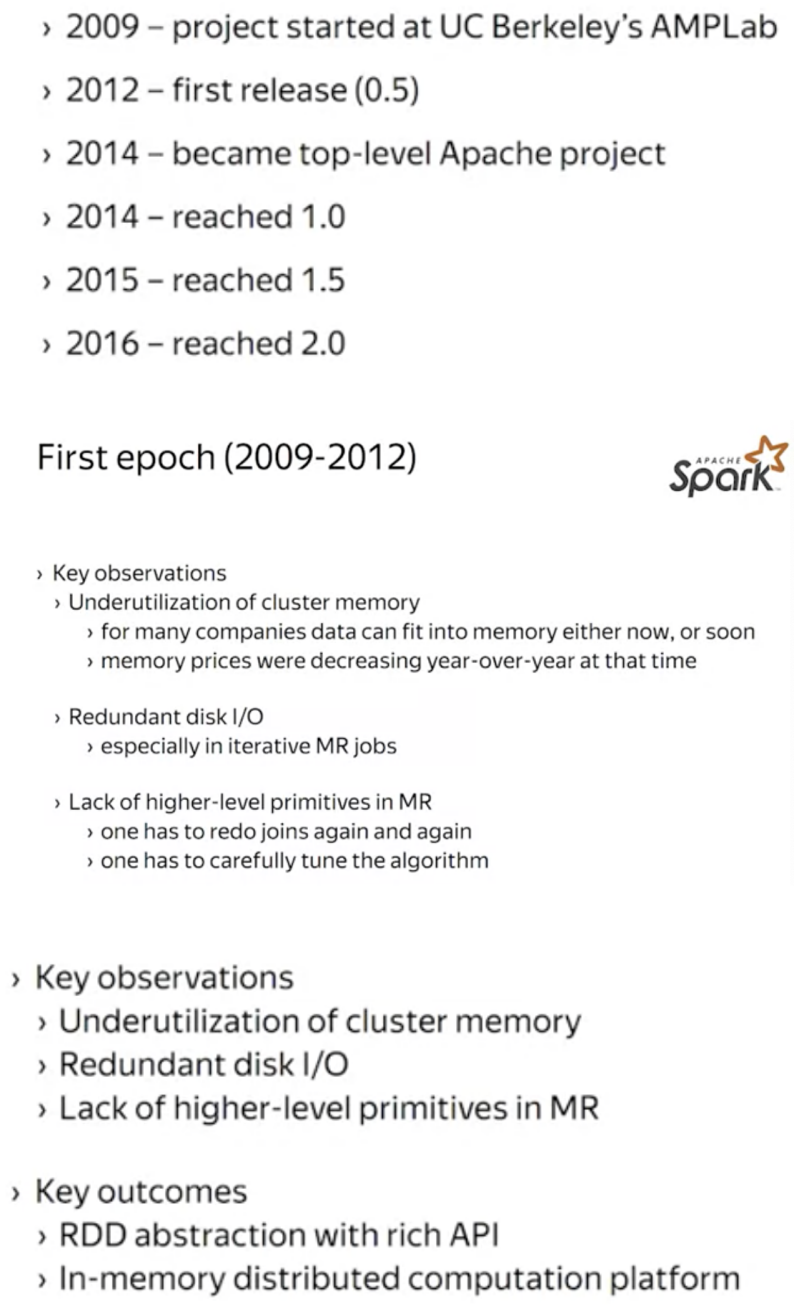
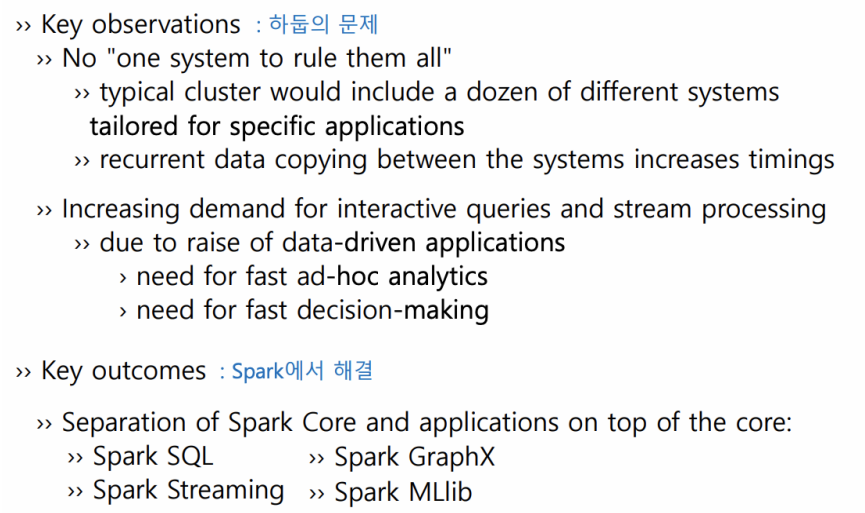
Week4: Apache Spark Basic concepts

