Yandex

Introduction to Spark

Week outline

- Lesson 1 – Basic concepts
  - RDDs, transformations, actions, resiliency
- Lesson 2 – Advanced topics
  - execution, persistence & caching, broadcast variables, accumulators
- Lesson 3 – Spark and Python
  - integration, examples

Historical background

›› 2009 – project started at UC Berkeley's AMPLab

›› 2012 – first release (0.5)

›› 2014 – became top-level Apache project

›› 2014 – reached 1.0

›› 2015 – reached 1.5

›› 2016 – reached 2.0

First epoch (2009-2012)

- Key observations : 하둡의 문제
  - Underutilization of cluster memory
    MapReduce가 memory를 제대로 사용하지 않는다.
    - for many companies data can fit into memory either now, or soon
    - memory prices were decreasing year-over-year at that time
  - Redundant disk I/O => 속도가 느려짐
    - especially in iterative MR jobs
  - Lack of higher-level primitives in MR => Framework 사용이 어려움
    - one has to redo joins again and again
    - one has to carefully tune the algorithm
- Key outcomes : Spark에서 해결
  - RDD abstraction with rich API
  - In-memory distributed computation platform

Second epoch (2012-2014)

›› Key observations : 하둡의 문제
  ›› No "one system to rule them all"
    ›› typical cluster would include a dozen of different systems tailored for specific applications
    ›› recurrent data copying between the systems increases timings
  ›› Increasing demand for interactive queries and stream processing
    ›› due to raise of data-driven applications
      › need for fast ad-hoc analytics
      › need for fast decision-making

›› Key outcomes : Spark에서 해결
  ›› Separation of Spark Core and applications on top of the core:
    ›› Spark SQL
    ›› Spark Streaming
    ›› Spark GraphX
    ›› Spark MLlib

batch processing, stream processing, graphical computation 등을 하나의 Framework로 통합

Third epoch (2014-now)

- Key observations
  - Increasing use of machine learning
  - Increasing demand for integration with other software (Python, R, Julia...)

- Key outcomes
  - Focus on ease-of-use
  - Spark Dataframes as first-class citizens

RDDs

Why do we need a new abstraction?

output이 다른 job의 input이 되는 연산

- Example: iterative computations (K-means, PageRank, …)
  - relation between consequent steps is known only to the user code
    Framework가 computation을 optimize할 수 없다.
  - framework must reliably persist data between steps (even if it is temporary data)
    intermediate 데이터가 disk에 저장되어 불필요한 I/O 발생

- Example: joins
  - join operation is used in many MapReduce applications
  - not-so-easy to reuse code

Resilient Distributed Datasets

› Resilient — able to withstand failures
› Distributed — spanning across multiple machines
› Formally, a read-only, partitioned collection of records

›› To adhere to RDD[T] interface, a dataset must implement:

구현해야 할 필수 3가지

››partitions() ➡ Array[Partition]
››iterator(p: Partition, parents: Array[Iterator[_]]) ➡ Iterator[T]
››dependencies() ➡ Array[Dependency]

›› ...and may implement other helper functions

›› Typed! RDD[T] — a dataset of items of type T

타입을 명시해야 한다.

Example: a binary file in HDFS

- partitions() ➡ *Array[Partition]* HDFS의 block이 Spark의 Partition으로 대체 가능
  - lookup blocks information from the NameNode
  - make a partition for every block
  - return an array of the partitions

- iterator*(p: Partition, parents: Array[Iterator[_]])* ➡
➡ *Iterator[Byte]*
  - parents are not used
  - return a reader for the block of the given partition

- dependencies*()* ➡ *Array[Dependency]*
  - ›return an empty array File 읽는 데는 dependency가 필요없음

Example: a binary data* file in HDFS

- partitions*()* ➡ *Array[Partition]*
  - ~~Lookup blocks information from the NameNode~~
    use InputFormat to compute InputSplits
  - make a partition for every ~~block~~ InputSplit
  - return an array of the partitions

- iterator*(p: Partition, parents: Array[Iterator[_]])* ➡
  ➡ *Iterator[~~Byte~~ InputRecord]*
  - parents are not used
  - use InputFormat to create a reader for the InputSplit of the given partition
  - return ~~a reader for the block of the given partition~~ the reader

- dependencies*()* ➡ *Array[Dependency]*
  - return an empty array

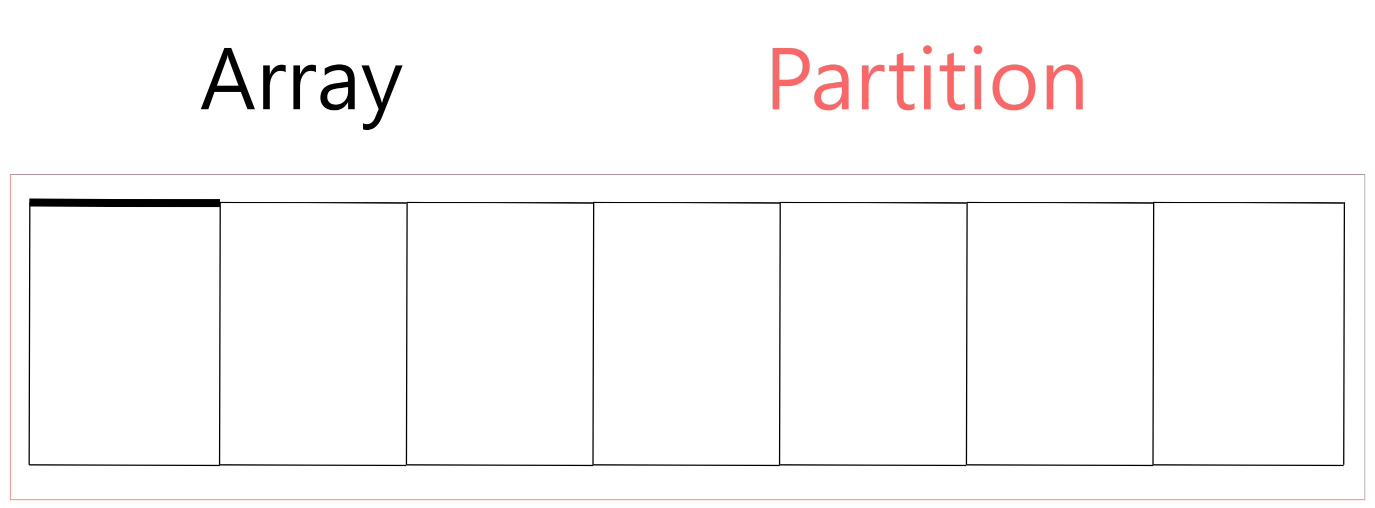
*a file encoded with the file format: see W1; think: text file, SequenceFile, Avro, RCFile

Example: an in-memory array

- partitions*()* ➡ *Array[Partition]*
  - return an array of a single partition with the source array

- iterator*(p: Partition, parents: Array[Iterator[_]])* ➡ *Iterator[T]*
  - parents are not used
  - return an iterator over the source array in the given partition

- dependencies*()* ➡ *Array[Dependency]*
  - return an empty array (no dependencies)

