RESILIENT DISTRIBUTED DATASETS

Presented by: Le Ngoc Thanh

Outline

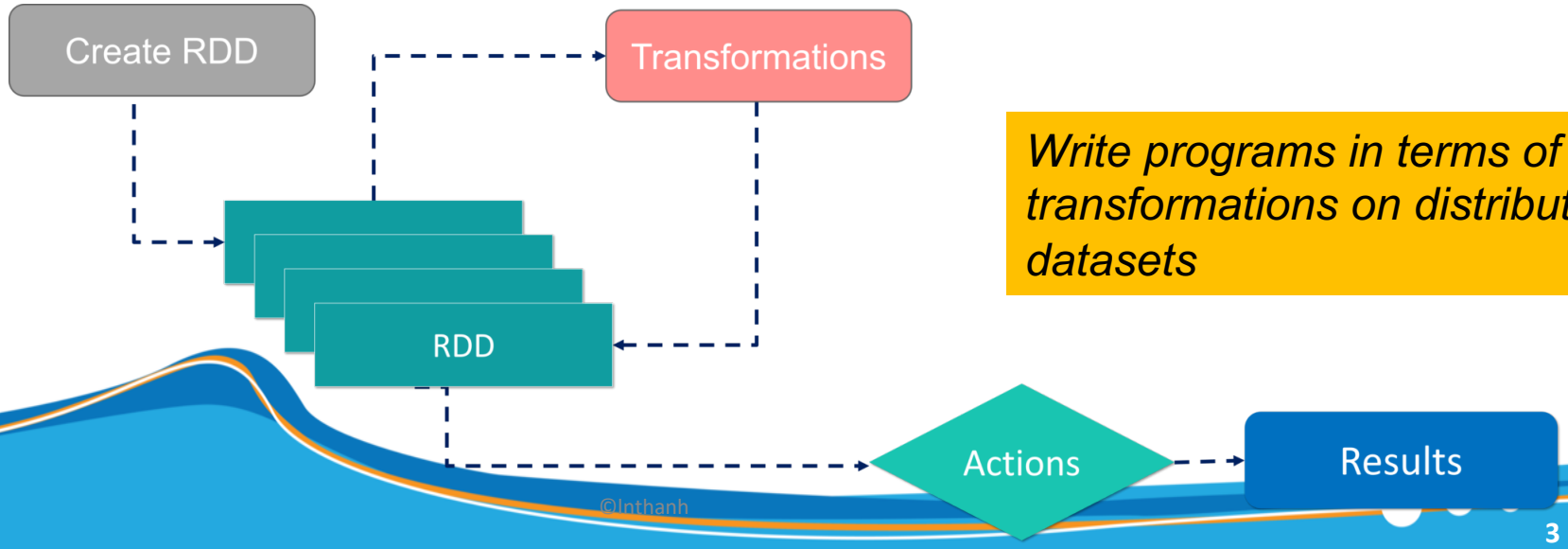
- What Are the Low-Level APIs?
- Resilient Distributed Datasets (RDDs)
- Advanced RDDs
- Broadcasts and Accumulators

@lnthanh

Life of a Spark Program

fit@hcmus

1) Create some input RDDs from external data or parallelize a collection in your driver program.
2) Lazily ***transform*** them to define new RDDs using transformations like filter() or map()
3) Ask Spark to cache() any intermediate RDDs that will need to be reused.
4) Launch ***actions*** such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark

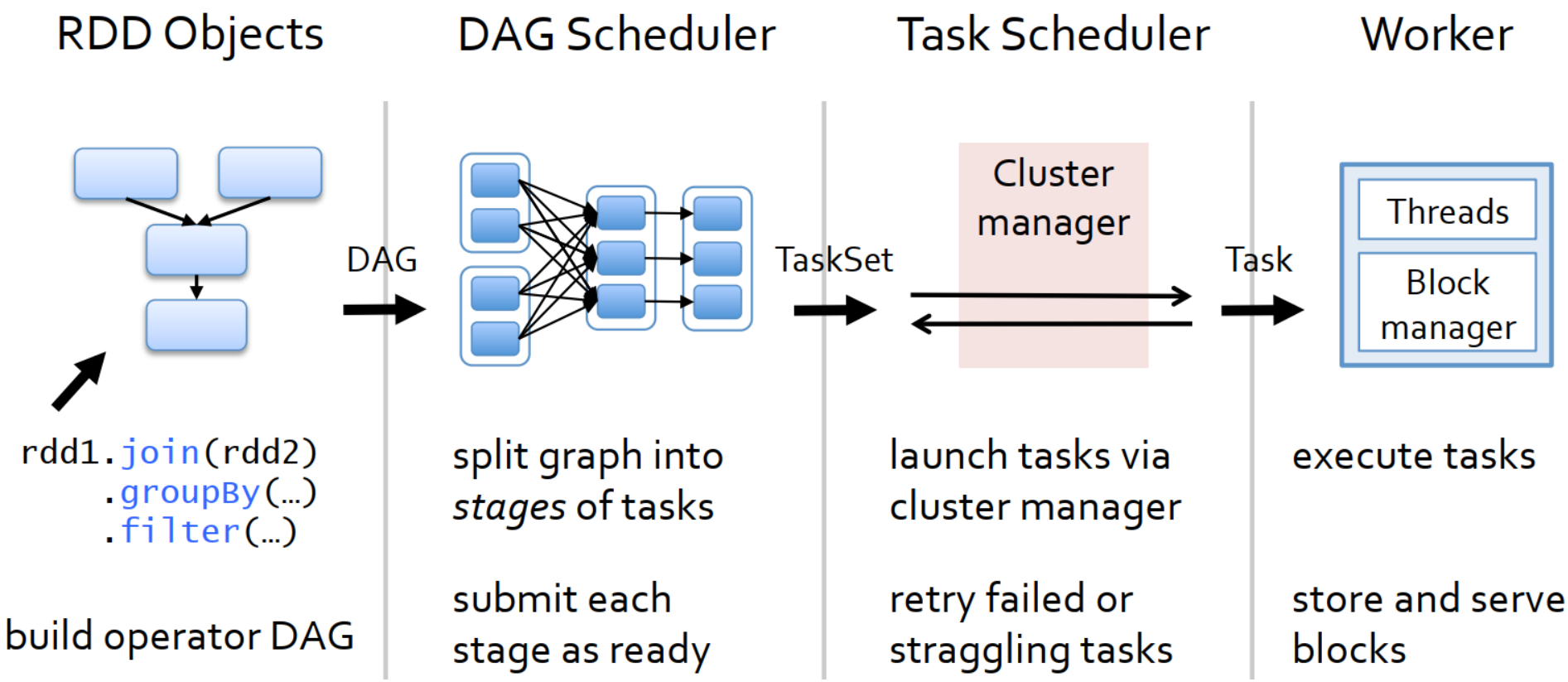


Write programs in terms of transformations on distributed datasets

@lnthanh

RDD -> Stages -> Tasks

fit@hcmus

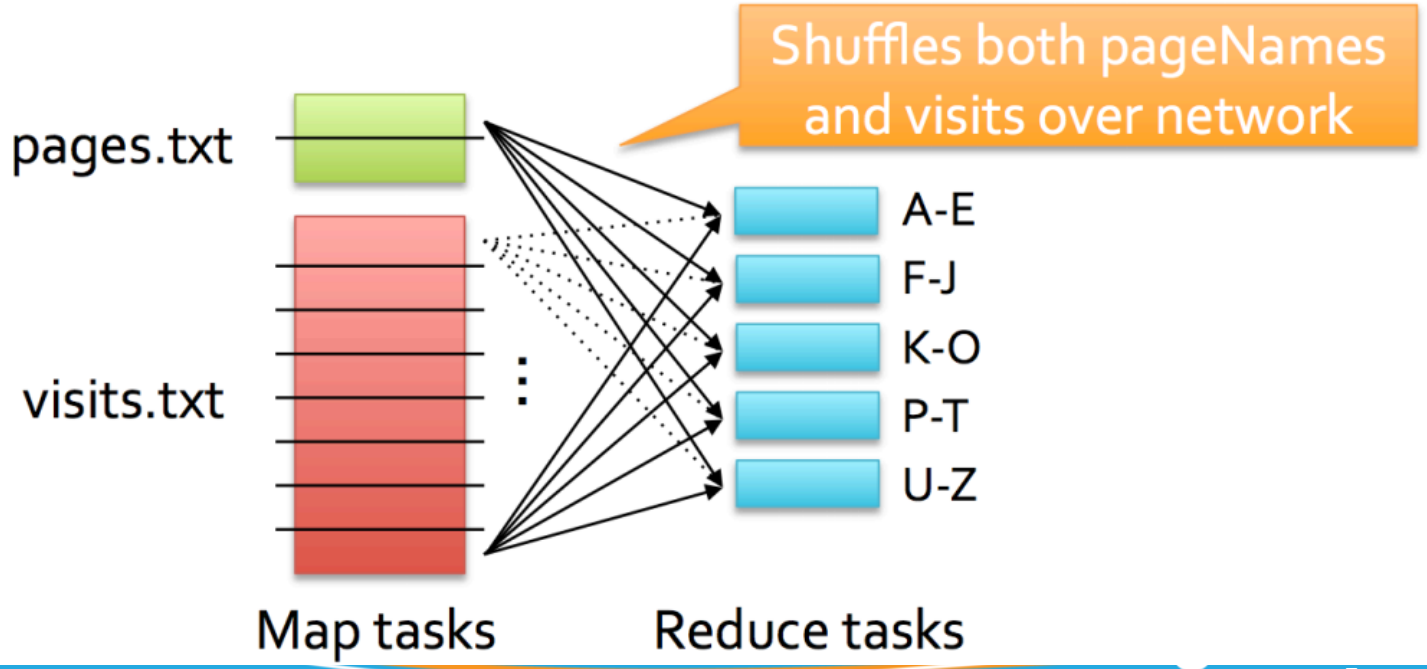


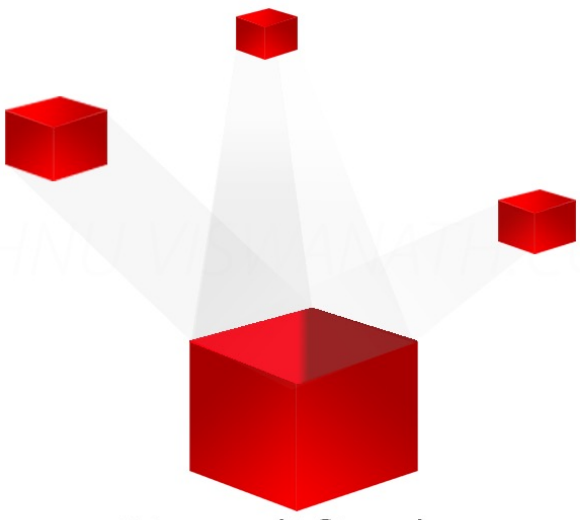
Simple Example

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.join(pageNames)
```



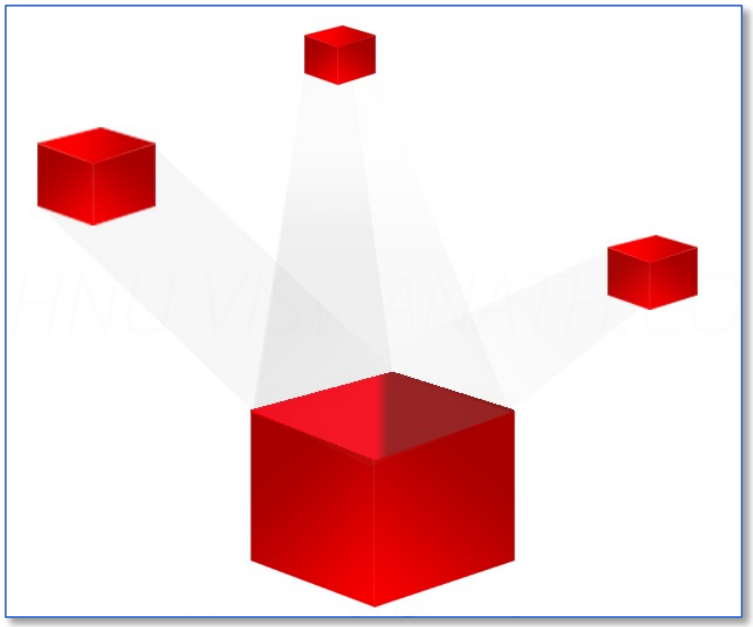


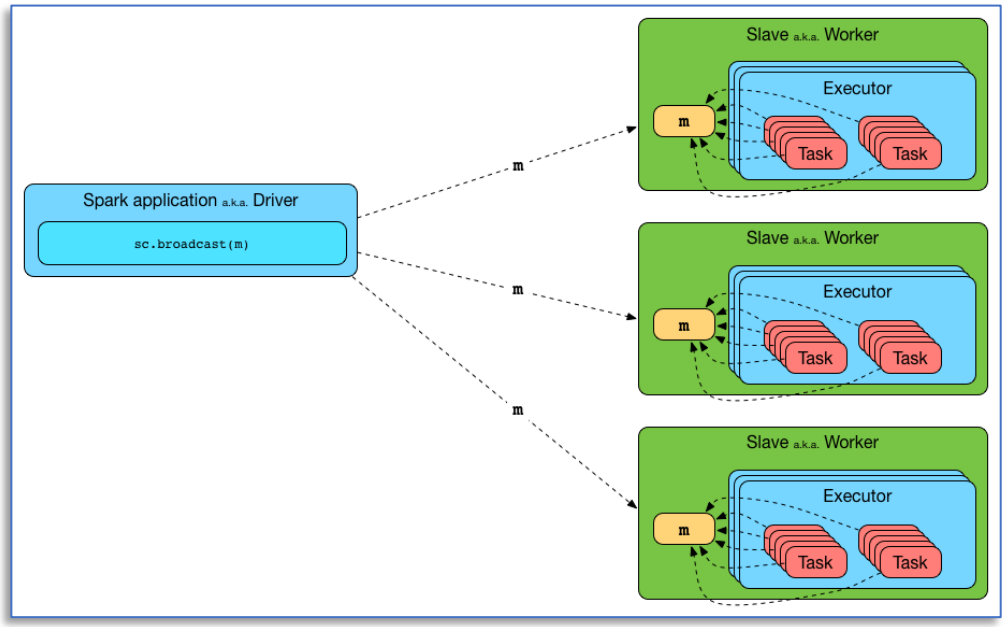
What are Low-level APIs?

Those for manipulating distributed data and distributed shared variables

What are Low-level APIs?

fit@hcmus





Resilient Distributed Datasets (RDDs)

Manipulating distributed data

Broadcast variables and accumulators

Distributing and manipulating distributed shared variables

@lnthanh

When to use Low-level APIs

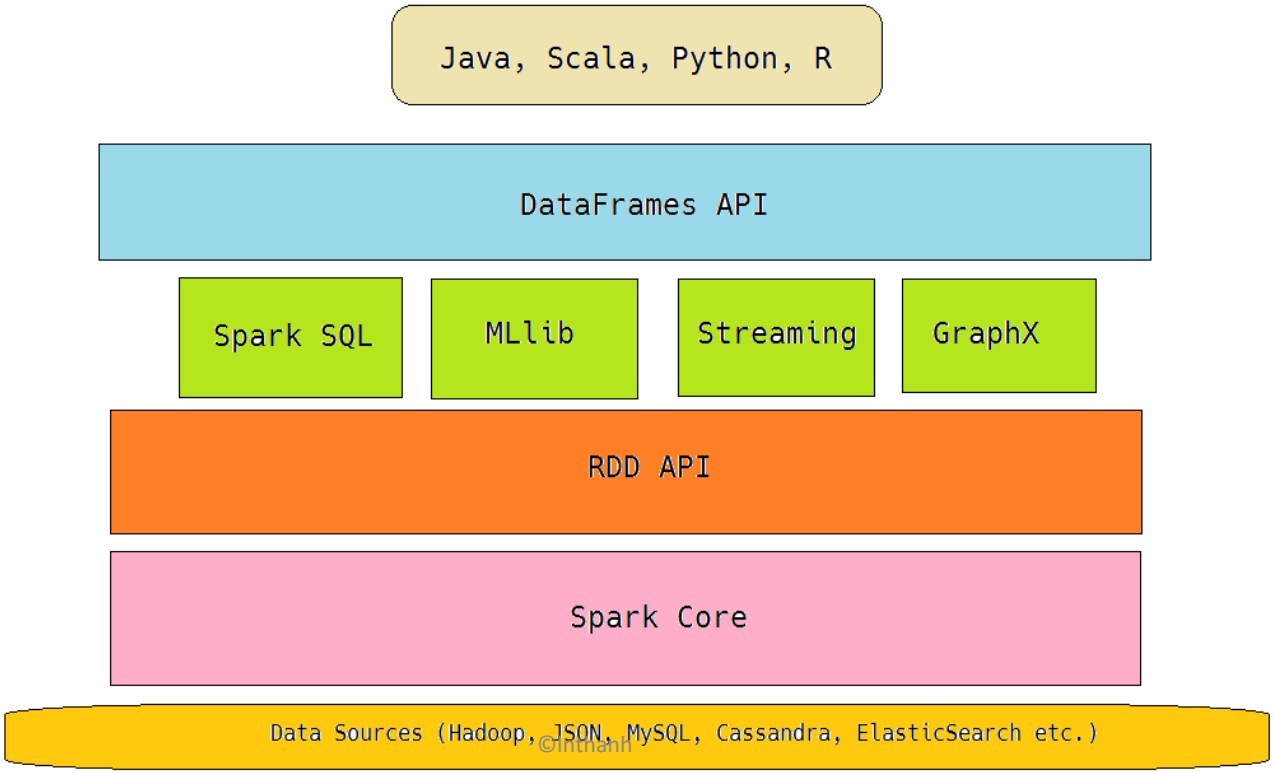
fit@hcmus

- Need some functionality not available in higher level APIs
  - E.g., very tight control over physical data placement across cluster
- Maintain some legacy codebase written using RDDs
- Do some custom shared variable manipulation

- *For the vast majority of use cases, DataFrames will be more efficient, more stable, and more expressive than RDDs.*

Why to understand Low-level APIs

- All Spark workloads compile down to fundamental primitives.
  - E.g., a DataFrame transformation actually becomes a set of RDD transformations

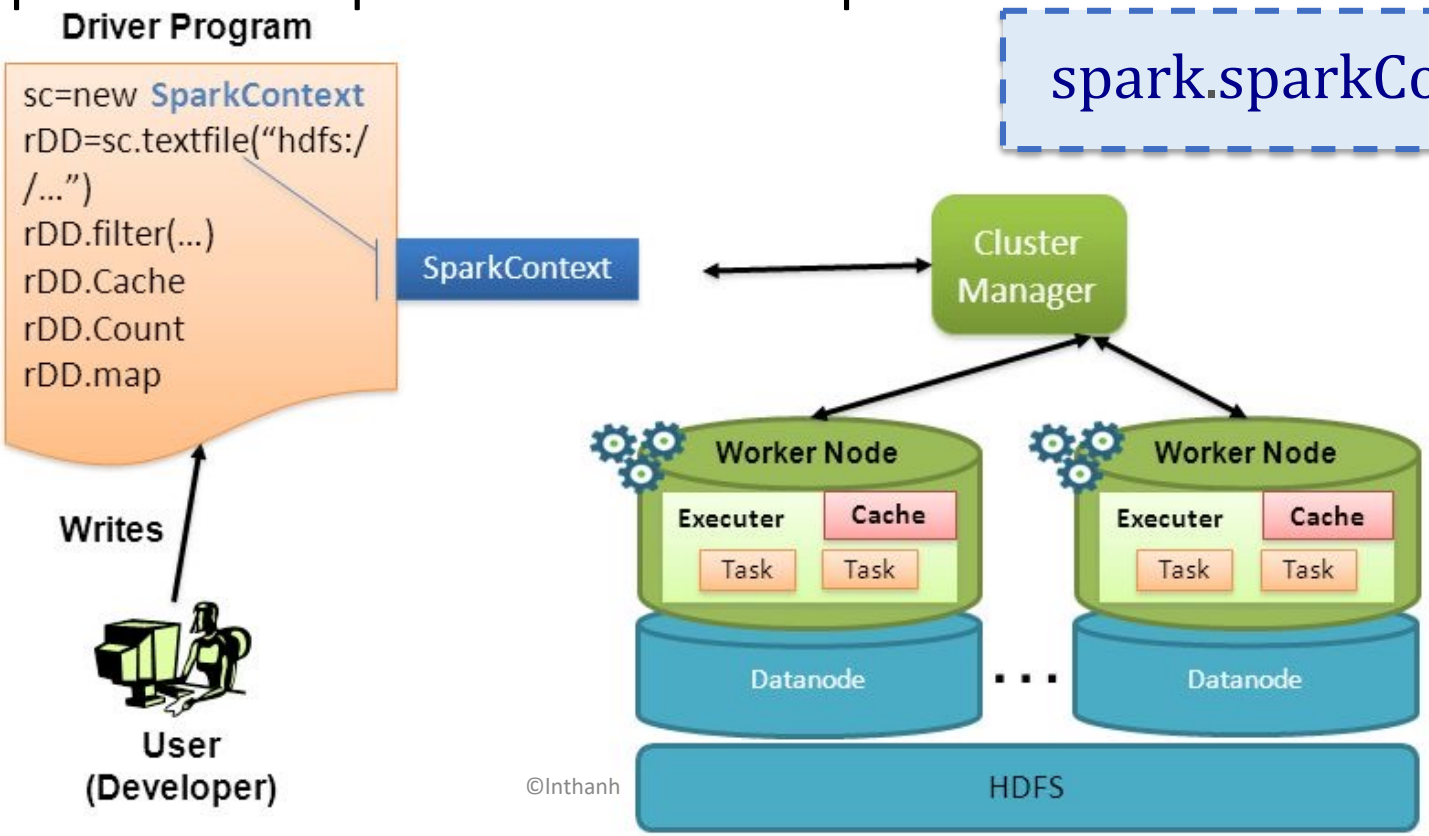


©lntnanh

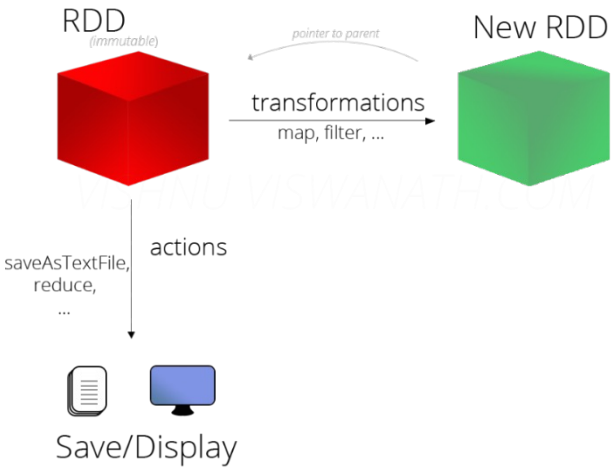
How to use the low-level APIs?

- **SparkContext** is the entry point for low-level API functionality
  - Assess through **SparkSession**, which is the tool you use to perform computation across a Spark cluster.

spark.sparkContext



©lnthanh



Resilient Distributed Datasets

An immutable, partitioned set of records that can be operated on in parallel

Resilient Distributed Datasets

fit@hcmus

- A RDD represents an immutable, partitioned collection of records that can be operated on in parallel.
- Records are Java/Scala/Python objects, following the choice of the programmer.
  - Meanwhile, each DataFrame record is a structured row containing fields with a known schema.

- *RDDs were the primary API in the Spark 1.X series*
- *They are still available in 2.X, yet not as commonly used.*

@lnthanh

Resilient Distributed Datasets

fit@hcmus

- **Great power** but not without **potential issues**.

✓ These objects can store anything in any format.

✗ Every manipulation and interaction between values must be defined by hand.

✗ Optimizations are going to require much more manual work.

  - Spark does not understand the inner structure of the records as it does with the Structured APIs

Types of RDDs

fit@hcmus

- There are lots of subclasses of RDD
- Most of which are for DataFrame APIs to create optimized physical execution plans.
- Users will likely only be creating two types of RDDs

Generic RDD

Key-value RDD

  - Key-value RDDs have special operations as well as a concept of custom partitioning by key.

@lnthanh

Properties of RDDs

fit@hcmus

Required

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs

Optional

- A Partitioner for key-value RDDs
  - E.g., to say that the RDD is hash-partitioned
- A list of preferred locations on which to compute each split
  - E.g., block locations for a HDFS file

Spark's ability to schedule and execute the user program

@lnthanh

RDD APIs availability

fit@hcmus

- RDD APIs are available in Python, Scala and Java
- Scala and Java: the same performance for the most part
  - The large costs incurred in manipulating the raw objects
- Python: a substantial amount of performance lost.
  - High overhead of serialization between Python process and the JVM

Creating an RDD

- An RDD can be created in ways:
  - By Converting from the Existing Source
  - By Loading an External Dataset
  - By Parallelizing the Collection of Objects
  - By Performing Transformations on the Existing RDDs

By Converting from the Existing Source

fit@hcmus

- Create an RDD from an existing DataFrame or Dataset

```
# in Python
spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

```
# in Python
spark.range(10).rdd
```

  - Scala and Java: from Dataset[T] to RDD[T]
    - E.g., Dataset[Long] → RDD[Long]
  - Python: only from DataFrames to RDDs of type Row

- Inversely, create a DataFrame or Dataset from an RDD

```
# in Python
spark.range(10).rdd.toDF( )
```

@lnthanh

By Loading an External Dataset

fit@hcmus

- You can load an external file onto an RDD.
  - The types of files you can load are csv, txt, JSON, etc.
- Each record represents a line in that text file or files

```
# in Python
spark.sparkContext.textFile("/some/path/withTextFiles")
```

- The whole text file becomes a single record
  - First object: filename, second string object: content

```
# in Python
spark.sparkContext.wholeTextFile("/some/path/withTextFiles")
```

@lnthanh

By Parallelizing the Collection of Objects

fit@hcmus

- When Spark's parallelize method is applied to a group of elements, a new distributed dataset is created. This dataset is an RDD.
- Use the **parallelize** method on a **SparkContext** to turn a single node collection into a parallel collection

```
# in Python
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
.split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

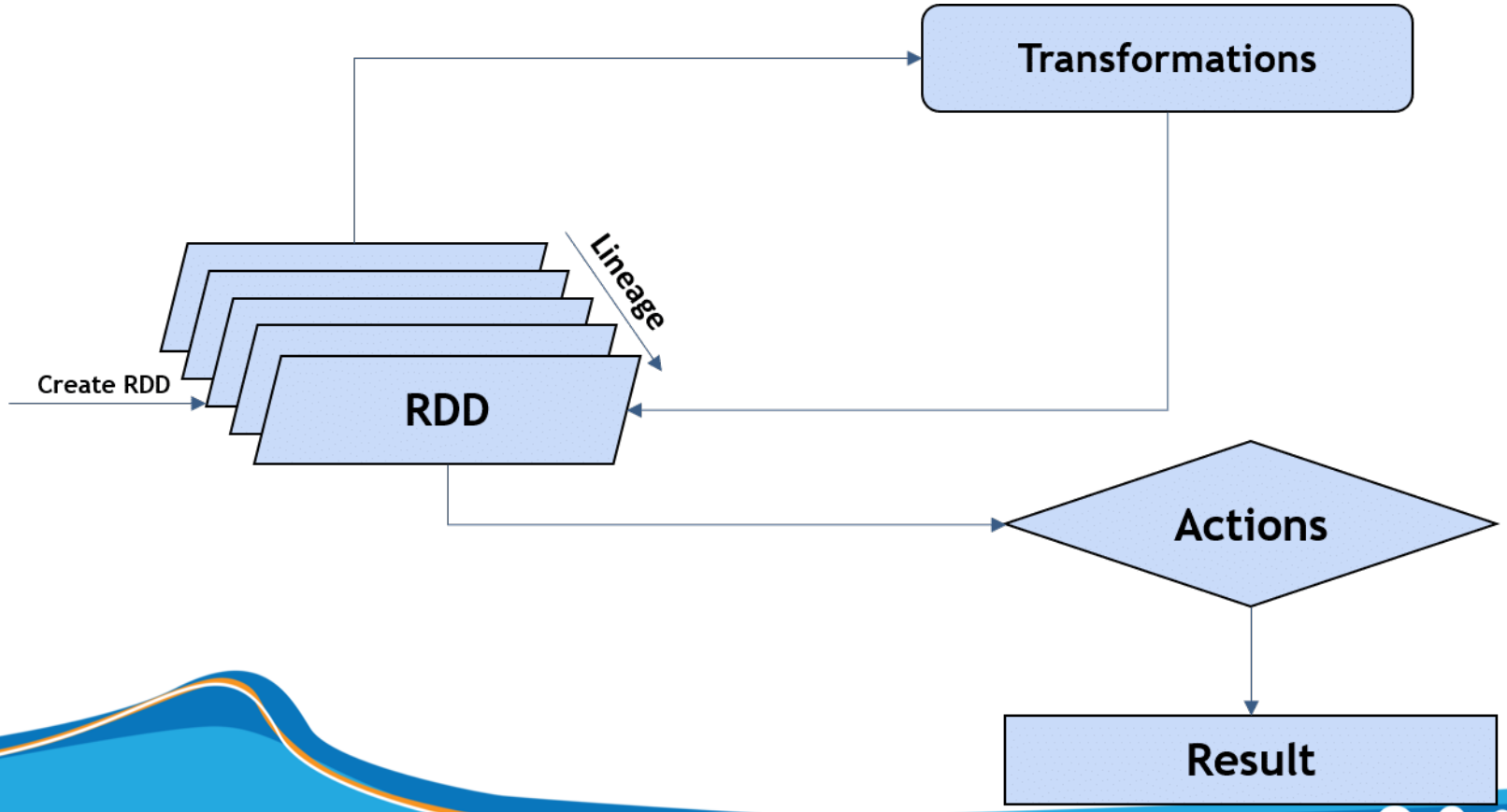
- Name the RDD to show up in the Spark UI