Historical background

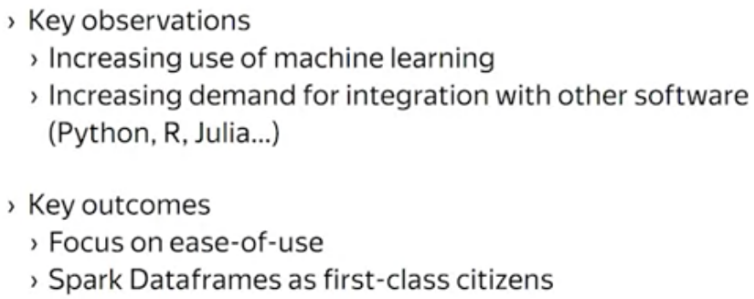


하둡의 문제: MapReduce가 memory를 제대로 사용하지 않음, 속도가 느려지고 framework 사용이 어려움 ⇒ spark에서 해결

the framework has to guarantee durability of the result and hence read and write the dataset on every iteration..



batch processing, stream processing, graphical computation 등 하나의 framework로 통합



RDDs (Resilient Distributed Datasets)

Why do we need a new abstraction?

› Example: iterative computations (K-means, PageRank, ...)
  › relation between consequent steps
    is known only to the user code
  › framework must reliably persist data between steps
    (even if it is temporary data)

computational optimization 불가, intermediate 데이터 disk에 저장되어 불필요한 IO 발생

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

Resilient: able to withstand failures

Distributed: spanning across multiple machines

› 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:
  › 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

typedness is an important property to catch bugs early on..

- 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

- 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

- 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

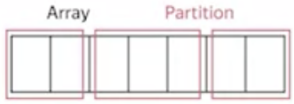
› dependencies() → *Array[Dependency]*
  › return an empty array (no dependencies)

- sliced In-memory array: chunk of source array (병렬처리를 위한)

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



› 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

› Next video: Transformations

▼ data in stable storage.. ex) files in HDFS, objects in Amazon S3 bucket, lines in a text file,...

▼ RDD for data in a stable storage has no dependencies

Transformations

▼ From existing RDDs by applying a transformations ex) filtered file, grouped records..

▼ RDD for a transformed data depends on the source data

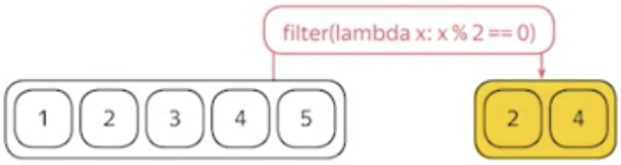
› Allow you to create new RDDs from the existing RDDs by specifying how to obtain new items from the existing items

› The transformed RDD depends implicitly on the source RDD

create new RDDs.. NOT MODIFY data in-place in Spark..