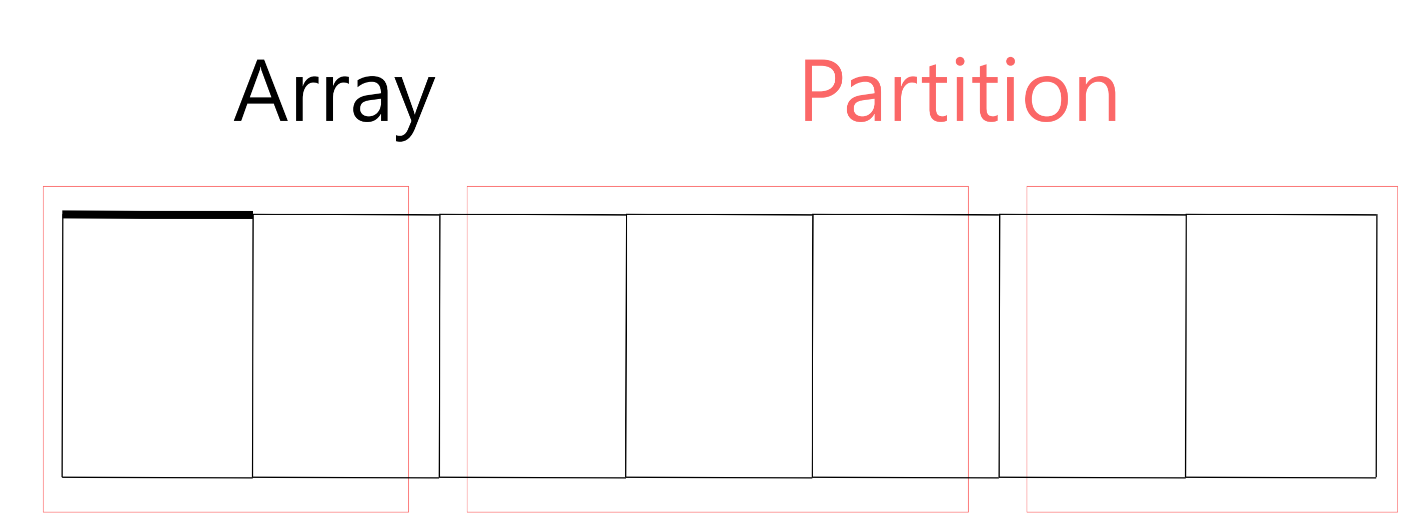


Example: an sliced* in-memory array

- partitions*()* ➡ *Array[Partition]*
  - slice array in chunks of size N
  - make a partition for every chunk
  - return an arra~~y of a single partition with the source a~~rray of the partitions

- iterator*(p: Partition, parents: Array[Iterator[_]])* ➡ *Iterator[T]*
  - parents are not used
  - return an iterator over the source array chunk in the given partition

- dependencies*()* ➡ *Array[Dependency]*
  - return an empty array (no dependencies)



*can be used to parallelize in-memory computations

Summary

- RDD is a read-only, partitioned collection of records
  - a developer can access the partitions and create iterators over them
  - RDD tracks dependencies (to be explained in the next video)

- Examples of RDDs
  - Hadoop files with the proper file format
  - In-memory arrays

Transformations 1

Two ways to construct RDDs

- Data in a stable storage (previous video)
  - Example: files in HDFS, objects in Amazon S3 bucket, lines in a text file, ...
  - RDD for data in a stable storage has no dependencies

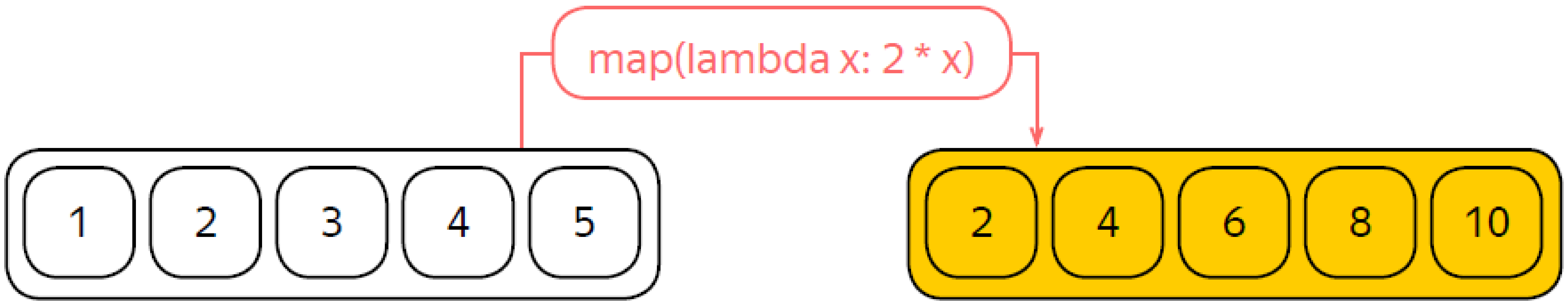
- From existing RDDs by applying a transformation (this video)
  - Example: filtered file, grouped records, ...
  - RDD for a transformed data depends on the source data

Transformations

- Allow you to create new RDDs from the existing RDDs by specifying how to obtain new items from the existing items
- The transformed RDD depends implicitly on the source RDD

Transformations

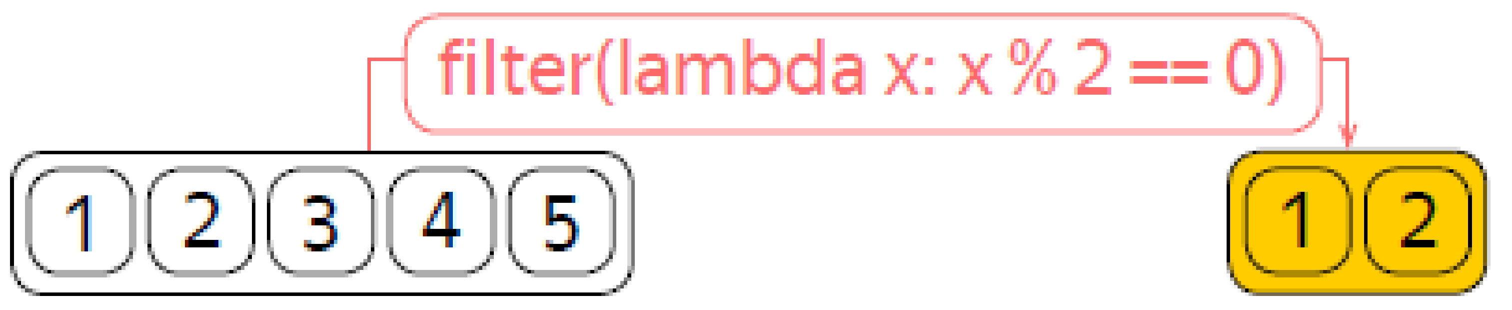
- Def: *filter(p: T ➡ Boolean): RDD[T] ➡ RDD[T]*
  - returns a filtered RDD with items satisfying the predicate p
- Def: *map(f: T ➡ U): RDD[T] ➡ RDD[U]*
  - returns a mapped RDD with items f(x) for every x in the source RDD



Transformations

- <u>Def</u>: filter(p: $T \to$ Boolean): $RDD[T] \to RDD[T]$
  - returns a filtered RDD with items satisfying the predicate p
- <u>Def</u>: map(f: $T \to U$): $RDD[T] \to RDD[U]$
  - returns a mapped RDD with items f(x) for every x in the source RDD
- <u>Def</u>: flatMap(f: $T \to Array[U]$): $RDD[T] \to RDD[U]$
  - same as map but flattens the result of f
  - generalizes map and filter

Filtered RDD



- Y = X.filter(p) # where X : RDD[T]
  - Y.partitions() → *Array[Partition]*
    - return the same partitions as X

  - Y.iterator(p: *Partition*, parents: *Array[Iterator[T]]*) → *Iterator[T]*
    - take a parent iterator over the corresponding partition of X
    - wrap the parent iterator to skip items that do not satisfy the predicate
    - return the iterator over partition of Y

  - Y.dependencies() → *Array[Dependency]*
    - k-th partition of Y depends on k-th partition of X

      partition을 그대로 사용하기에 1대1 대응

Filtered RDD

- Y = X.filter(p) # where X : RDD[T]
  - Y.partitions() ➞ *Array[Partition]*

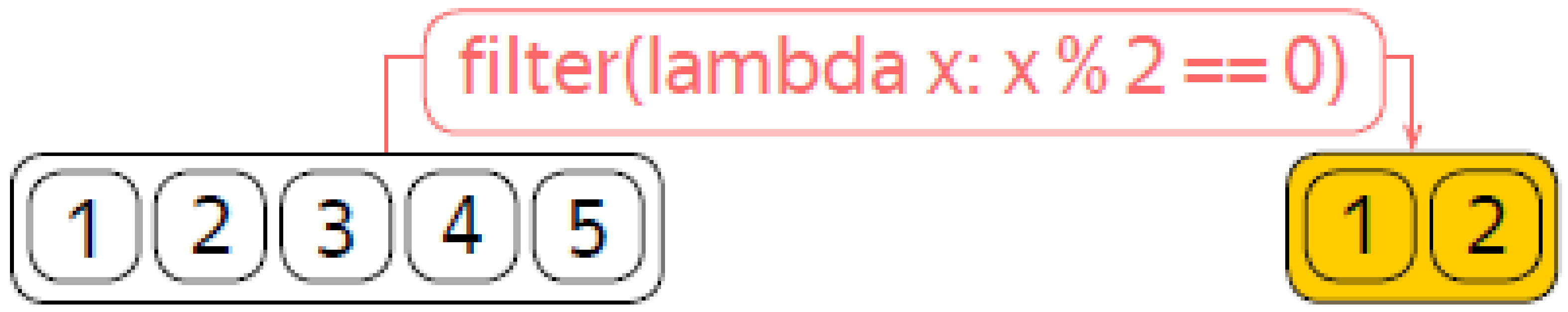
Note that actual filtering happens not at
the creation time of Y, but at *the access time*
to the iterator over a partition of Y.

