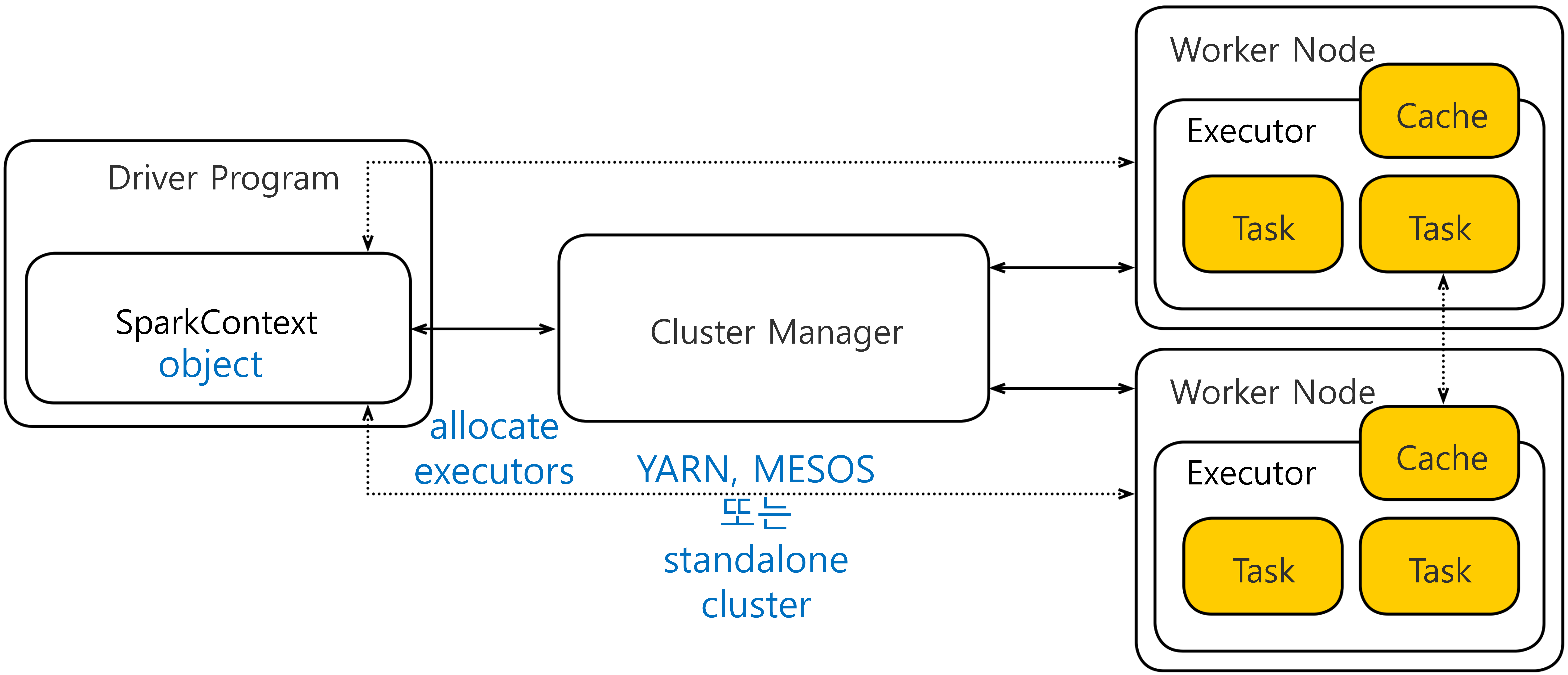
Yandex

Execution & scheduling

SparkContext

- Tells your application how to access a cluster
- Coordinates processes on the cluster to run your application



Jobs, stages, tasks

› Task is a unit of work to be done

›› Tasks are created by a job scheduler for every job stage

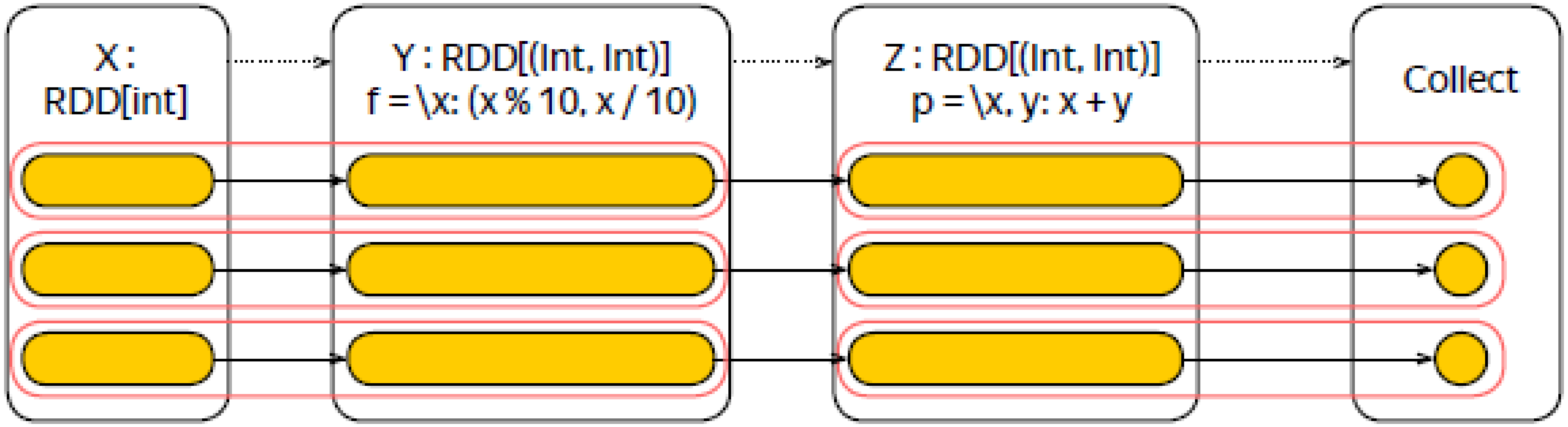
› Job is spawned in response to a Spark action