- **filter()**

› Def: filter(p: *T → Boolean*): *RDD[T] → RDD[T]*
  › returns a filtered RDD with items satisfying the predicate p



- **map()**

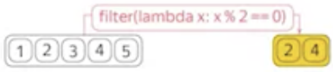
› Def: map(f: *T → U*): *RDD[T] → RDD[U]*
  › returns a mapped RDD with items f(x) for every x in the source RDD

- **FlatMap()**

› Def: flatMap(f: *T → Array[U]*): *RDD[T] → RDD[U]*
  › same as map but flattens the result of f
  › generalizes map and filter

- **Filtered RDD**

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

Lazy 기법

- Y.partitions() → *Array[Partition]*

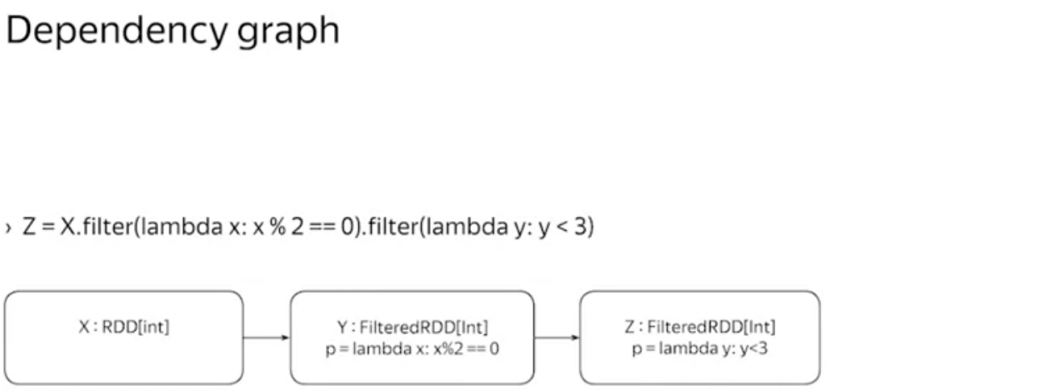
> Note that actual filtering happens not at *the creation time* of Y, but at *the access time* to the iterator over a partition of Y.
>
> Same holds for other transformations – they are lazy, i.e. they compute the result only when accessed.

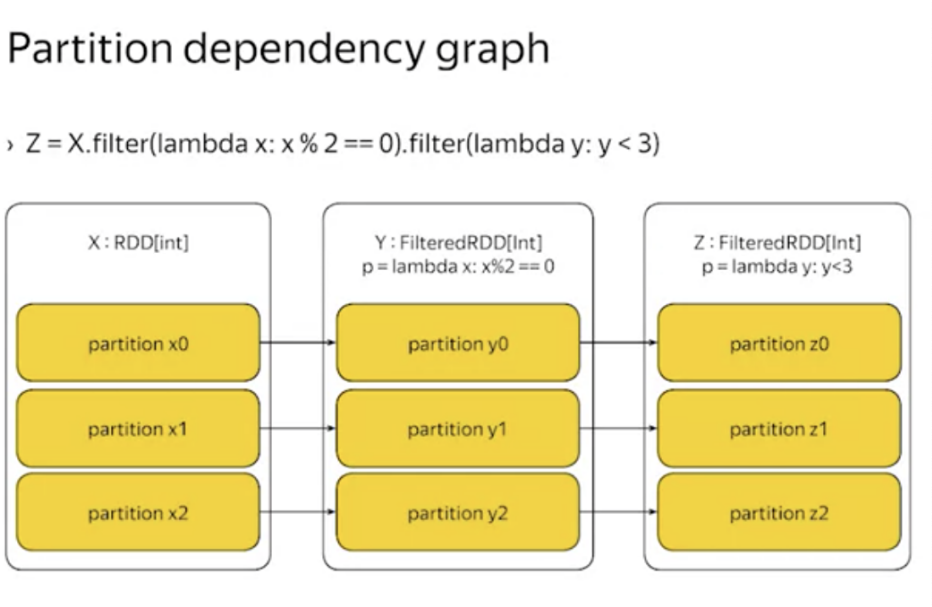
- Y.dependencies() → *Array[Dependency]*

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)

implicitly 생성..
used to schedule
by framework





the function actually applied to the items (ex if you invoke the 'map' transformation on the dataset) in the FUTURE, when the data is ACTUALLY REQUESTED. on the EXECUTOR MACHINE which may or may not be the local machine..

