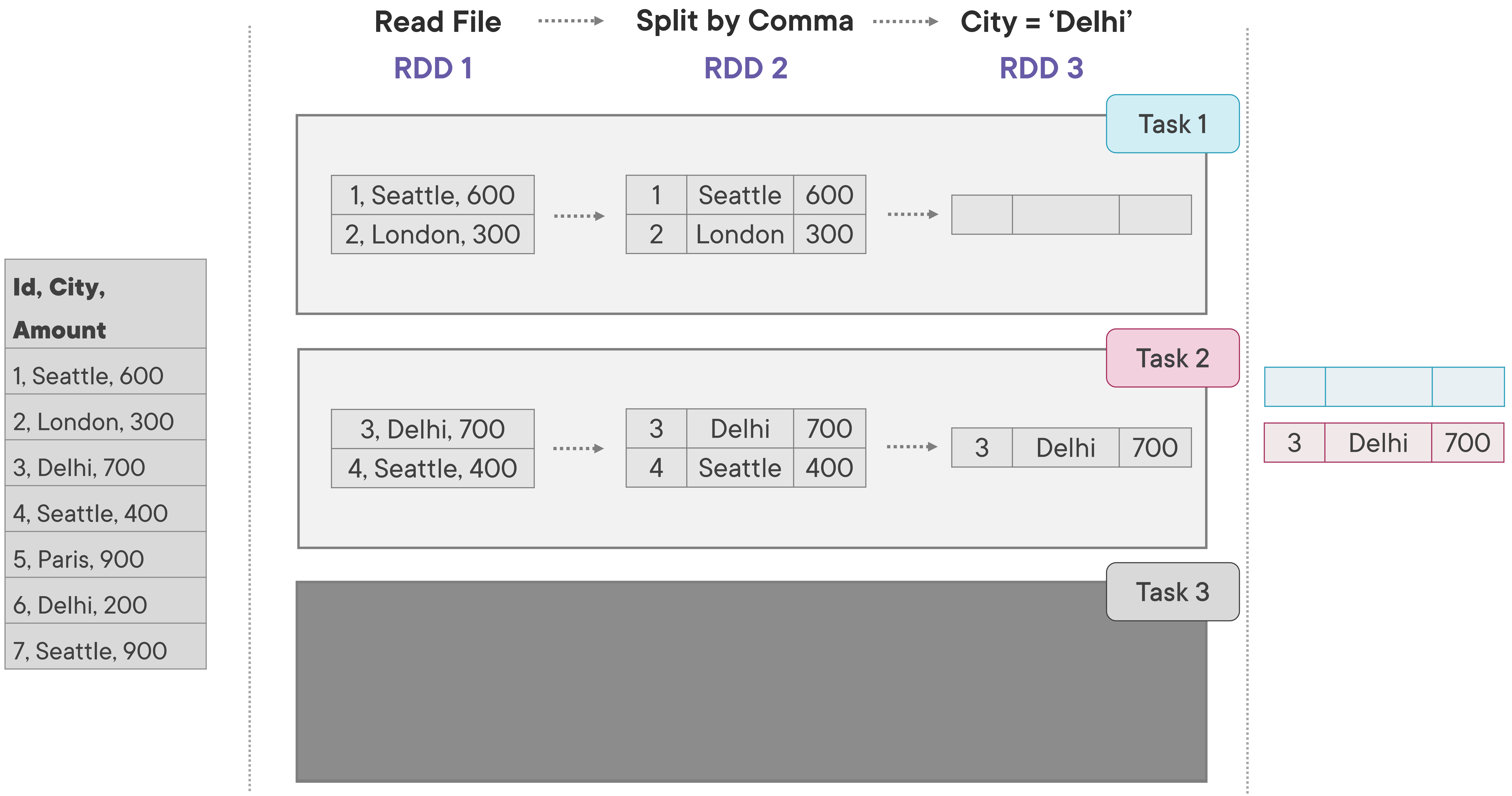

Id, City, Amount
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Task 1
Task 2
Task 3
1, Seattle, 600
2, London, 300
1 Seattle 600
2 London 300
3, Delhi, 700
4, Seattle, 400
3 Delhi 700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

Spark will re-execute failed Tasks
using their Lineage Graph

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

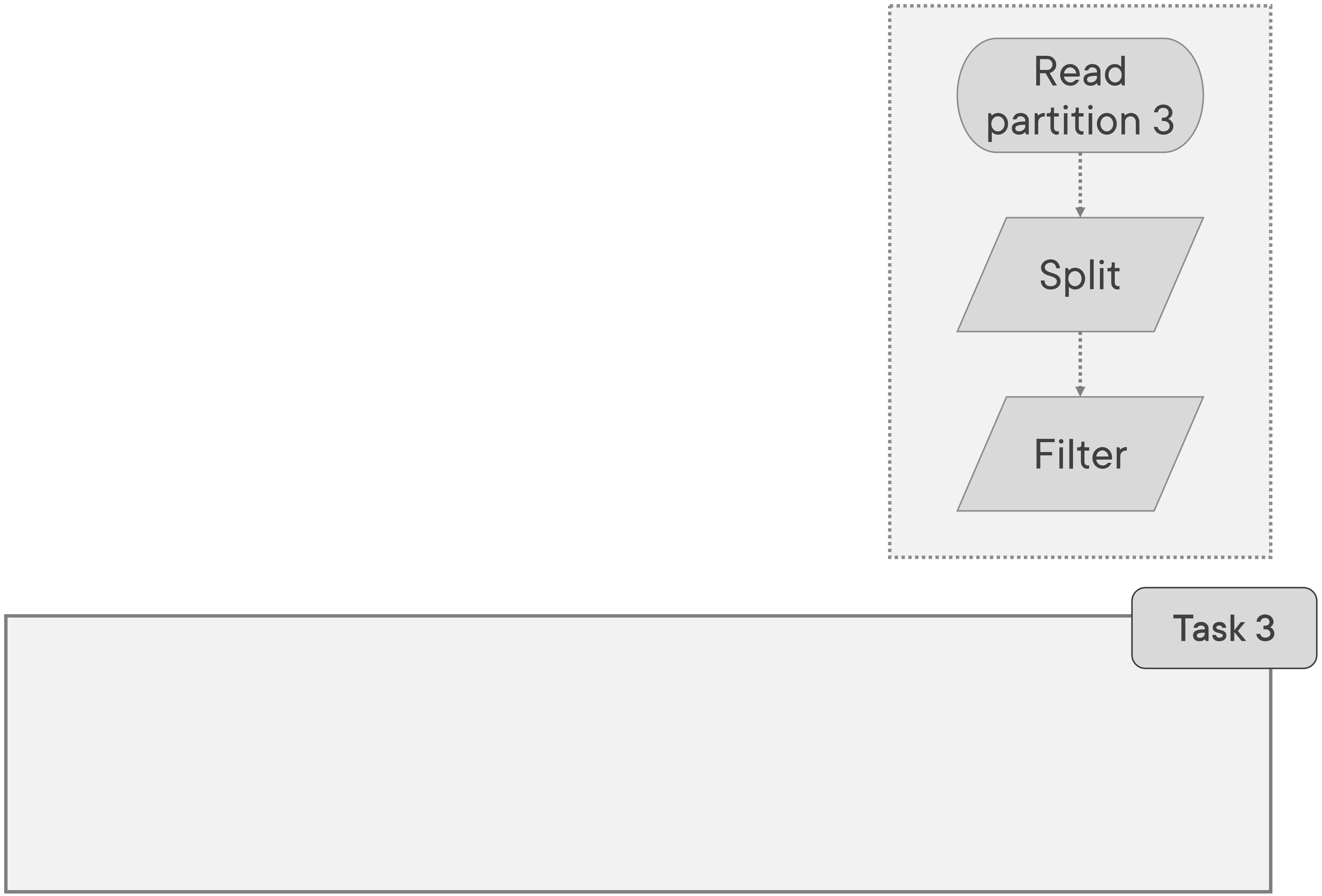


Read File
Split by Comma
City = 'Delhi'
RDD 1
RDD 2
RDD 3
Task 1
Task 2
Task 3
Id, City, Amount
1, Seattle, 600
2, London, 300
3, Delhi, 700
4, Seattle, 400
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
1, Seattle, 600
2, London, 300
1 Seattle 600
2 London 300
3, Delhi, 700
4, Seattle, 400
3 Delhi 700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

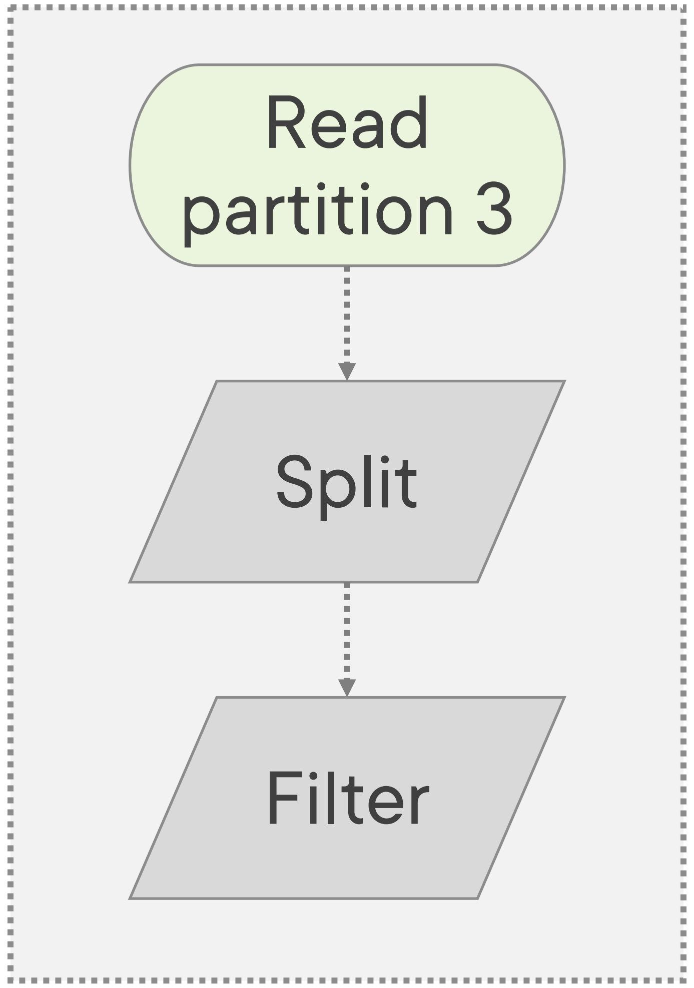


3	Delhi	700

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

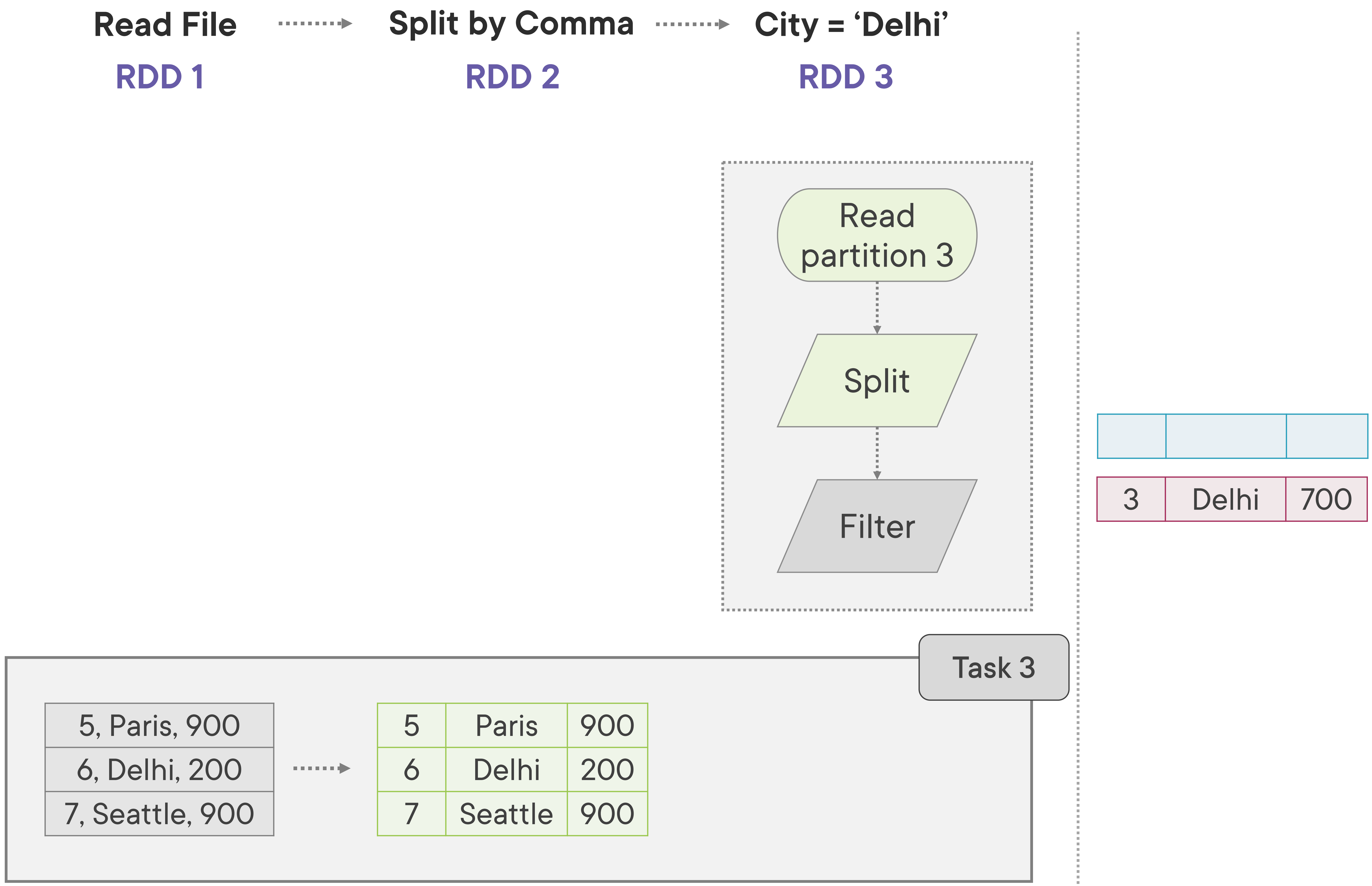


Task 3

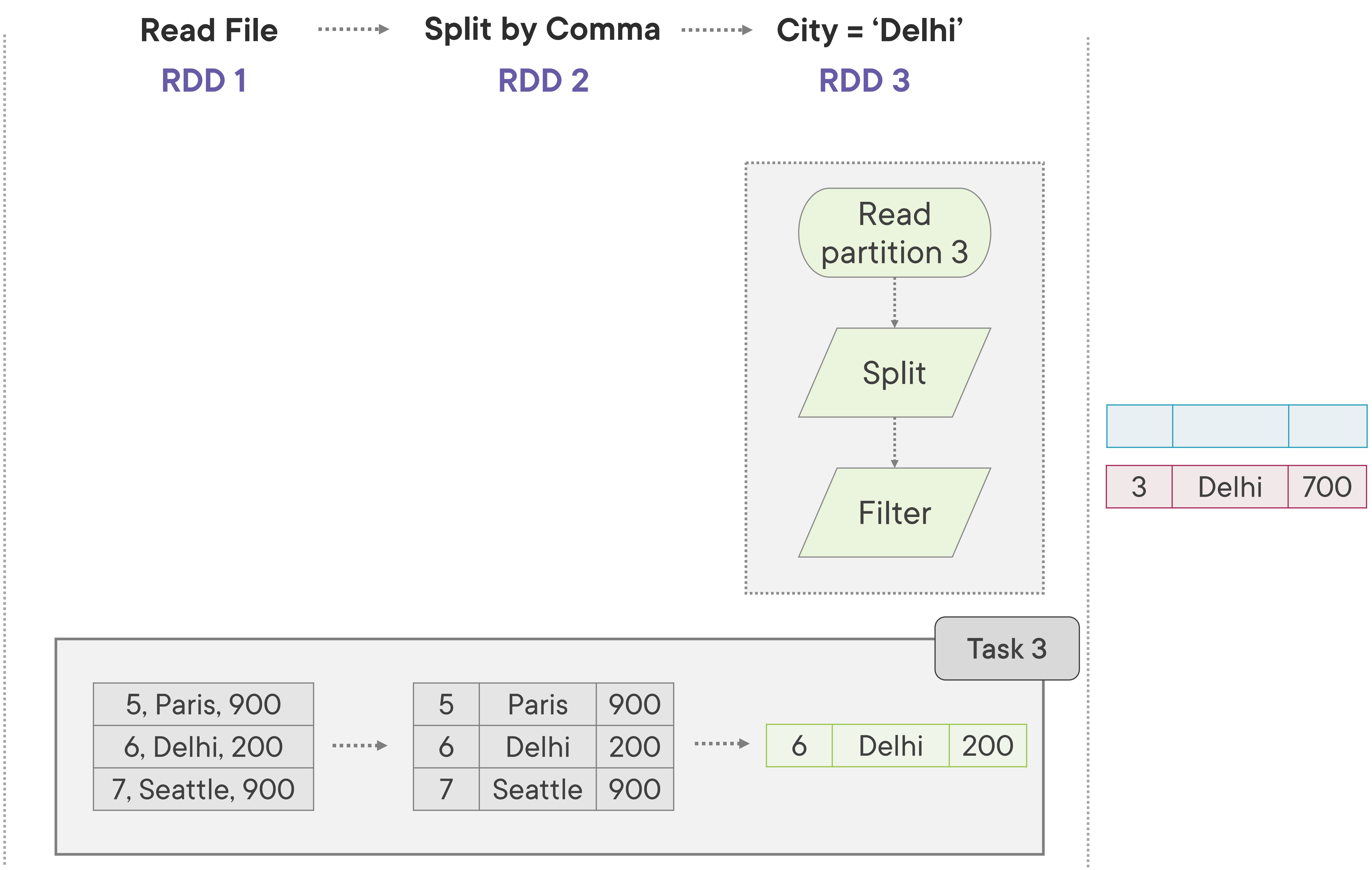
5, Paris, 900
6, Delhi, 200
7, Seattle, 900

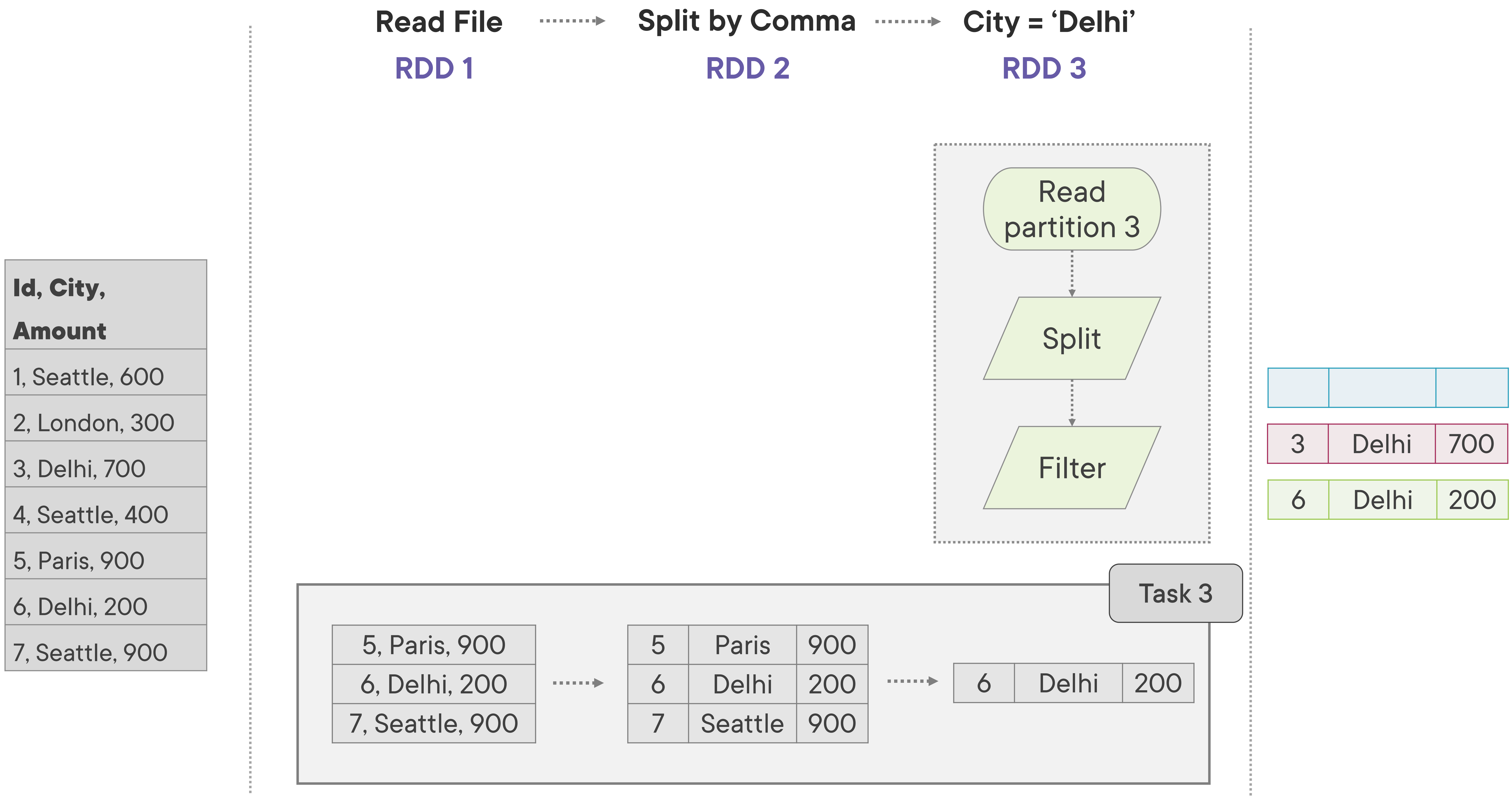
3	Delhi	700

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
RDD 1
Split by Comma
RDD 2
City = ‘Delhi’
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
3 Delhi 700
6 Delhi 200
Task 3
5, Paris, 900
6, Delhi, 200
7, Seattle, 900
5 Paris 900
6 Delhi 200
7 Seattle 900
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