›› Job is divided in smaller sets of tasks called stages

Jobs, stages, tasks (example)

```
›› Z = X
   .map(lambda x: (x % 10, x / 10))
   .reduceByKey(lambda x, y: x + y)
   .collect()
```

1. Invoking an action… collect()
2. …spawns the job…
3. … that gets divided into the stages by the job scheduler…
4. …and tasks are created for every job stage.



Jobs, stages, tasks

Stage는 파이프라인을 위한 구조

› Job stage is a pipelined computation spanning between materialization boundaries
  ››not immediately executable RDD Level

› Task is a job stage bound to particular partitions
  ››immediately executable Partition Level

›› Materialization happens when reading, shuffling or passing data to an action Materializatoin == building
  ››narrow dependencies allow pipelining
  ››wide dependencies forbid it

SparkContext의 역할

- Tracks liveness of the executors
  - required to provide fault-tolerance
- Schedules multiple concurrent jobs
  - to control the resource allocation within the application
- Performs dynamic resource allocation
  - to control the resource allocation between different applications

Summary

›› The SparkContext is the core of your application

›› The driver communicates directly with the executors

›› E xecution goes as follows:
Action ➞ Job ➞ Job Stages ➞ Tasks

›› Transformations with narrow dependencies allow pipelining

Caching & persistence

Intermediate Data

Quick reminder

- RDDs are partitioned
- Execution is build around the partitions
- Block is a unit of input and output in Spark

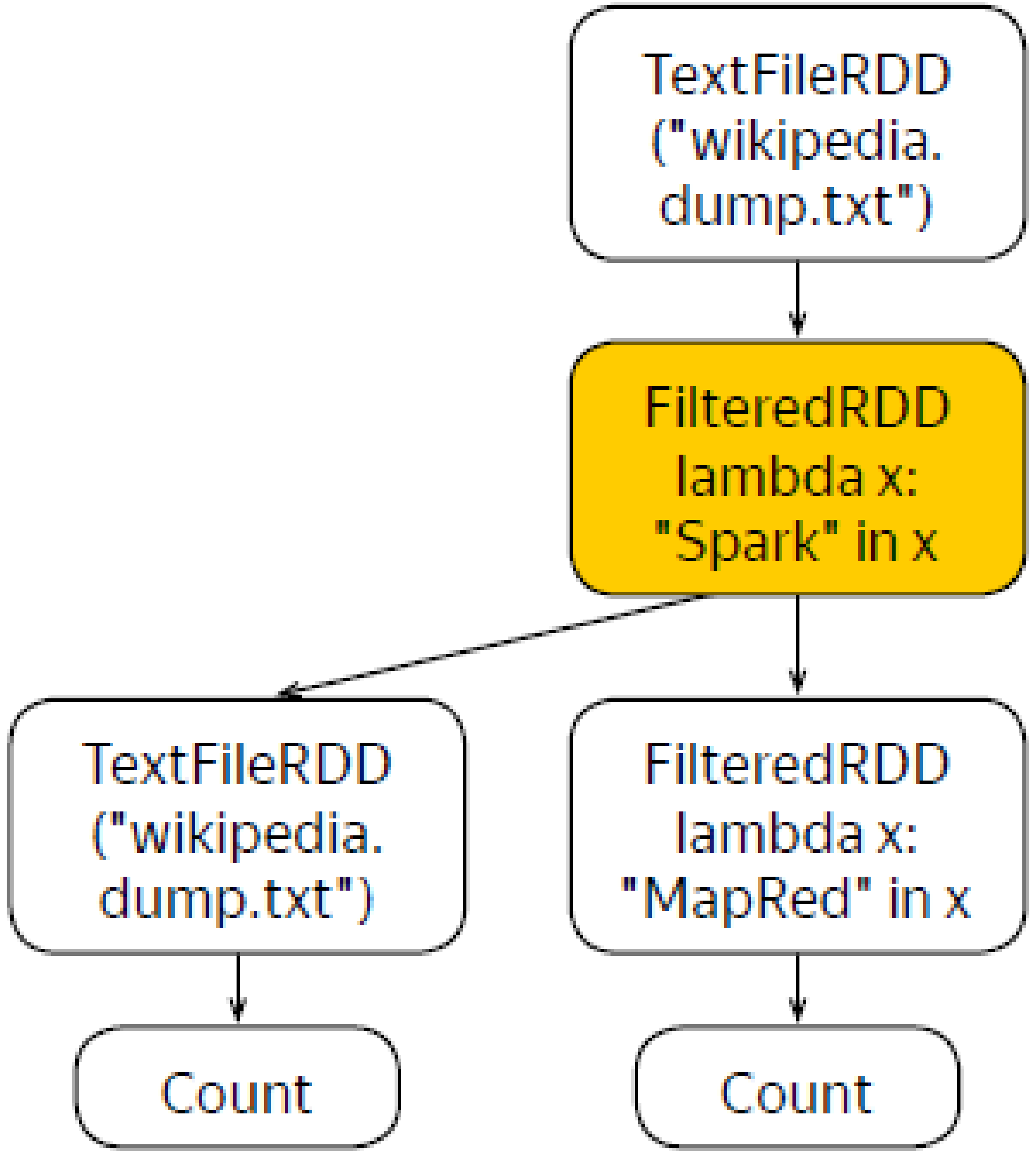
Motivating example

```
sc = SparkContext(...)
wiki = sc.textFile("wikipedia.dump.txt")

spark_articles = wiki.filter(
lambda x: "Spark" in x)

spark_articles.cache()

hadoop_articles = spark_articles.filter(
lambda x: "Hadoop" in x)
mapreduce_articles = spark_articles.filter(
lambda x: "MapRed" in x)
print(hadoop_articles.count())
print(mapreduce_articles.count())
```