Keyed transformations

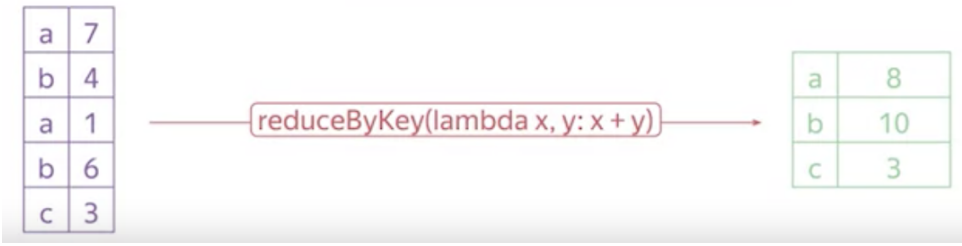
keyed RDD is an RDD of key-value pairs

- **groupByKey()**

› Def: 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

- **reduceByKey()**

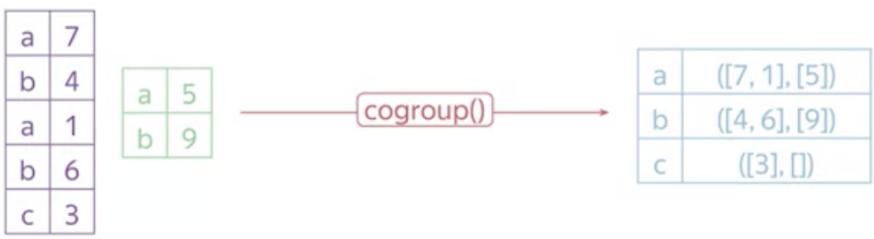
› Def: reduceByKey(f: (V, V) → V): RDD[(K, V)] → RDD[(K, V)]
› folds all values with the same key using the given function f
› returns a set of the folded values with corresponding keys



- **cogroup()**

in MapReduce framework, in the MAPPER, add an input tag (left or right) to every value in the input key-value pairs; emit the tagged values with the original keys. in the REDUCER, distribute values to an array according to the value tag.

› 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



How to compute an inner join from the result of cogroup?

That is, all triples (k, x, y) where (k, x) is in X and (k, y) is in Y.

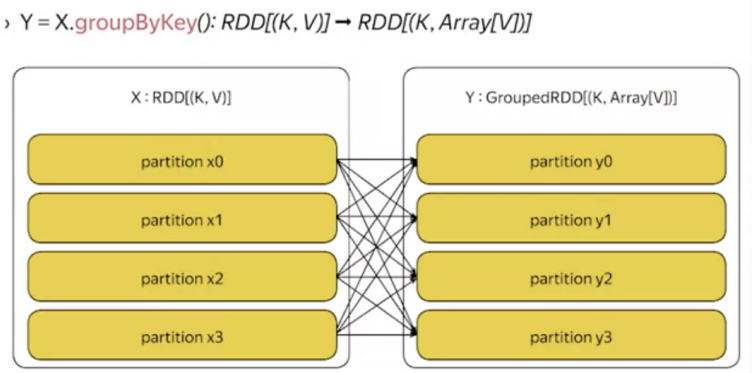
- **join()**

› 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

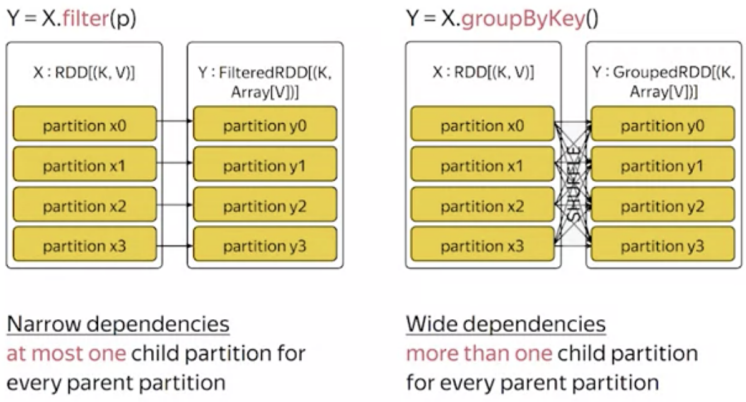
› Also: X.leftOuterJoin, X.rightOuterJoin, X.fullOuterJoin