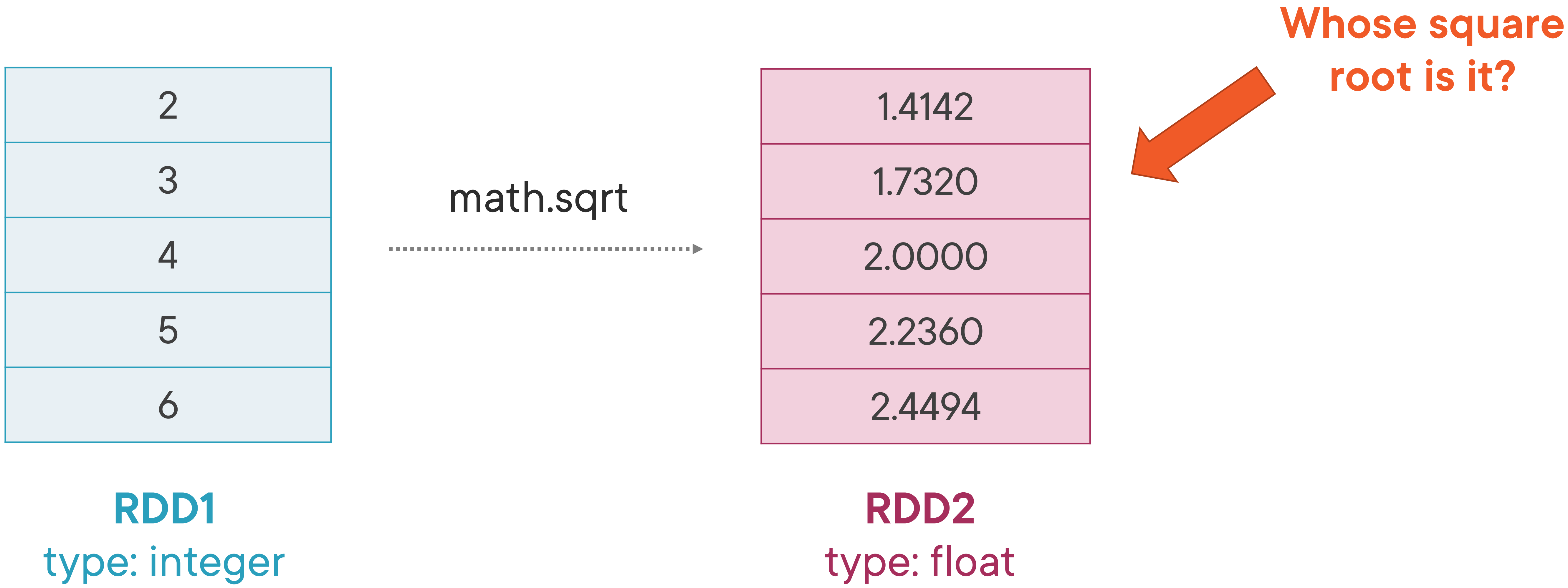
Read a File

From another RDD

Setup Instructions, and Code & Data Files are available in Exercise Files section of the course

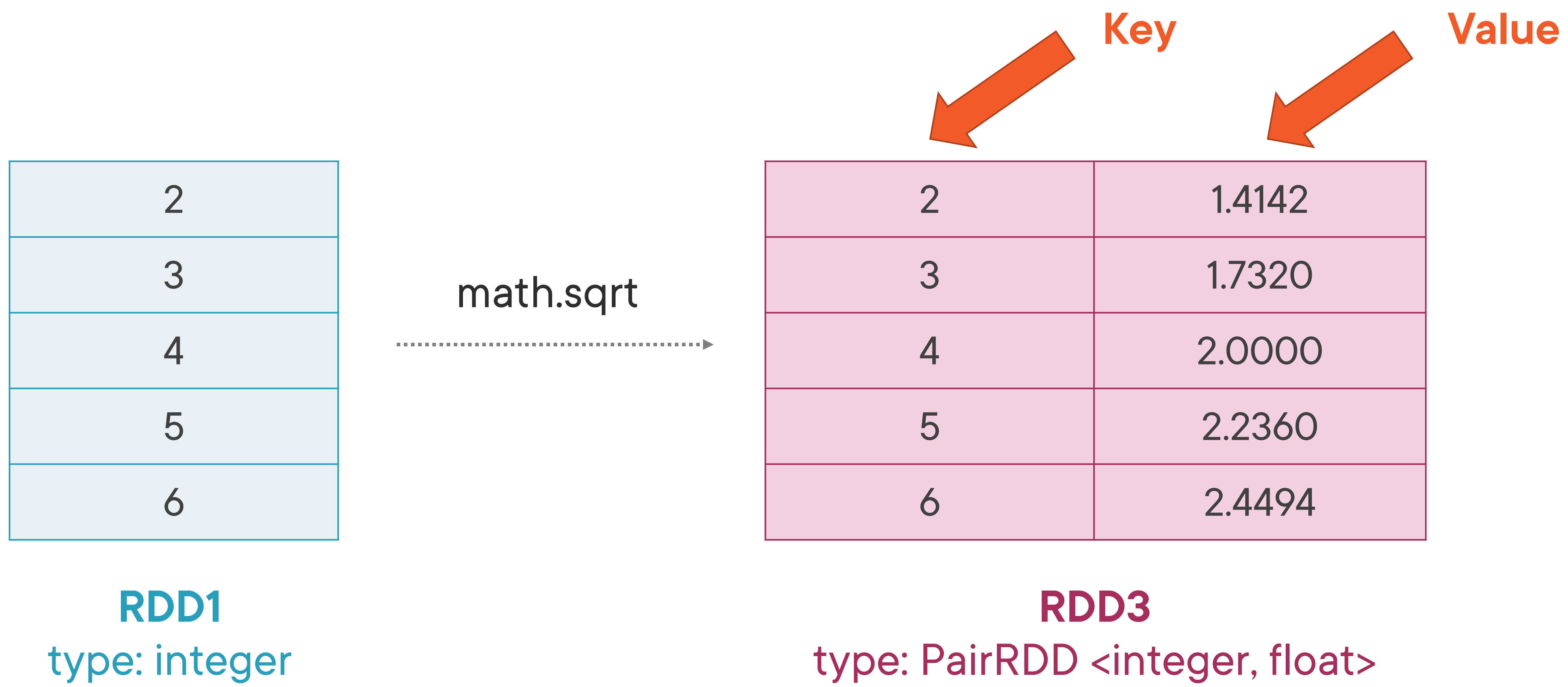
Working with Pair RDDs

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

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

Task 1

Seattle	600
Seattle	300
Seattle	400

Delhi	700

value 1

value 2

Lambda Function

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

Calculate sum of Amount by City

Task 1

Seattle	900
Seattle	400

value 1

value 2

Delhi	700

Delhi	900
Delhi	200
Seattle	900

Lambda Function

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

Calculate sum of Amount by City

Task 1

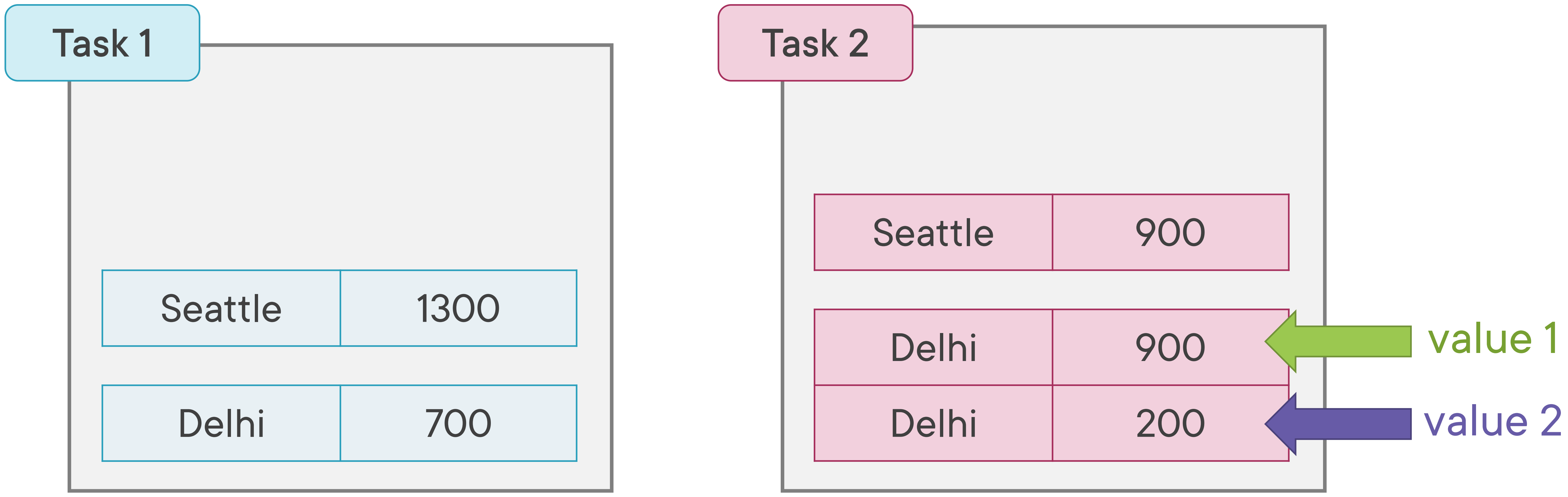
Seattle	1300
Delhi	700

Delhi	900
Delhi	200
Seattle	900

Lambda Function

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

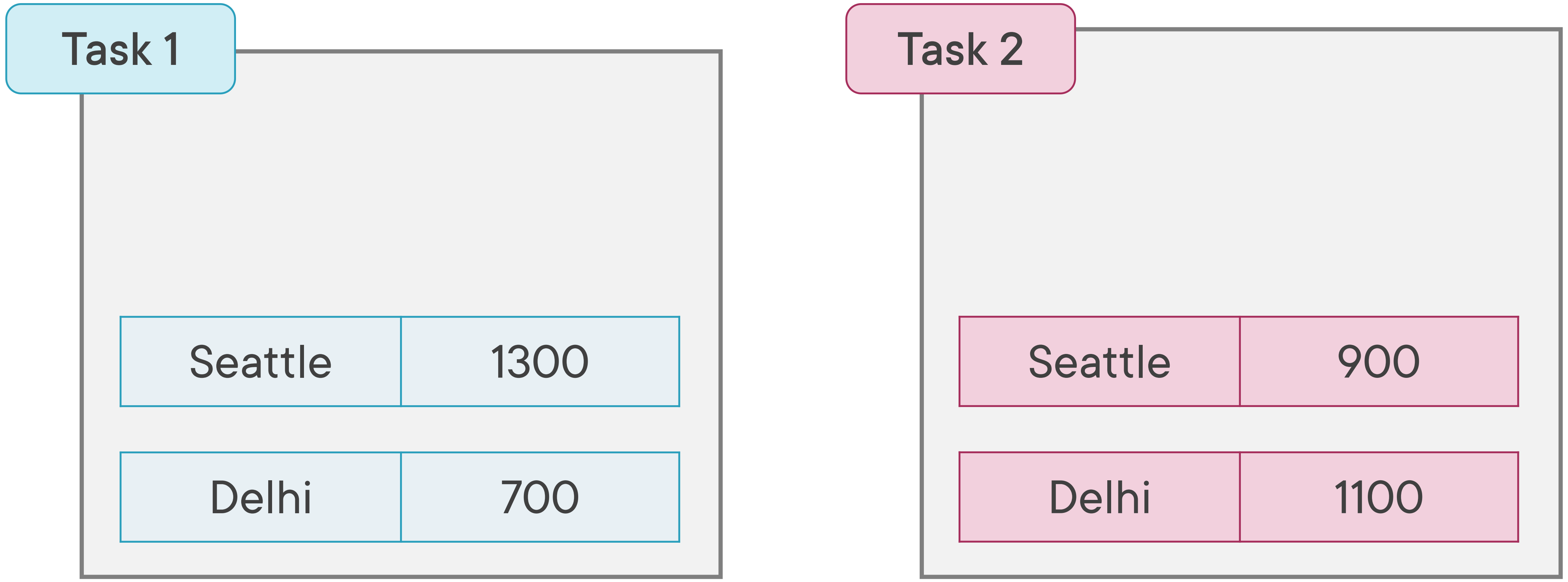
Calculate sum of Amount by City



Lambda Function

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

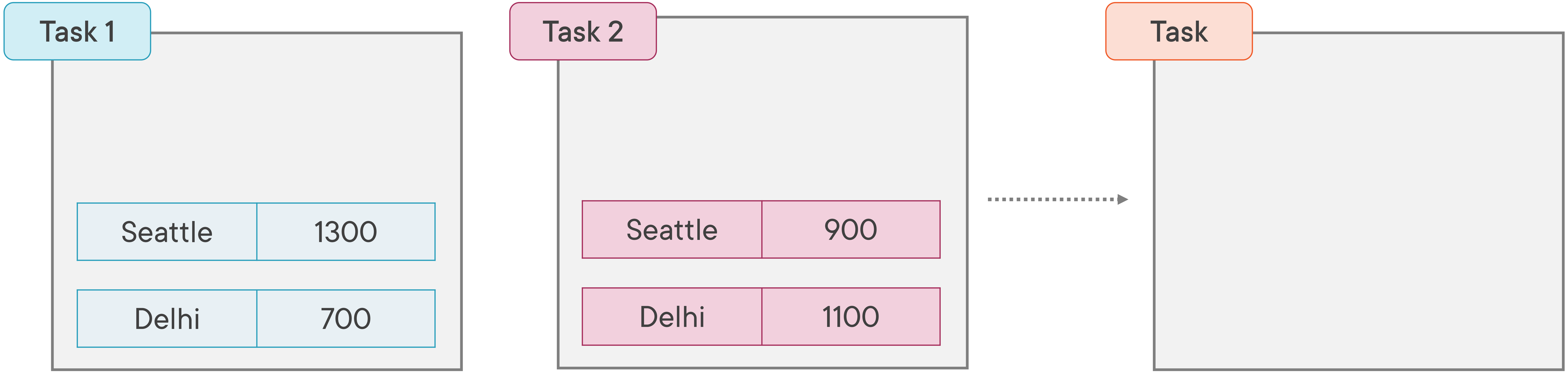
Calculate sum of Amount by City



Lambda Function

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

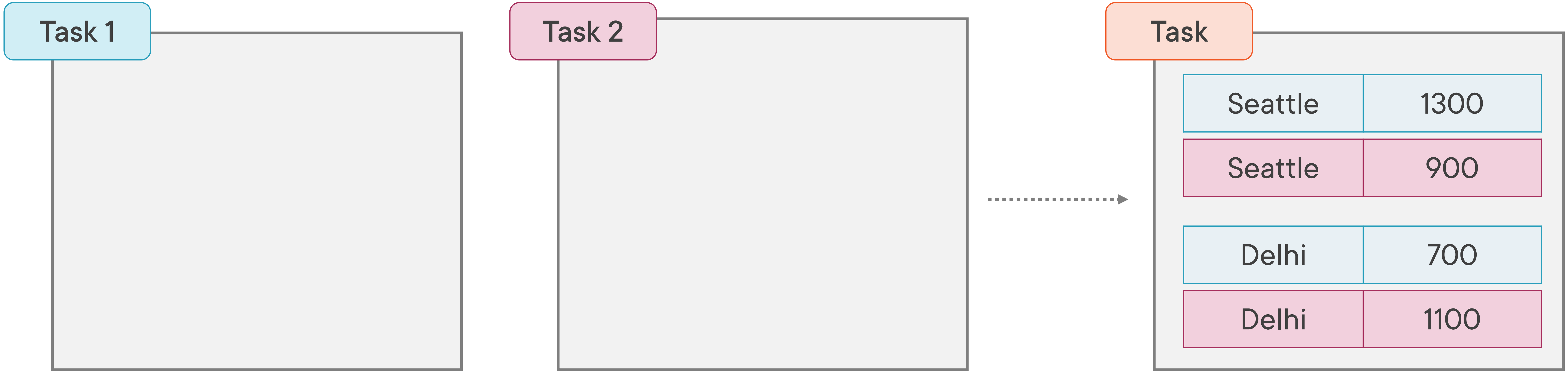
Calculate sum of Amount by City



Lambda Function

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

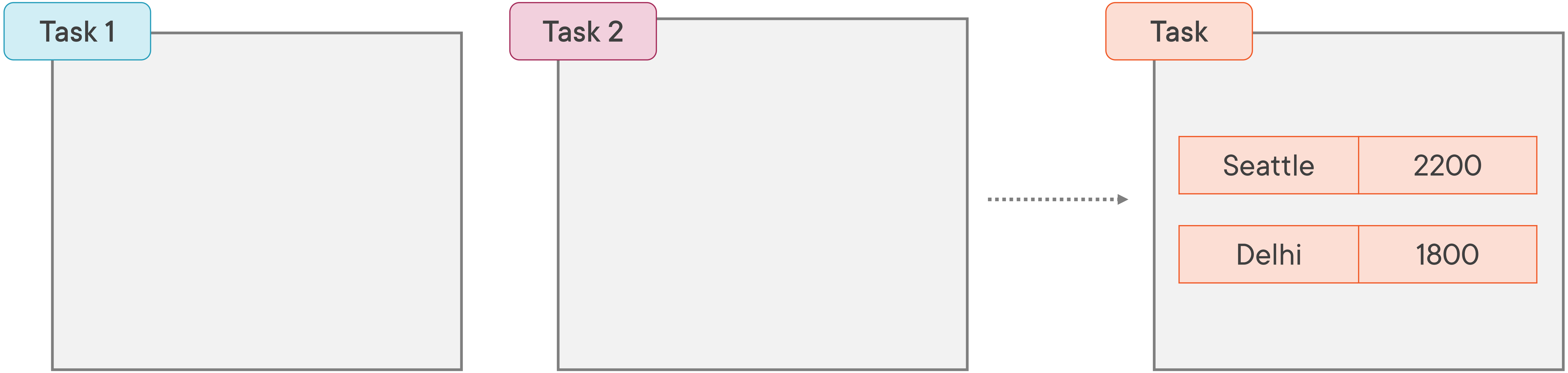
Calculate sum of Amount by City



Lambda Function

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

Calculate sum of Amount by City



Lambda Function

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

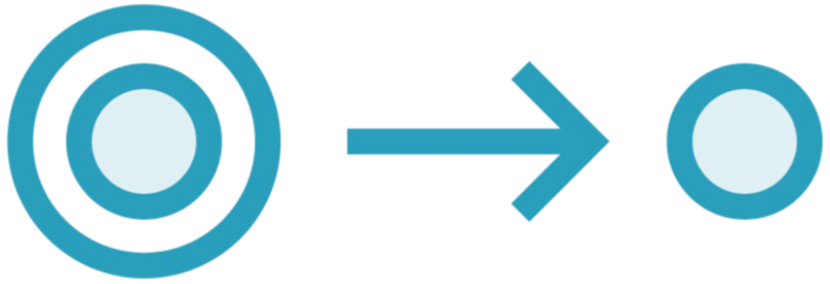
Applying Operations on RDDs

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

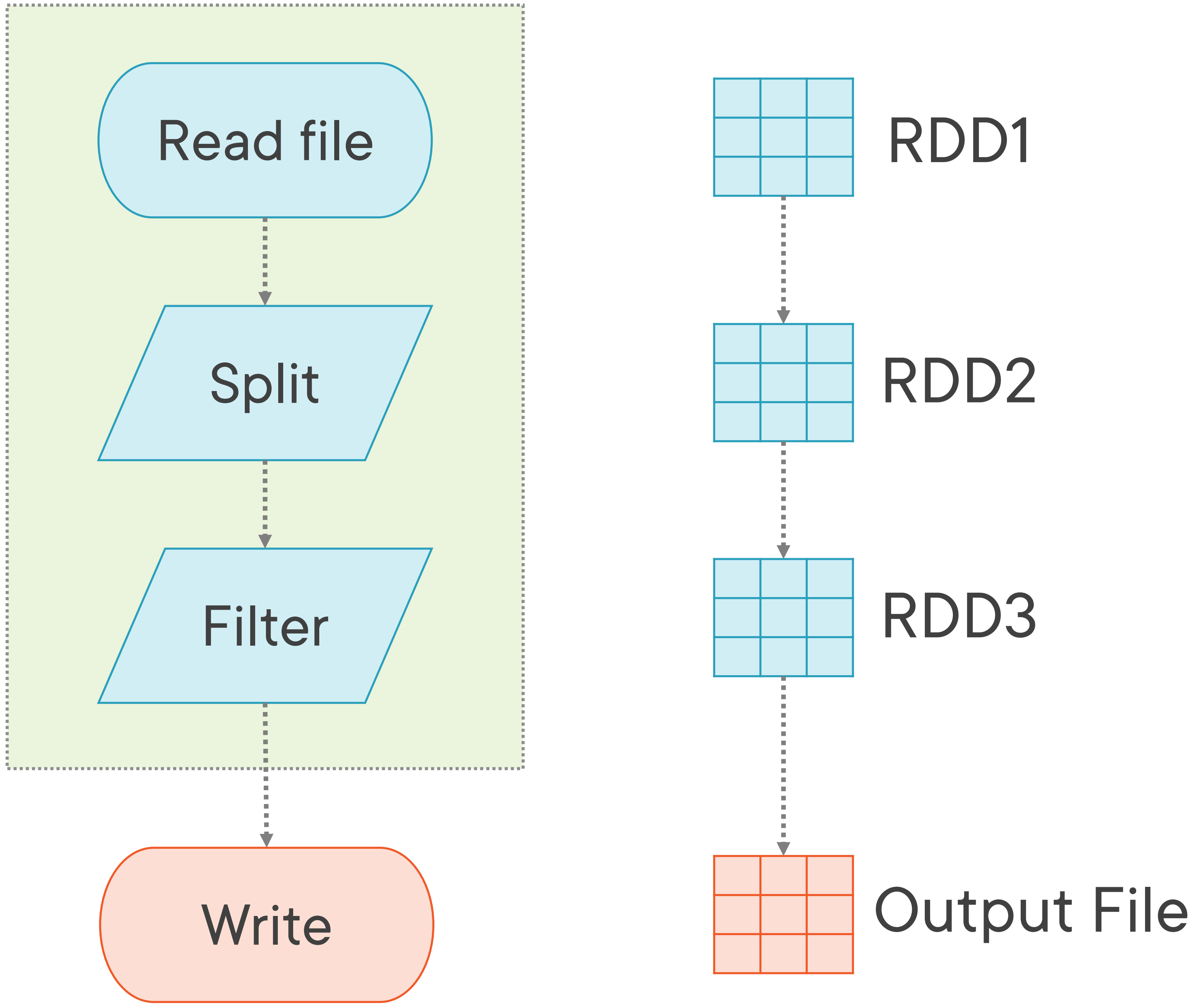
1. Read file1.csv from Storage
2. Split data by comma
3. Filter where City='Delhi'
4. Write processed data to storage