메모리에 RDD를 cache로 저장하면 불필요한 작업을 줄일 수 있다.
(같은 작업 반복 X)

Controlling persistence level

›› rdd.persist(storageLevel)

››sets RDD's storage to persist across operations after it is computed for the first time

››storageLevel is a set of flags controlling the persistence, typical values are

DISK_ONLY

– save the data to the disk,

MEMORY_ONLY

– keep the data in the memory

MEMORY_AND_DISK

– keep the data in the memory; when out of memory – save it to the disk

DISK_ONLY_2, MEMORY_ONLY_2, MEMORY_AND_DISK_2

– same as about, but make two replicas ← improves failure recovery times!

›› rdd.cache() = rdd.persist(MEMORY_ONLY)

cache함수는 메모리에 저장한다는 함수의 shortcut이다.

Best practices

일반적인 데이터 persist 방법

- For interactive sessions
  - cache preprocessed data

- For batch computations
  - cache dictionaries
  - cache other datasets that are accessed multiple times

- For iterative computations
  - cache static data

- And do benchmarks!

Summary

- Performance may be improved by persisting data across operations
  - in interactive sessions, iterative computations and hot datasets
- You can control the persistence of a dataset
  - whether to store in the memory or on the disk
  - how many replicas to create

Broadcast variables

Shared Data

Broadcast variable

›› Broadcast variable is a read-only variable that is efficiently shared among tasks

›› Distribution is done by a torrent-like protocol (extremely fast!)