```
# in Python
words.setName("myWords")
words.name()
# myWords
```

@lnthanh

By Performing Transformations on the Existing RDDs

- RDDs can be created by performing transformations on the existing RDDs



RDD Operations

fit@hcmus

- **Transformations**
  - create a new dataset from an existing one
  - The data in the existing RDD in Spark does not change as it is immutable.
- **Actions**
  - return value to the driver program after running a computation on the dataset.
  - Actions are operations that provide non-RDD values.
- Example:
  - *map* is a transformation that passes each dataset element through a function and returns a new RDD representing the results.
  - *reduce* is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program

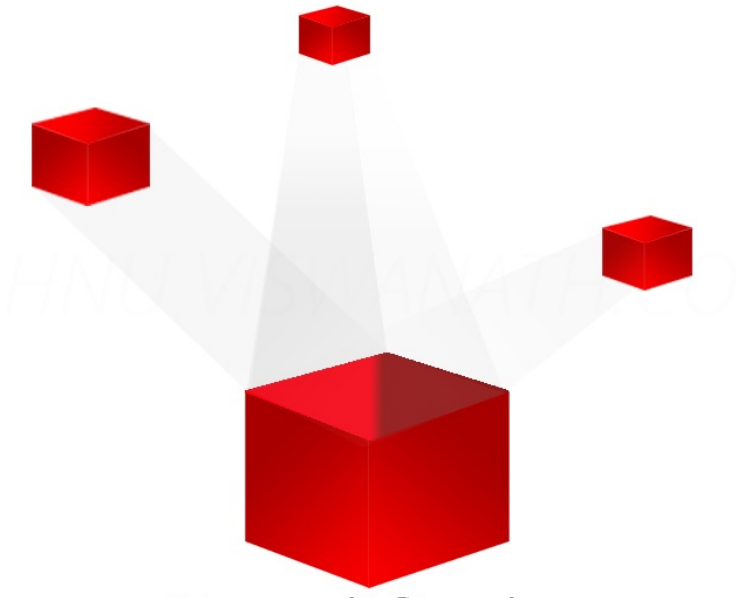
Transformations and Actions

fit@hcmus

- RDDs provide **transformations** and **actions** that work the same as those in DataFrames and DataSets.
  - Evaluate transformations lazily whereas actions eagerly

Transformations	Actions
map(func)	take(N)
flatMap(func)	count()
filter(func)	collect()
groupByKey()	reduce(func)
reduceByKey(func)	takeOrdered(N)
mapValues(func)	top(N)
...	...

@lnthanh



Transformations

RDD Operations

- **Transformations**
  - create a new dataset from an existing one
  - The data in the existing RDD in Spark does not change as it is immutable.
- We will discuss some functions in transformations

Transformations: distinct

fit@hcmus

- Many transformations mirror the functionality found in the Structured APIs.
- **distinct** removes duplicates from the RDD

```
# in Python
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
words.distinct( ).count( )
# 10
```

Transformations: filter

fit@hcmus

- Filtering is equivalent to creating a SQL-like **where** clause.
  - Look through the records in the RDD and see which ones match some predicate function
- Define the filter function

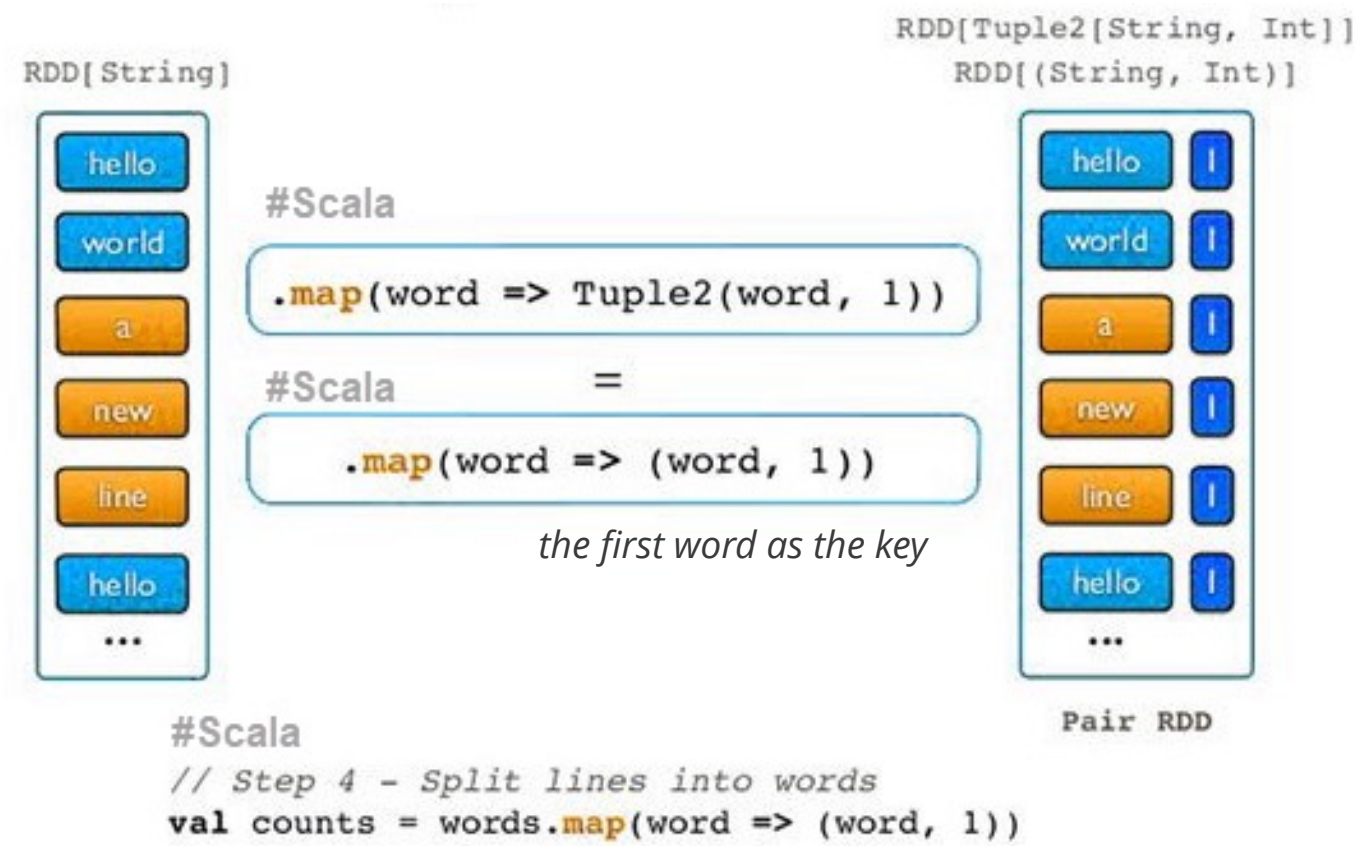
```
# in Python
def startsWithS(individual):
        return individual.startswith("S")
```

```
# in Python
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
words.filter(lambda word: startsWithS(word)).collect()
# ['Spark', 'Simple']
```

@lnthanh

Transformations: map

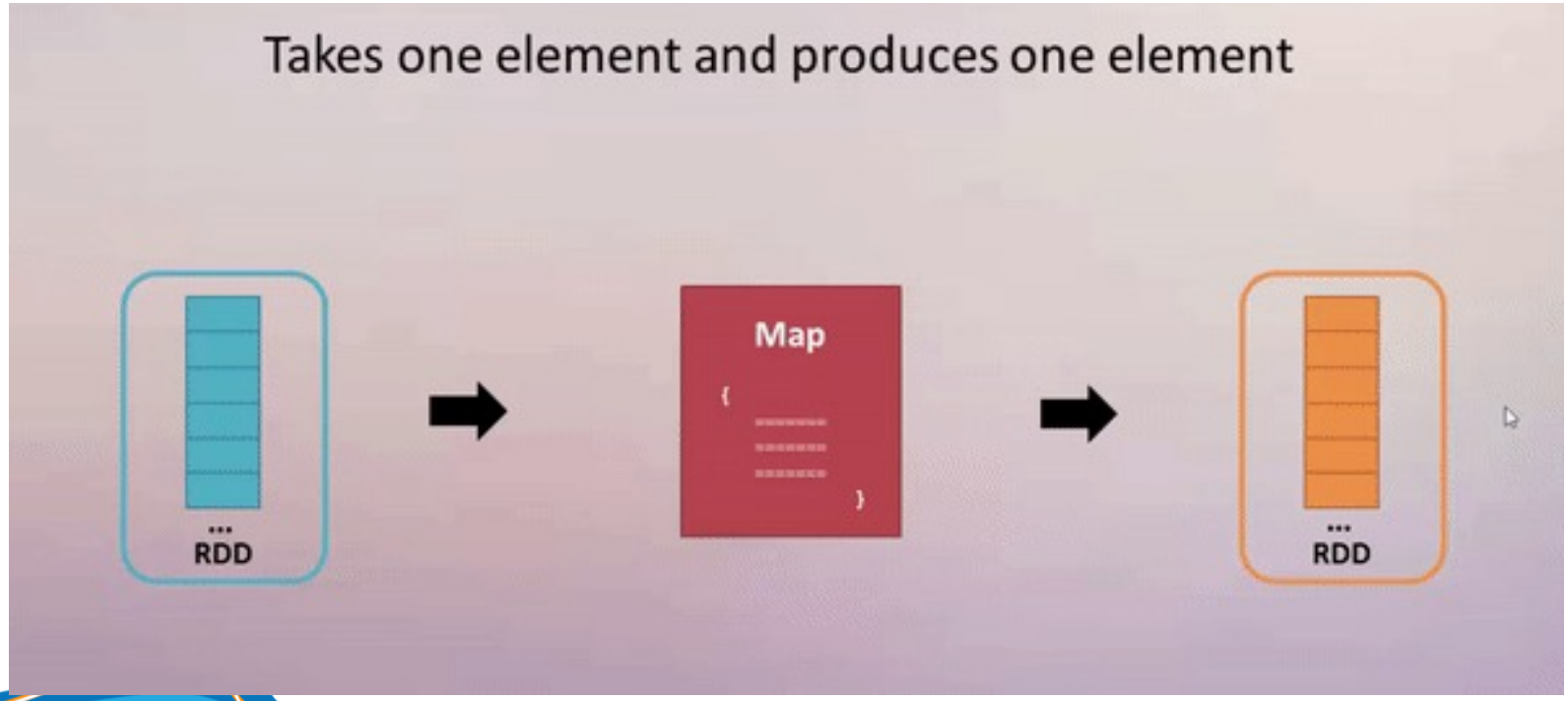
- Paired RDDs can be created by running a map() function that returns key/value pairs.




```
#Scala
// Step 4 - Split lines into words
val counts = words.map(word => (word, 1))
```

Transformations: map

- Spark Map function takes one element as input process it according to custom code (specified by the developer) and returns one element at a time.
- Map transforms an **RDD** of length N into another RDD of length N.
- The input and output RDDs will typically have the same number of records.



Transformations: map

fit@hcmus

- **map** defines functions completely inline using lambda syntax

```
# in Python
words2 = words.map(lambda word: (word, word[0], word.startswith("S")))
```

```
# [('Spark', 'S', True),
# ('The', 'T', False),
# ('Definitive', 'D', False),
# ('Guide', 'G', False),
#  (':', ':', False),
#  ('Big', 'B', False),
#  ('Data', 'D', False),
#  ('Processing', 'P', False),
#  ('Made', 'M', False),
#  ('Simple', 'S', True)]
```

```
# in Python
words2.filter(lambda record: record[2]).take(5)
```

```
# [('Spark', 'S', True),
#  ('Simple', 'S', True)]
```

- More filtering can be done subsequently

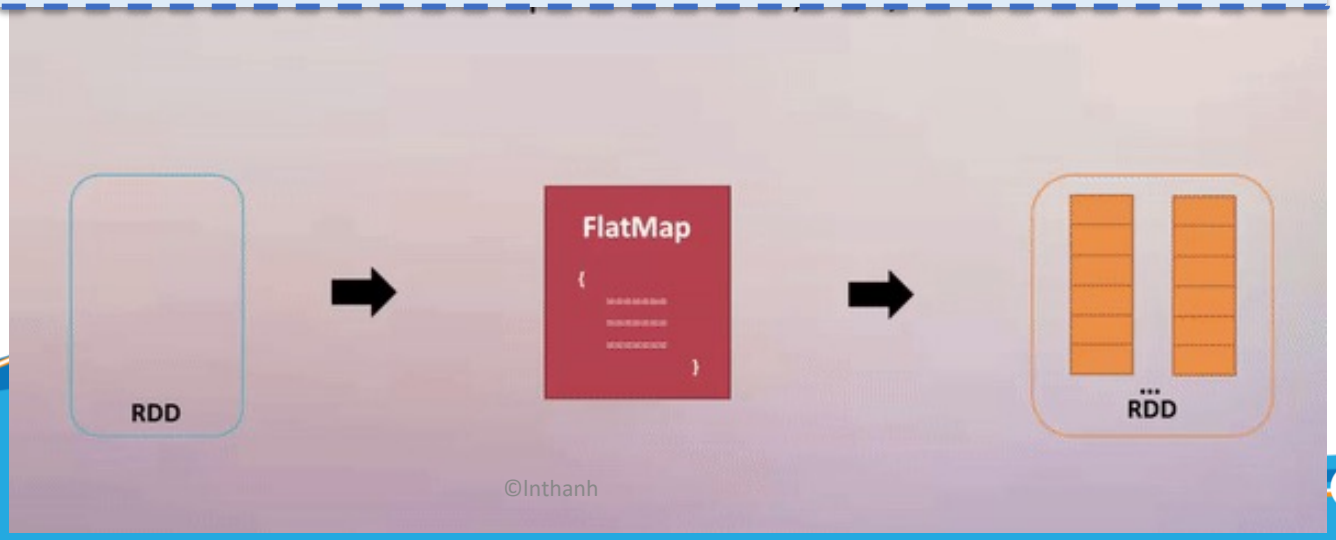
@lnthanh

Transformations: flatMap

fit@hcmus

- **flatMap** provides a simple extension of the **map** function
  - It is similar to Map, but FlatMap allows returning 0, 1 or more elements from map function. In the FlatMap operation, a developer can define his own custom business logic.