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



Narrow & Wide dependencies



Narrow dependencies
at most one child partition for every parent partition

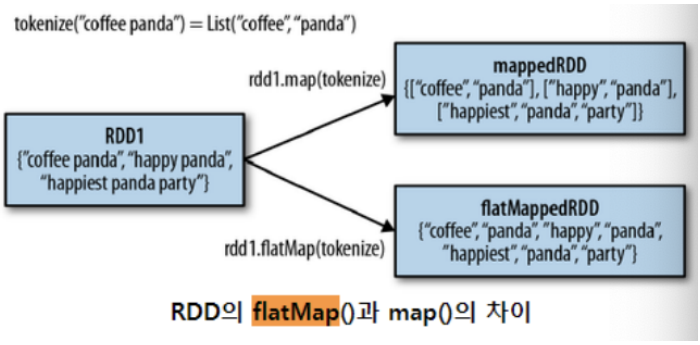
Wide dependencies
more than one child partition for every parent partition

groupByKey()는 pair 찾기위해 모든 RDD 스캔해야 함

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

Plenty of transformations!

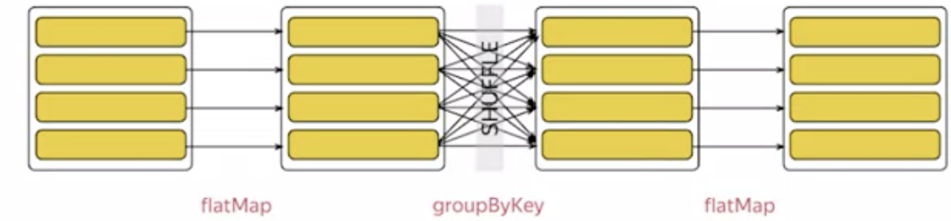
- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- mapValues
- sample
- distinct
- union
- intersection
- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join
- cogroup
- cartesian
- coalesce
- repartition
- ... and others!



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]



You CANNOT produce a CYCLIC dependency graph by applying Spark transformations to RDDs.. the transformation creates a new RDD every time.

- 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 delegates tasks to executors to use cluster resources
- In local mode, executors are collocated with the driver
- In cluster mode, executors are located on other machines
- More in the next lesson

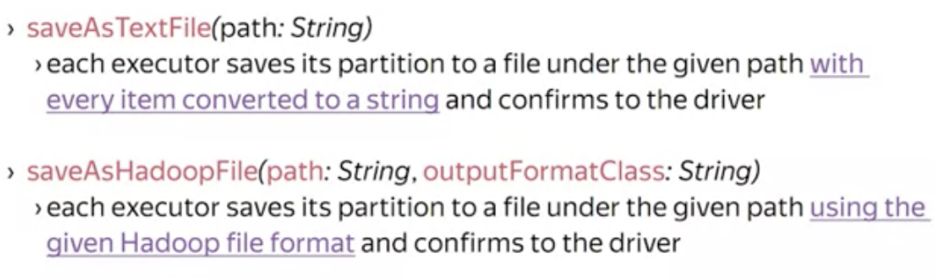
hadoop 의 manager.. driver program receives only the outcom.

Actions

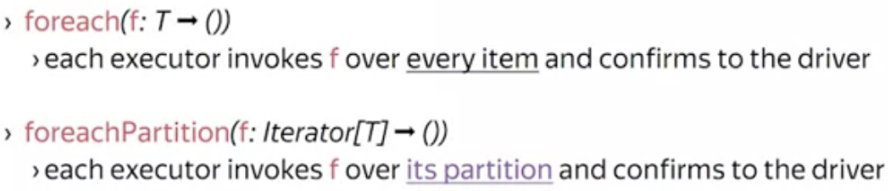
- Triggers data to be materialized and processed on the executors and then passes the outcome to the driver
- Example: actions are used to collect, print and save data