Same holds for other transformations – they are lazy,
i.e. they compute the result only when accessed.

RDD를 만들 때는 전혀 operation을 실행하지 않는다. => action을 실행해야 함

  - Y.dependencies() ➞ *Array[Dependency]*
    - k-th partition of Y depends on k-th partition of X

On closures



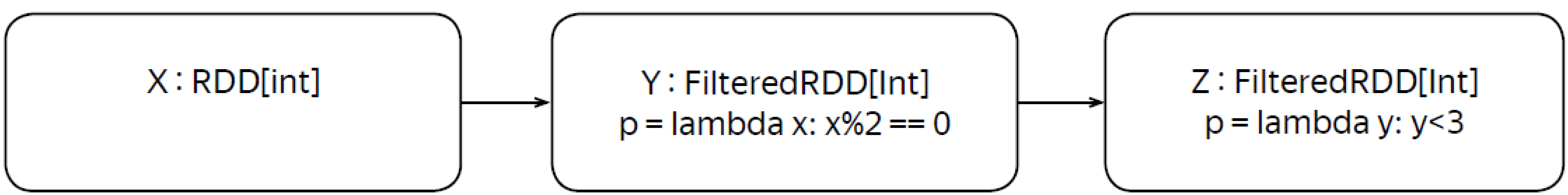
- Y = X.filter(lambda x: x % 2 == 0)
  - predicate closure is captured within the Y (it is a part of the definition of Y)
  - predicate is not guaranteed to execute locally (closure may be sent over the network to the executor)

Dependency graph

자동으로(implicitly) 생성된다.

Framework가 scheduling할 때 활용된다.

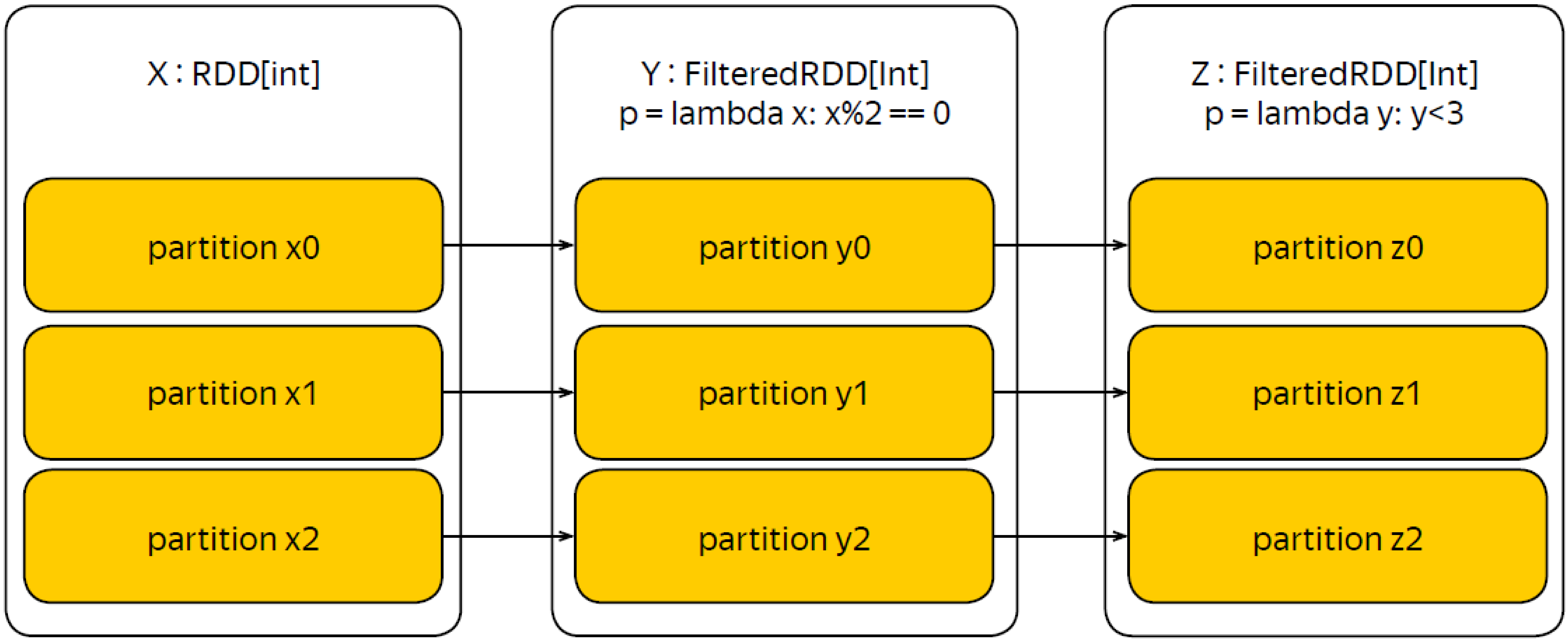
- Y = X.filter(lambda x: x % 2 == 0)
- Z = X.filter(lambda x: x % 2 == 0).filter(lambda y: y < 3)



Partition dependency graph

interpartition dependencies

- Z = X.filter(lambda x: x % 2 == 0).filter(lambda y: y < 3)

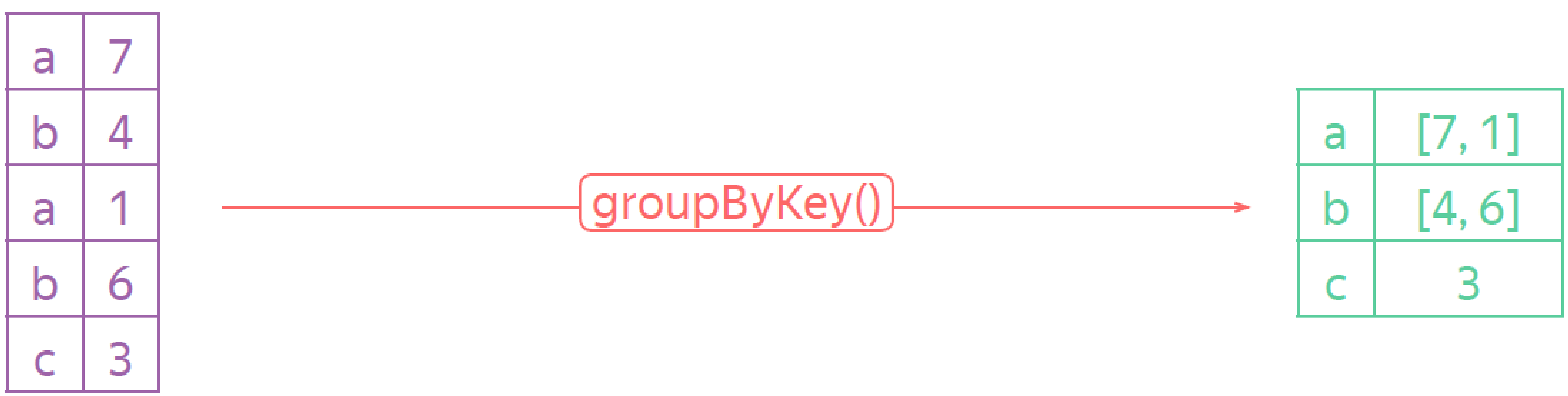


Transformations 2

Keyed RDDs

Keyed transformations

- <u>Def</u>: groupByKey(): *RDD[(K, V)]* ➞ *RDD[(K, Array[V])]*
  - groups all values with the same key into the array
  - returns a set of the arrays with corresponding keys