```
# in Python
words.flatMap(lambda word: list(word)).take(5)
# S, p, a, r, k
```



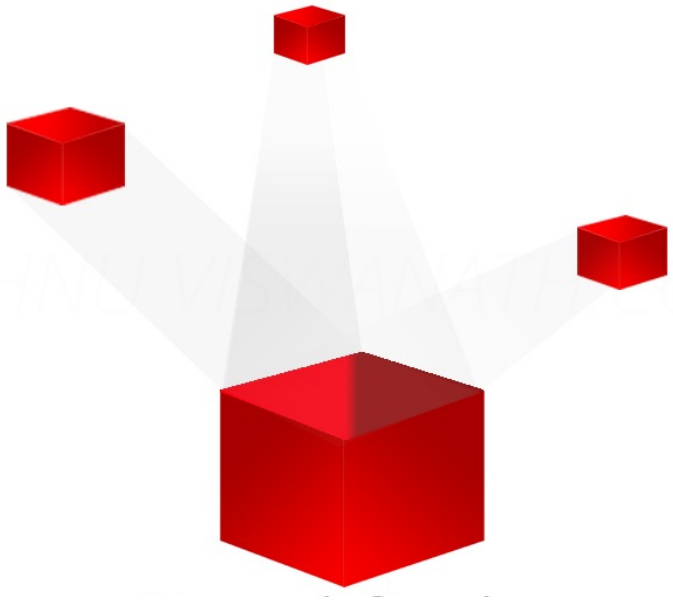
Transformations: sortBy – randomSplits

- **sortBy** specifies a function to extract a value from objects in RDDs and then sort based on that

```
# in Python
words.sortBy(lambda word: len(word) * -1).take(5)
# ['Definitive', 'Processing', 'Simple', 'Spark', 'Guide']
```

- **randomSplit** accepts an Array of weights and a random seed then randomly split an RDD into an Array of RDDs.

```
# in Python
fiftyFiftySplit = words.randomSplit([0.5, 0.5])
```



Actions

RDD Operations

- **Actions**
  - return value to the driver program after running a computation on the dataset.
  - Actions are operations that provide non-RDD values.
- We will discuss some functions in actions

@lnthanh

Actions: reduce

fit@hcmus

- Actions kick off specified transformations and either collect data to the driver or write to an external data source.
- **reduce** reduces an RDD of any kind of value to one

```
# in Python
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y)
# 210
```

```
# in Python
def wordLengthReducer(leftWord, rightWord):
    if len(leftWord) > len(rightWord):
        return leftWord
    else:
        return rightWord
words.reduce(wordLengthReducer)
# Undeterministic: 'Processing' or 'Definitive'
```

Actions: count

fit@hcmus

- **count** counts the number of rows in the RDD

```
# in Python
words.count( )
# 10
```

- **countApprox** is an approximation of count method but it must execute within a timeout → maybe incompleted results

```
# in Python
confidence = 0.95
timeoutMilliseconds = 400
words.countApprox(timeoutMilliseconds, confidence)
```

  - Confidence (∈ [0,1] is the probability that the error bounds of the result will contain the true value.
    - E.g., call countApprox repeatedly with confidence 0.9 → 90% of the results to contain the true count

@lnthanh

Actions: countByValue

fit@hcmus

- **countByValue** counts the number of values in a given RDD.
  - Use only if the resulting map is expected to be small because the entire thing is loaded into the driver's memory
  - It makes sense only in a scenario in which either the total number of rows is low or the number of distinct items is low

```
# in Python
words.countByValue( )
```

```
# defaultdict(int,
#         {'Spark': 1,
#          'The': 1,
#          'Definitive': 1,
#          'Guide': 1,
#          ':': 1,
#          'Big': 1,
#          'Data': 1,
#          'Processing': 1,
#          'Made': 1,
#          'Simple': 1})
```

- **countApproxDistinct**
- **countByValueApprox**

@lnthanh

Actions: Other methods

fit@hcmus

- **first** returns the first value in the dataset.

```
# in Python
words.first( )
# Spark
```

- **max** and **min** return the maximum and minimum values, respectively

```
# in Python
spark.sparkContext.parallelize(1 to 20).max()
# 20
spark.sparkContext.parallelize(1 to 20).min()
# 1
```

Actions: take

- **take** and its derivative methods take a number of values from your RDD.
  - First scan one partition and then use the results to estimate the number of additional partitions needed to satisfy the limit

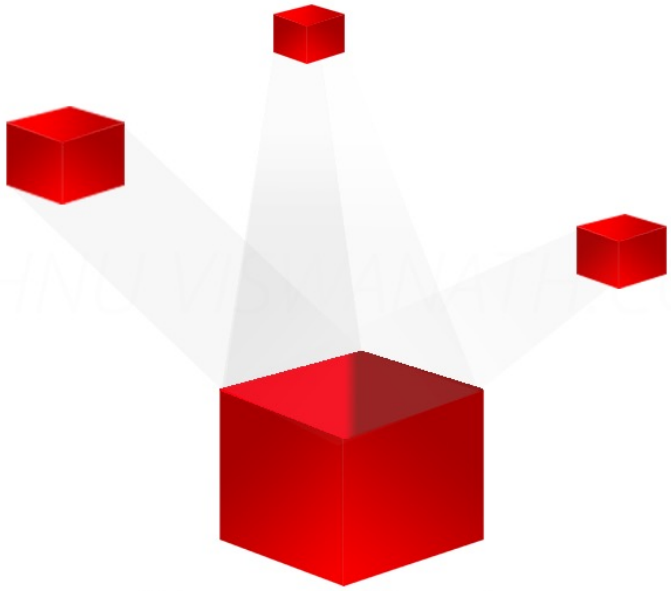
```
# in Python
words.take(5 )
# ['Spark', 'The', 'Definitive', 'Guide', ':']
words.takeOrdered(5 )
# [':', 'Big', 'Data', 'Definitive', 'Guide']
words.top(5 )
# ['The', 'Spark', 'Simple', 'Processing', 'Made']
```

- **top** is effectively the opposite of **takeOrdered** in that it selects the top values according to the implicit ordering.

Actions: take

- **takeSample** specifies a fixed-size random sample.

```
withReplacement = True
numberToTake = 6
randomSeed = 100
words.takeSample(withReplacement, numberToTake, randomSeed)
# ['Data', 'Definitive', 'Data', 'The', 'Definitive', 'Spark']
# ['The', ':', 'Simple', 'Spark', 'Data', 'Big']
```



Save RDD

Saving files

fit@hcmus

- Saving files means writing to plain-text files.

```
# in Python
words.saveAsTextFile("file:/tmp/bookTitle")
```

- To set a compression codec, import the proper codec from Hadoop – org.apache.hadoop.io.compress library

```
# in Python
codec = "org.apache.hadoop.io.compress.GzipCodec"
words.saveAsTextFile("file:/tmp/bookTitle“, codec)
```

- Saving to some external database requires iteration over every partition of a given RDD.
  - This reveals the underlying operation performed in higher-level APIs.

@lnthanh

SequenceFiles and Hadoop files

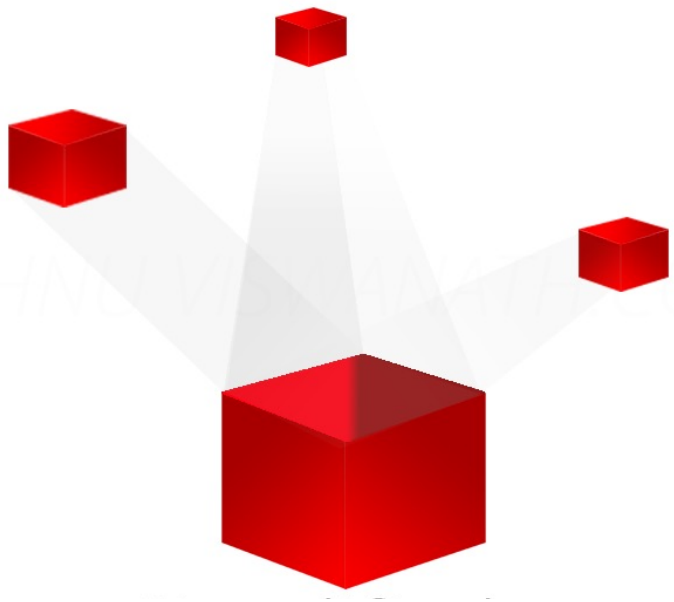
fit@hcmus

- A **sequenceFile** is a flat file of binary key–value pairs.
  - It is extensively used in MapReduce as input/output formats.
- Spark can write to sequenceFiles using **saveAsObjectFile** or by explicitly writing key–value pairs.

```
# in Python
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

- A variety of different Hadoop file formats are also supported.
  - These allow specifying classes, output formats, Hadoop configurations, and compression schemes.

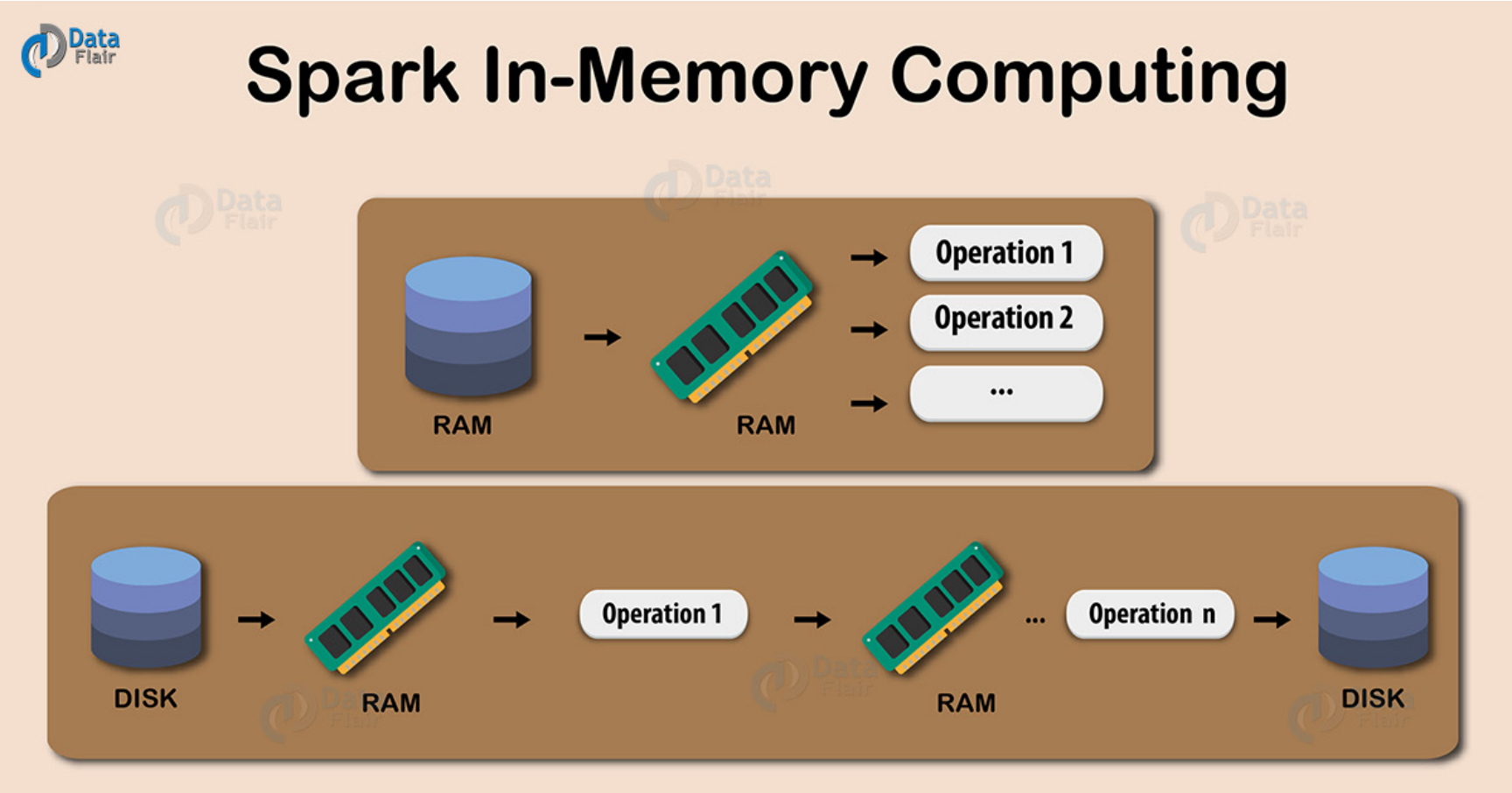
@lnthanh



In-Memory Computing

Spark In-Memory Computing

The data is kept in *random access memory(**RAM**)* instead of some slow disk drives and is processed in parallel