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

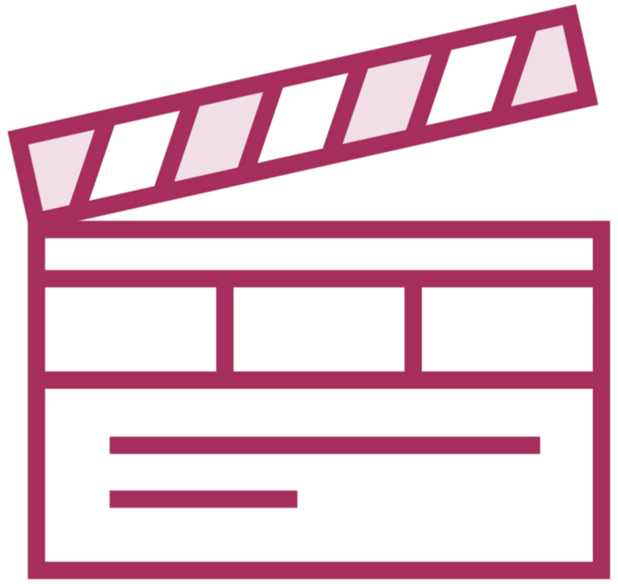
Lineage Graph

Built using transformation operations

Action operation



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

Using Narrow Transformations

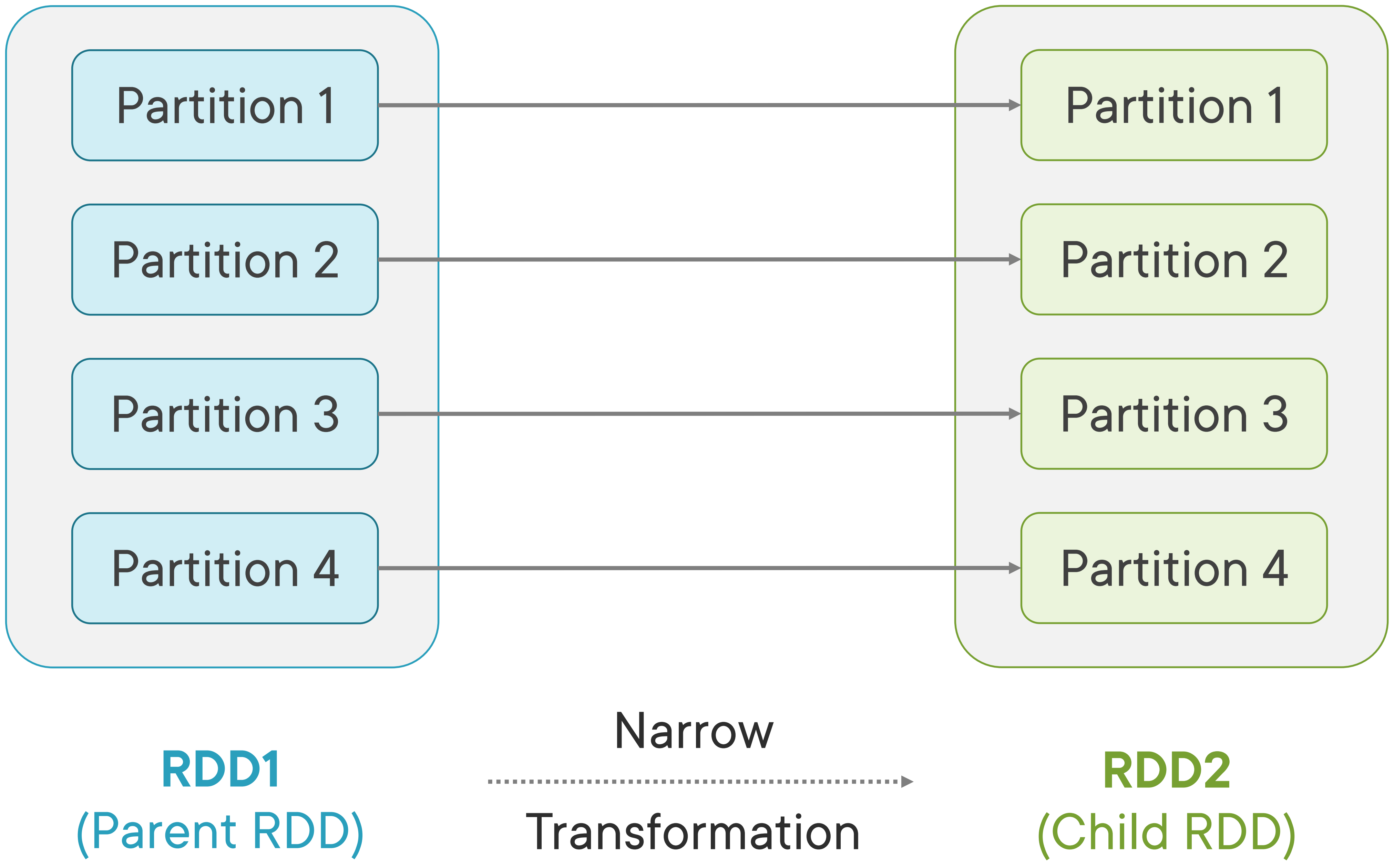
Transformation Operations

Narrow Dependency Transformation

Wide Dependency Transformation

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

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

Read file,
split by comma

RDD 1

Task 1

Task 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

Filter
City = ‘Seattle’
RDD 2

Task 1

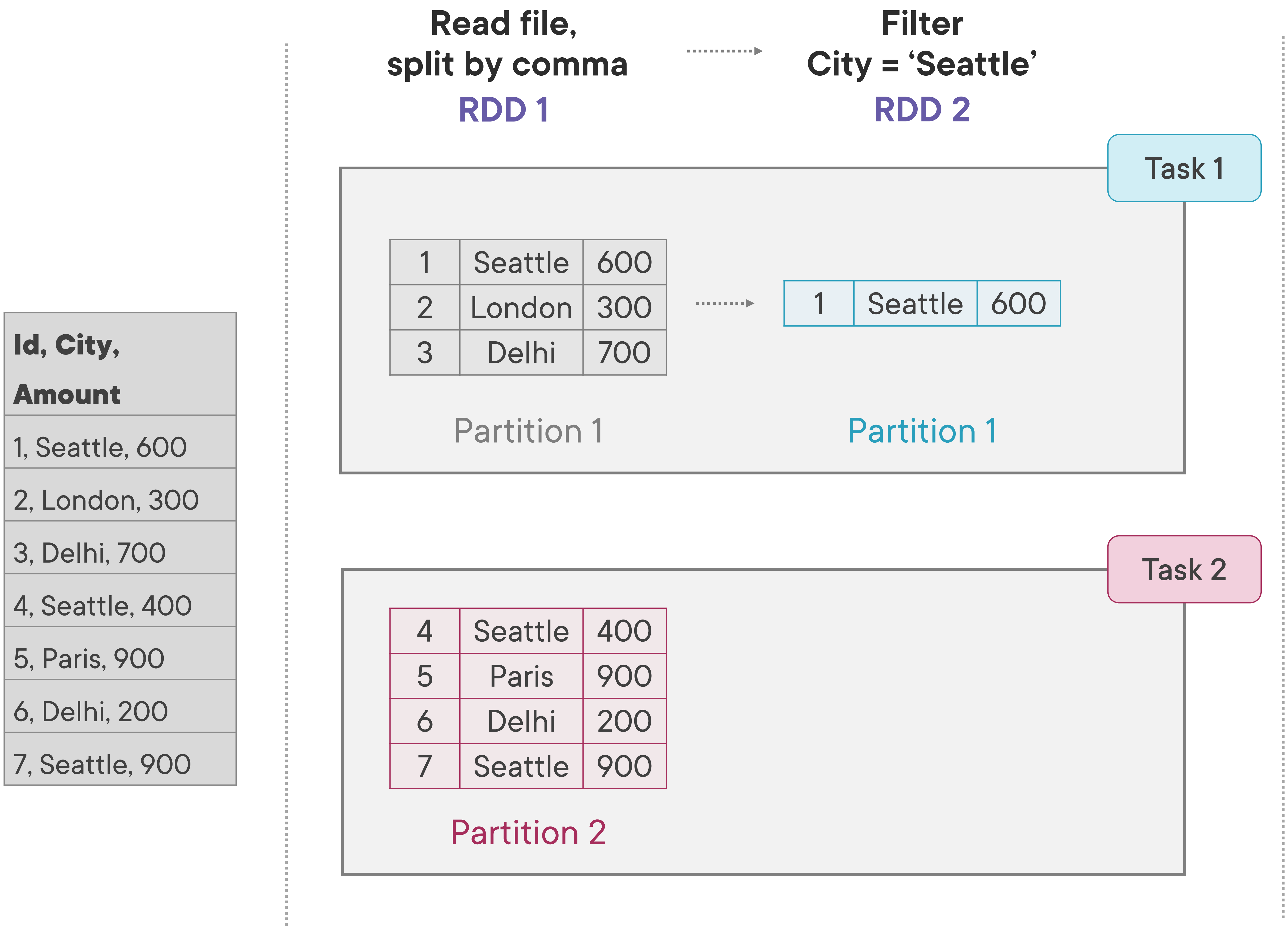
1	Seattle	600
2	London	300
3	Delhi	700

Partition 1

Task 2

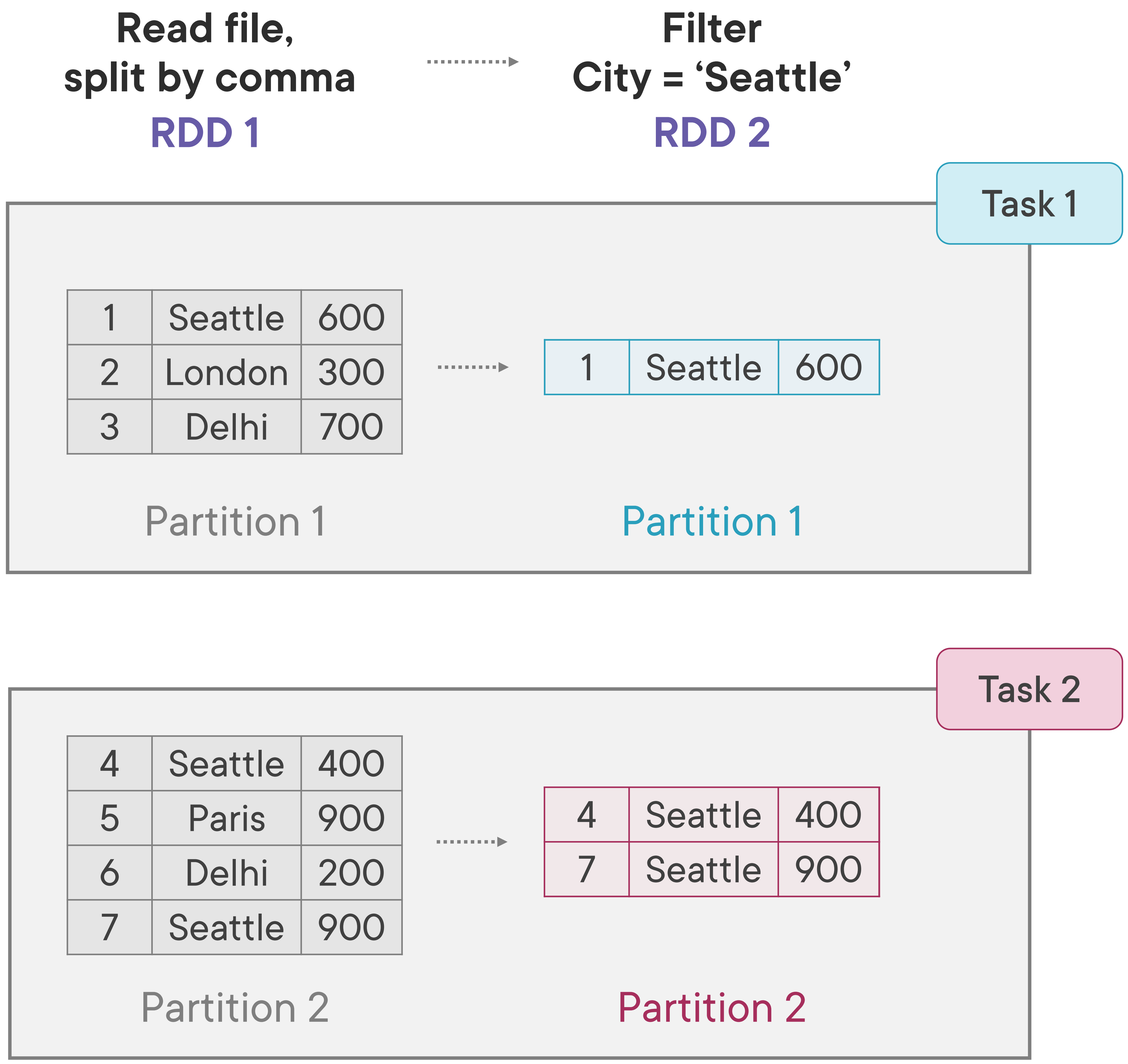
4	Seattle	400
5	Paris	900
6	Delhi	200
7	Seattle	900

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
Read file, split by comma
RDD 1
Filter City = 'Seattle'
RDD 2
Task 1
1 Seattle 600
2 London 300
3 Delhi 700
1 Seattle 600
Partition 1
Partition 1
Task 2
4 Seattle 400
5 Paris 900
6 Delhi 200
7 Seattle 900
Partition 2

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



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

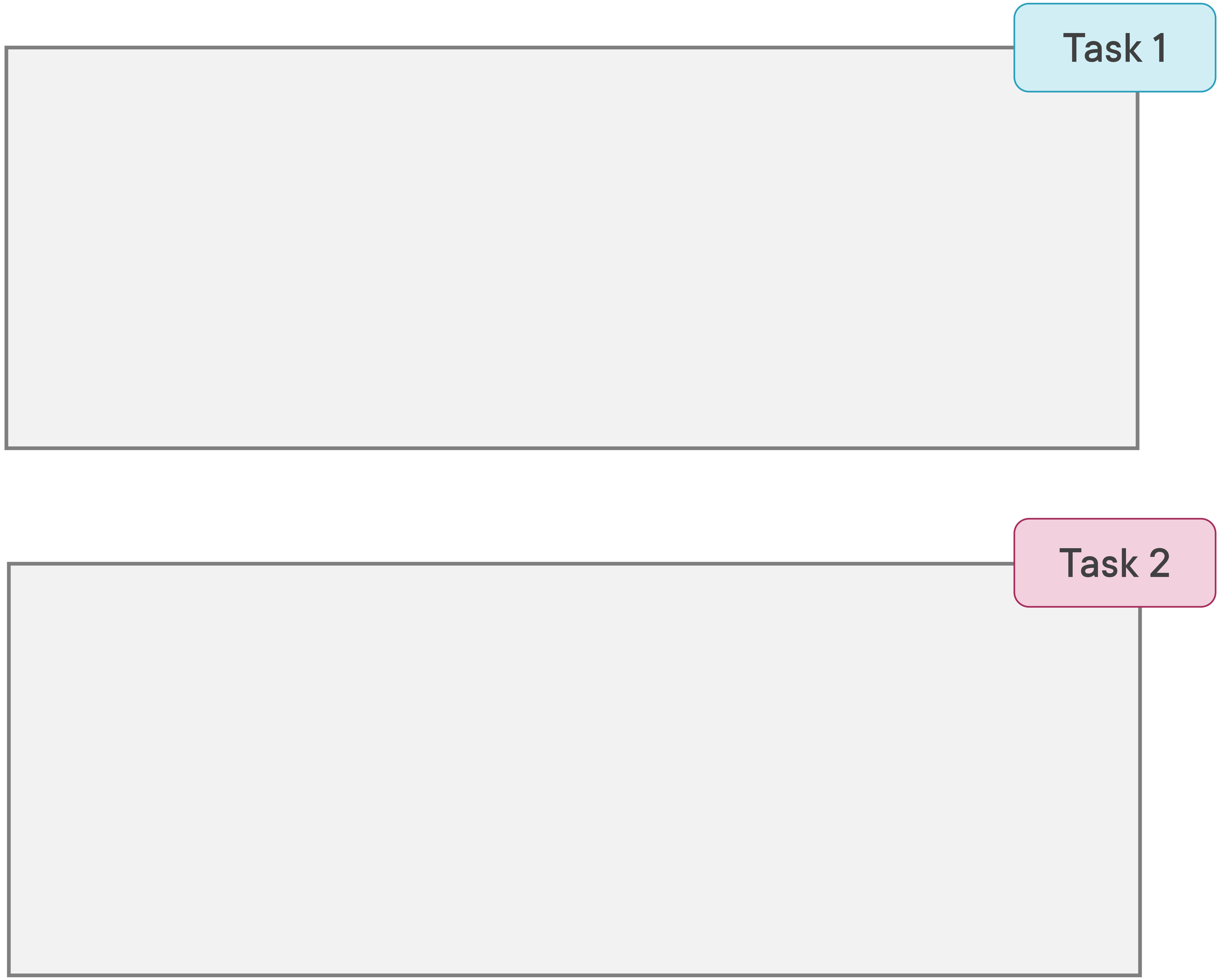
1	Seattle	600

4	Seattle	400
7	Seattle	900

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



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

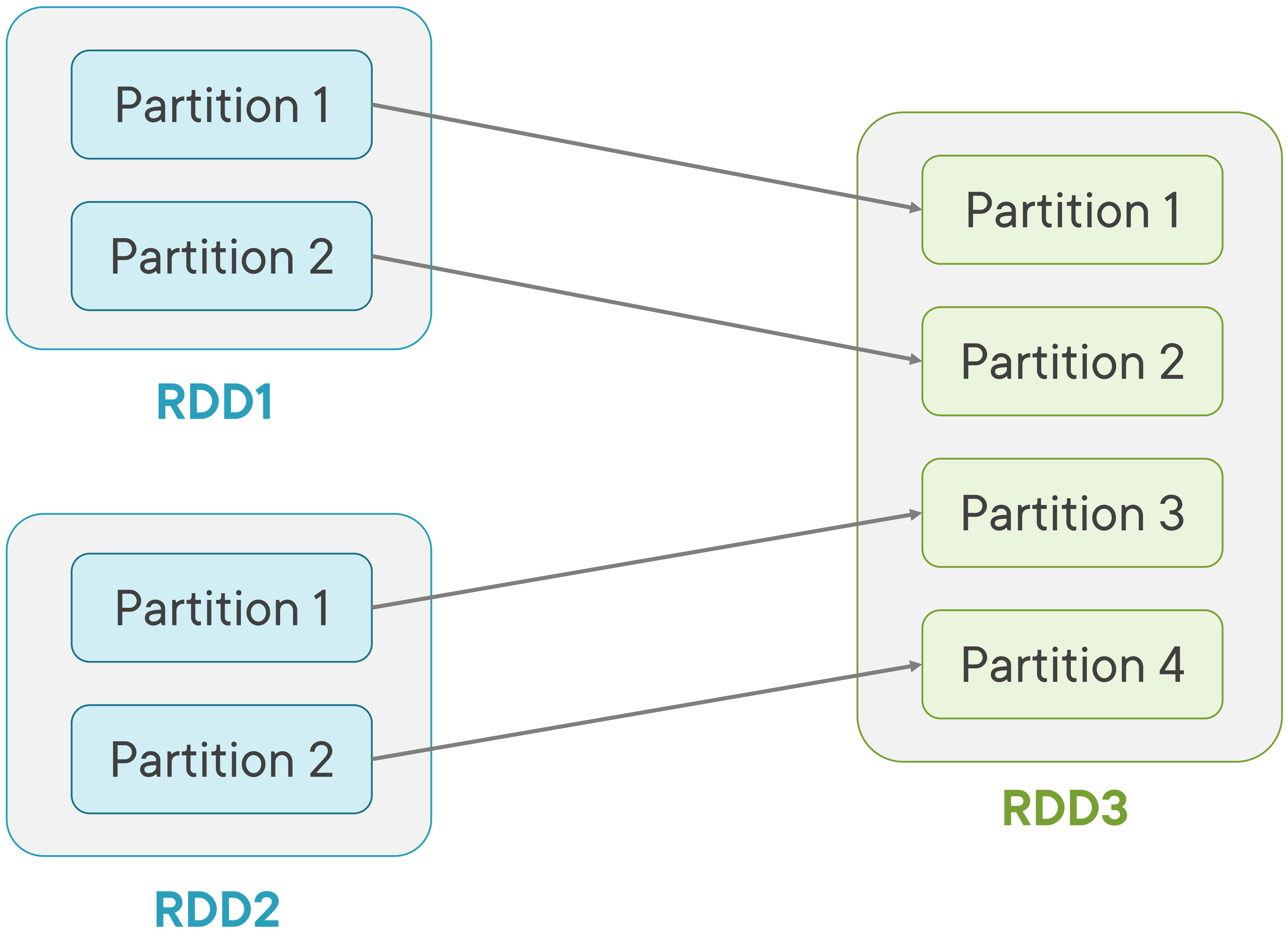
Partition 2

Map is a Narrow Transformation

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

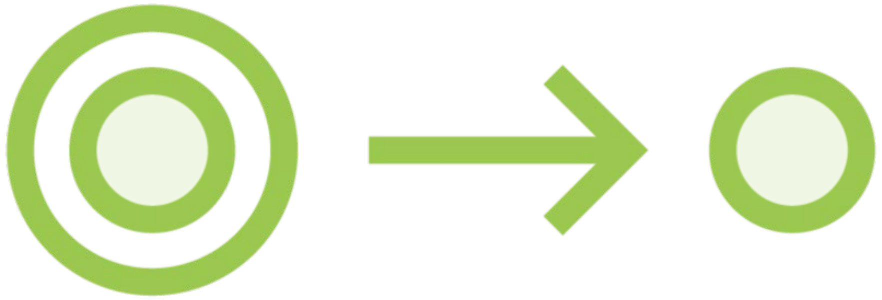
4	Seattle	400	4800
5	Paris	900	10800
6	Delhi	200	2400
7	Seattle	900	10800