Keyed transformations

- <u>Def</u>: groupByKey*(): RDD[(K, V)] ➞ RDD[(K, Array[V])]*
  - groups all values with the same key into the array
  - returns a set of the arrays with corresponding keys

- <u>Def</u>: reduceByKey*(f: (V, V) ➞ V): RDD[(K, V)] ➞ RDD[(K, V)]*
  - folds all values with the same key using the given function f
  - returns a set of the folded values with corresponding keys

a	7
b	4
a	1
b	6
c	3

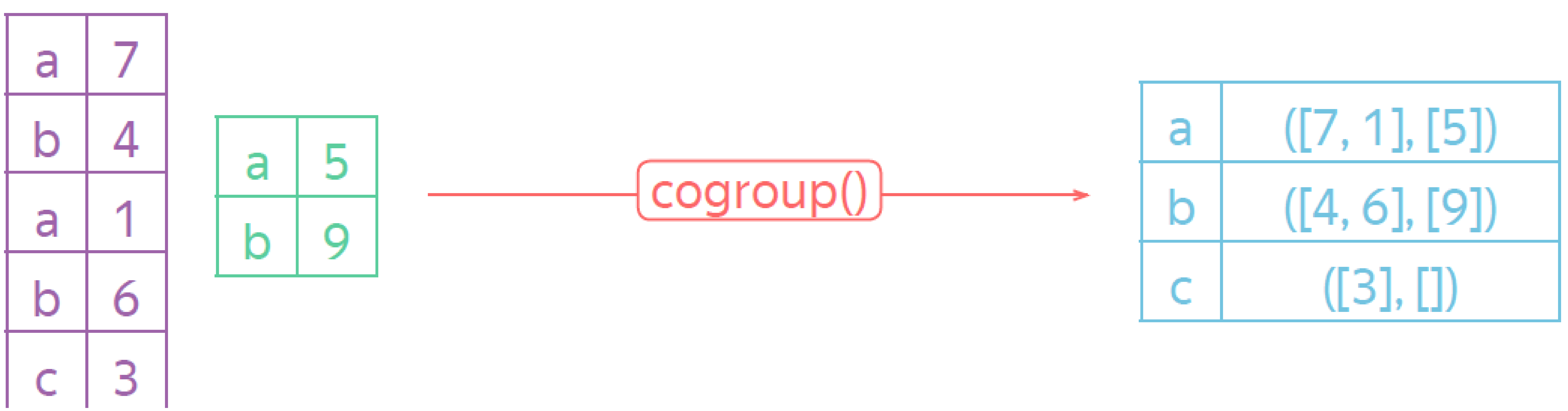
reduceByKey(lambda x, y: x + y) →

a	8
b	10
c	3

Cogroup transformation

- Def: X.cogroup(*Y: RDD[(K, W)]):*
  RDD[(K, V)] ➡ RDD[(K, (Array[V], Array[W]))]
  - given two keyed RDDs, groups all values with the same key
  - returns a triple (k, X-values, Y-value) for every key where X-values are all values found under the key k in X and Y-values are similar

두 개의 RDD를 Group화한다.

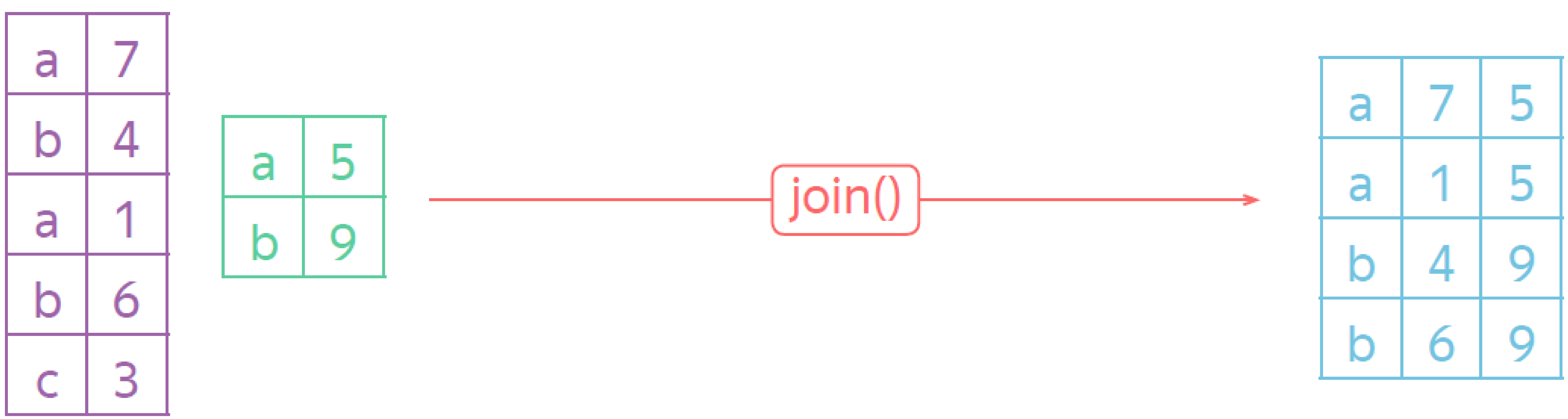


Joins

자주 쓰이는 연산이어서 구현되어 있다.

- Def: X.join(*Y: RDD[(K, W)]): RDD[(K, V)]* ➡ *RDD[(K, V, W)]*
  - given two keyed RDDs, returns all matching items in two datasets
  - that are triples (k, x, y) where (k, x) is in X and (k, y) is in Y