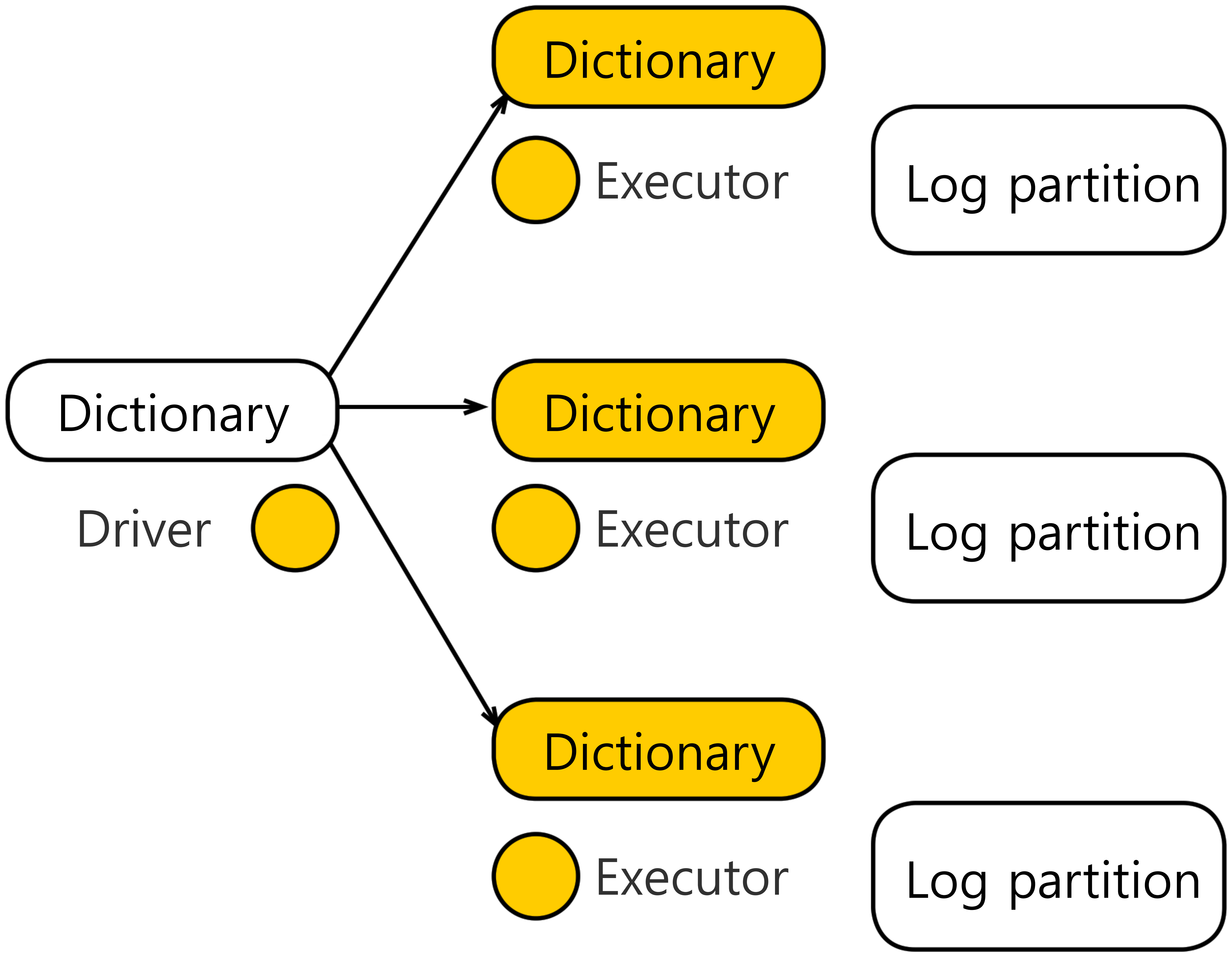
›› Distributed efficiently compared to captured variables

일반 variable을 closure에 넣으면 1 to many protocol
(한 곳에서 많은 executor로 전달해야 함)

Broadcast variable은 many to many protocol (토렌트와 같은 방식)

Motivating example

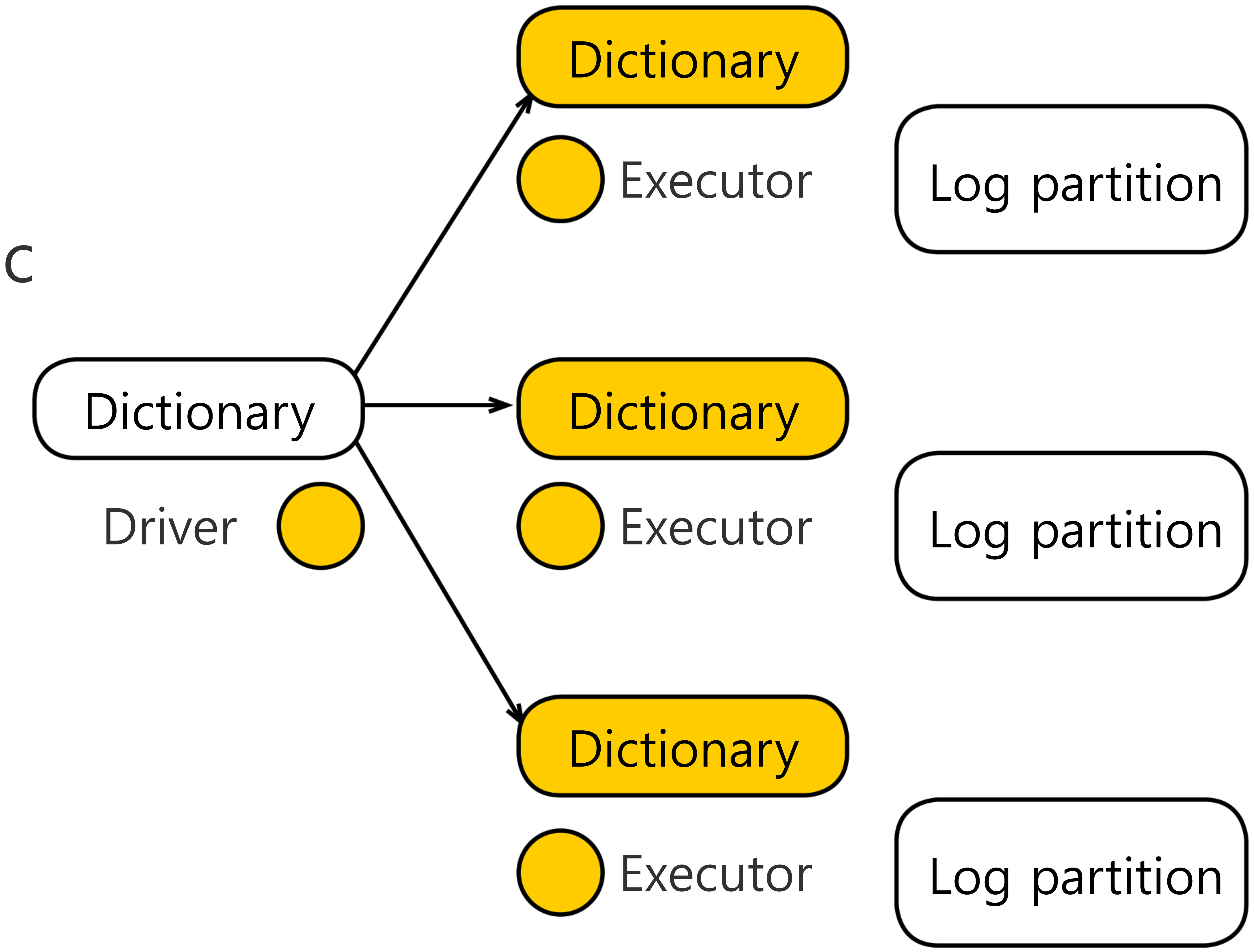
- Input:
  1TB partitioned log, 1GB IP dictionary
- Task:
  resolve IP addresses
- Idea:
  distribute the dictionary
  query it locally