Frequently used actions

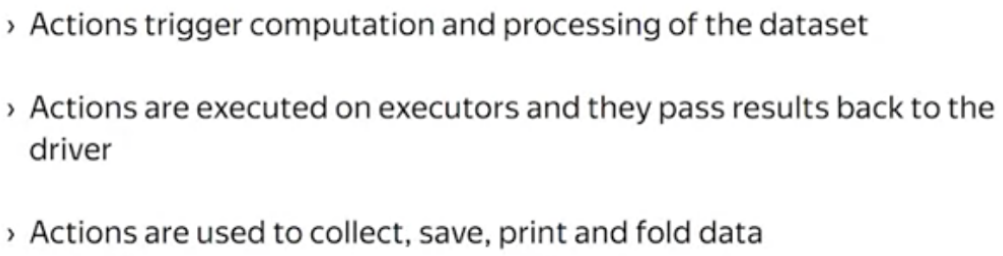
- collect()
  - collects items and passes them to the driver
  - for small datasets! all data is loaded to the driver memory
- take(n: *Int*)
  - collects only n items and passes them to the driver
  - tries to decrease amount of computation by peeking on partitions
- top(n: *Int*)
  - collects n largest items and passes them to the driver
- reduce(f: *(T, T) → T*)
  - reduces all elements of the dataset with the given associate, commutative binary function and passes the result back to the driver



saveAsHadoopFile() most common way to save file to HDFS. USER-PROVIDED file format is used to serialize data in this action.



foreach는 map과 달리 return이 없고 RDD를 만들지 않음

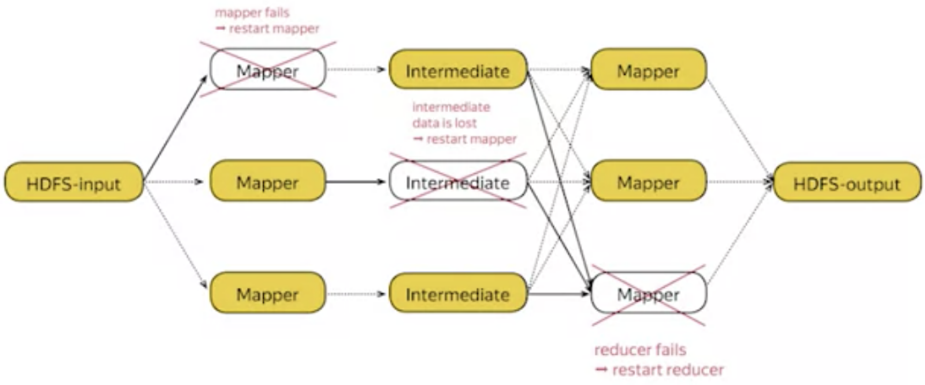


Resiliency

is about how it is possible to continue operation despite machine failures in the cluster..

Fault-tolerance in MapReduce

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



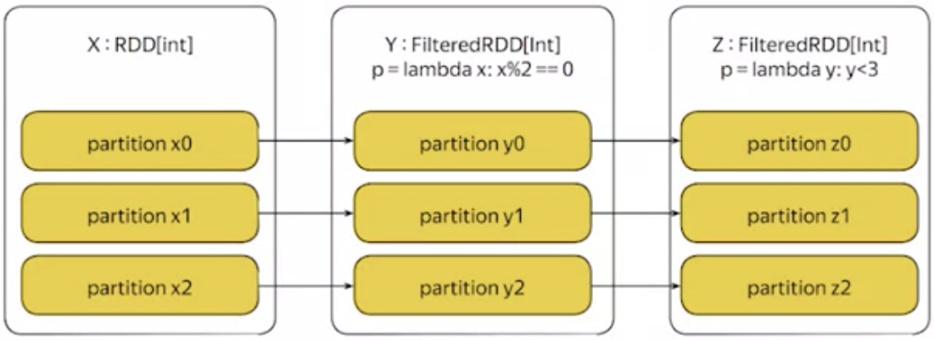
› Determinism — every invocation of the function results in the same returned value
  › e. g. do not use random numbers, do not depend on a hash value order