- Also: X.leftOuterJoin, X.rightOuterJoin, X.fullOuterJoin



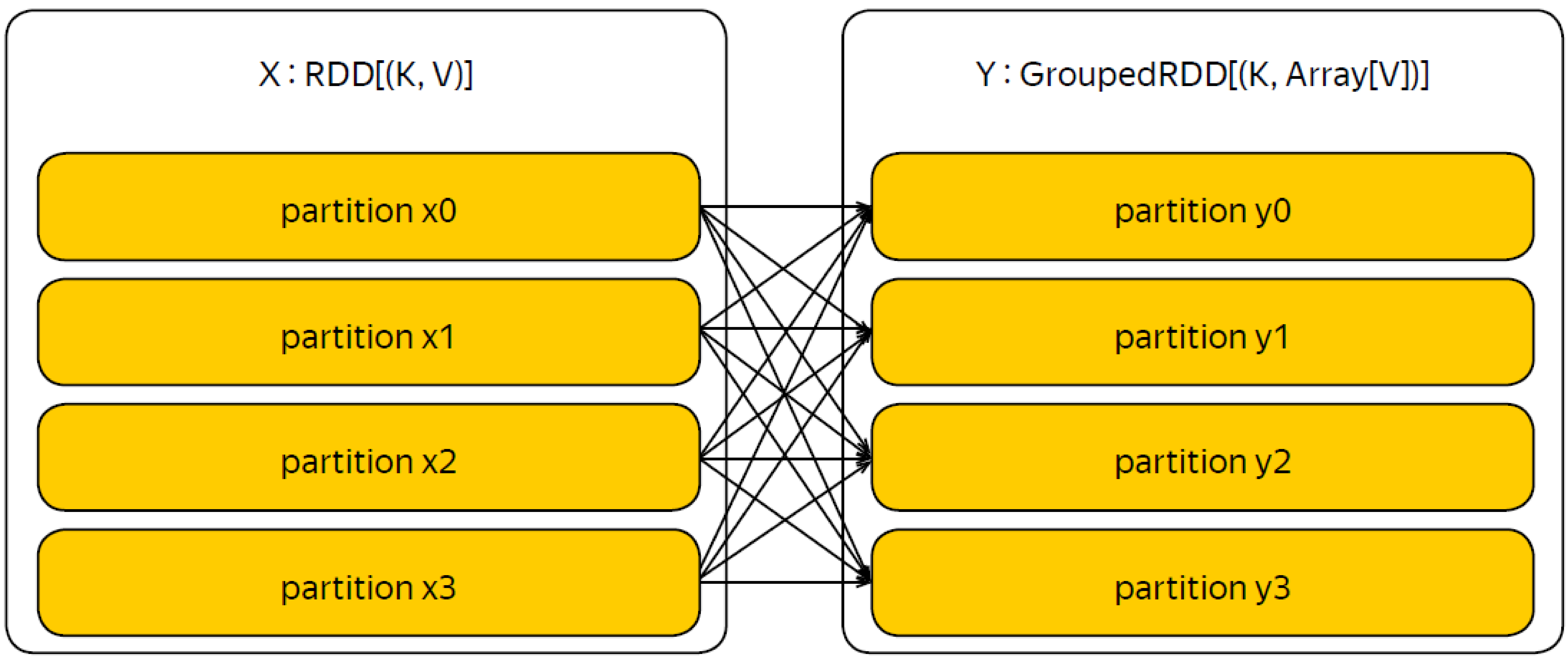
Grouped RDD

- Y = X.groupByKey(): *RDD[(K, V)]* ➡ *RDD[(K, Array[V])]*
  - Y.partitions() ➡ *Array[Partition]*
    - returns a set of partitions of the key space
  - Y.iterator(p: *Partition*, parents: *Array[Iterator[(K,V)]]*)
    ➡ *Iterator[(K, Array[V])]*
    - iterate over every parent partition to select pairs with the key in the partition range, group the pairs by the key – a shuffle operation!
    - return an iterator over the result
  - Y.dependencies() ➡ *Array[Dependency]*
    - k-th output partition depends on all input partitions

Grouped RDD – Shuffle

pair를 찾기 위해 모든 RDD를 스캔해야 한다.

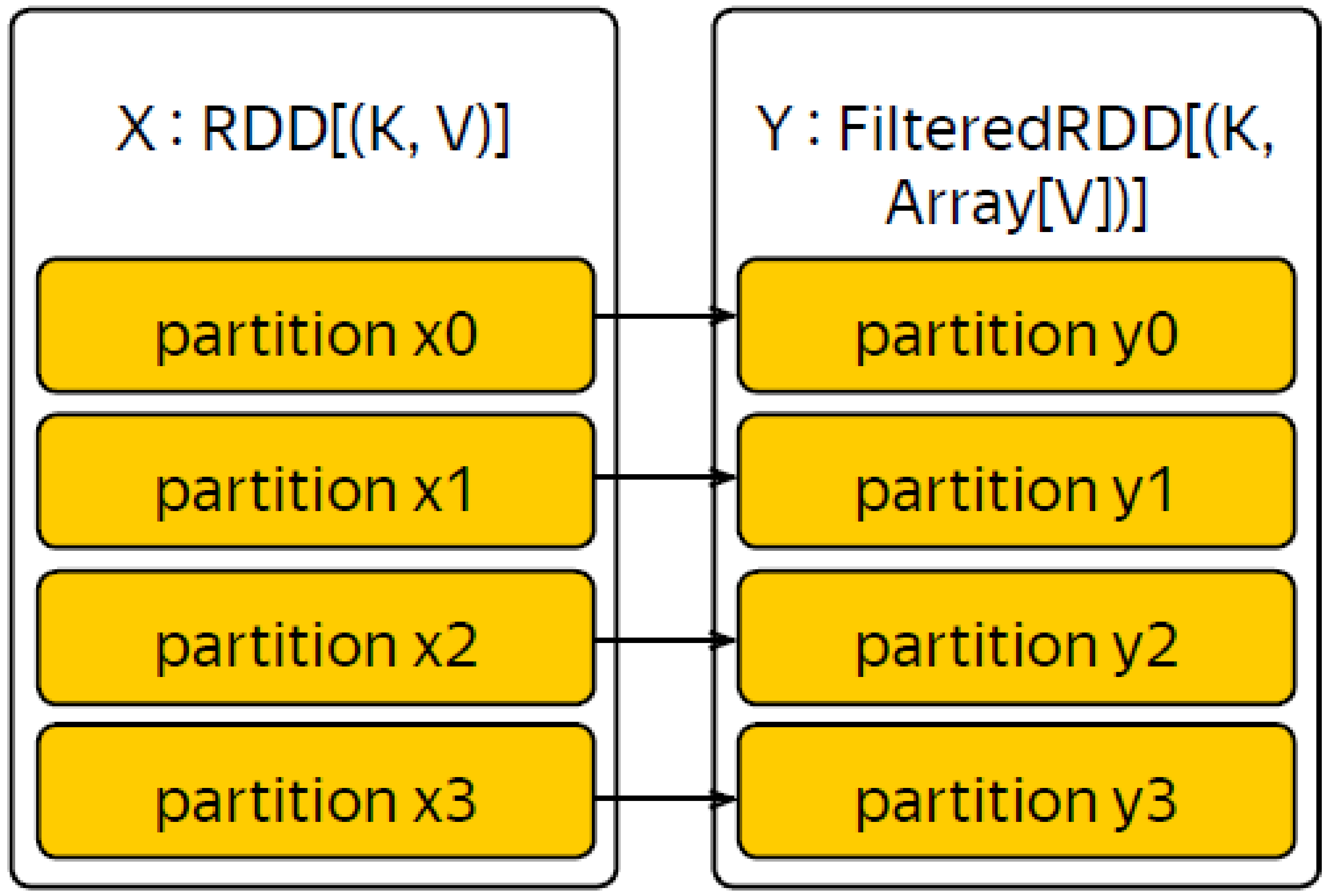
› Y = X.groupByKey(): *RDD[(K, V)] ➡ RDD[(K, Array[V])]*



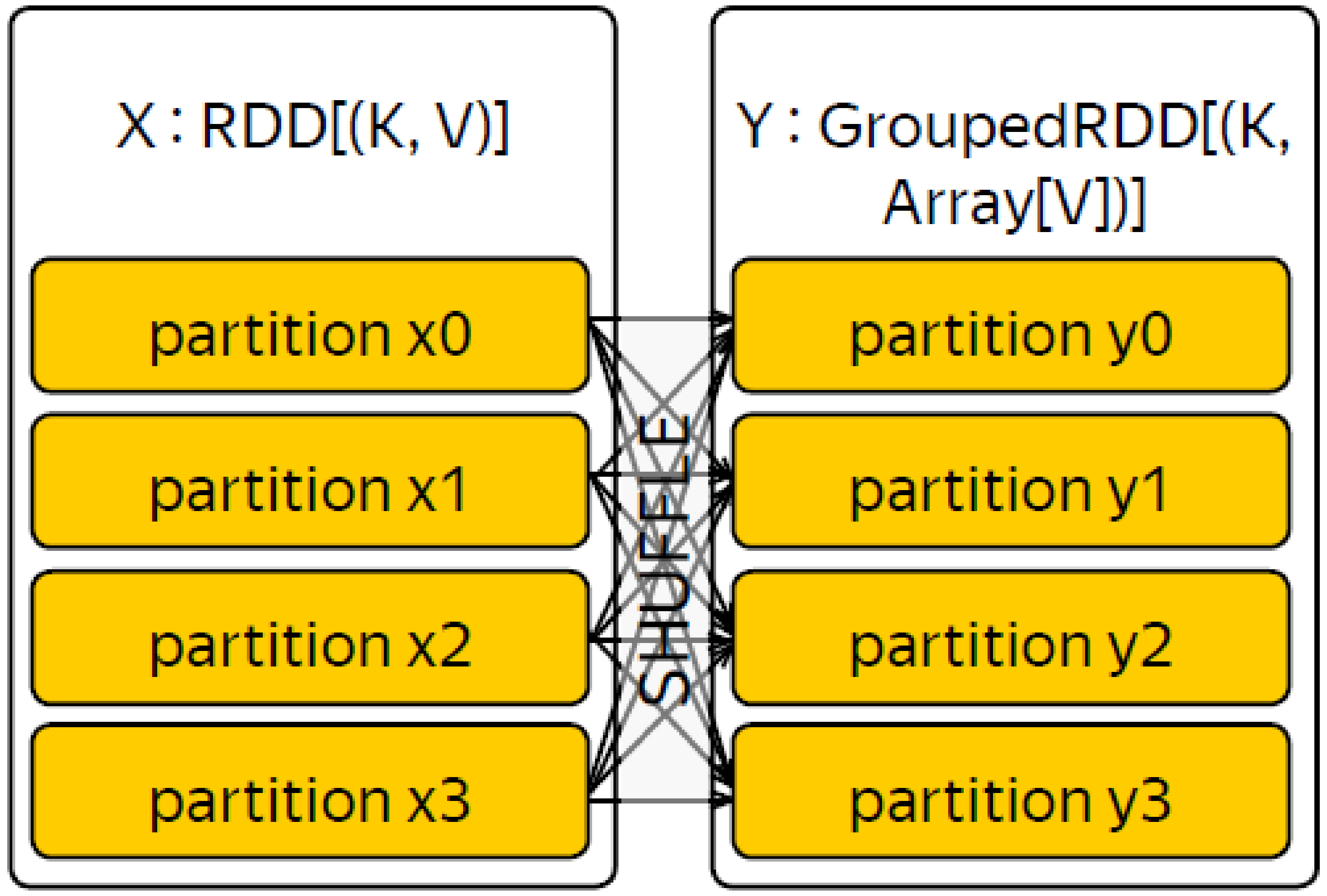
MapReduce처럼 네트워크와 memory 관련 performance 이슈가 있다.
(특정 partition에 몰리는 경우)