Motivating example

Serial distribution via the closure
(from the driver to every executor)
~1000 (tasks) * 1GB = 1TB of traffic

1GB 데이터를 driver가 모두 전달



Parallel distribution via
the broadcast variable
(torrent-like)
~1-2 GB of traffic Faster!

Motivating example 2

```
sc = SparkContext(conf=...)

# compute the dictionary
my_dict_rdd = sc.textFile(...).map(...).filter(...)
my_dict_data = my_dict_rdd.collect()

# distributed the dictionary via the broadcast variable
broadcast_var = sc.broadcast(my_dict_data)

# use the broadcast variable within the task
my_data_rdd = sc.textFile(...).filter(
lambda x: x in broadcast_var.value)
```

Summary

›› Broadcast variables are read-only shared variables with effective sharing mechanism

단, memory에 맞는 양의 데이터를 활용가능하다.

›› Useful to share dictionaries, models

Accumulator variables

Useful for the control flow, monitoring, profiling & debugging

Accumulator variable

› Accumulator variable is a read-write variable that is shared among tasks

›› Writes are restricted to increments! synchronization 문제를 피하기 위해

››i. e.: var += delta

››addition may be replaced by any associate, commutative operation

›› Reads are allowed only by the driver program! task에서는 읽을 수 없다.

Guarantees on the updates

›› In actions updates are applied exactly once

action에서는 accumulator에 한 번만 적용된다.

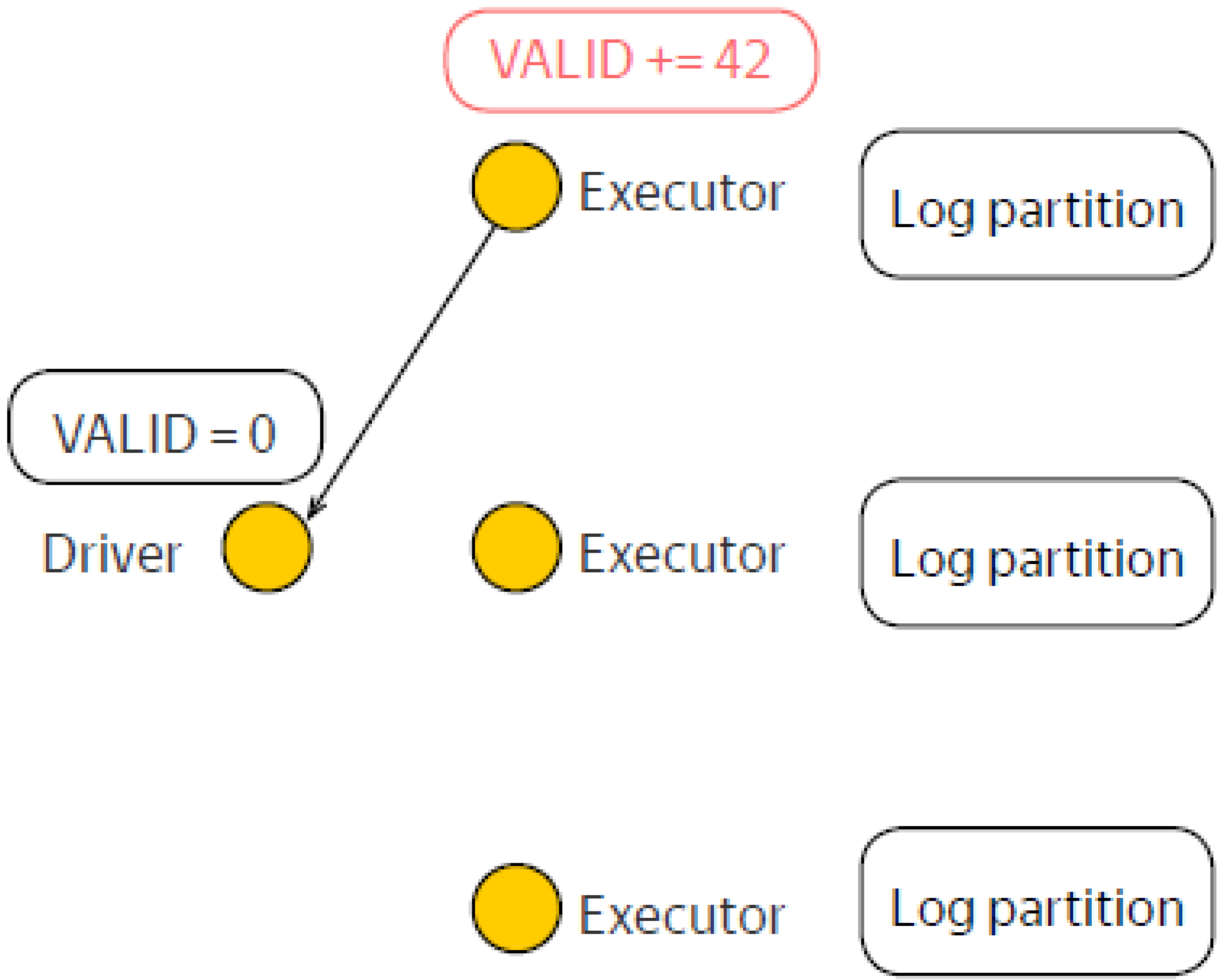
›› In transformations there are no guarantees as the transformation code may be re-executed