Spark In-memory Computing

fit@hcmus

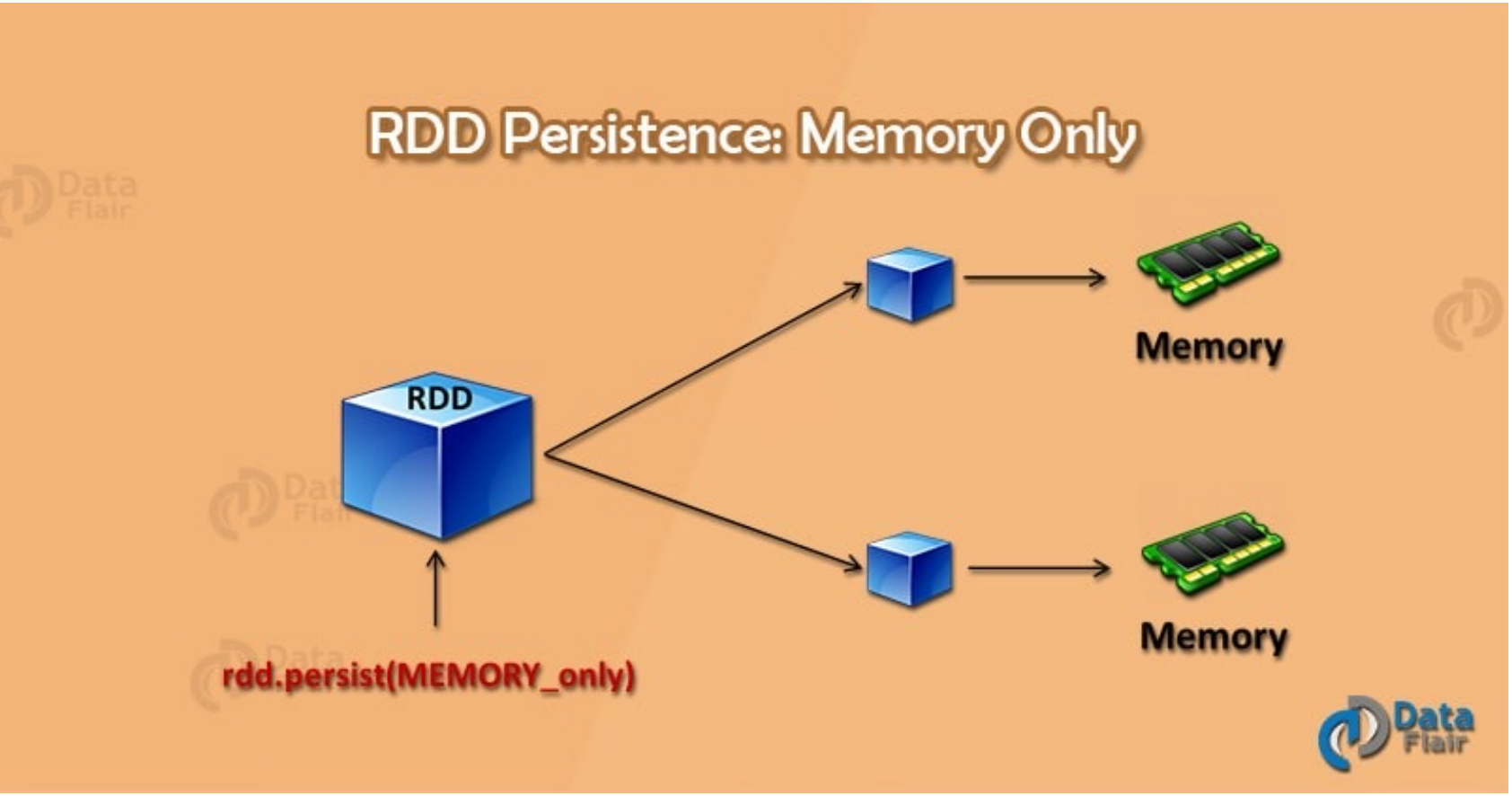
- The in-memory capability of Spark is good for machine learning and micro-batch processing.
  - It provides faster execution for iterative jobs.
- The RDDs are cached using the cache() or persist() method.
  - The RDDs can be stored in-memory.
  - In cache() method, if the data that does not fit in memory, it is either recalculated or the excess data is sent to disk.
  - The difference between cache() and persist() is that using cache() the default storage level is MEMORY_ONLY while using persist() we can use various storage levels.

Storage levels of RDD Persist()

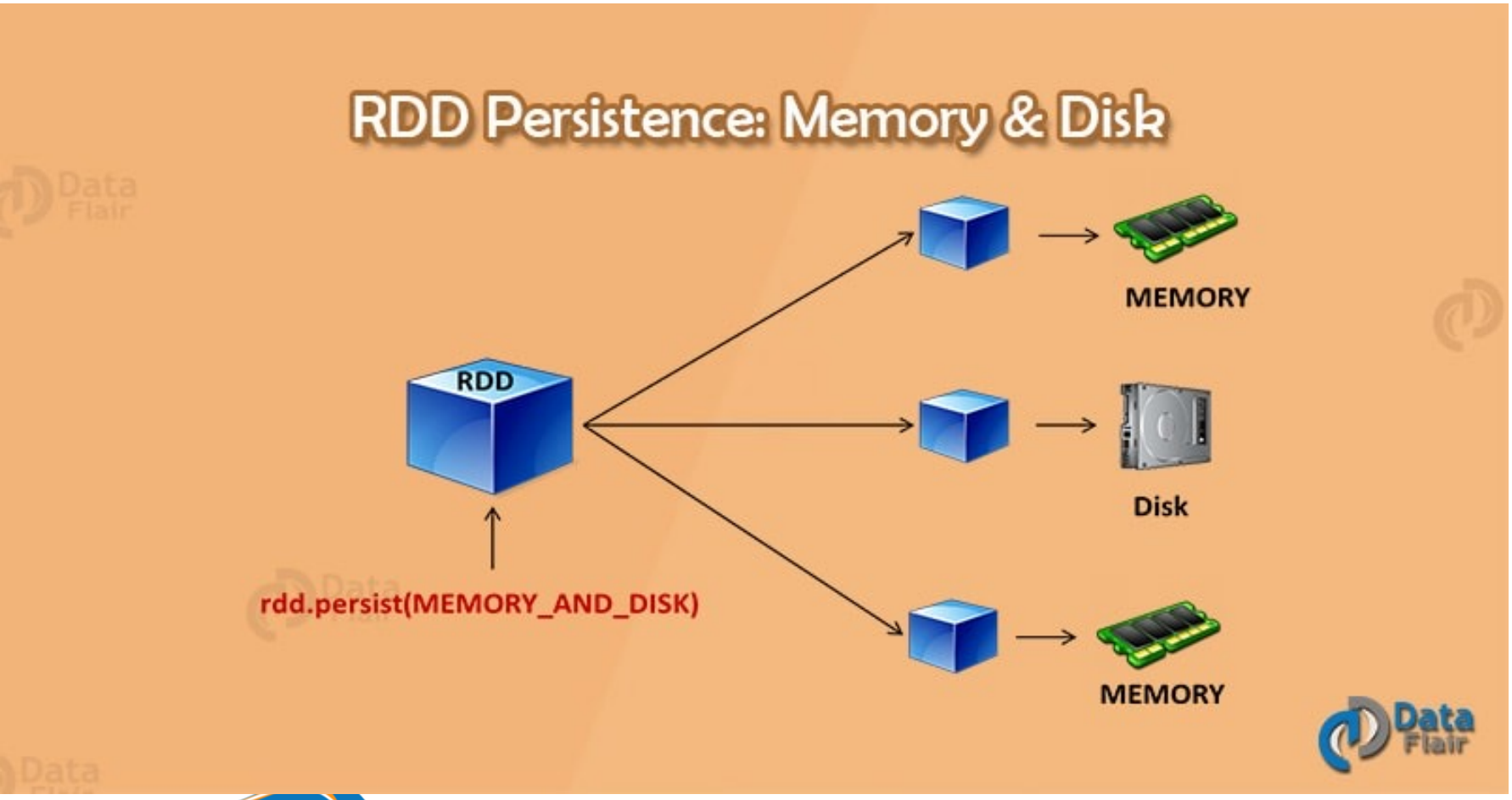
fit@hcmus

- The various storage level of persist() method in Apache Spark RDD are:
  - MEMORY_ONLY
  - MEMORY_AND_DISK
  - MEMORY_ONLY_SER
  - MEMORY_AND_DISK_SER
  - DISK_ONLY
  - MEMORY_ONLY_2 and MEMORY_AND_DISK_2

MEMORY_ONLY

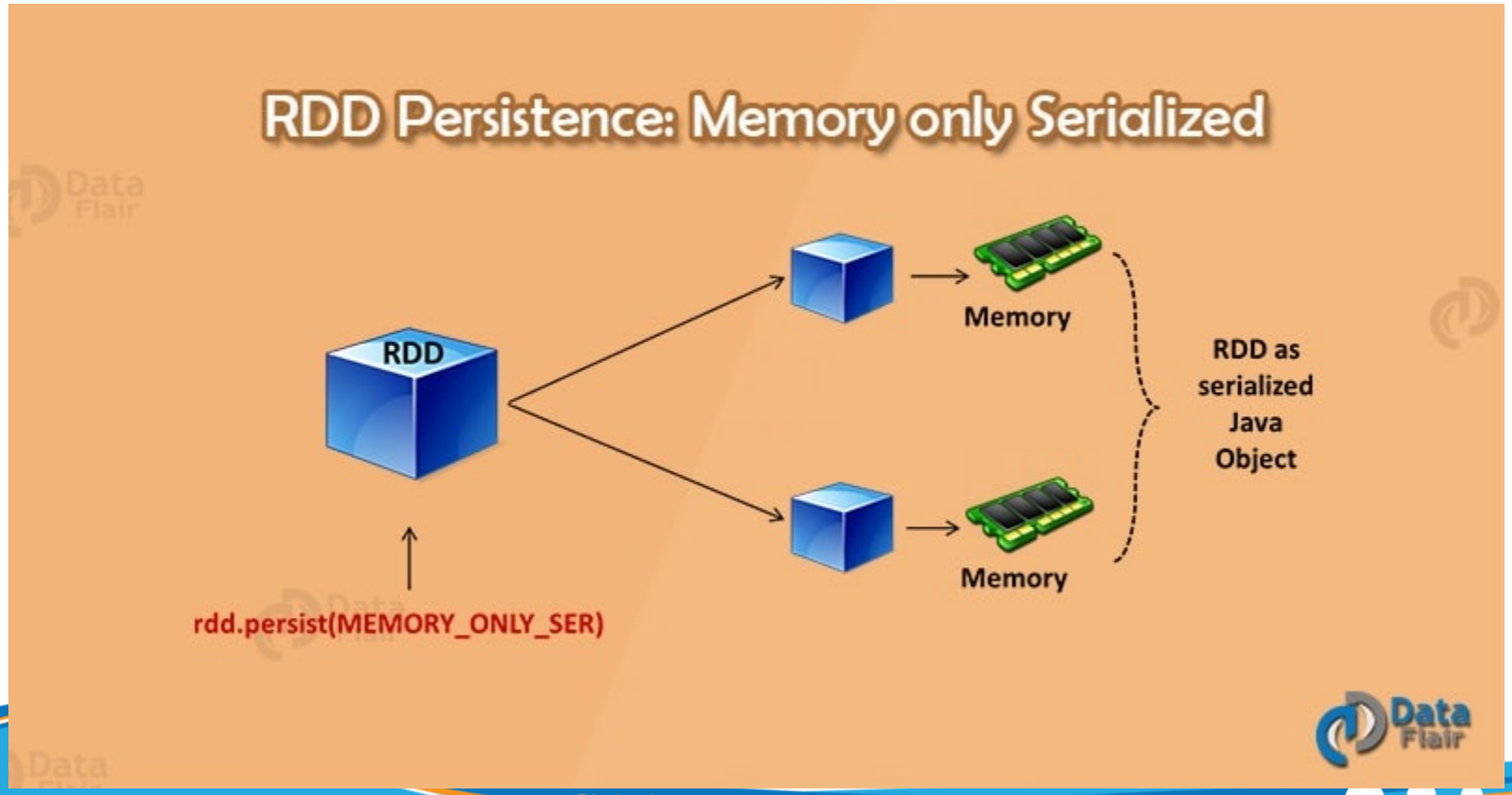

RDD Persistence: Memory Only
RDD
Memory
Memory
rdd.persist(MEMORY_only)
Data Flair

MEMORY_AND_DISK



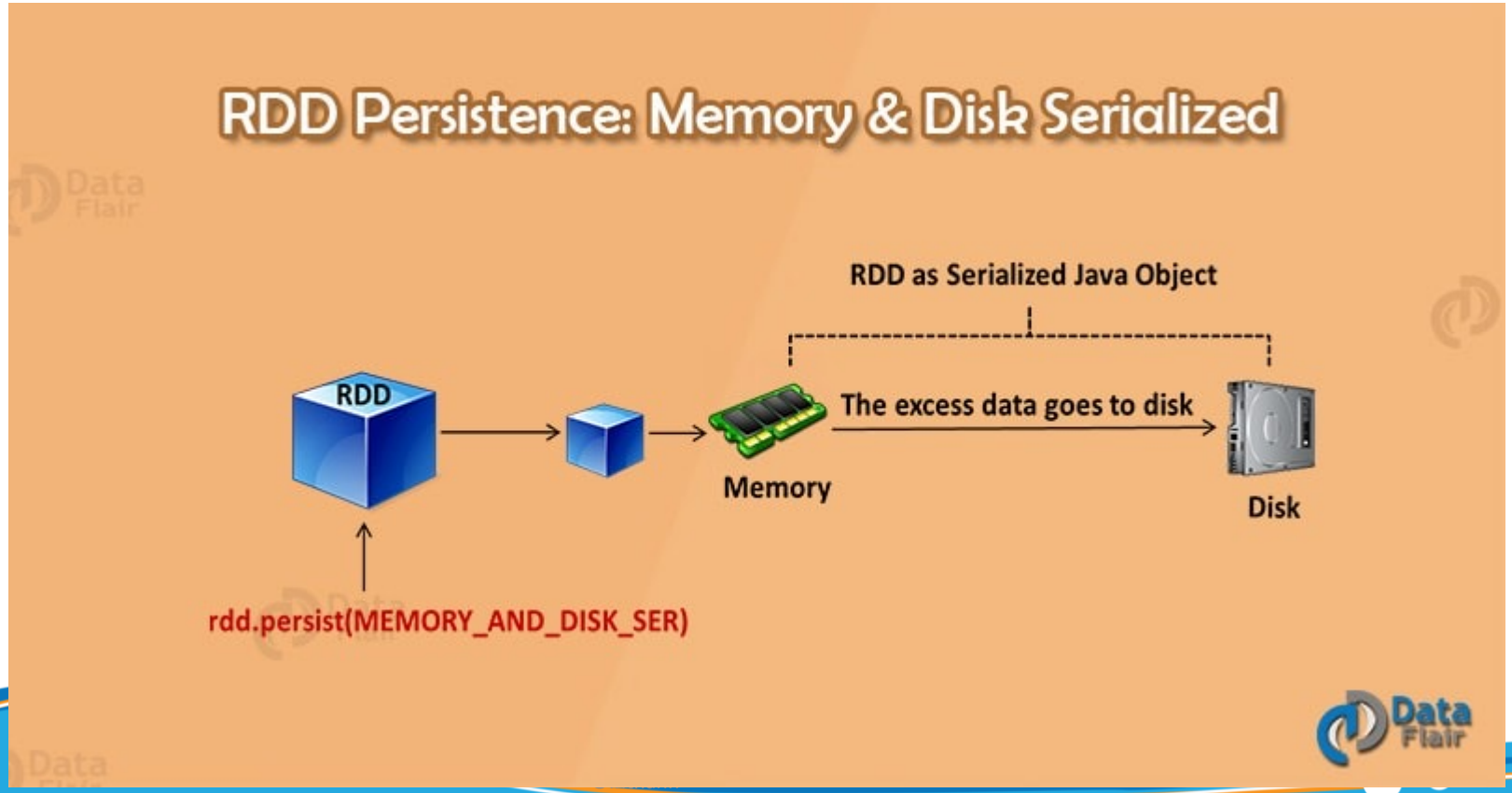
MEMORY_ONLY_SER

- This level stores RDDs as serialized JAVA object. It stores one-byte array per partition. It is like *MEMORY_ONLY* but is more space efficient especially when we use fast serializer.