Narrow & Wide dependencies

Y = X.filter(p)



Y = X.groupByKey()



Narrow dependencies

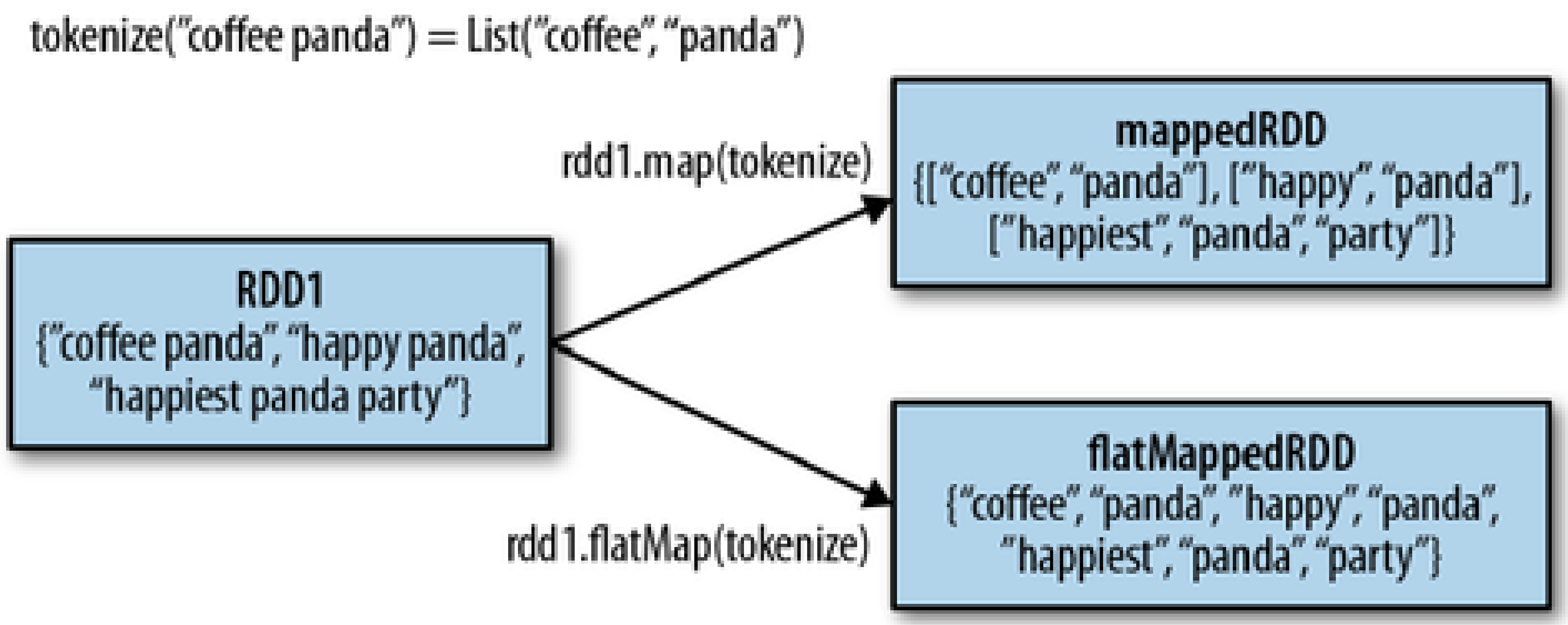
at most one child partition for every parent partition

Wide dependencies

more than one child partition for every parent partition

Plenty of transformations!

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- mapValues
- sample
- distinct
- union
- intersection
- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join
- cogroup
- cartesian
- coalesce
- repartition
- ... and others!

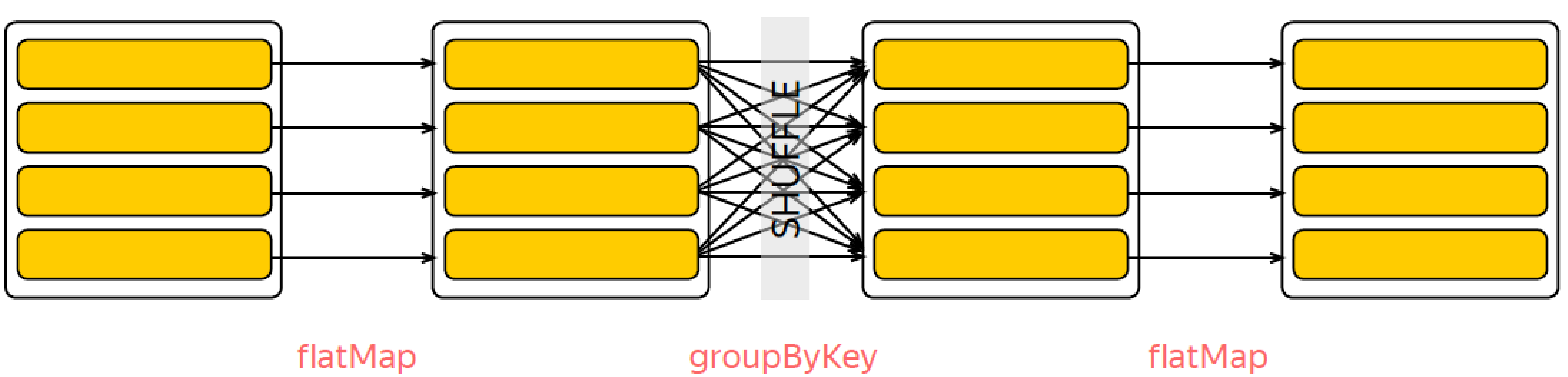


RDD의 flatMap()과 map()의 차이

MapReduce in Spark

› Example: Y = X.flatMap(*m*).groupByKey*()*.flatMap(r*)*

X	: *RDD[T]*	
.flatMap(m*)*	: *RDD[(K, V)],*	m: *T* ➡ *Array[(K, V)]*
.groupByKey*()*	: *RDD[(K, Array[V])]*	
.flatMap(r*)*	: *RDD[U],*	r: *(K, Array[V])* ➡ *Array[U]*



Summary

- Transformation
  - is a description of how to obtain a new RDD from existing RDDs
  - is the primary way to "modify" data (given that RDDs are immutable)
- Transformations are lazy, i.e. no work is done until data is explicitly requested (next video!)
- There are transformations with narrow and wide dependencies
- MapReduce can be expressed with a couple of transformations
- Complex transformations (like joins, cogroup) are available

Actions

Trigger computation

Driver & executors

› Driver program runs your Spark application

driver program receives only the outcome

›› Driver delegates tasks to executors to use cluster resources

하둡의 resource manager 역할과 비슷하다.