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

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

Task 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

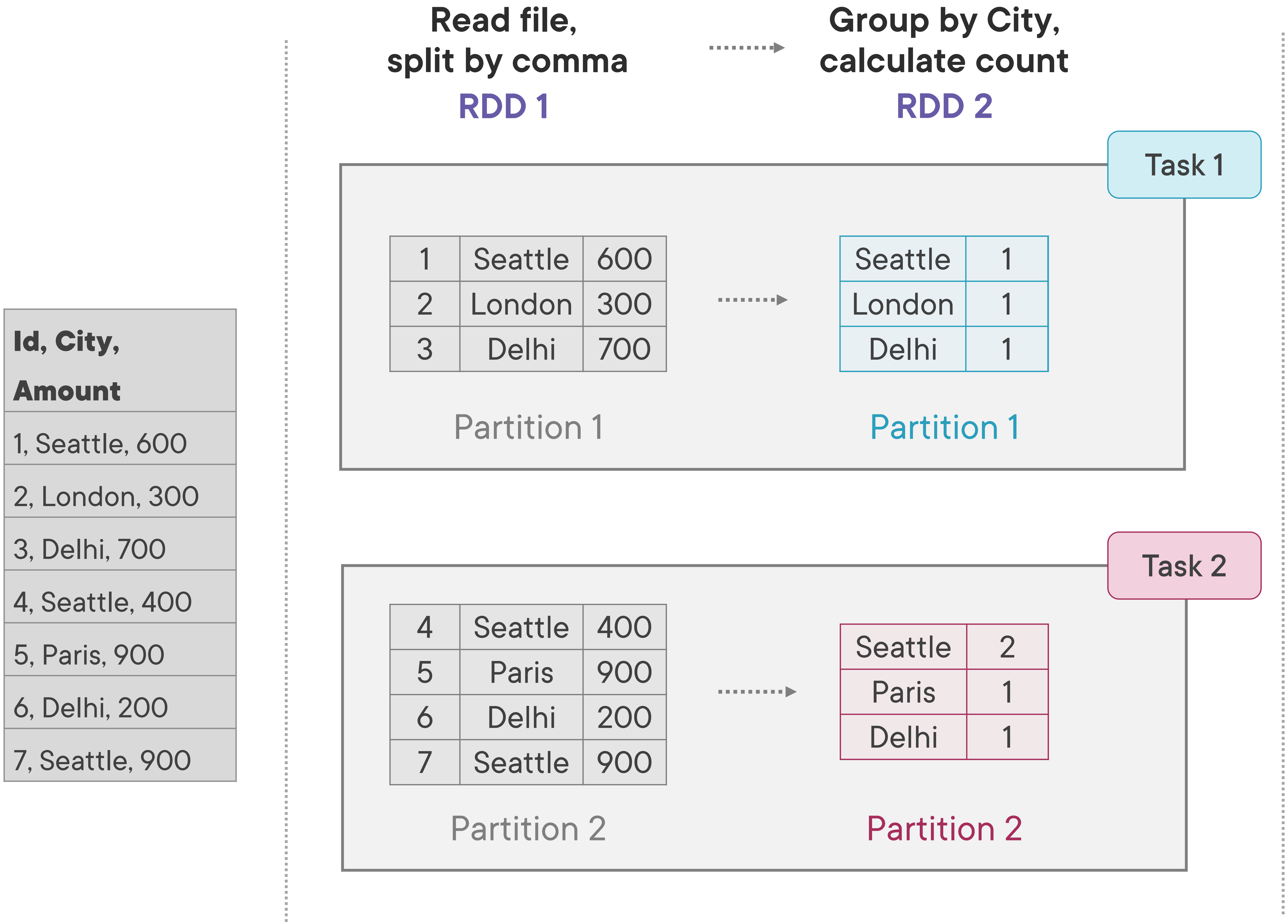
1	Seattle	600
2	London	300
3	Delhi	700

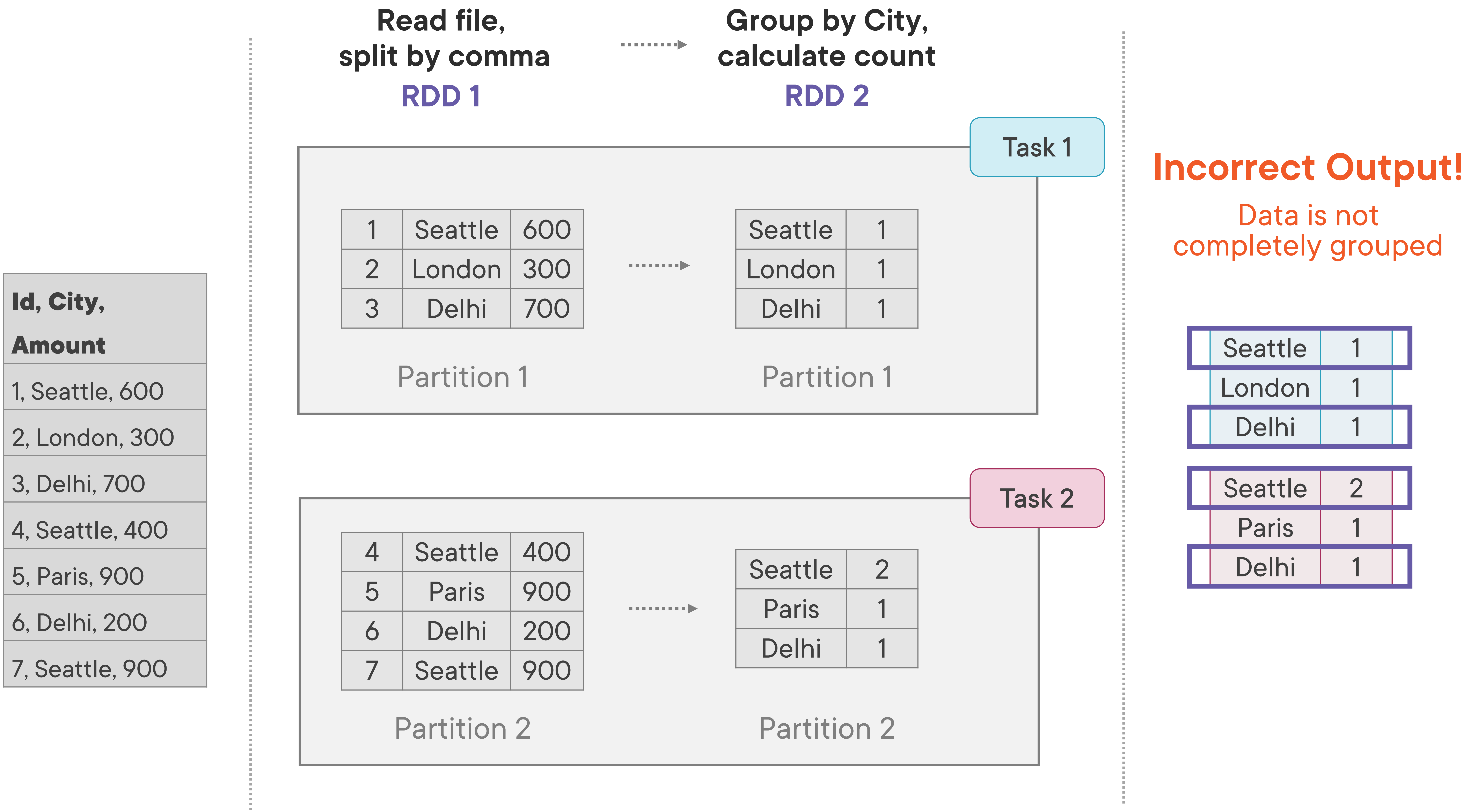
Partition 1

Task 2

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

Partition 2


Read file,
split by comma
RDD 1
Group by City,
calculate count
RDD 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
Task 1
1 Seattle 600
2 London 300
3 Delhi 700
Seattle 1
London 1
Delhi 1
Partition 1
Partition 1
Task 2
4 Seattle 400
5 Paris 900
6 Delhi 200
7 Seattle 900
Seattle 2
Paris 1
Delhi 1
Partition 2
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

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

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Group by City, calculate sum of Amount
RDD 2

Step 1 – Group: Shuffle / Exchange

Shuffle Write

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

Group by City, calculate sum of Amount

RDD 2

Step 1 – Group: Shuffle / Exchange

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

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Group by City, calculate sum of Amount

RDD 2

Step 1 – Group: Shuffle / Exchange

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

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Group by City, calculate sum of Amount

RDD 2

Step 1 – Group: Shuffle / Exchange

Step 2 – Sum

Seattle	600
London	300
Delhi	700

Shuffle Block 1

Seattle	1300
Paris	900
Delhi	200

Shuffle Block 2

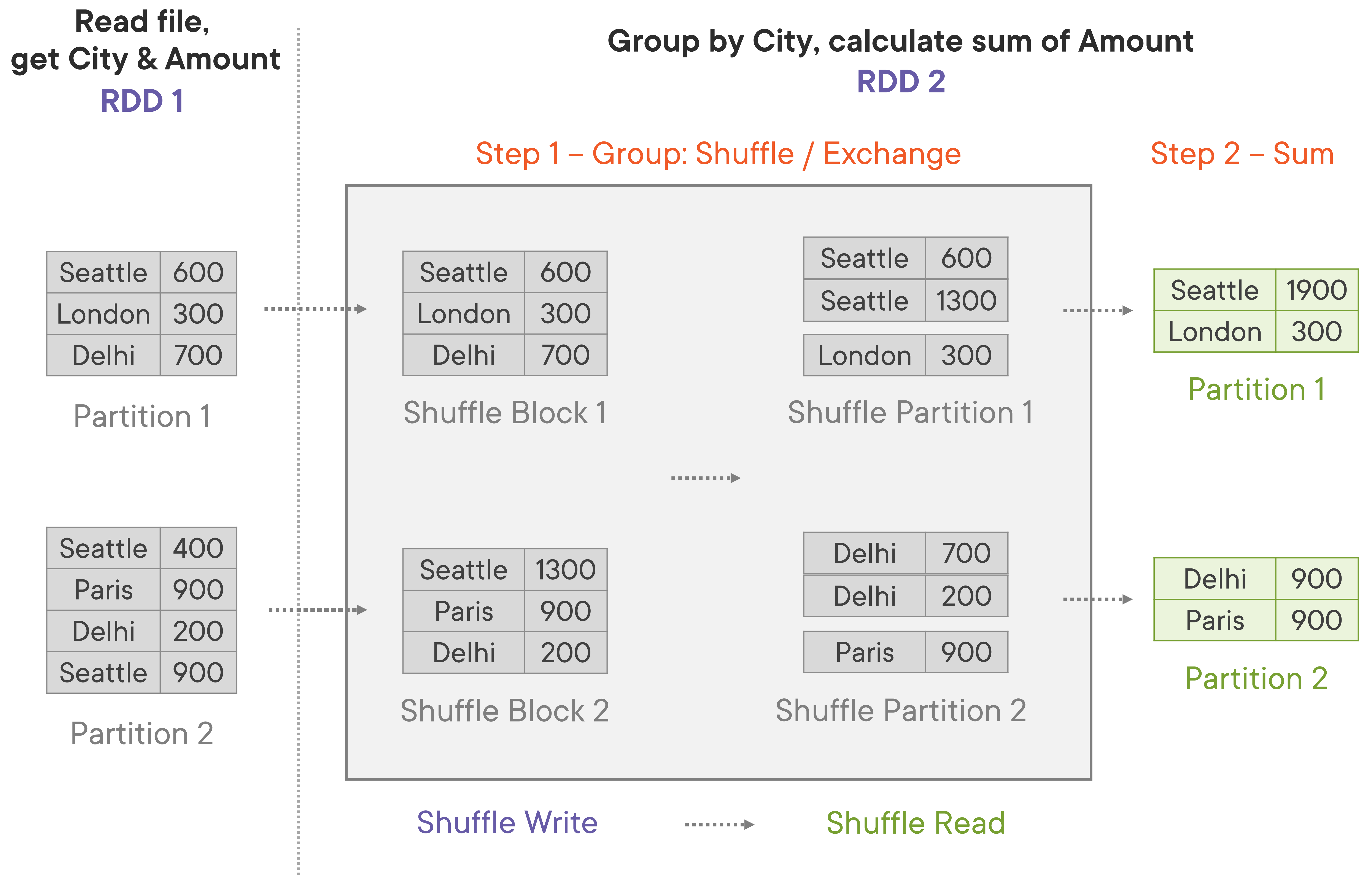
Seattle	600
Seattle	1300
London	300

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

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

Partition 2

Group by City, calculate sum of Amount

RDD 2

Step 1 – Group: Shuffle / Exchange

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

Step 2 – Sum

Seattle	1900
London	300

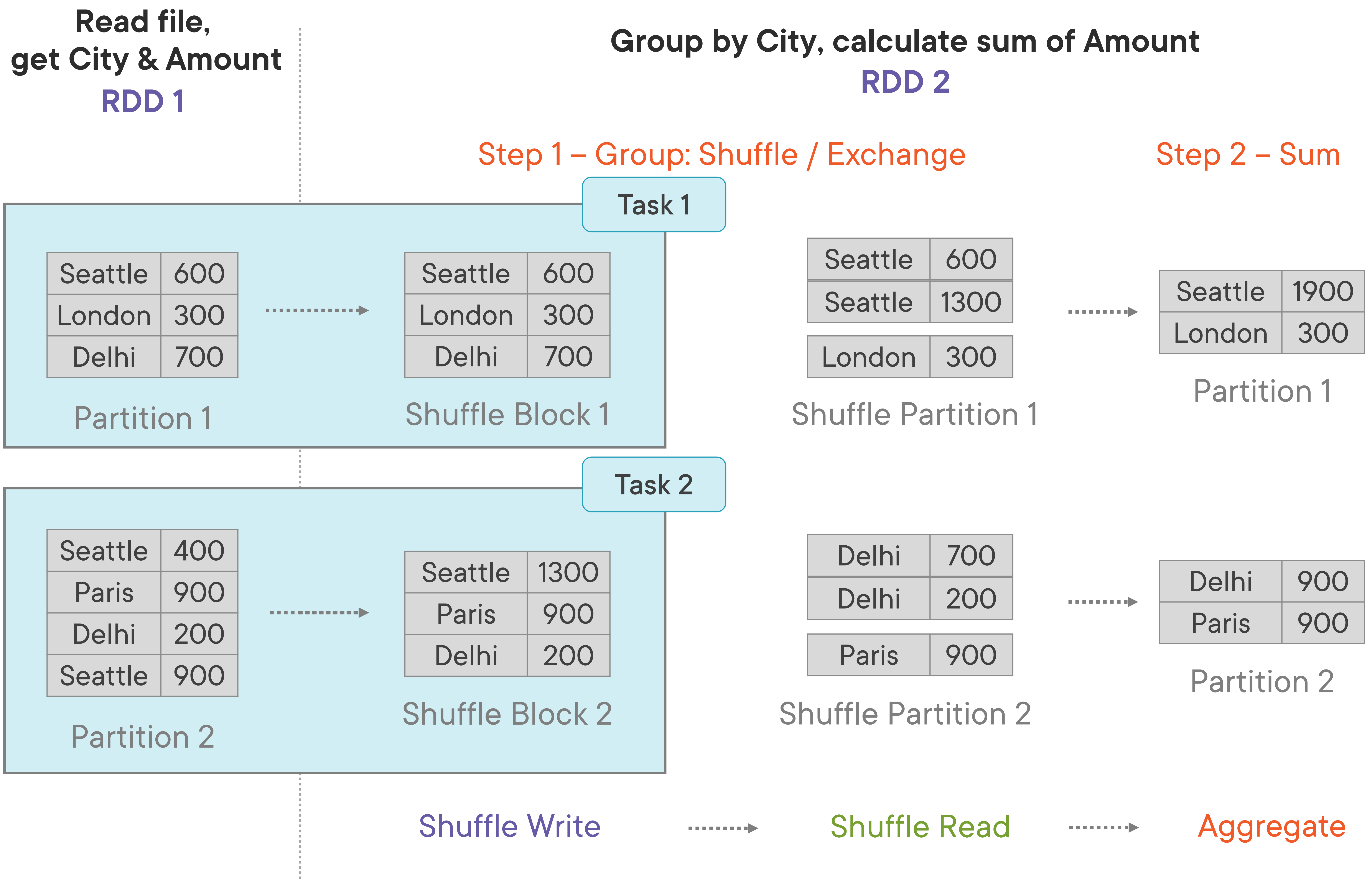
Partition 1

Delhi	900
Paris	900

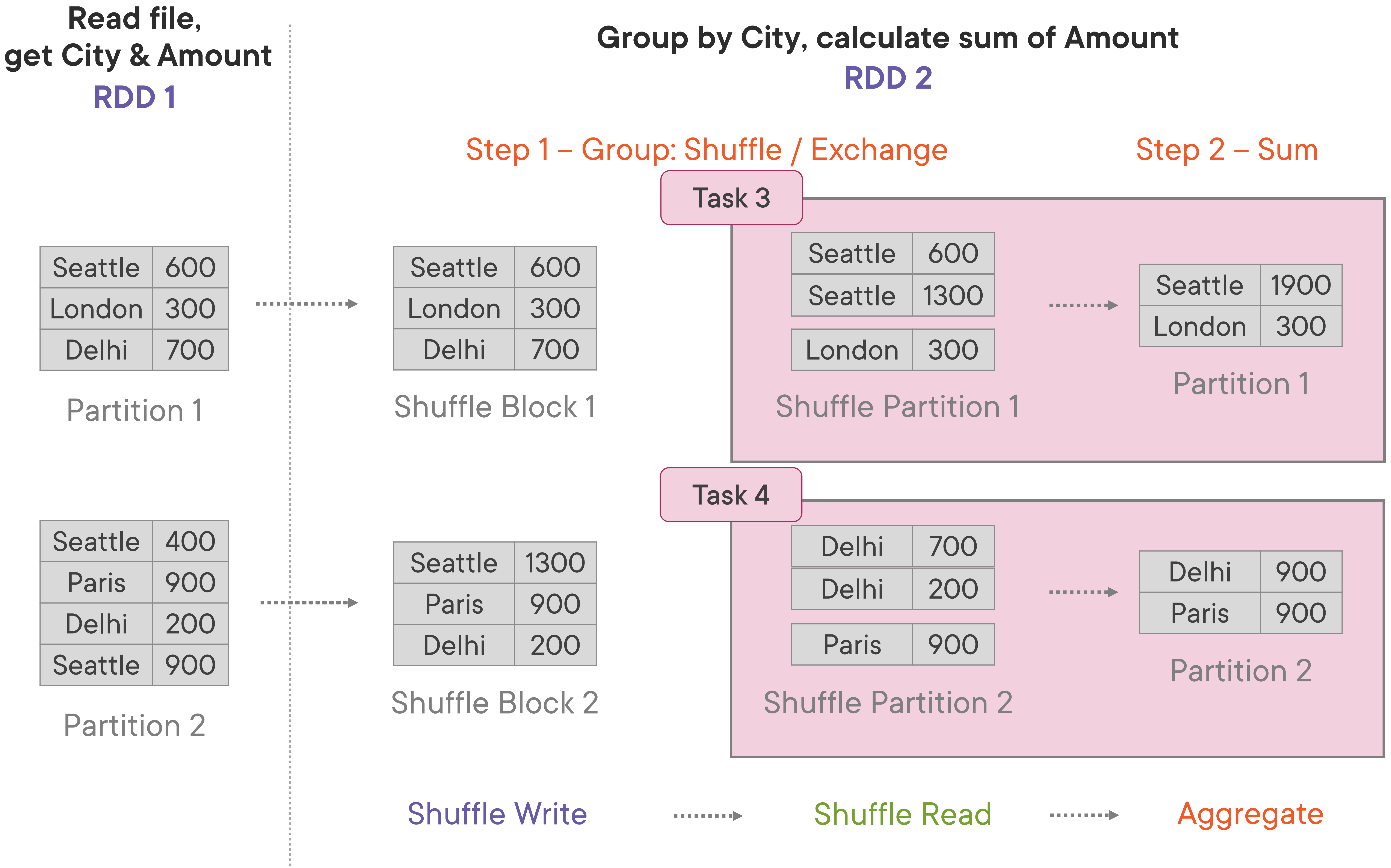
Partition 2

Correct Output!