transformation은 재실행될 수 있기 때문에 보장할 수 없다.

Example (similar to the previous video)

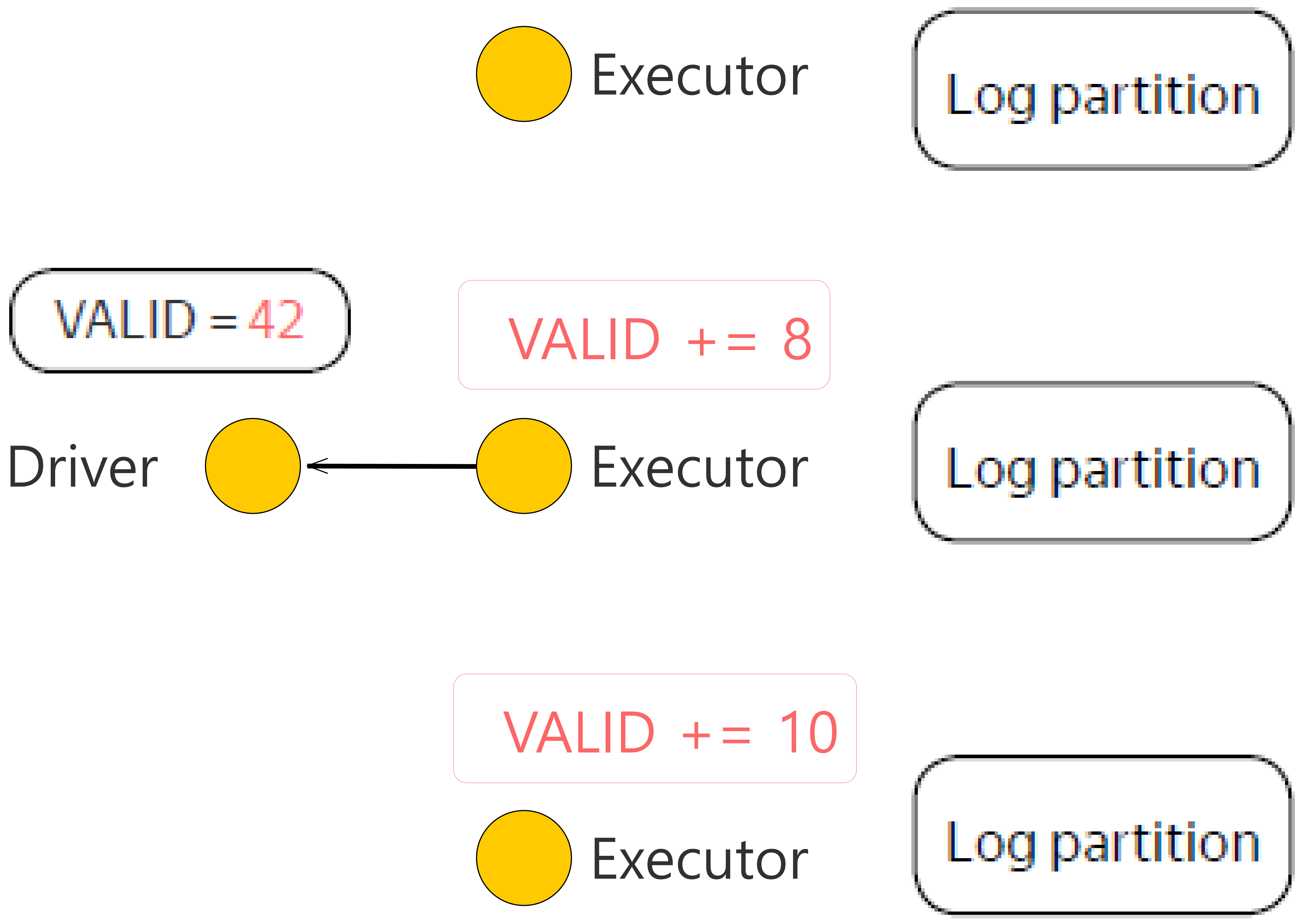
- Input:
  1TB partitioned log

- Task:
  resolve IP addresses
  AND
  collect metrics:
  # of valid records



Example (similar to the previous video)

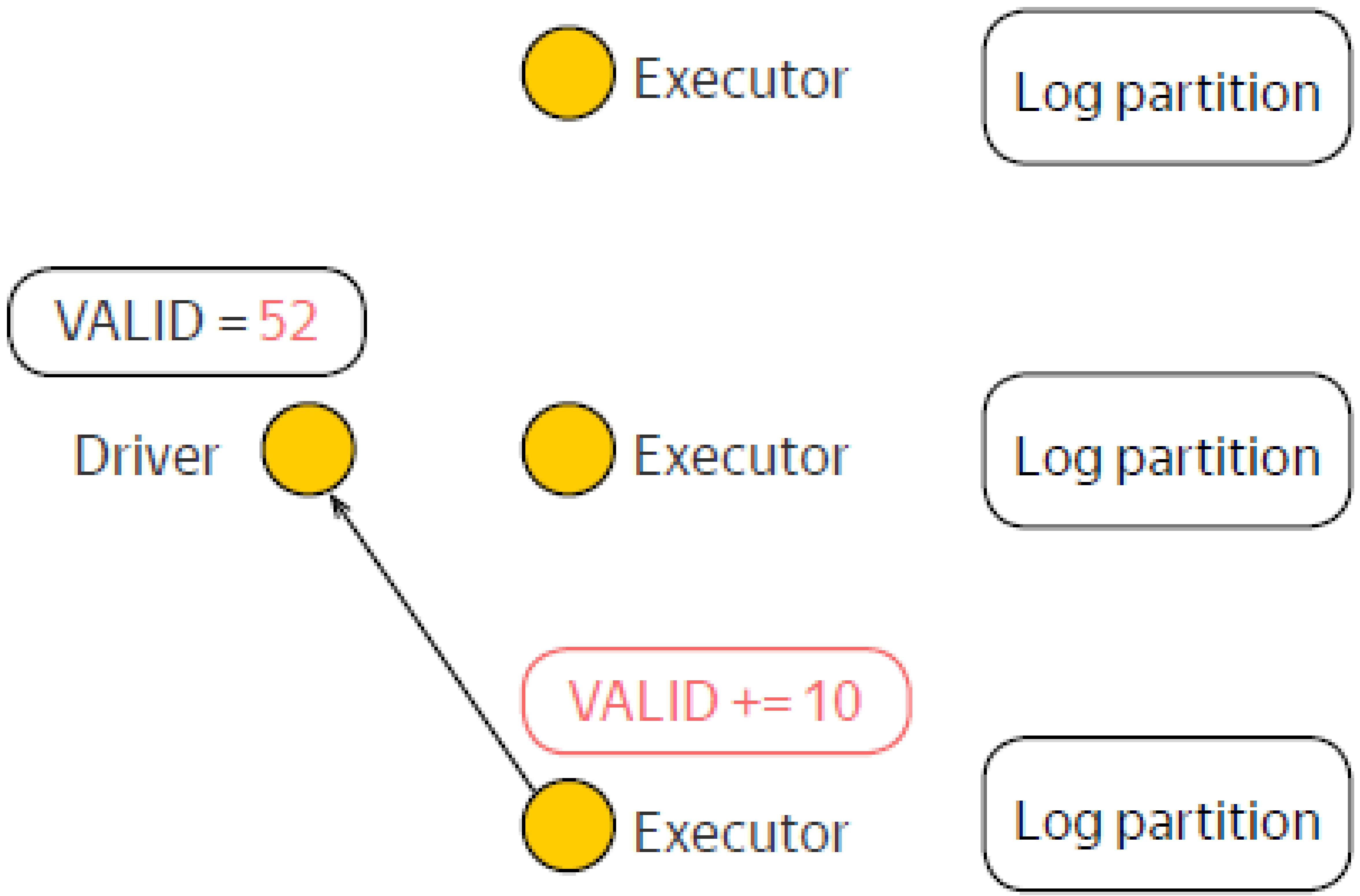
- Input:
  1TB partitioned log
- Task:
  resolve IP addresses
  AND
  collect metrics:
  # of valid records