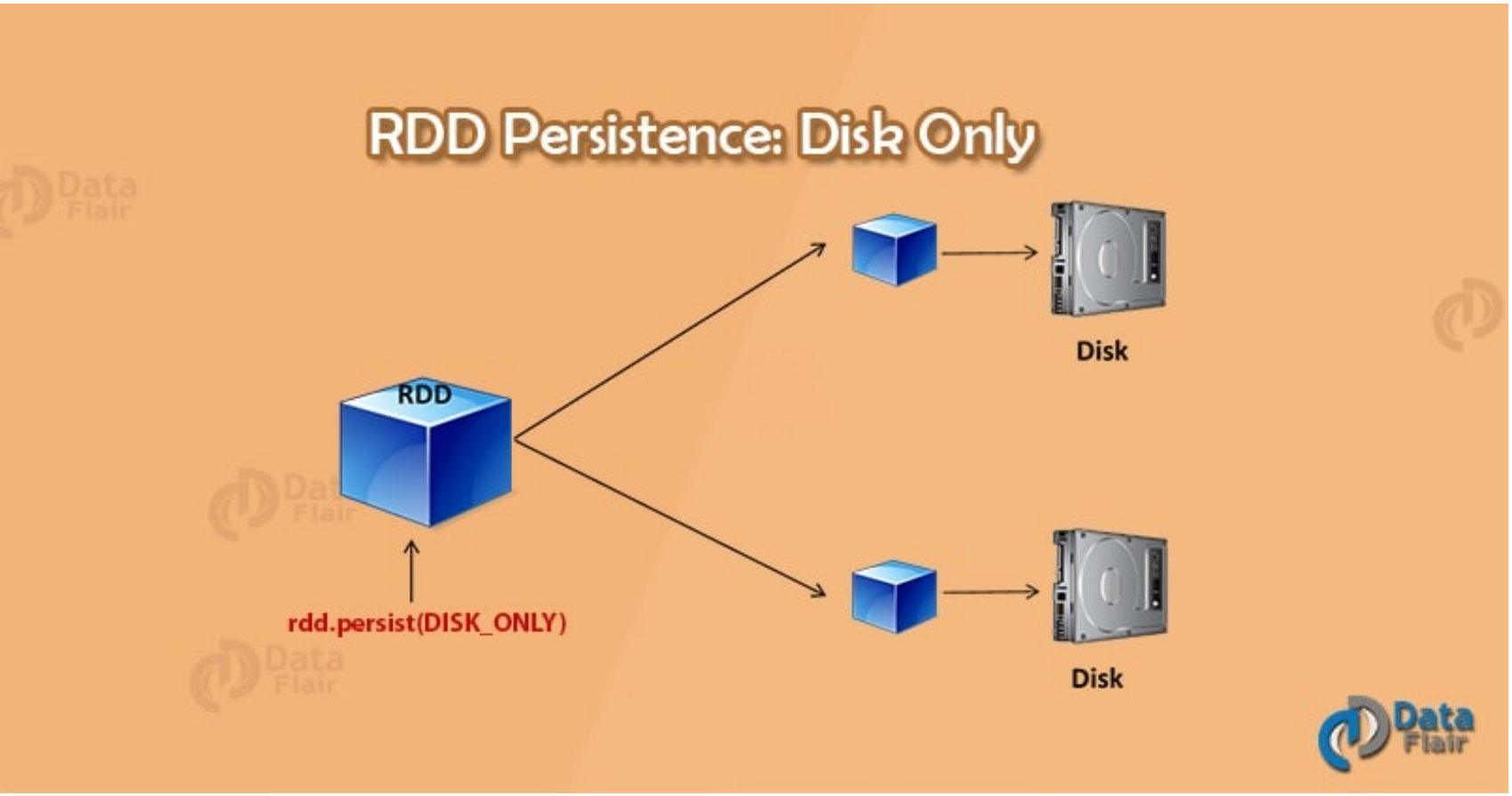


MEMORY_AND_DISK_SER

- This level stores RDD as serialized JAVA object. If the full RDD does not fit in the memory then it stores the remaining partition on the disk, instead of recomputing it every time when we need.



DISK_ONLY


RDD Persistence: Disk Only
RDD
rdd.persist(DISK_ONLY)
Disk
Disk

Caching

- The same principles for RDDs, DataFrames and Datasets.
- It is possible to either cache or persist an RDD.
  - By default, cache and persist only handle data in memory.
  - Refer to http://spark.apache.org/docs/2.0.0/api/python/pyspark.html

```
# in Python
words.cache( )
```

```
# in Python
words.persist( )
```

- It is possible to subsequently query for this storage level

```
# in Python
words.getStorageLevel( )
# StorageLevel(False, False, False, False, 1)
```

Checkpointing

fit@hcmus

- Save an RDD to disk → future references point to those intermediate partitions rather than recomputing the RDD from its original source.

```
# in Python
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
words.checkpoint()
```

- Helpful optimization.
- Similar to caching except that it is stored only in disk
- Feature not available in the DataFrame API

Finally, our Word Count!

fit@hcmus

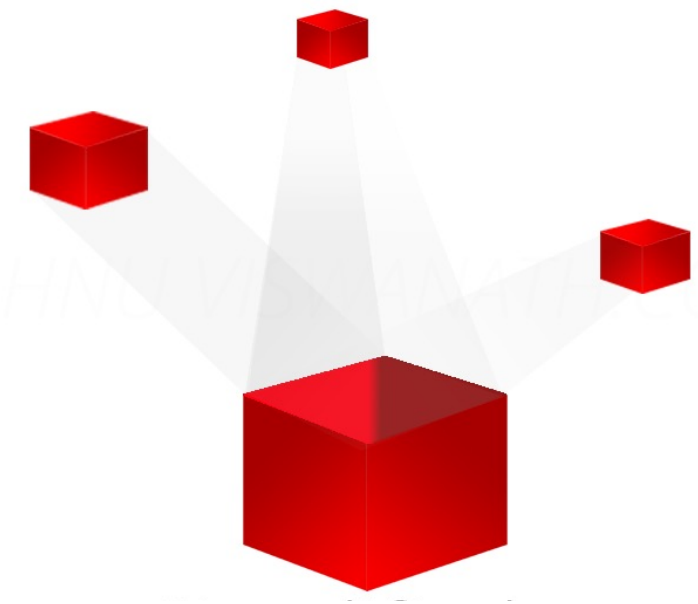
```
text_file = sc.textFile("input.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)
count.collect()
```

VS.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
linesDF = sqlContext.read.text("input.txt")
linesDF.show(df.count(),truncate=False)

import pyspark.sql.functions as f
wordsDF = linesDF.withColumn('word',
f.explode(f.split(f.col('value'), ' ')))\
    .groupBy('word')\
    .count()\
    .sort('count', ascending=False)\
    .show()
```

©Intel



Advanced RDDs

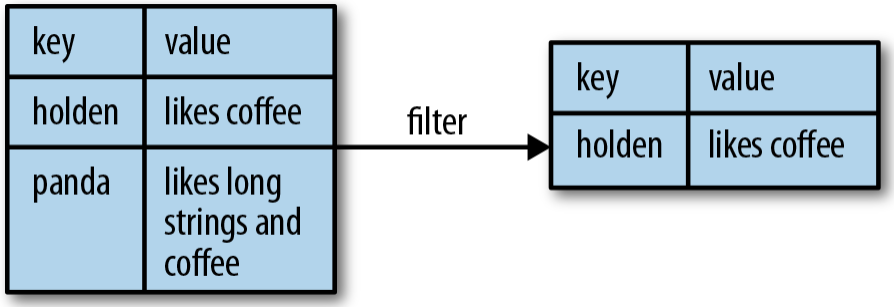
Key-value RDDs is a powerful abstraction for manipulating data

Key-value RDDs

- There are many cases that require data in key–value format.
- Method including **<some-operation>ByKey** can only perform on **PairRDD** type.
- A key–value structure can be achieved by **mapping**.

```
# in Python
words.map(lambda word: (word.lower(), 1))
```

```
# [('spark', 1),
#  ('the', 1),
#  ('definitive', 1),
#  ('guide', 1),
#  (':', 1),
#  ('big', 1),
#  ('data', 1),
#  ('processing', 1),
#  ('made', 1),
#  ('simple', 1)]
```



Key-value RDDs

fit@hcmus

- **keyBy** provides similar results by specifying a function that creates the key from a current value.

```
# in Python
keyword = words.keyBy(lambda word: word.lower()[0])
```

```
# [('s', 'Spark'),
#   ('t', 'The'),
#   ('d', 'Definitive'),
#   ('g', 'Guide'),
#   (':', ':'),
#   ('b', 'Big'),
#   ('d', 'Data'),
#   ('p', 'Processing'),
#   ('m', 'Made'),
#   ('s', 'Simple')]
```

Map over values

fit@hcmus

- In a tuple, the first element is considered as a key and the second as the corresponding value.
- **mapValues** explicitly maps over the values and ignores the individual keys.

```
# in Python
keyword.mapValues(lambda word: word.upper()).
collect()
```

```
# [('s', 'SPARK'),
#  ('t', 'THE'),
#  ('d', 'DEFINITIVE'),
#  ('g', 'GUIDE'),
#  (':', ':'),
#  ('b', 'BIG'),
#  ('d', 'DATA'),
#  ('p', 'PROCESSING'),
#  ('m', 'MADE'),
#  ('s', 'SIMPLE')]
```

@lnthanh

Map over values

fit@hcmus

- **flatMap** over the rows will expand the number of rows so that each row represents a character.

```
# in Python
keyword.flatMapValues(lambda word: word.upper()).collect()
```

```
# [('s', 'S'),
#  ('s', 'P'),
#  ('s', 'A'),
#  ('s', 'R'),
#  ('s', 'K'),
#  ('t', 'T'),
#  ('t', 'H'),
#  ('t', 'E'),
#  ('d', 'D'),
#  ('d', 'E'),
#  ...
```

@lnthanh

Extracting keys and values

fit@hcmus

- With a key-value formatted RDD, it is possible to extract the specific keys or values.

```
# in Python
keywords.values ().collect()
```

```
# ['Spark',
#  'The',
#  'Definitive',
#  'Guide',
#  ':',
#  'Big',
#  'Data',
#  'Processing',
#  'Made',
#  'Simple']
```

```
# in Python
keywords.key().collect()
# ['s', 't', 'd', 'g', ':', 'b', 'd', 'p', 'm', 's']
```

@lnthanh

lookup

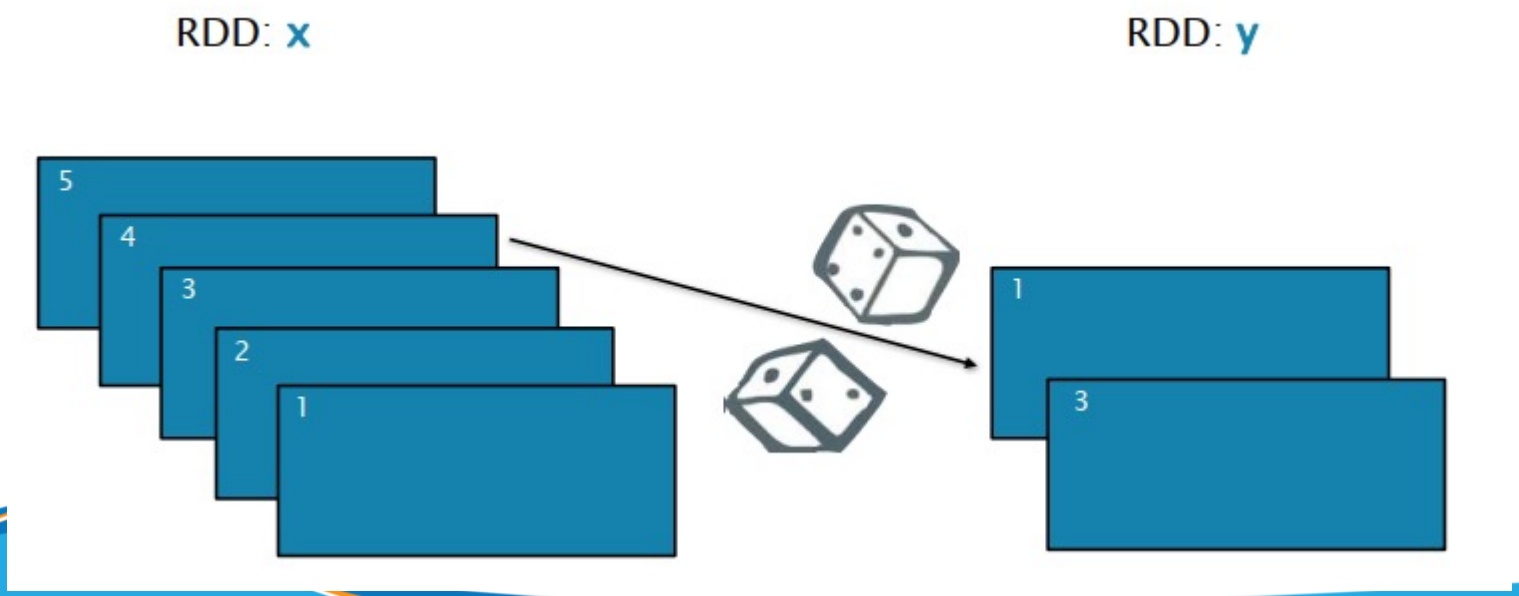
- **lookup** looks up the result for a particular key in the RDD.
  - returns the list of values in the RDD for key 'key'
- Note that there is no enforcement mechanism with respect to there being only one key for each input
  - E.g., lookup "s" gets both associated values, "Spark" and "Simple"

```
# in Python
keyword.lookup("s")
# ['Spark', 'Simple']
```

@lnthanh

Random sampling

- Sampling an RDD by a set of keys can be done via an approximation or exactly.
- Both operations can perform with or without replacement as well as sampling by a fraction by a given key.



Random sampling: sampleByKey

fit@hcmus

sampleByKey(withReplacement, fractions, seed)

- **sampleByKey** performs random sampling with one pass over the RDD.
  - It produces a sample of size that is approximately equal to the sum of math.ceil(numItems * samplingRate) over all key values.