Seattle	1900
London	300
Delhi	900
Paris	900



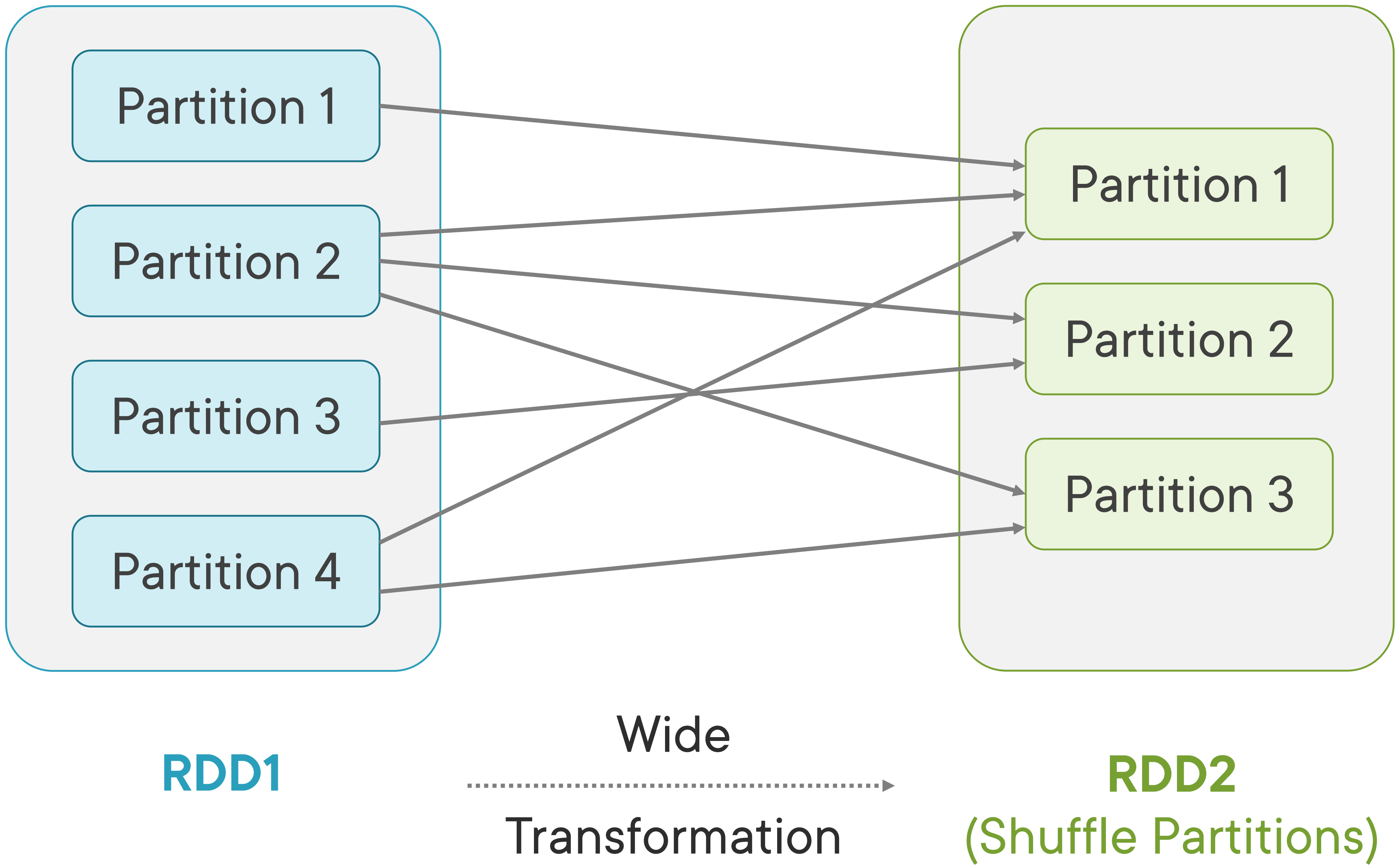
Read file,
get City & Amount
RDD 1
Group by City, calculate sum of Amount
RDD 2
Step 1 – Group: Shuffle / Exchange
Step 2 – Sum
Task 1
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 600
London 300
Delhi 700
Shuffle Block 1
Seattle 600
Seattle 1300
London 300
Shuffle Partition 1
Seattle 1900
London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

Seattle	600
London	300
Delhi	700

Partition 1

Seattle	400
Paris	900
Delhi	200
Seattle	900

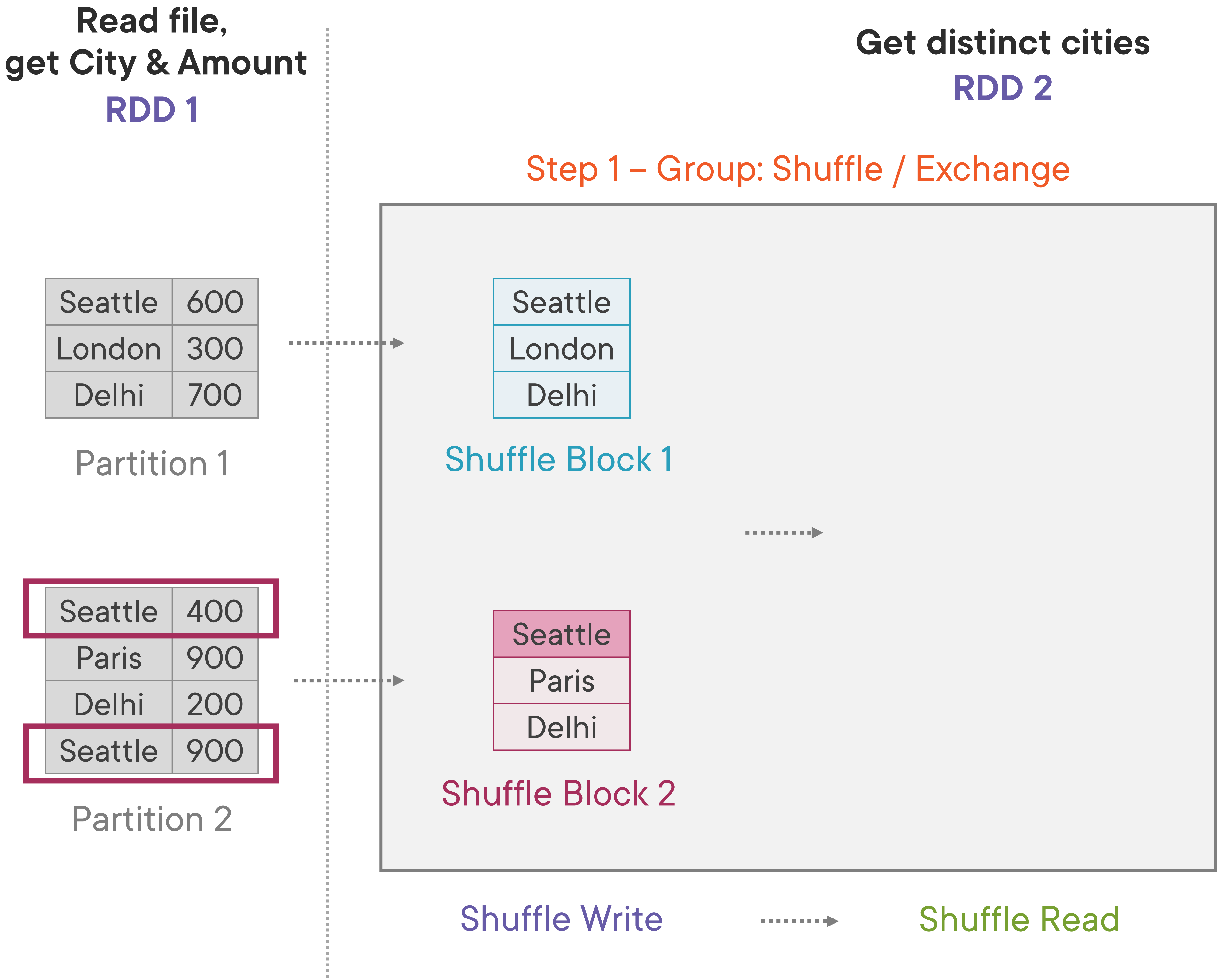
Partition 2

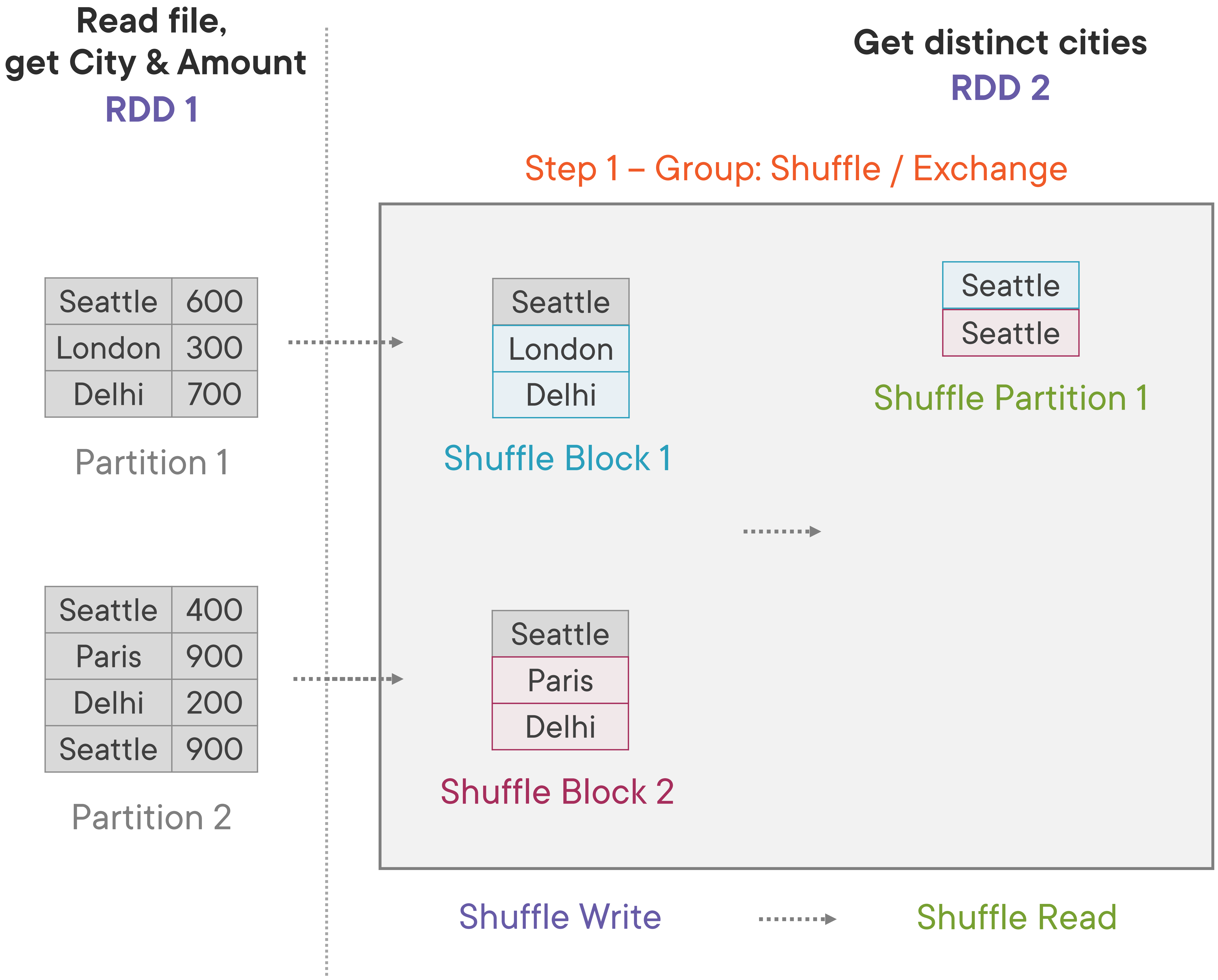
Get distinct cities

RDD 2

Step 1 – Group: Shuffle / Exchange

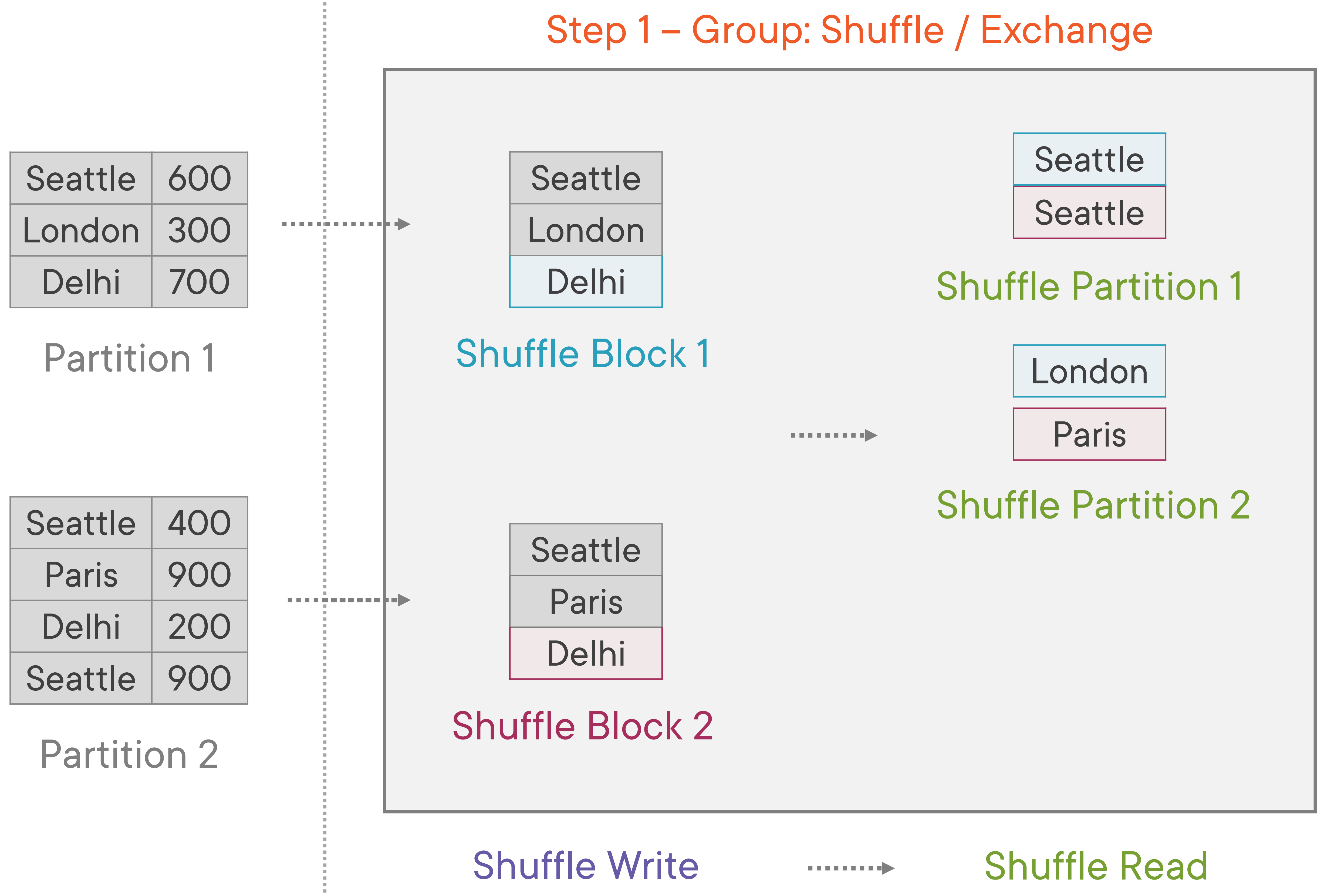
Shuffle Write

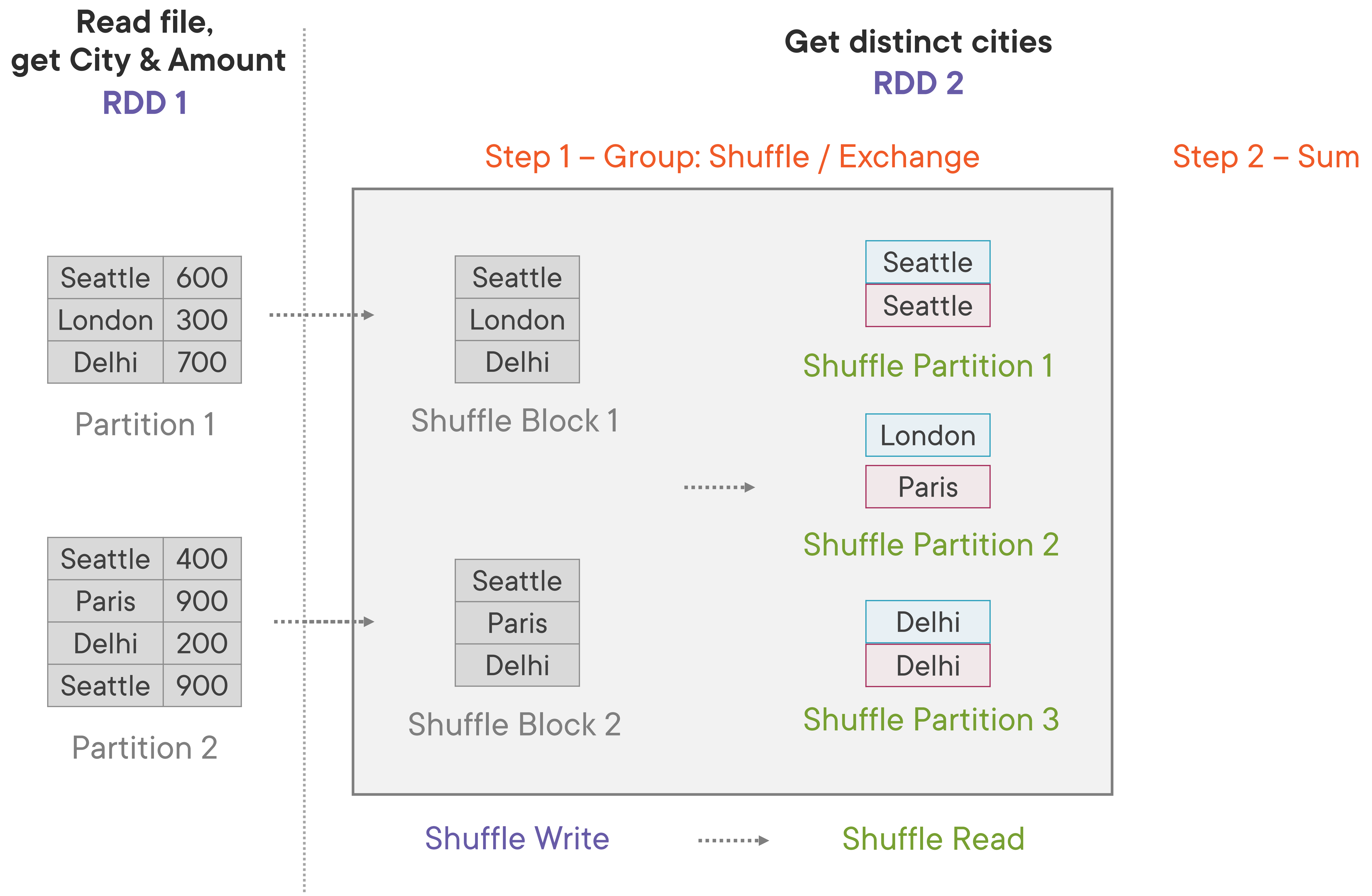

Read file,
get City & Amount
RDD 1
Get distinct cities
RDD 2
Step 1 – Group: Shuffle / Exchange
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Shuffle Write
Shuffle Read

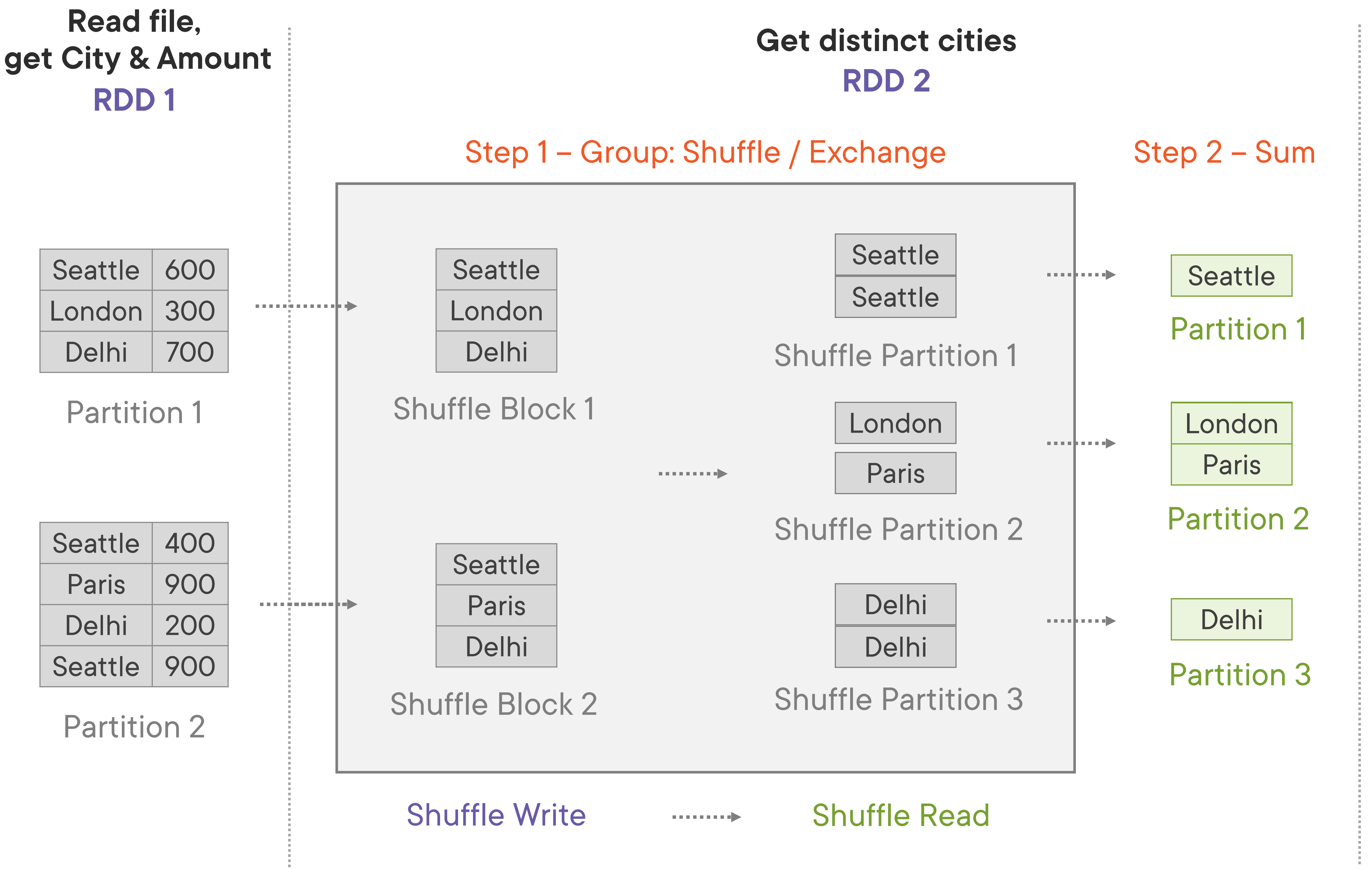

Read file,
get City & Amount
RDD 1
Get distinct cities
RDD 2
Step 1 – Group: Shuffle / Exchange
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle Partition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