› 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

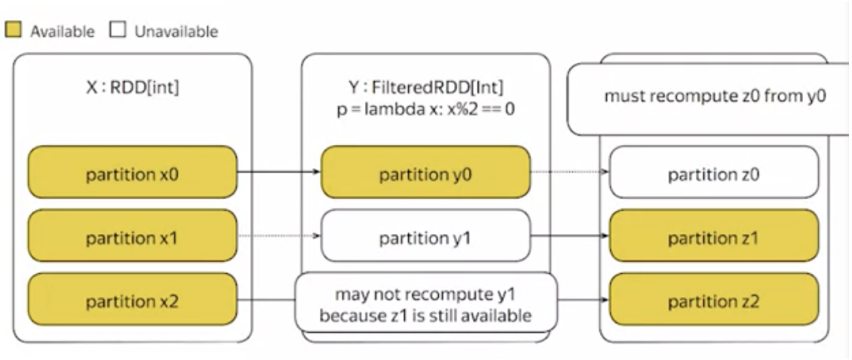
함수는 항상 같은 값을 반환해야 한다, 함수로 인해 외부 세계의 그 무엇도 바뀌지 않아야한다

Fault-tolerance & transformations

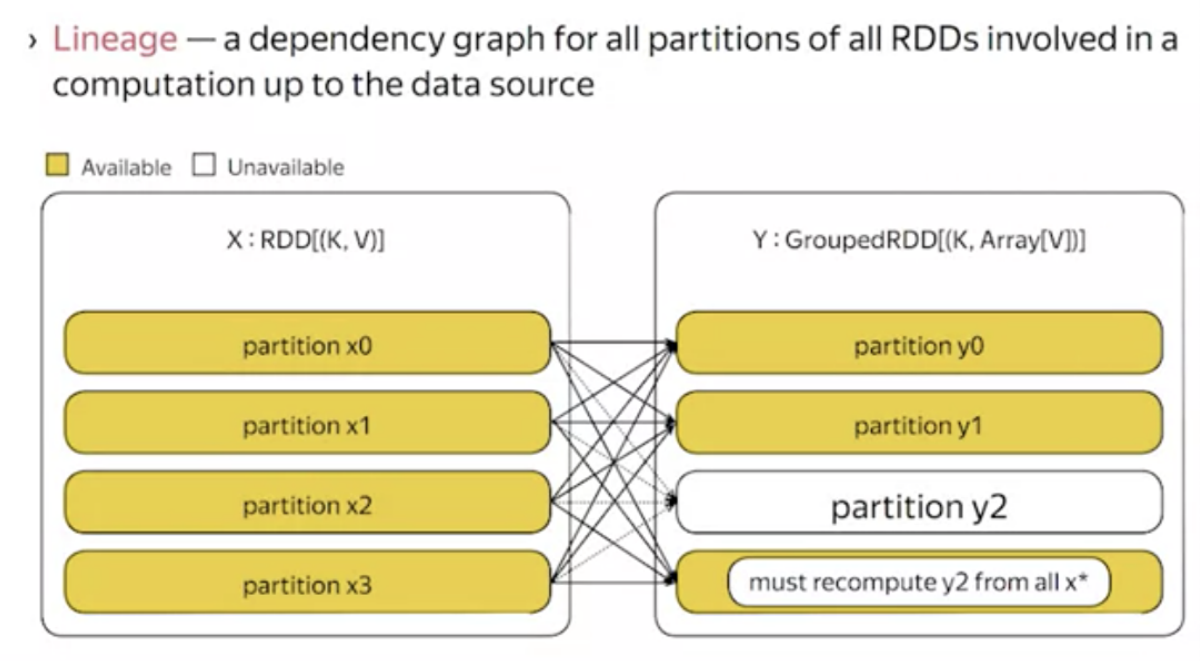
› Lineage — a dependency graph for all partitions of all RDDs involved in a computation up to the data source



어디서 오류가 발생했는지 알기 위해 tracking



dependency역시 fail이면 재귀적으로 restart



wide dependencies의 경우 모든 partition이 살아있어야하기 때문에 restart가 expensive함

Fault-tolerance & actions

› Actions are side-effects in Spark

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

› Example: collect()
  › the dataset is immutable;
  thus reading it multiple times is safe

› Example: saveAsTextFile()
  › the dataset is immutable;
  thus file would be the same after every write

restart 때문에 action이 딱 한번 실행된다는 보장 없음. → 여러번 실행되더라도 결과는 항상 같아야함

⇒ immutable

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

- May improve resiliency by increasing durability of RDDs
  - in the next lesson!

Lesson 1 (Quiz)