›› In local mode, executors are collocated with the driver

›› In cluster mode, executors are located on other machines

Actions

›› Triggers data to be materialized and processed on the executors and then passes the outcome to the driver

› Example: actions are used to collect, print and save data

Frequently used actions

- collect()
  - collects items and passes them to the driver
  - for small datasets! all data is loaded to the driver memory

    memory 용량에 대한 인식이 필요하다.

- take(n: Int)
  - collects only n items and passes them to the driver
  - tries to decrease amount of computation by peeking on partitions

- top(n: Int)
  - collects n largest items and passes them to the driver

- reduce(f: (T, T) ➡ T)
  - r educes all elements of the dataset with the given associate, commutative binary function and passes the result back to the driver

Frequently used actions

- saveAsTextFile(path: *String*)
  - e ach executor saves its partition to a file under the given path with every item converted to a string and confirms to the driver

- saveAsHadoopFile(path: *String*, outputFormatClass: *String*)
  - ››each executor saves its partition to a file under the given path using the given Hadoop file format and confirms to the driver

- foreach(f: *T* ➡ *()*)
  - each executor invokes f over every item and confirms to the driver

  foreach는 map과 달리 return이 없고 RDD를 만들지 않는다.

- foreachPartition(f: *Iterator[T]* ➡ *()*)
  - each executor invokes f over its partition and confirms to the driver

Summary

›› Actions trigger computation and processing of the dataset

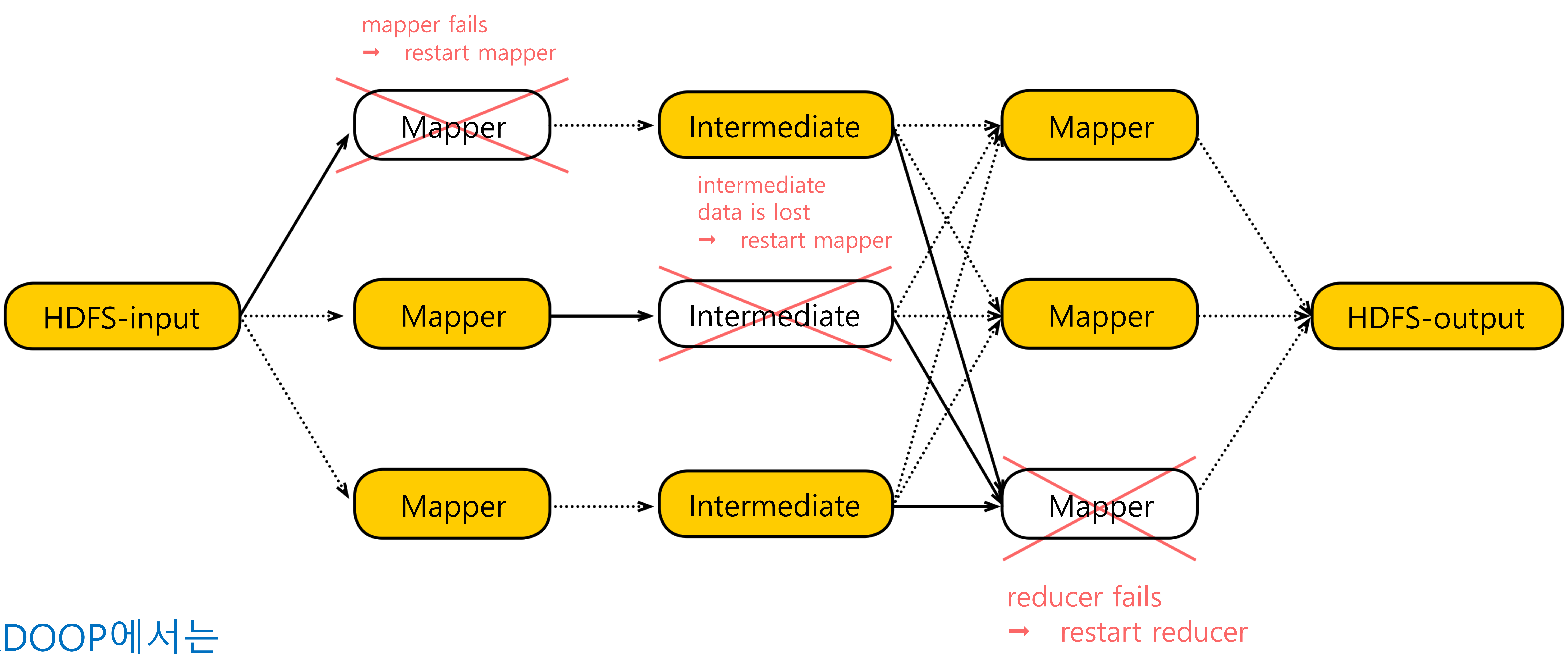
›› Actions are executed on executors and they pass results back to the driver

›› Actions are used to collect, save, print and fold data

Resiliency

Fault-tolerance in MapReduce

›› Two key aspects
  ››reliable storage for input and output data
  ››deterministic and side-effect free execution of mappers and reducers



HADOOP에서는
1. 데이터를 디스크에 안전하게 저장
2. deterministic function 으로 fault-toleranc를 보장한다.

Fault-tolerance in Spark

- Same two key aspects 하둡과 같은 기준을 사용
  - reliable storage for input and output data
  - deterministic and side-effect free execution of transformations(including closures)

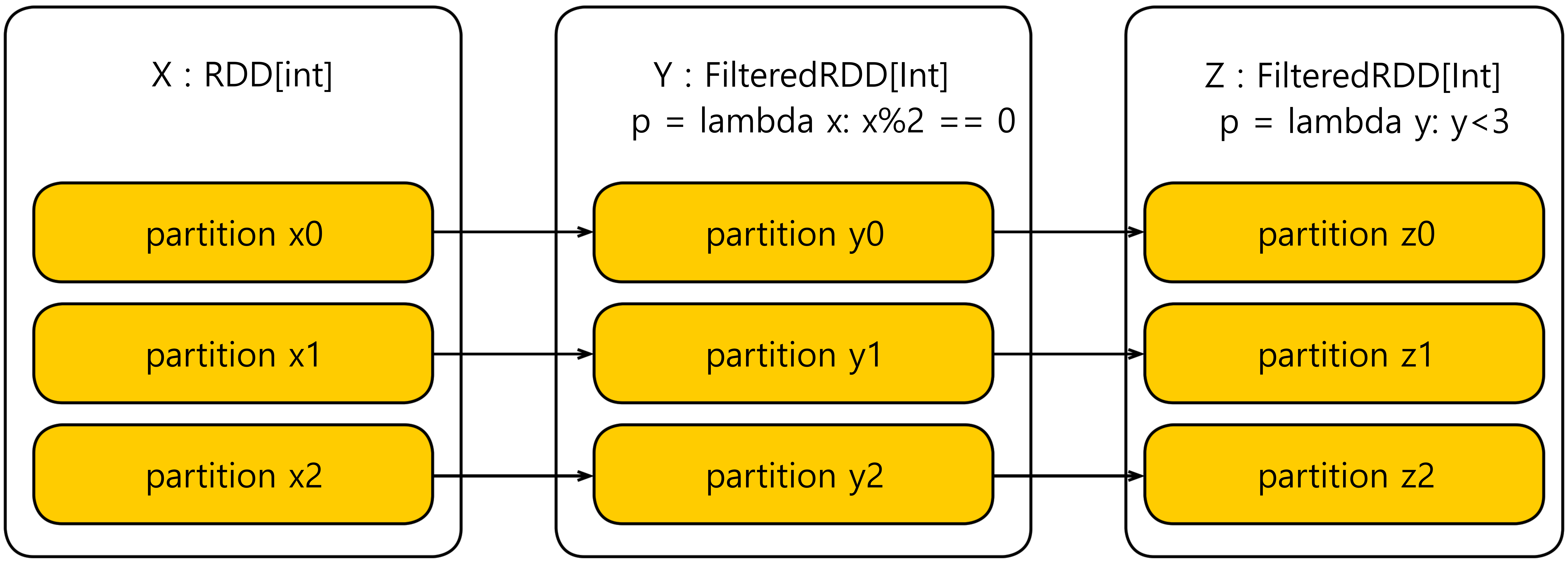
- Determinism — every invocation of the function results in the same returned value
  - e. g. do not use random numbers, do not depend on a hash value order