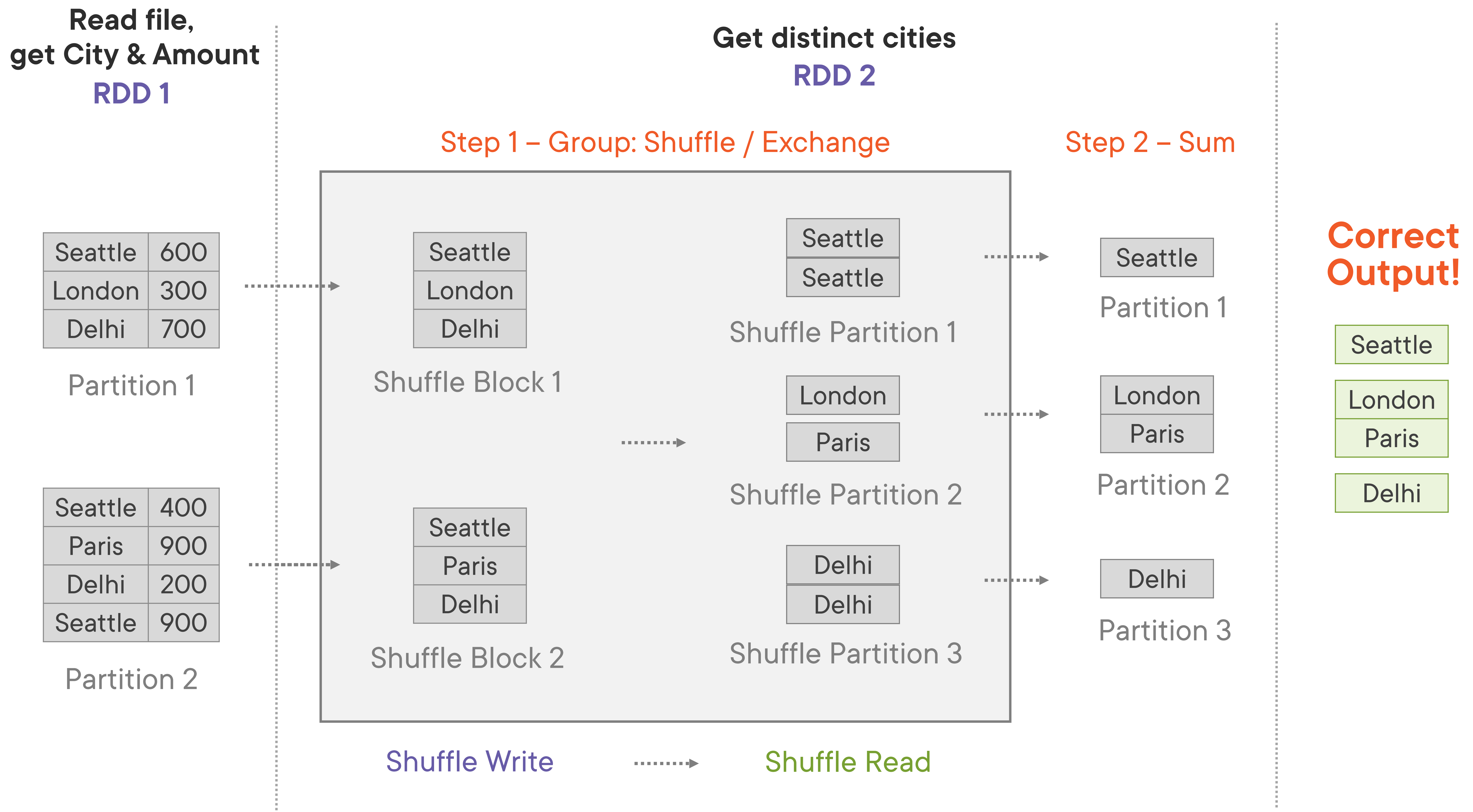
Read file, get City & Amount
RDD 1

Get distinct cities
RDD 2

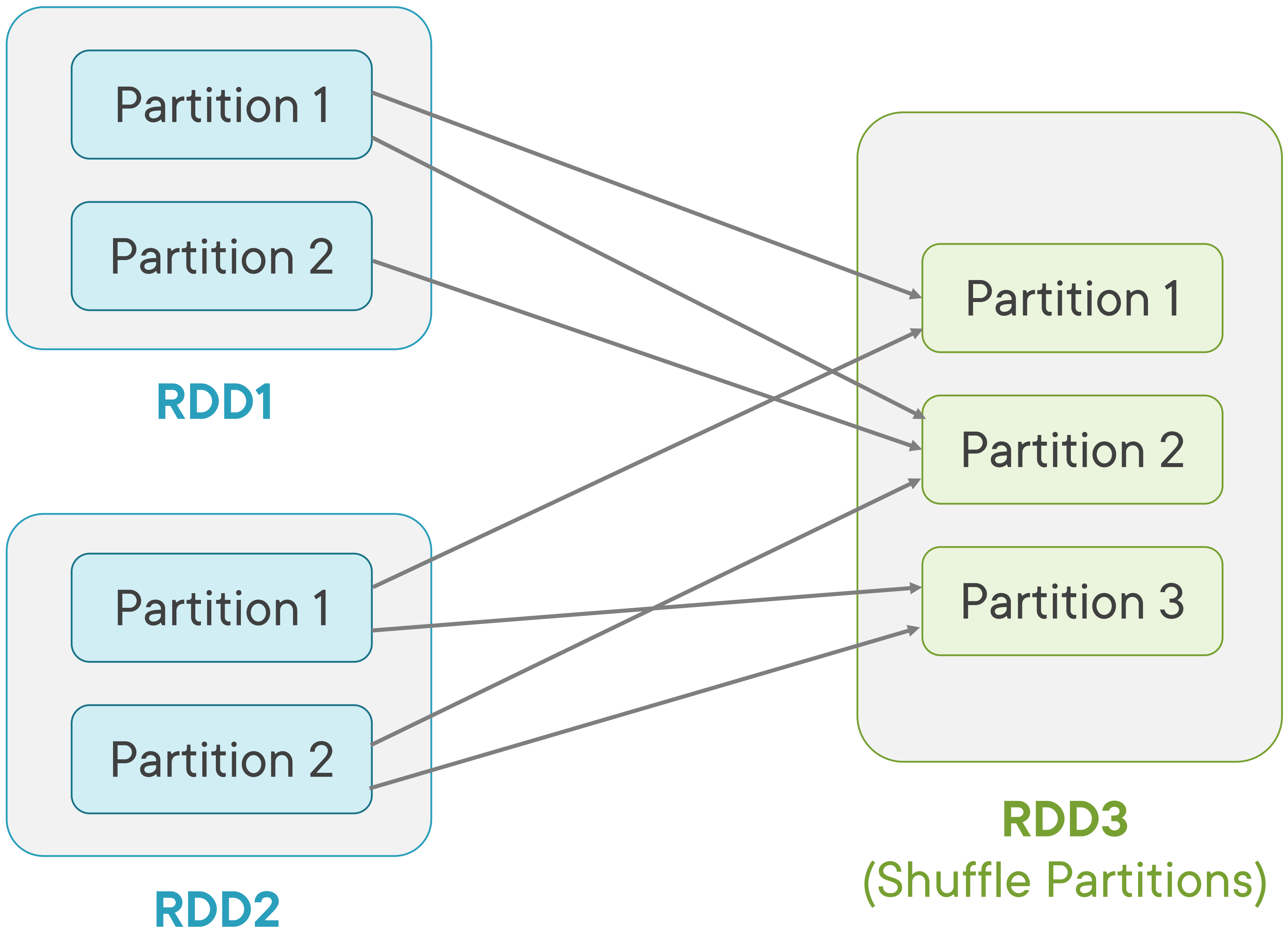



Read file,
get City & Amount
RDD 1
Get distinct cities
RDD 2
Step 1 – Group: Shuffle / Exchange
Step 2 – Sum
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle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

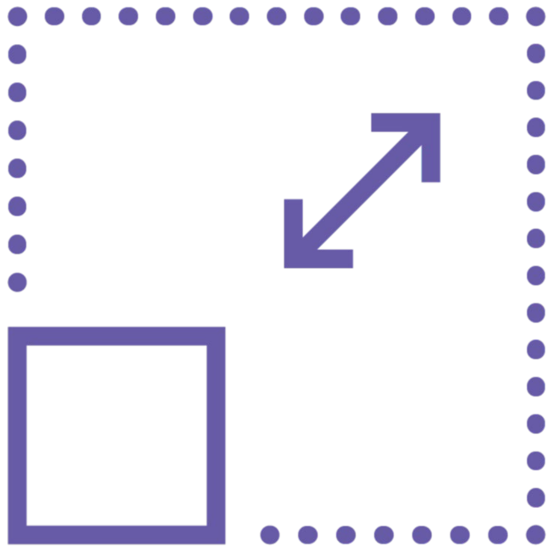

Read file,
get City & Amount
RDD 1
Get distinct cities
RDD 2
Step 1 – Group: Shuffle / Exchange
Step 2 – Sum
Seattle 600
London 300
Delhi 700
Partition 1
Seattle 400
Paris 900
Delhi 200
Seattle 900
Partition 2
Seattle
London
Delhi
Shuffle Block 1
Seattle
Paris
Delhi
Shuffle Block 2
Seattle
Seattle
Shuffle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

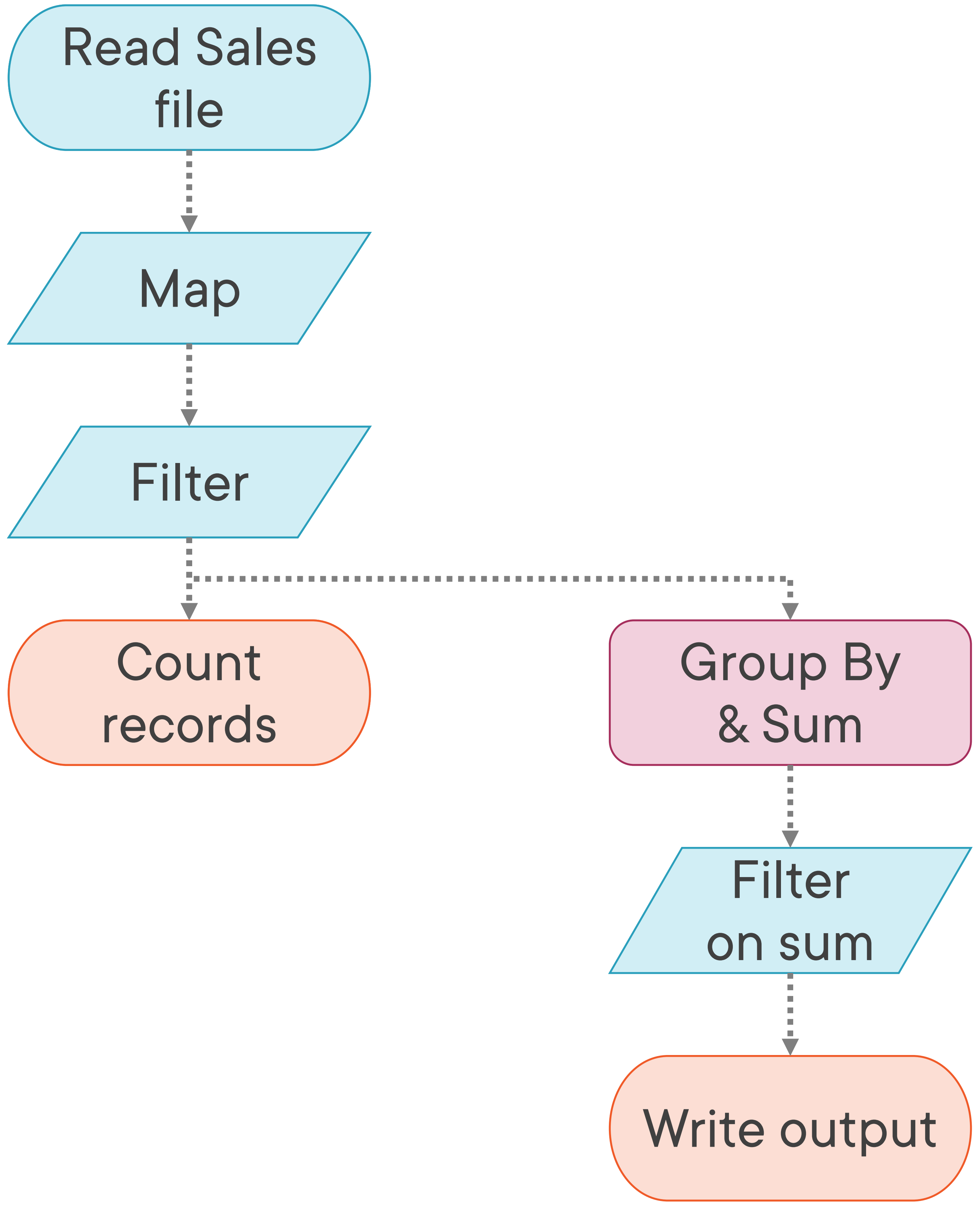
Number of shuffle partitions can be different than parent RDD partitions

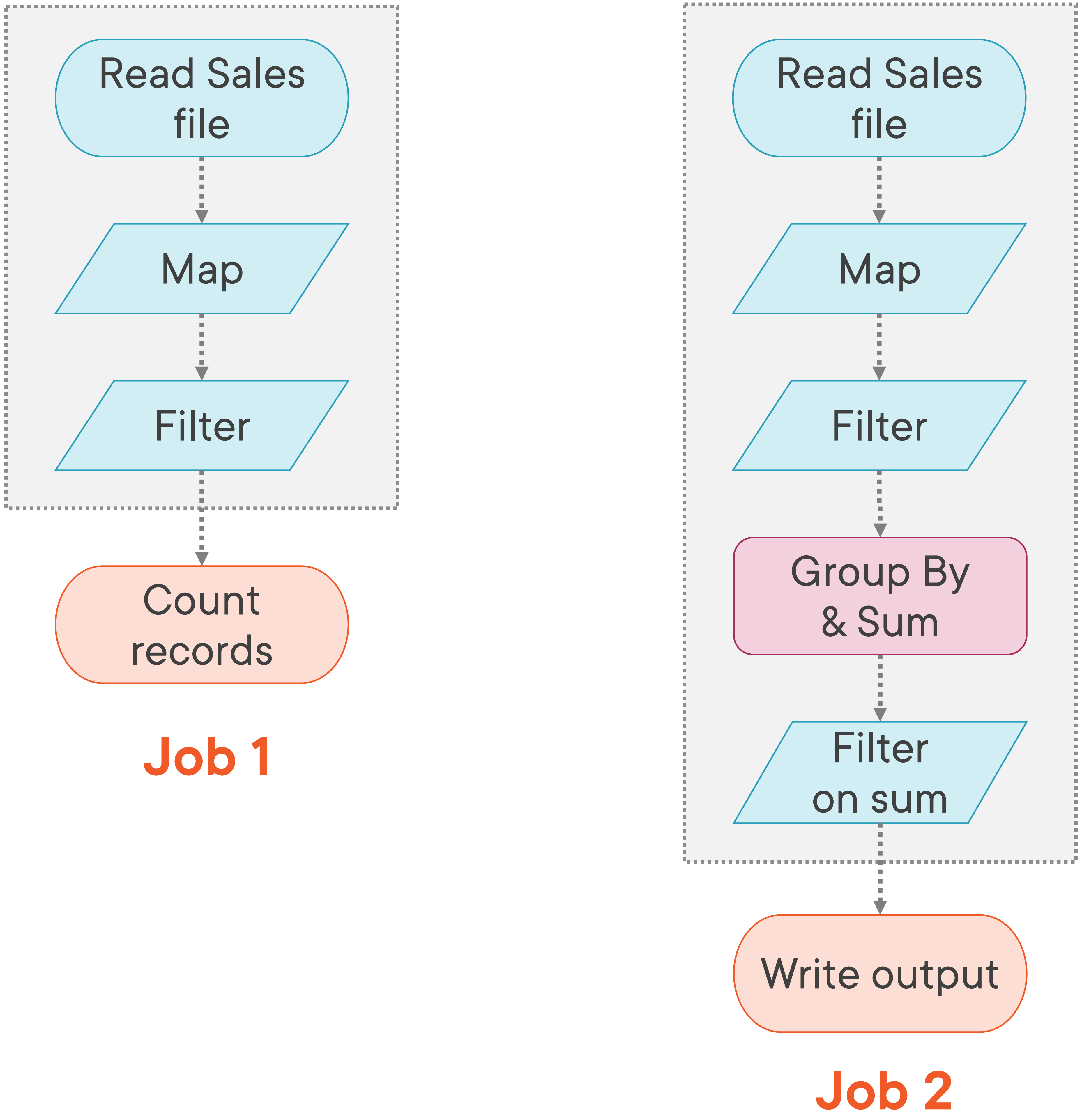
Examples

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

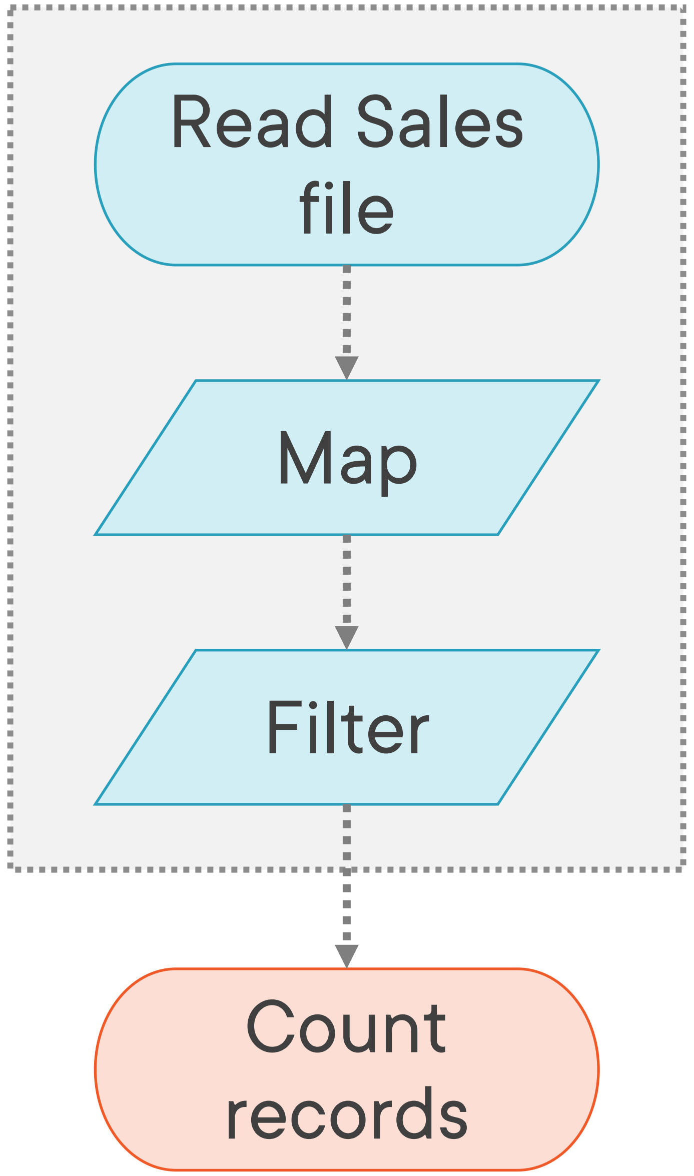
Spark Application Concepts: Jobs, Stages & Tasks

1. Read Sales csv file from Storage
2. Split Sales data by comma & define schema
3. Filter Sales where City = ‘Delhi’
4. Count number of Sales
5. Group by Product, calculate sum of Amount
6. Filter grouped data where TotalAmount > ‘10000’
7. Write processed data to storage