Example (similar to the previous video)

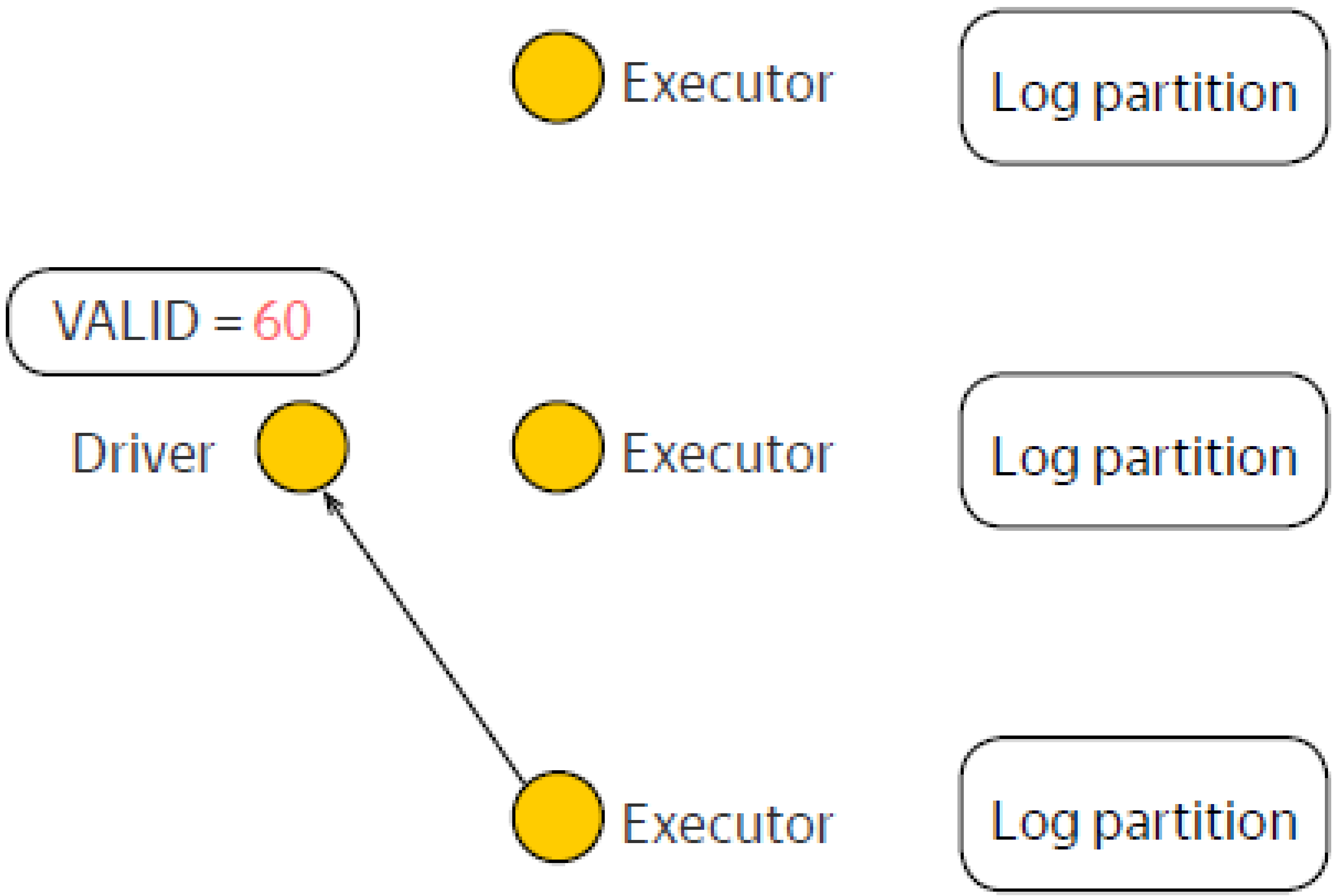
- Input:
  1TB partitioned log

- Task:
  resolve IP addresses
  AND
  collect metrics:
  # of valid records



Example (similar to the previous video)

- Input:
  1TB partitioned log
- Task:
  resolve IP addresses
  AND
  collect metrics:
  # of valid records



Use cases

- Performance counters
  - # of processed records, total elapsed time, total error and so on and so forth
- Simple control flow
  - conditionals: stop on reaching a threshold for corrupted records
  - loops: decide whether to run the next iteration of an algorithm or not
- Monitoring
  - export values to the monitoring system
- Profiling & debugging

Summary

- Accumulators are read-write shared variables with restricted updates
  - increments only
  - can use custom associative, commutative operation for the updates
  - can read the total value only in the driver

- Useful for the control flow, monitoring, profiling & debugging

BigDATAteam