- Freedom of side-effects — an invocation of the function does not change anything in the external world
  - e. g. do not commit to a database, do not rely on global variables

Fault-tolerance & transformations

› Lineage — a dependency graph for all partitions of all RDDs involved in a computation up to the data source

To decide what to restart : 문제점을 파악하기 위한 것

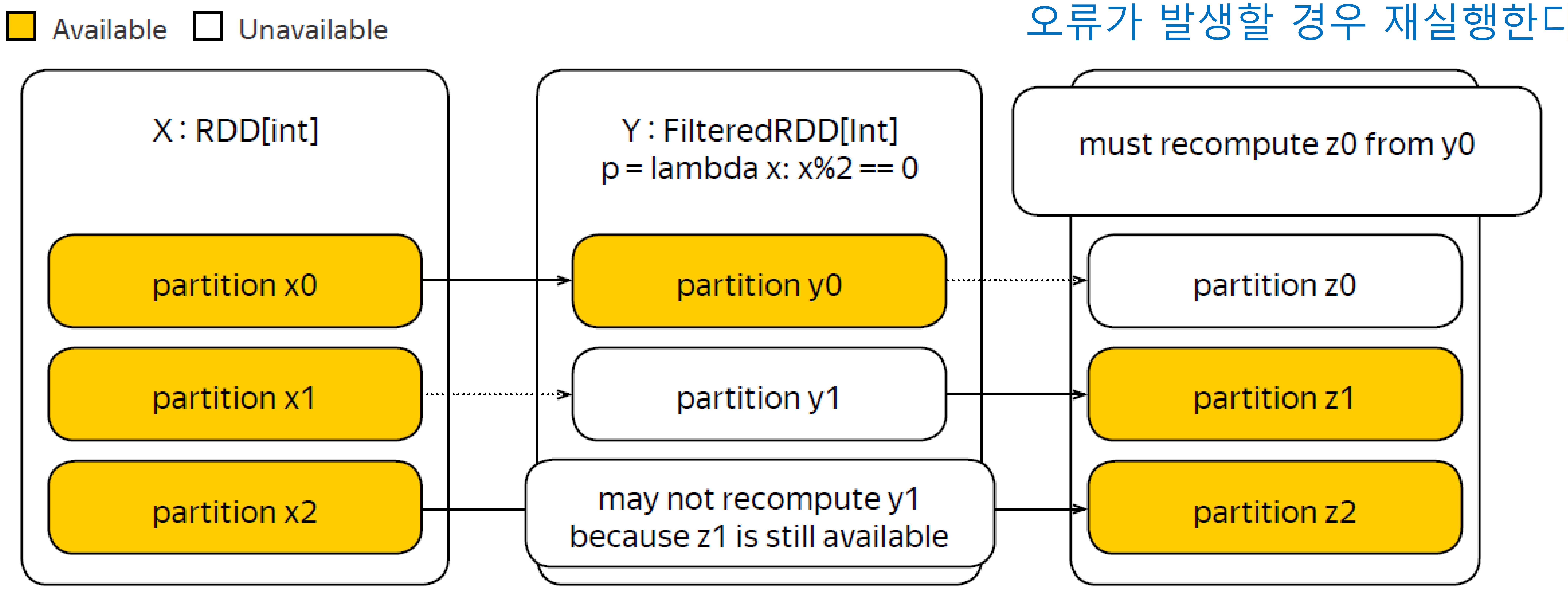


Fault-tolerance & transformations

› Lineage — a dependency graph for all partitions of all RDDs involved in a computation up to the data source

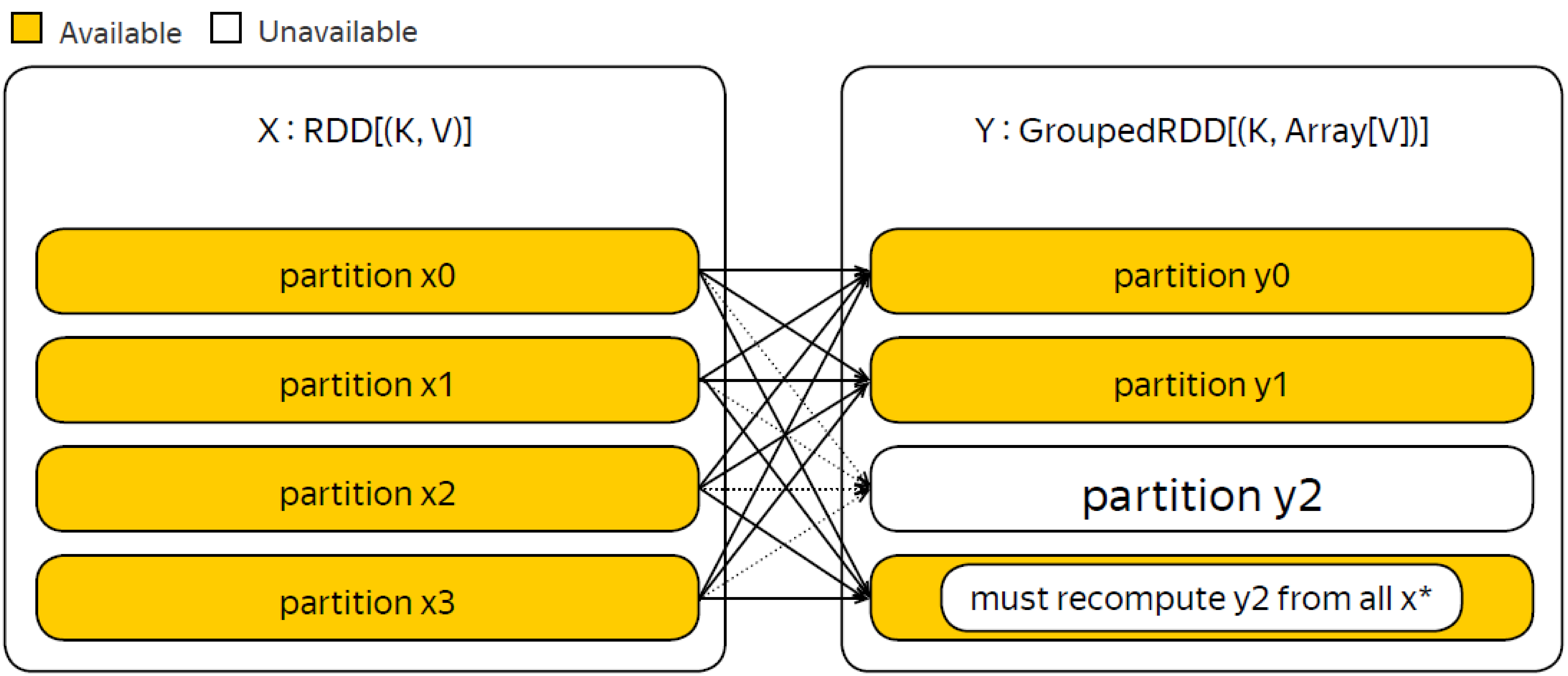
Detection is done in the driver

오류가 발생할 경우 재실행한다.



Fault-tolerance & transformations

› Lineage — a dependency graph for all partitions of all RDDs involved in a computation up to the data source



wide dependencies의 경우, restart가 expensive하다.
(모든 partition이 살아있어야 하기 때문)

Fault-tolerance & actions

›› Actions are side-effects in Spark

restart 때문에 action이 딱 한 번 실행된다는 보장을 할 수 없다.

›› Actions have to be idempotent that is safe to be re-executed multiple times given the same input

여러 번 실행되더라도 결과는 항상 같아야한다는 의미

› Example: collect()

› the dataset is immutable;
thus reading it multiple times is safe

› Example: saveAsTextFile()

›› the dataset is immutable; safely overwrite
thus file would be the same after every write

Summary

- Resiliency is implemented by
  - tracking lineage
  - assuming deterministic & side-effect free execution of transformations(including closures)
  - assuming idempotency for actions

BigDATAteam