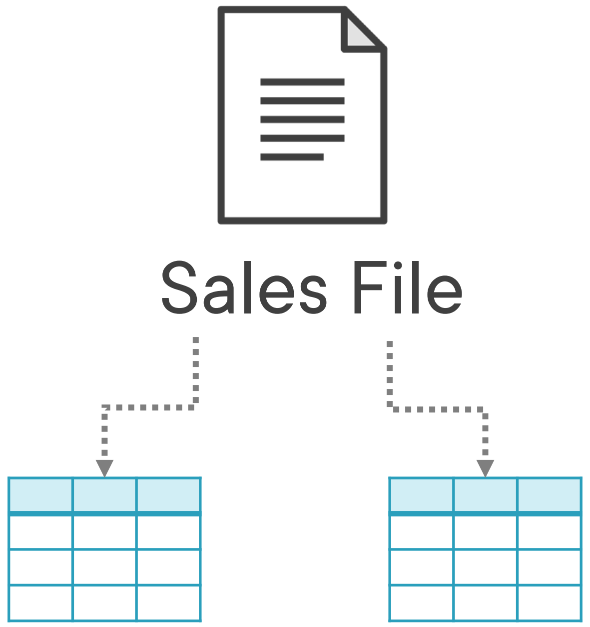


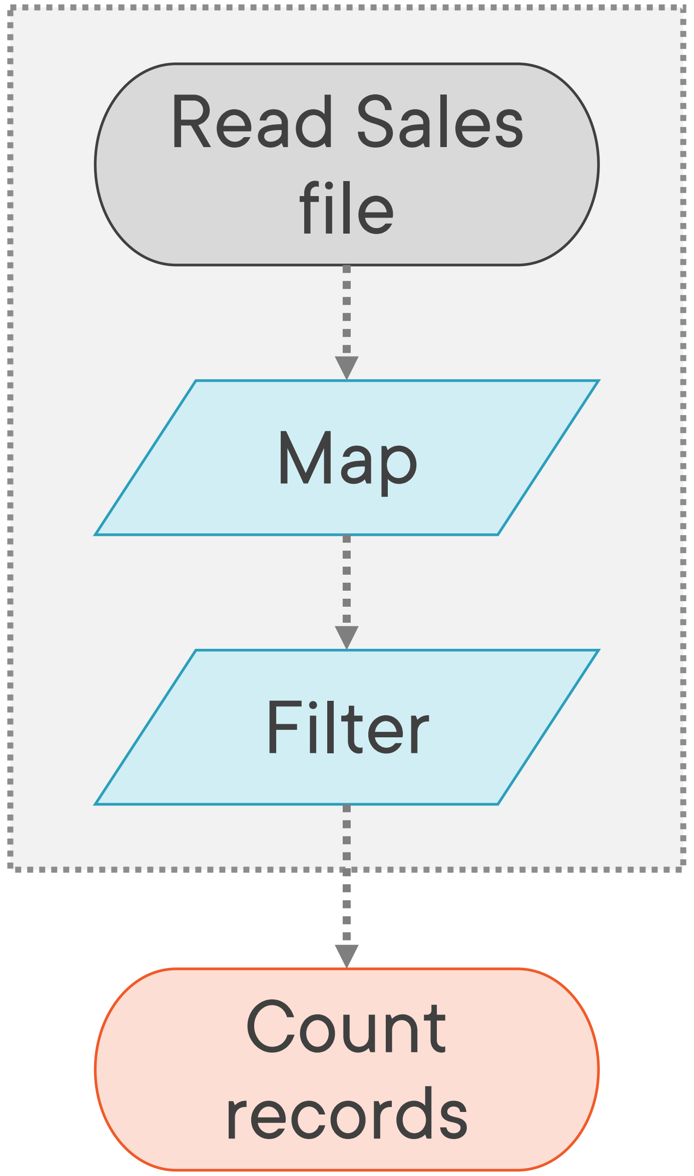


Jobs = Action Operations

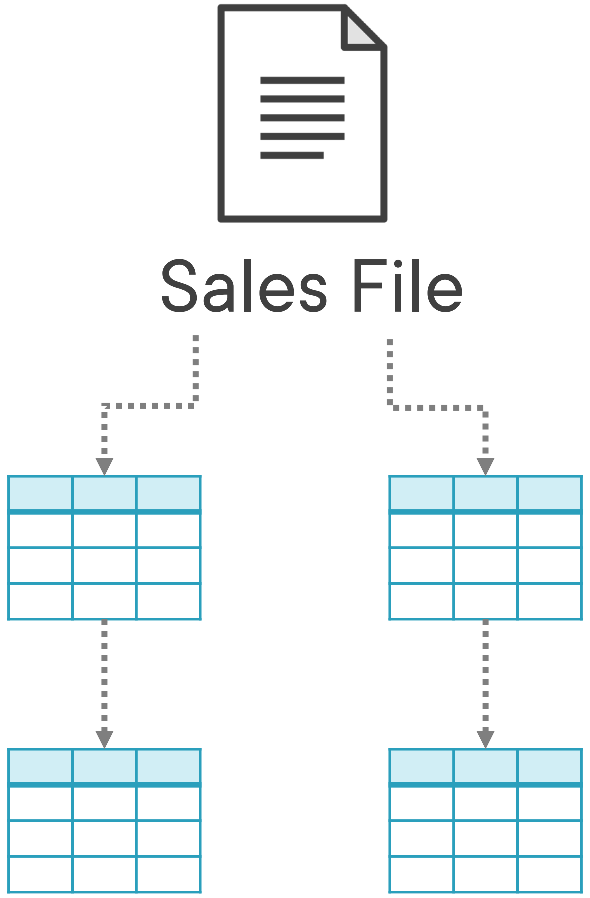


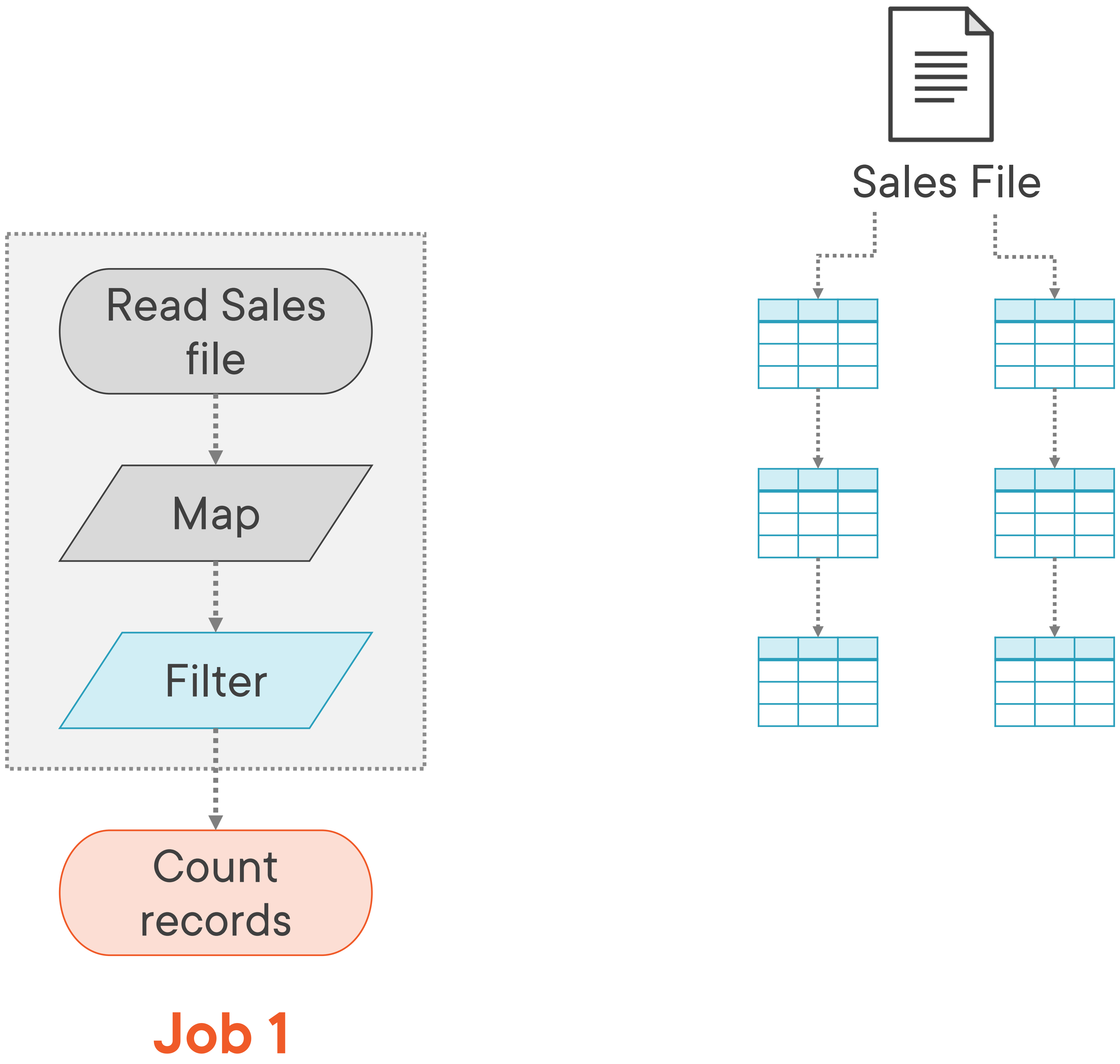
Job 1

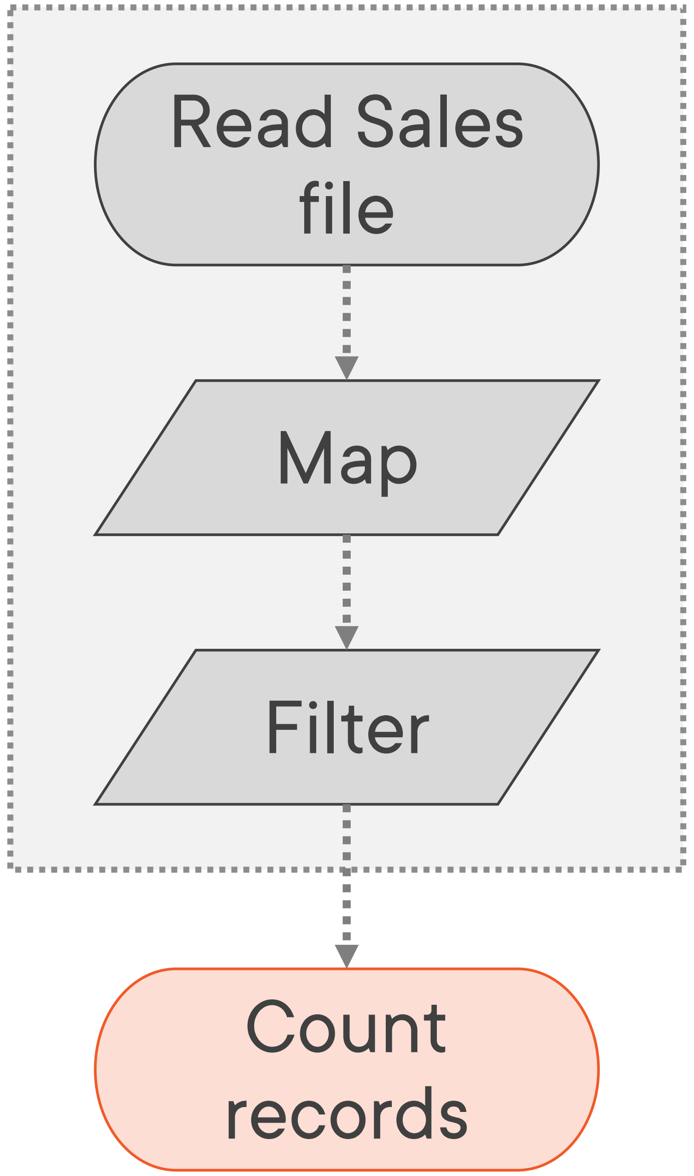




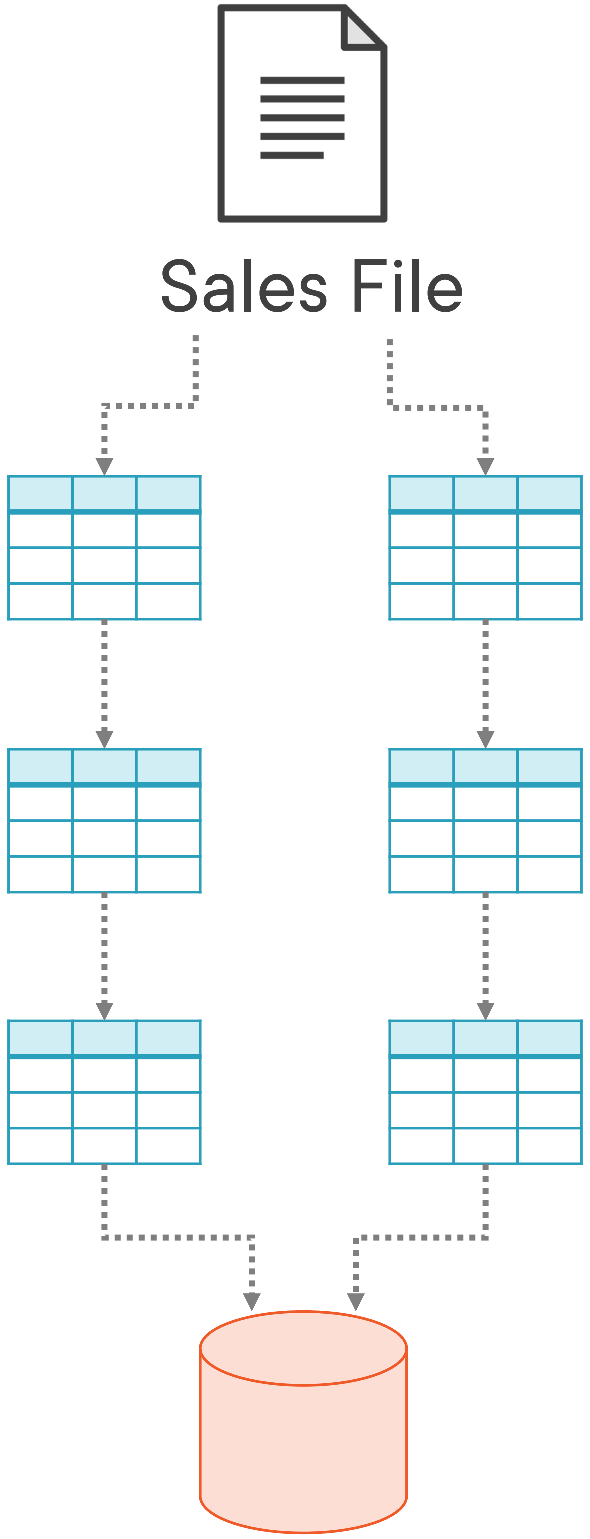
Job 1




Read Sales file
Map
Filter
Count records
Job 1
Sales File

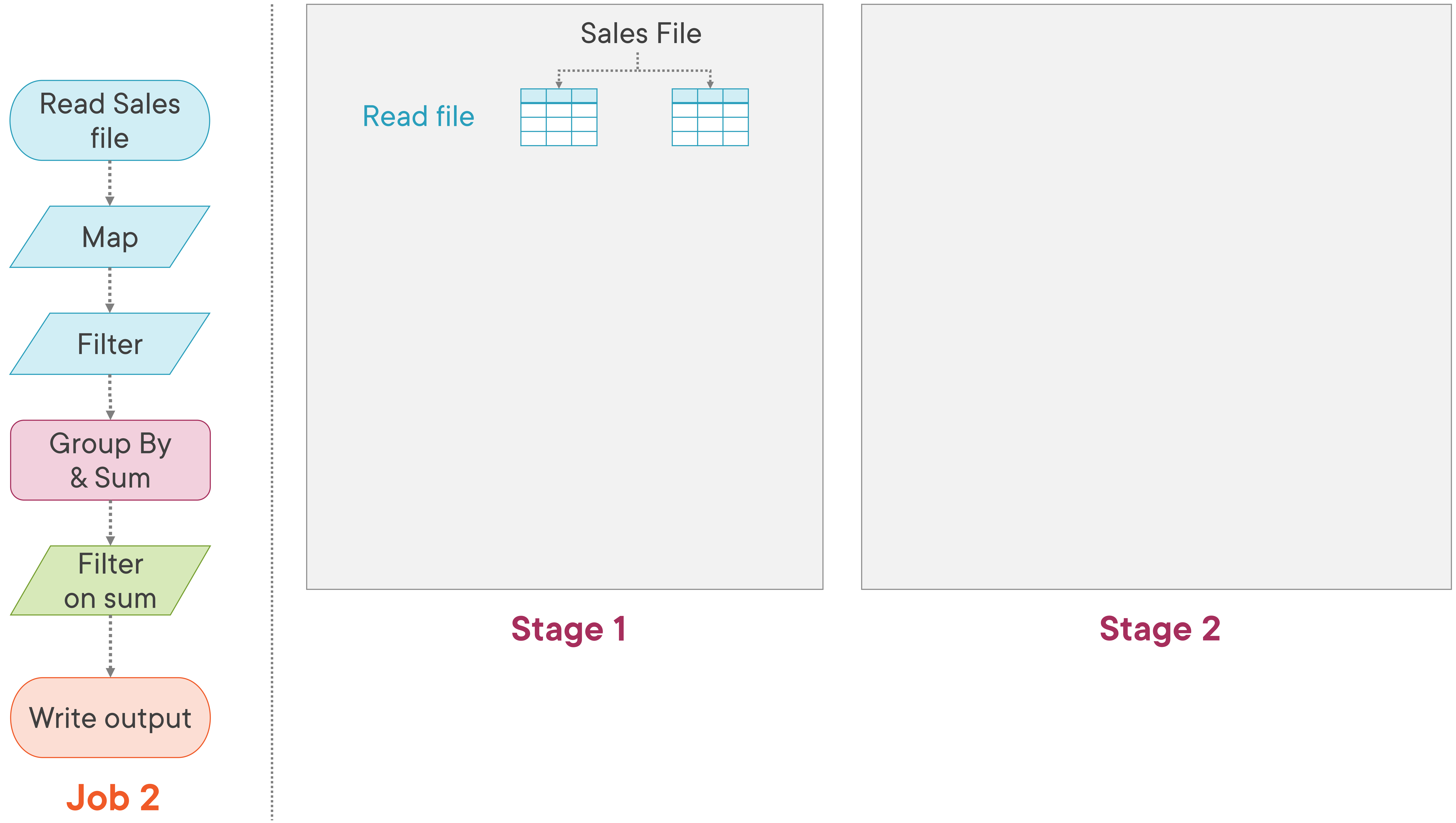


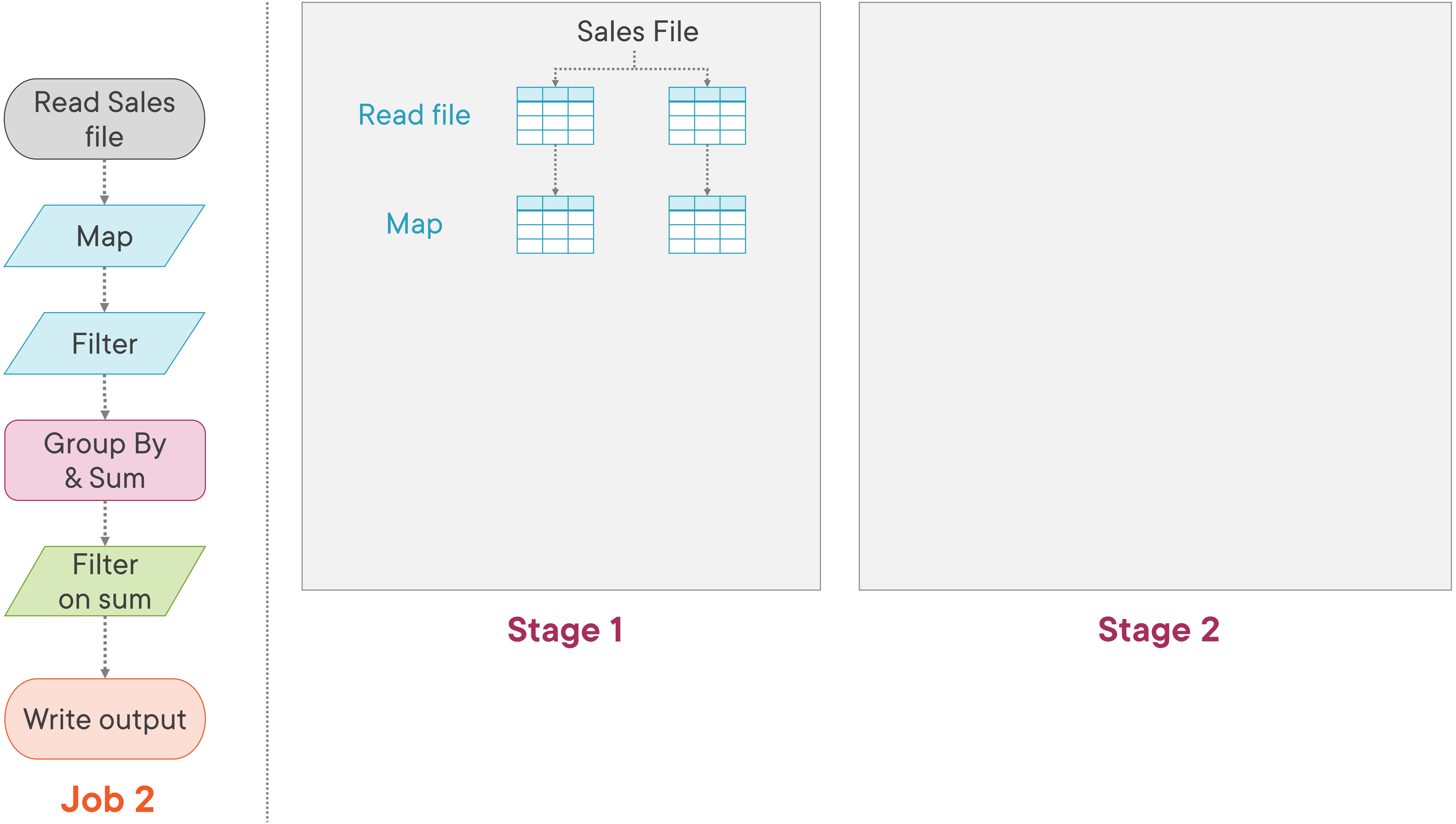
Job 1

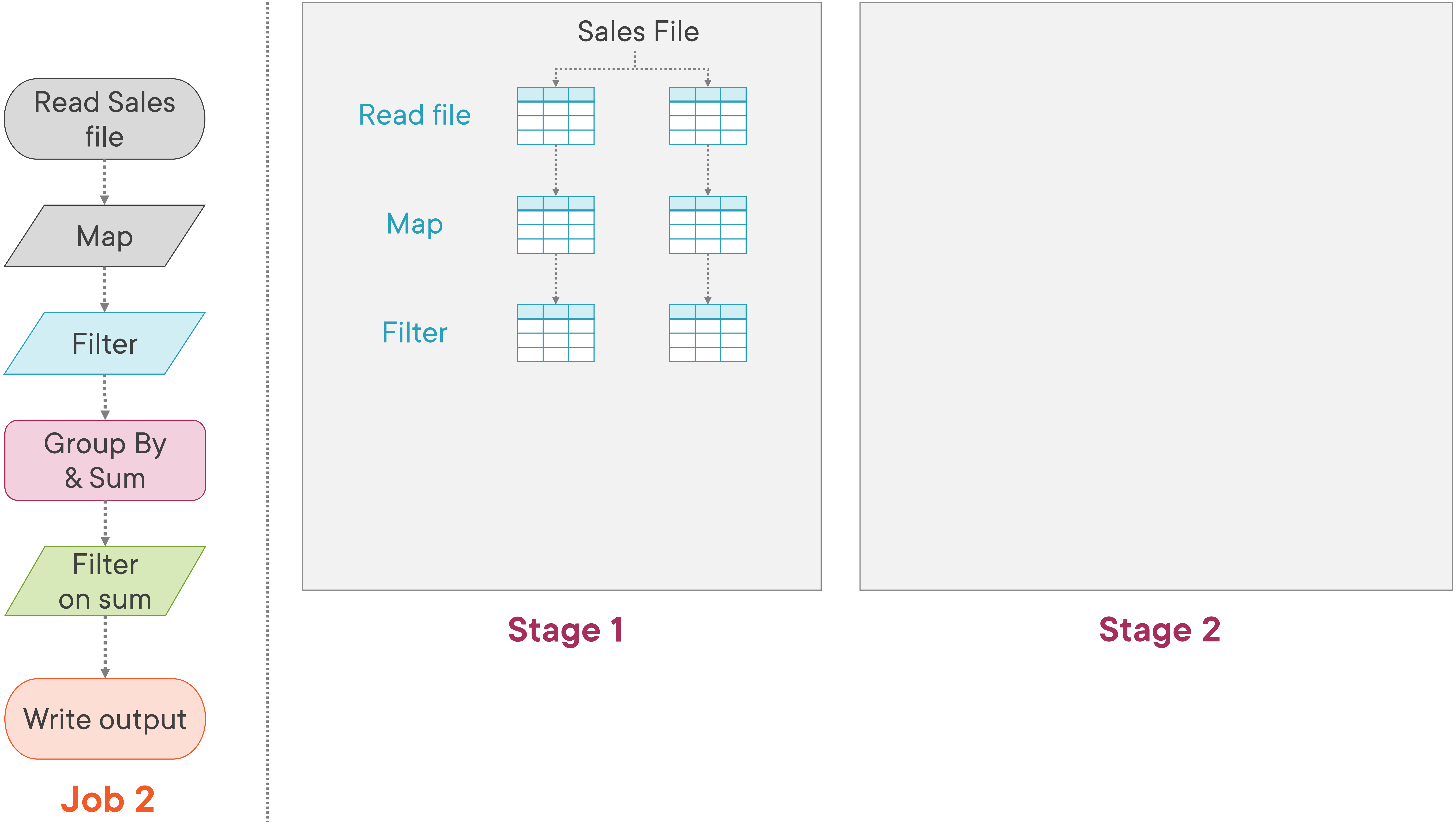


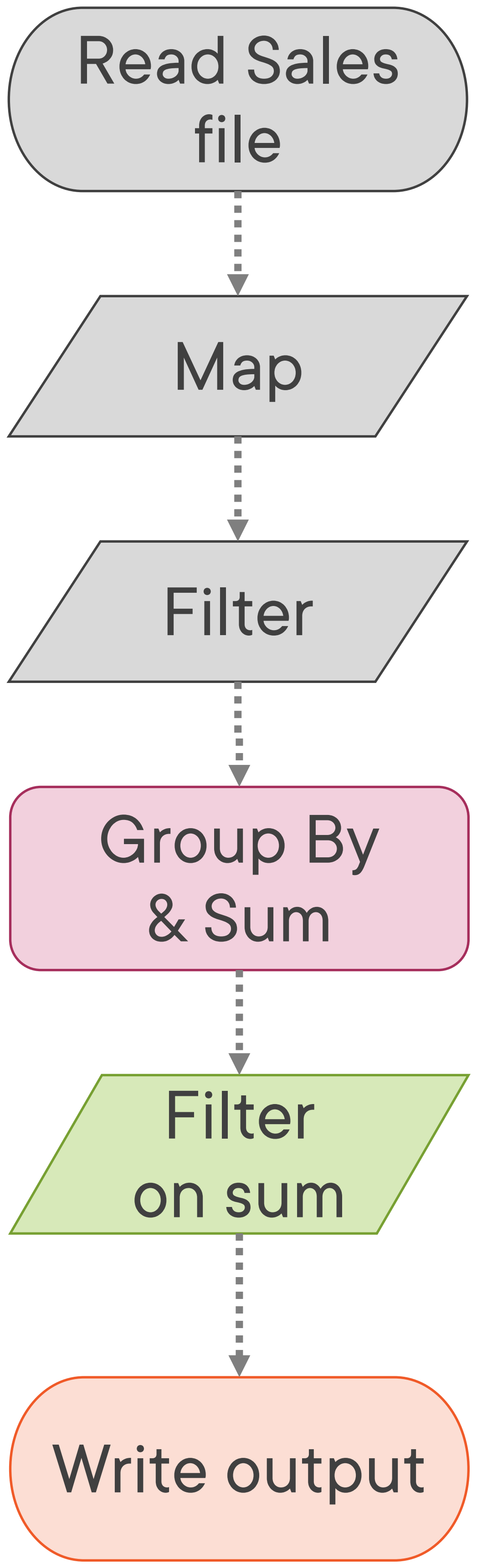
Narrow Transformation

Number of output partitions are typically the same as input partitions

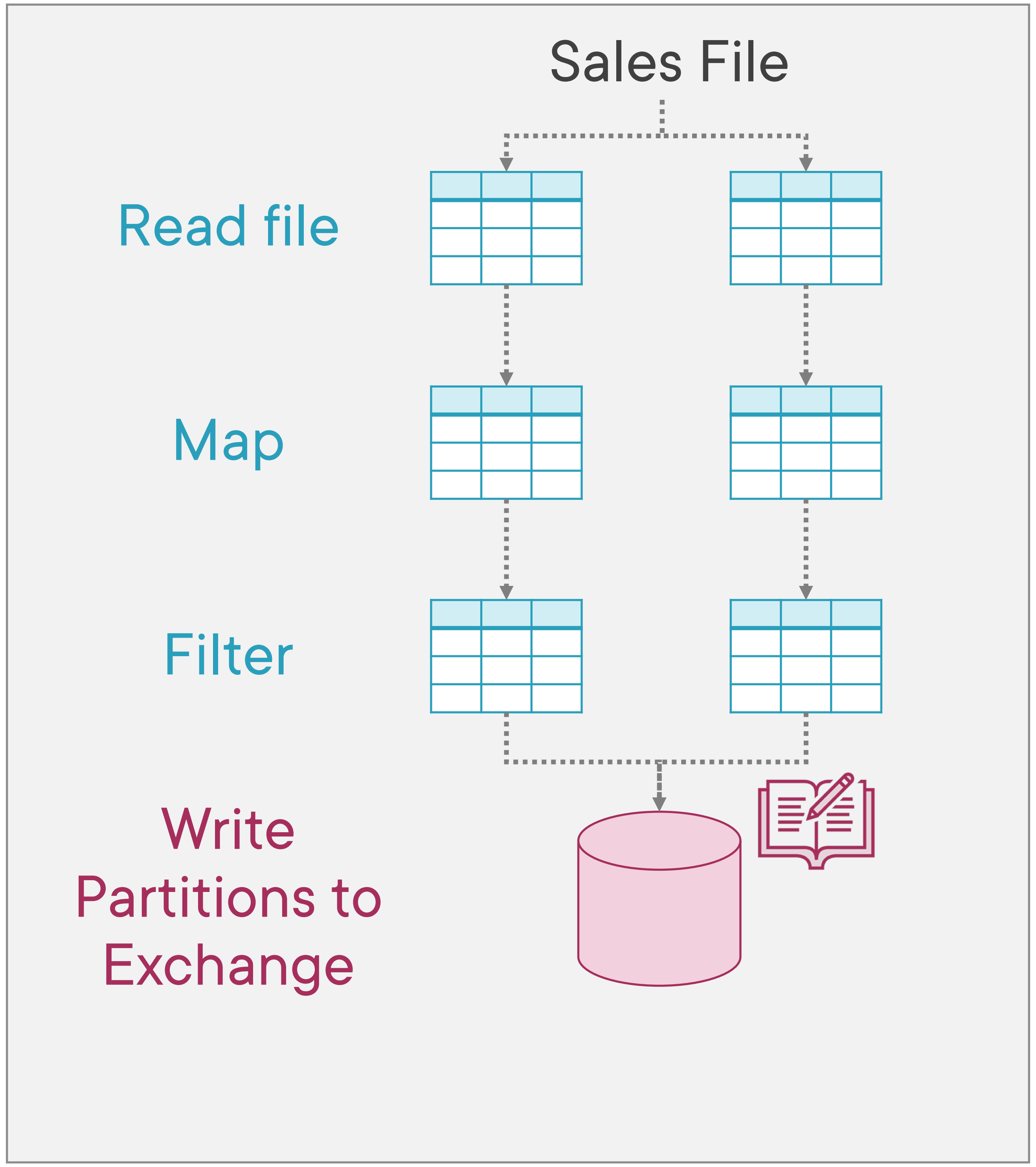

Read Sales file
Map
Filter
Group By & Sum