```
# in Python
import random
distinctChars = words.flatMap(lambda word:\
list(word.lower())).distinct().collect()
sampleMap = dict(map(lambda c: (c, random.random()), distinctChars))
words.map(lambda word: (word.lower()[0], word))\
.sampleByKey(False, sampleMap, 6).collect()
# [('t', 'The'), ('b', 'Big'), ('p', 'Processing')]
```

Random sampling: sampleByKey

fit@hcmus

sampleByKeyExact(withReplacement, fractions, seed)

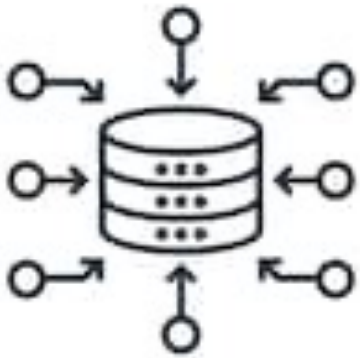
- **sampleByKeyExact** differs from **sampleByKey** by making additional passes over the RDD.
  - The sample size exactly equals to the sum of math.ceil(numItems* samplingRate) over all key values with a 99.99% confidence.
  - Sampling without replacement: one additional pass;

```
# in Scala
words.map(word => (word.toLowerCase.toSeq(0), word))
.sampleByKeyExact(true, sampleMap, 6L).collect()
```

- *This is currently not supported in python.*

Aggregations

- Aggregations can be done on plain RDDs or on PairRDDs, depending on the method be



```
# in Python
chars = words.flatMap(lambda word: word.lower())
KVcharacters = chars.map(lambda letter: (letter, 1))
def maxFunc(left, right):
    return max(left, right)
def addFunc(left, right):
    return left + right
nums = sc.parallelize(range(1,31), 5)
```

Aggregations

fit@hcmus

- **countByKey** counts the number of elements for each key, collecting the results to a local Map.

```
# in Python
KVcharacters.countByKey()
```

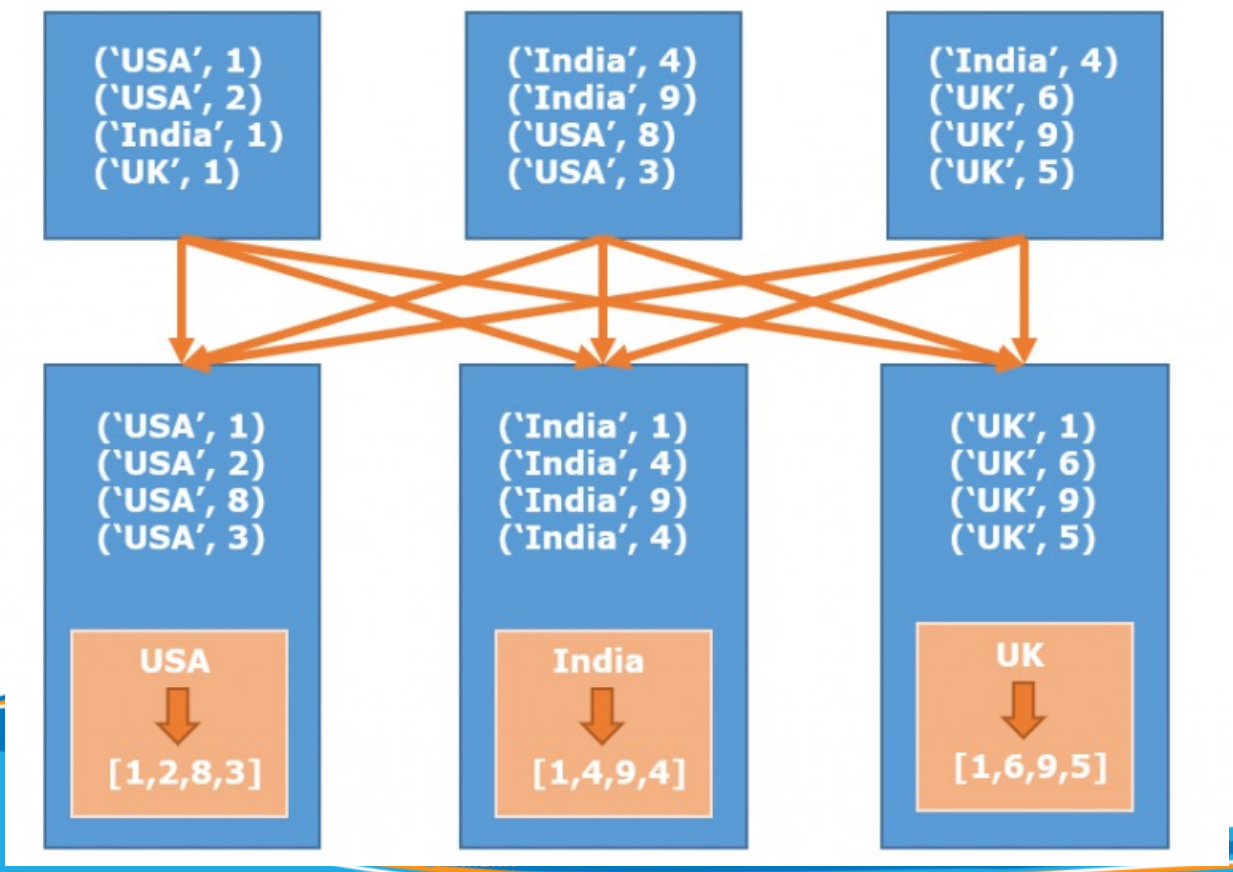
```
# defaultdict(int,
#          {'s': 4,
#           'p': 3,
#           'a': 4,
#           'r': 2,
#           'k': 1,
#           't': 3,
#           'h': 1,
#           ...
```

- **countByKeyApprox** makes it possible to specify a timeout and confidence, but it is available only in Scala or Java.

@lnthanh

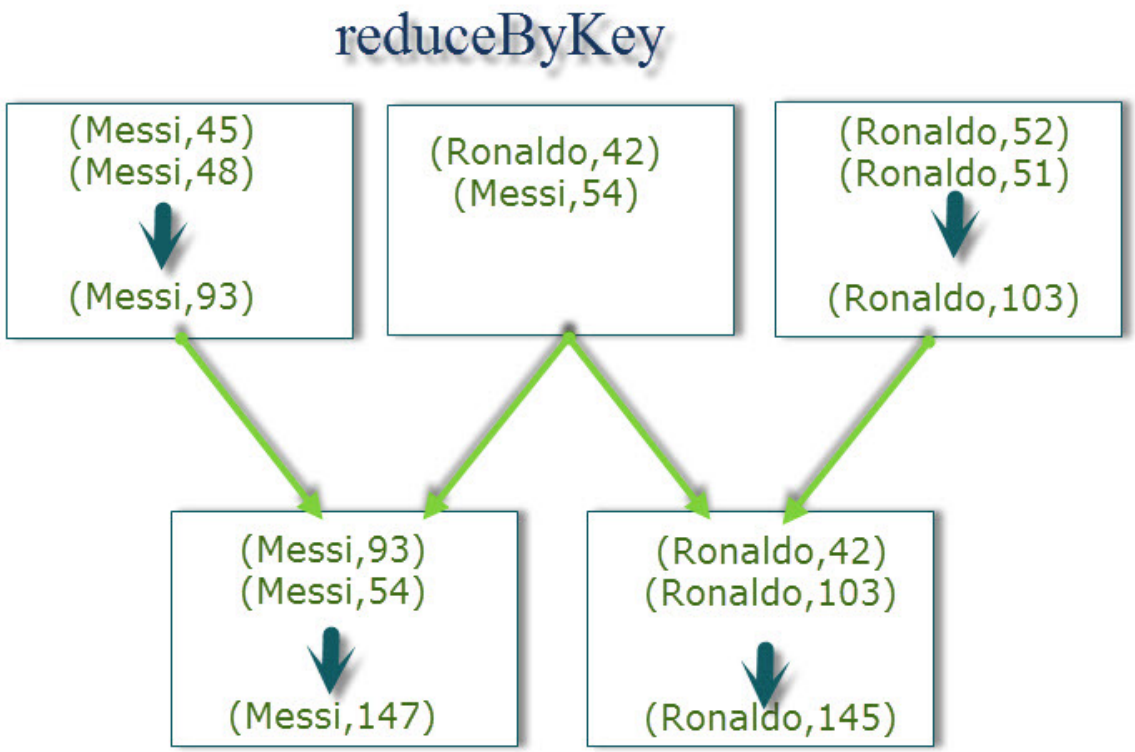
Understanding aggregation implementations

- groupBykey() - When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs.



Understanding aggregation implementations

- reduceByKey() - receives key-value pairs (K, V) as an input, aggregates the values based on the key and generates a dataset of (K, V) pairs as an output..



Unlike groupByKey , reduceByKey does not shuffle data at the beginning.

Understanding aggregation implementations

- Consider **groupByKey** and **reduceByKey** as an example.

```
# in Python
KVcharacters.groupByKey().map(lambda row: \
(row[0], reduce(addFunc, row[1]))).collect()
```

```
# in Python
KVcharacters.reduceByKey(addFunc).collect()
```

```
# [('s', 4),
#  ('p', 3),
#  ('r', 2),
#  ('h', 1),
#  ('d', 4),
#  ('i', 7),
#  ('g', 3),
#  ('b', 1),
#  ('c', 1),
#  ..
#  (':', 1),
#  ('o', 1),
#  ('m', 2)]
```

©lnthanh

Understanding aggregation implementations

- **groupByKey** is a wrong approach for the majority of cases.
  - Each executor must hold all values for a given key in memory before applying the function to them.
  - With massive key skew, some partitions might be overloaded.
  - *Appropriate when each key has consistent value sizes and fit in the memory of a given executor*
- **reduceByKey** is preferred for additive use cases.
  - Within each partition, everything in memory not required
  - No incurred shuffle during this operation, everything at each worker individually before performing a final reduce
  - *Appropriate when the workload is associative but inappropriate when the order matters*

©lnthanh

Other aggregation: aggregate

fit@hcmus

- These functions allow for very specific, low-level control on exactly how a given aggregation is performed on the cluster.
- **aggregate** requires a function that aggregate within partitions and another for across partitions.
  - The start value will be used at both aggregation levels:

```
# in Python
nums.aggregate(0, maxFunc, addFunc)
# 90
# [[1, 2, 3, 4, 5, 6],
#   [7, 8, 9, 10, 11, 12],
#   [13, 14, 15, 16, 17, 18],
#   [19, 20, 21, 22, 23, 24],
#   [25, 26, 27, 28, 29, 30]]
```

Other aggregation: treeAggregate

fit@hcmus

- **aggregate** performs the final aggregation on the driver

  → failed if the results from the executors are too large

- **treeAggregate** "pushes down" some subaggregations before performing the final aggregation on the driver.
  - Having multiple levels helps to ensure that the driver does not run out of memory in the process of the aggregation.

```
# in Python
depth = 3
nums.treeAggregate(0, maxFunc, addFunc, depth)
```

©Intnann

Other aggregation: combineByKey

fit@hcmus

- A combiner can work instead of an aggregation function.
- The combiner operates on a given key and merges the values according to some function
- It then merges different outputs of combiners for final result.

```
# in Python
def valToCombiner(value):
    return [value]
def mergeValuesFunc(vals, valToAppend):
    vals.append(valToAppend)
    return vals
def mergeCombinerFunc(vals1, vals2):
    return vals1 + vals2
outputPartitions = 6
KVcharacters.combineByKey(valToCombiner,mergeValuesFunc,\
mergeCombinerFunc,outputPartitions).collect()
```

```
# [('s', [1, 1, 1, 1]),
#  ('d', [1, 1, 1, 1]),
#  ('l', [1]),
#  ('v', [1]),
#  (':', [1]),
#  ('p', [1, 1, 1]),
#  ('r', [1, 1]),
#  ...
```

Other aggregation: foldByKey

fit@hcmus

- **foldByKey** merges values for each key using an associative function and a neutral "zero value".
  - This value can be added to the result an arbitrary number of times, and must preserve the result (e.g., 0 for addition, 1 for multiplication)

```
# in Python
depth = 3
nums.foldByKey(0, addFunc).collect()
```

```
# [('s', 4),
#  ('p', 3),
#  ('r', 2),
#  ('h', 1),
#  ('d', 4),
#  ('i', 7),
#  ('g', 3),
#  ('b', 1),
#  ('c', 1),
#  ..
#  (':', 1),
#  ('o', 1),
#  ('m', 2)]
```

@lnthanh

Cogroup

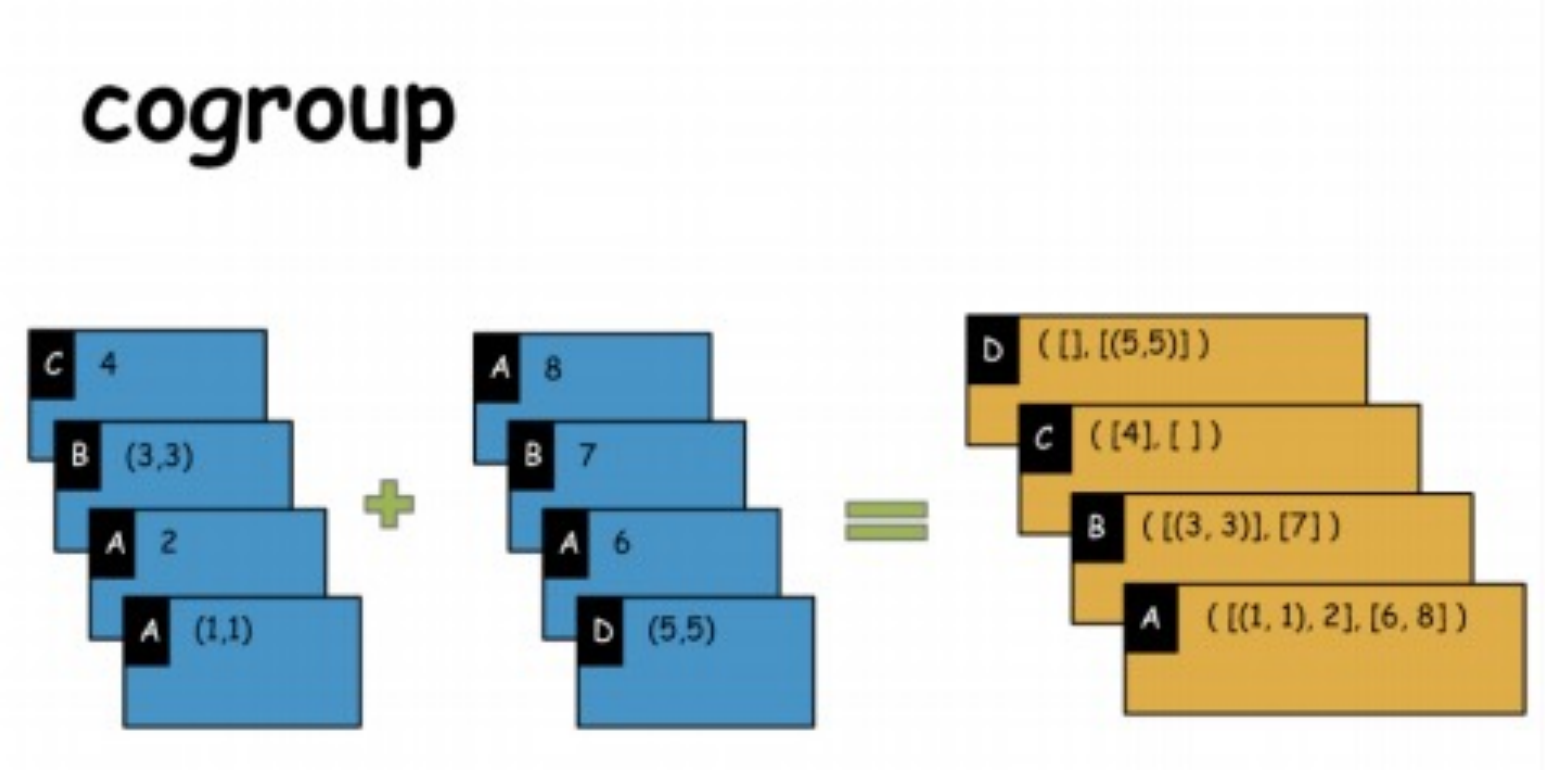
- **cogroup** results in a group with the given key on one side, and all of the relevant values on the other side.
  - A number of output partitions or a custom partitioning function is also available to control how this data is distributed across the cluster