Filter on sum
Write output
Job 2
Sales File
Read file
Stage 1
Stage 2


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

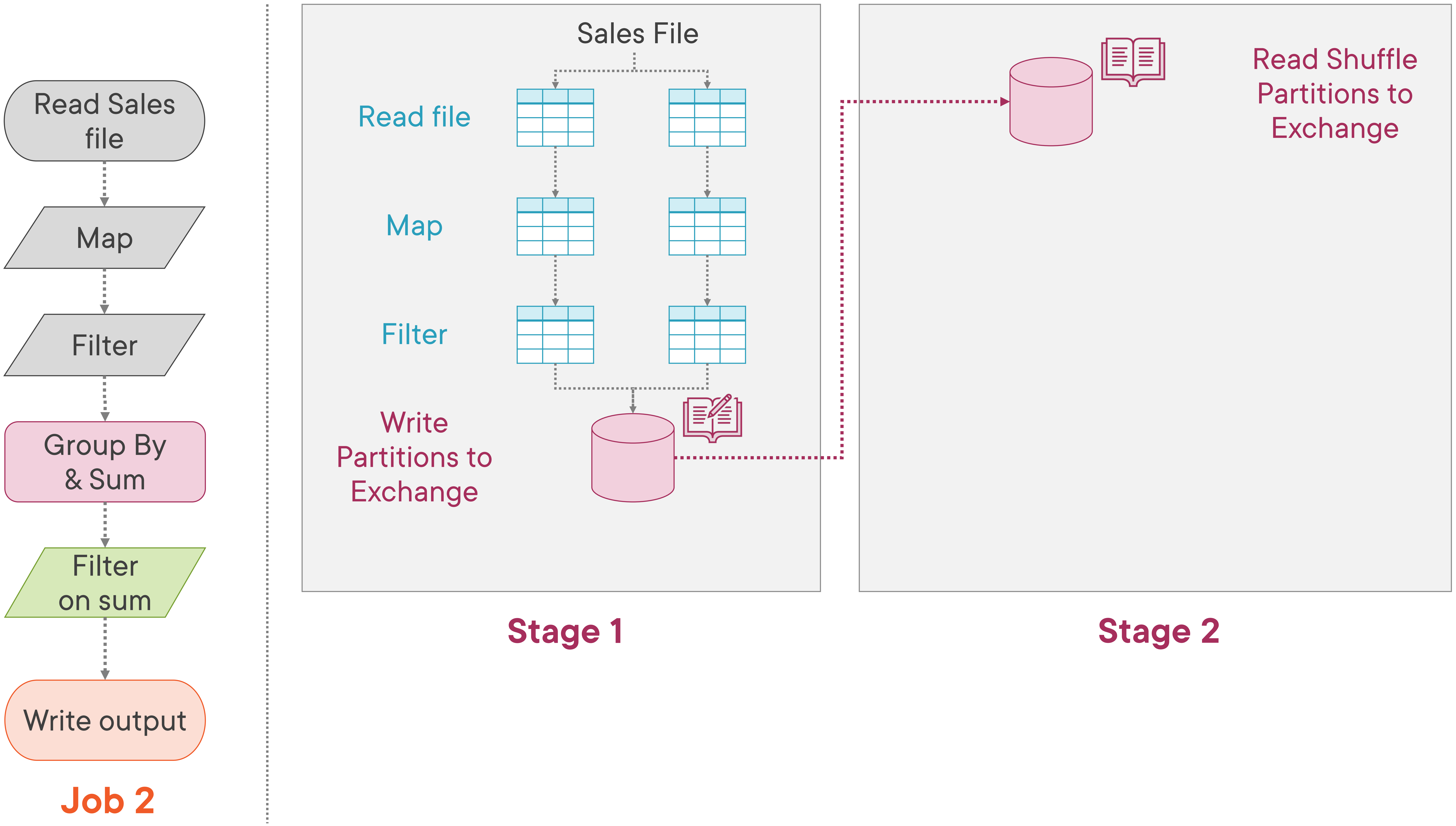


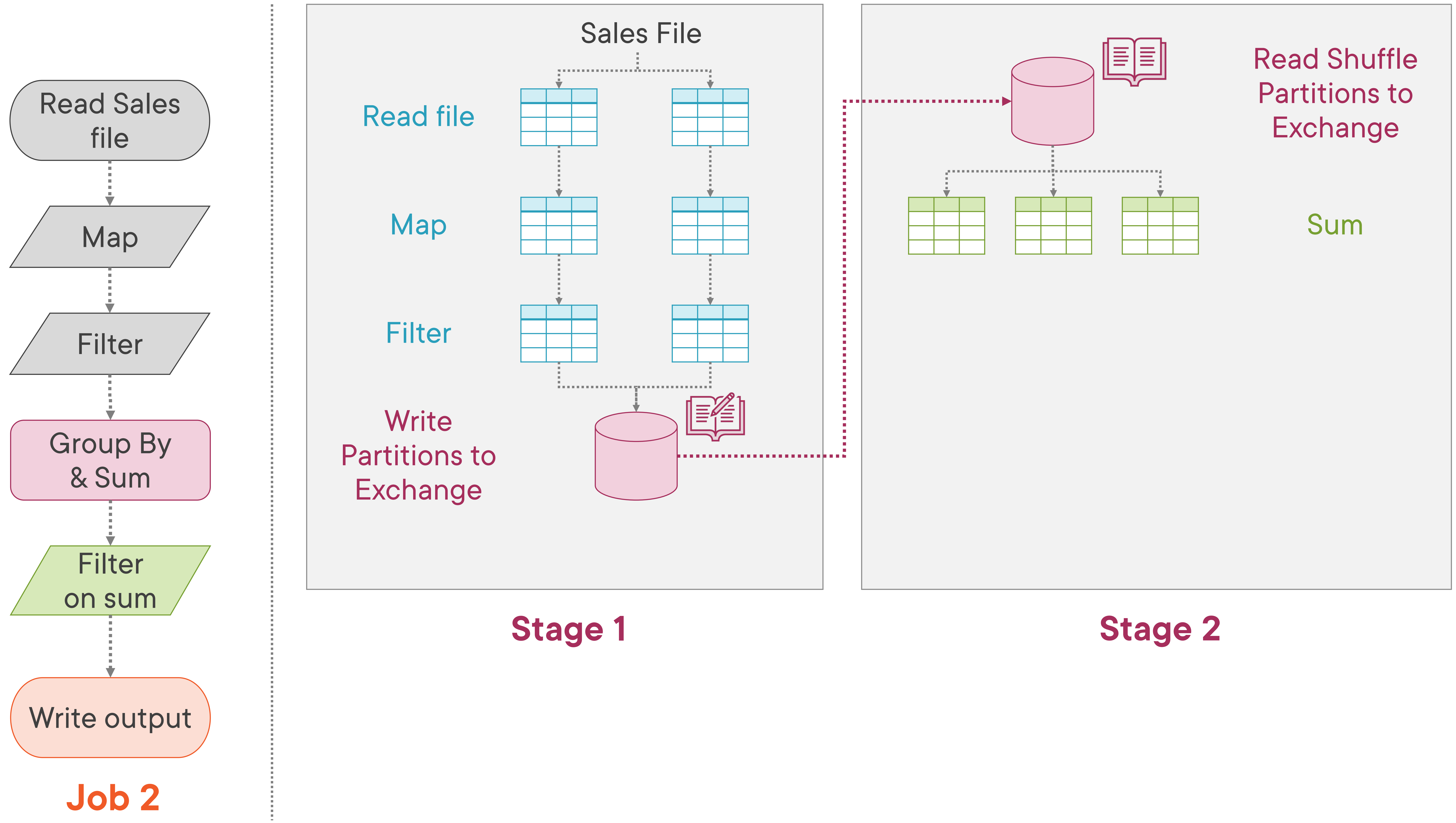
Job 2

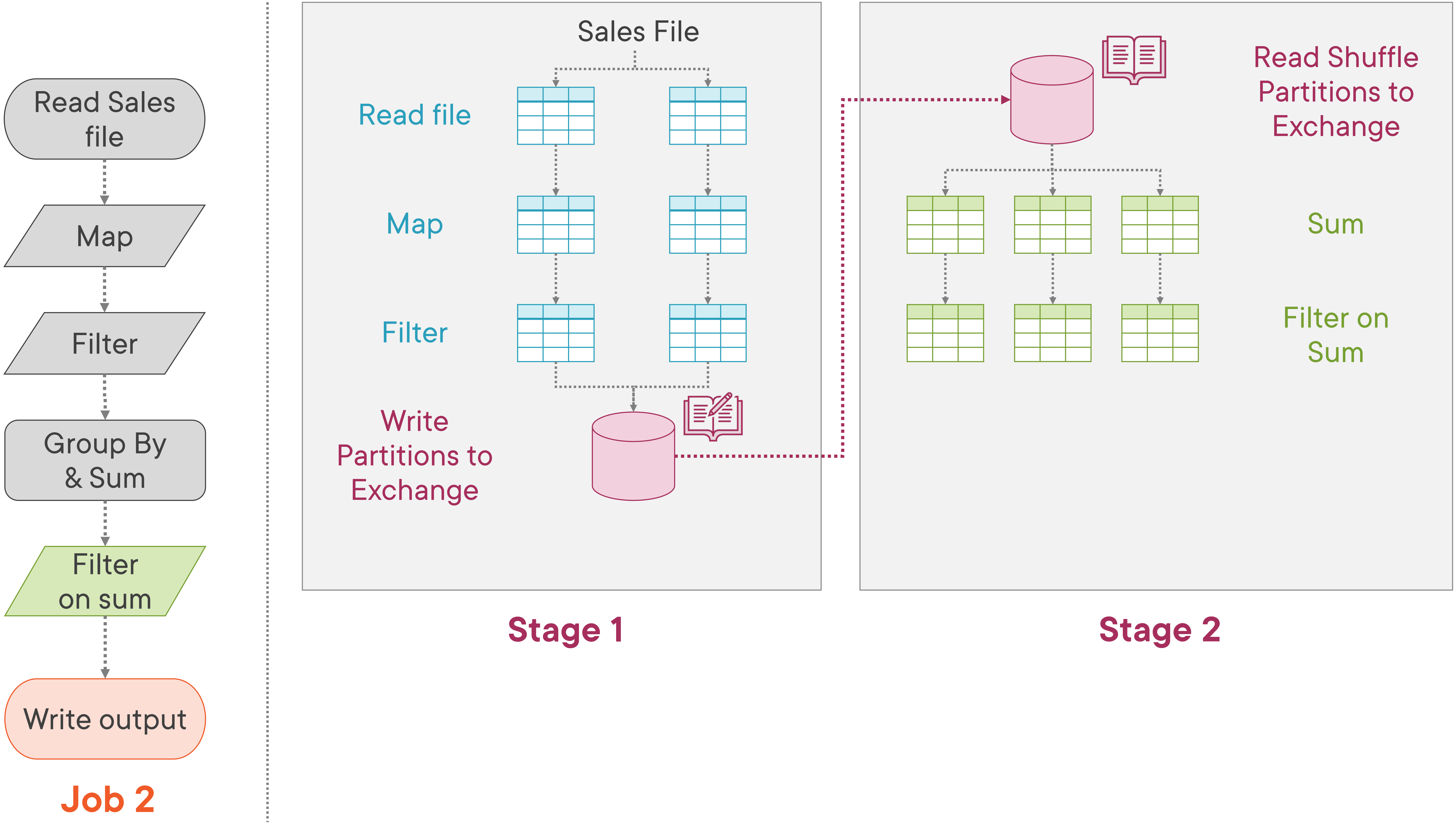


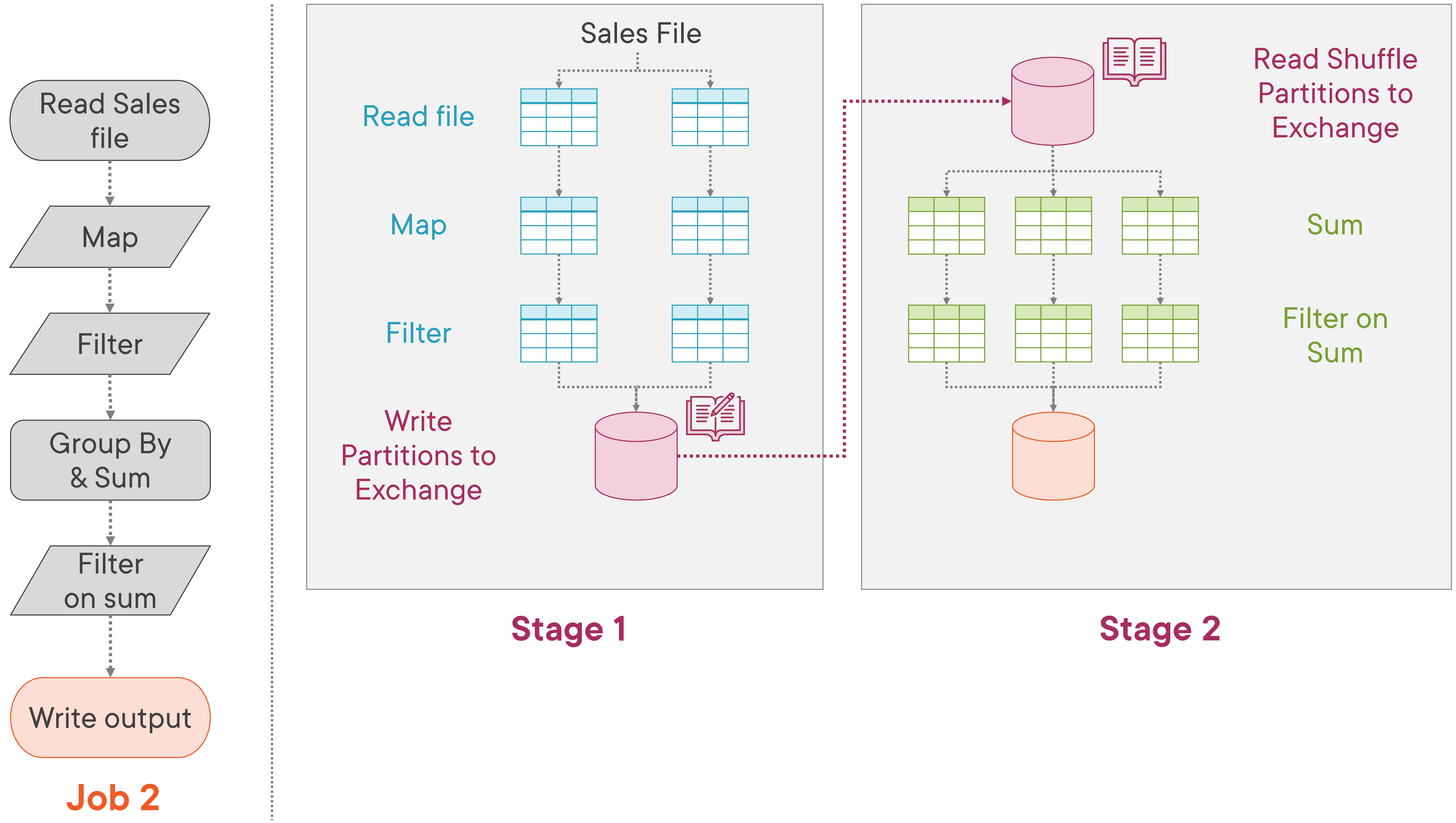
Stage 1

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
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


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
Filter on Sum
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
Filter on Sum
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

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

Up Next:
Cleaning & Transforming Data with DataFrames