```
# in Python
import random
distinctChars = words.flatMap(lambda word:
word.lower()).distinct()
charRDD = distinctChars.map(lambda c: (c, random.random()))
charRDD2 = distinctChars.map(lambda c: (c, random.random()))
charRDD.cogroup(charRDD2).take(5)
```

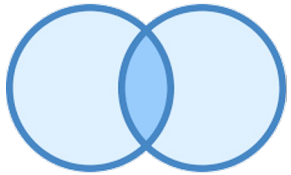
- It is able to group together up to three key–value RDDs together in Scala and two in Python.

©lnthanh

Cogroup



Joins: Inner join



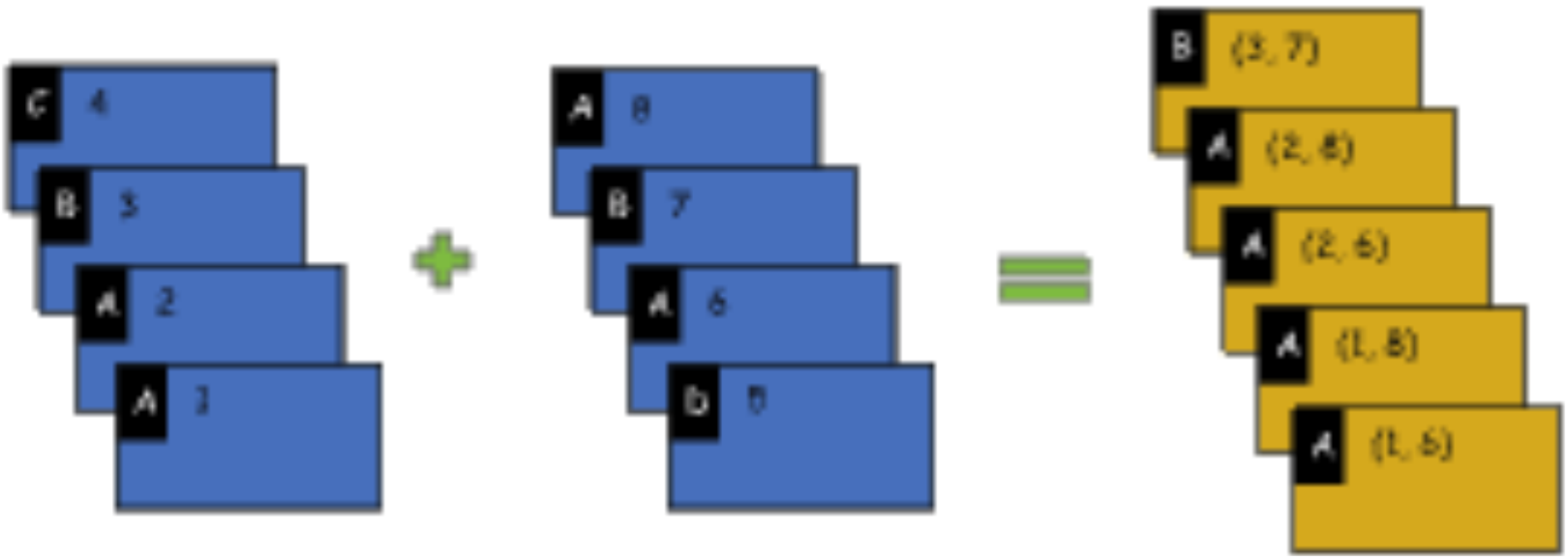
- **join** performs an inner join.

```
# in Python
keyedChars = distinctChars.map(lambda c: (c, random.random()))
outputPartitions = 10
KVcharacters.join(keyedChars).count()
KVcharacters.join(keyedChars, outputPartitions).count()
```

- Other joins all follow the same basic format
  - fullOuterJoin, leftOuterJoin, rightOuterJoin
  - cartesian (This is very dangerous! It does not accept a join key and can have a massive output.)
- Basic format: the two RDDs to join and (optionally) either the number of output partitions or the customer partition function

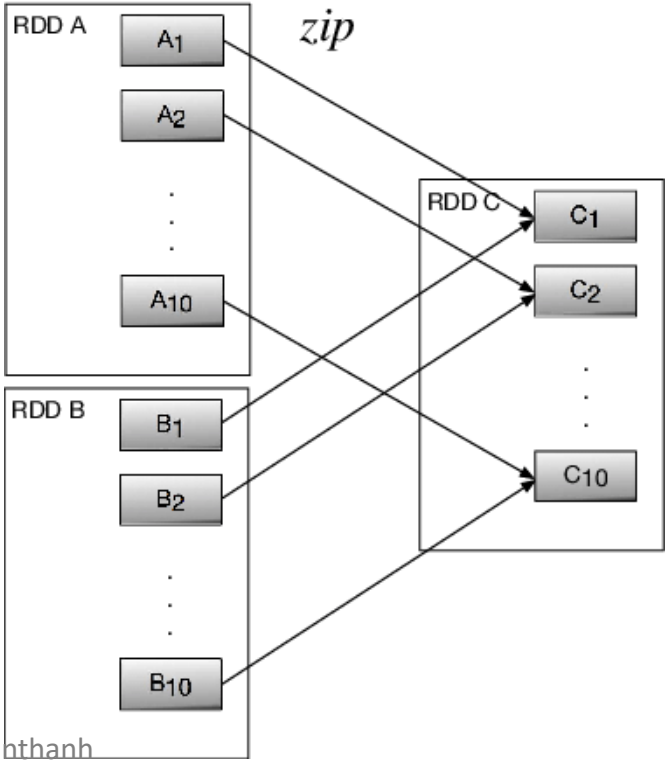
©lnthanh

Join



Joins: zip

- **zip** creates a PairRDD by combining two RDDs, assuming that they have the same length.
  - The two RDDs must have the same number of partitions as well as the same number of elements.




```
# in Python
numRange = sc.parallelize(range(10), 2)
words.zip(numRange).collect()
```

```
# [('Spark', 0),
#  ('The', 1),
#  ('Definitive', 2),
#  ('Guide', 3),
#  (':', 4),
#  ('Big', 5),
#  ('Data', 6),
#  ('Processing', 7),
#  ('Made', 8),
#  ('Simple', 9)]
```

©lnthanh

Controlling partitions

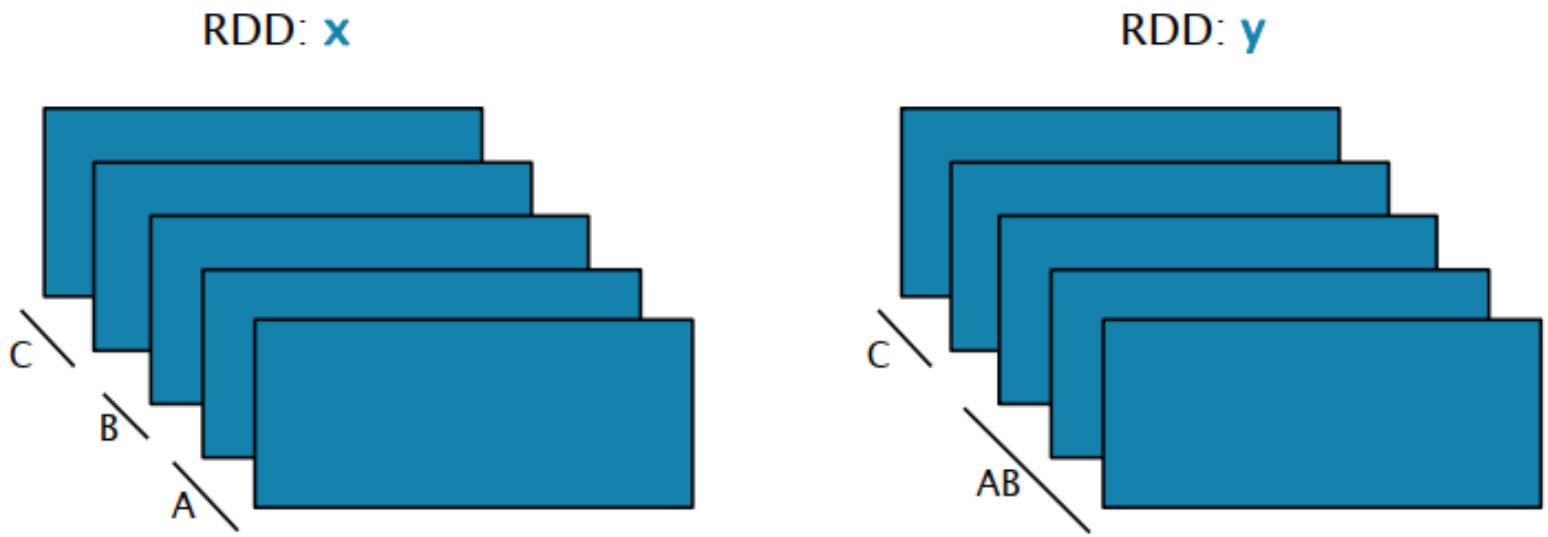
- Control over how data is exactly physically distributed across the cluster
  - Basically the same from what in the Structured APIs, but additionally specify a partitioning function (formally a custom Partitioner).
- **coalesce** effectively collapses partitions on the same worker in order to avoid a shuffle of the data when repartitioning.
- **repartition** allows repartitioning the data up or down but performs a shuffle across nodes in the process.

```
# in Python
words.coalesce(1).getNumPartitions()
# 1
```

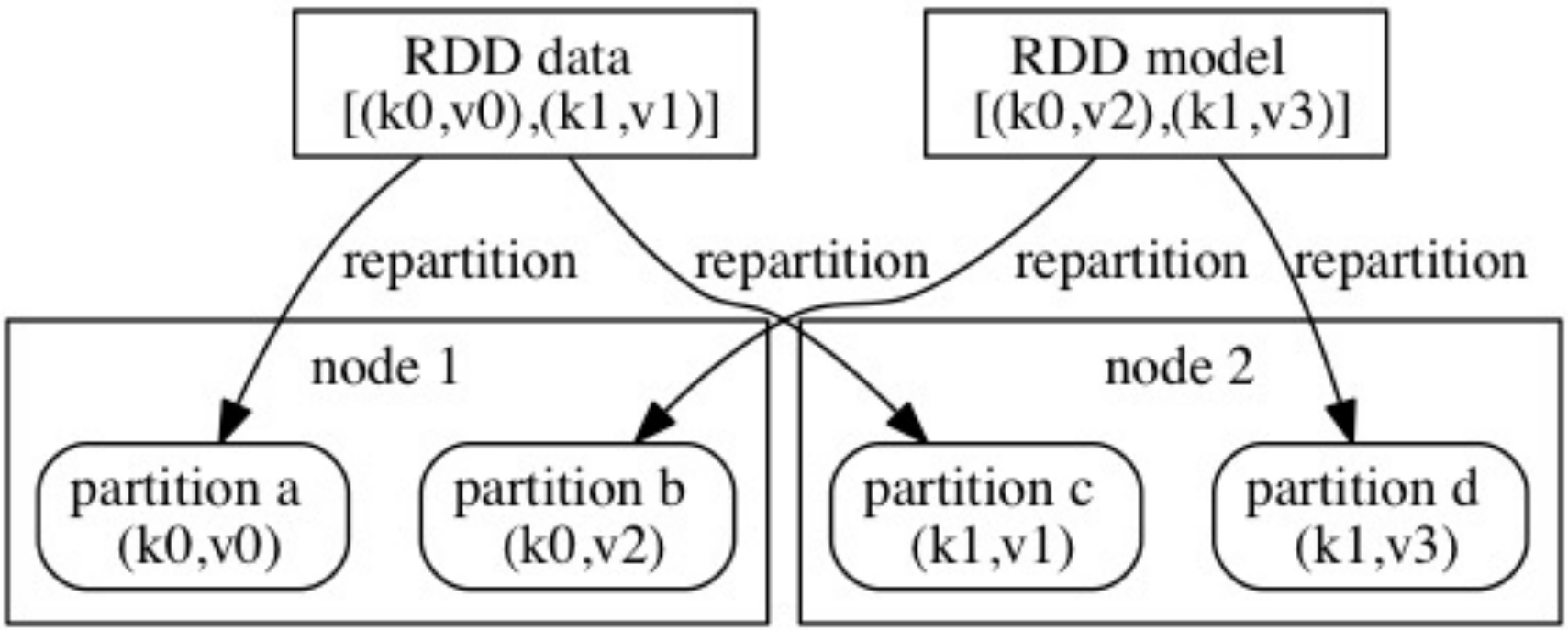
```
# in Python
words.repartition(10)
# gives us 10 partitions
```

©lnthanh

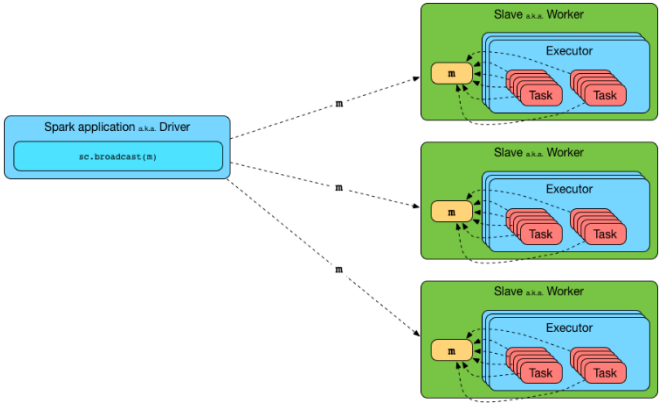
coalesce



Repartition



©lnthanh

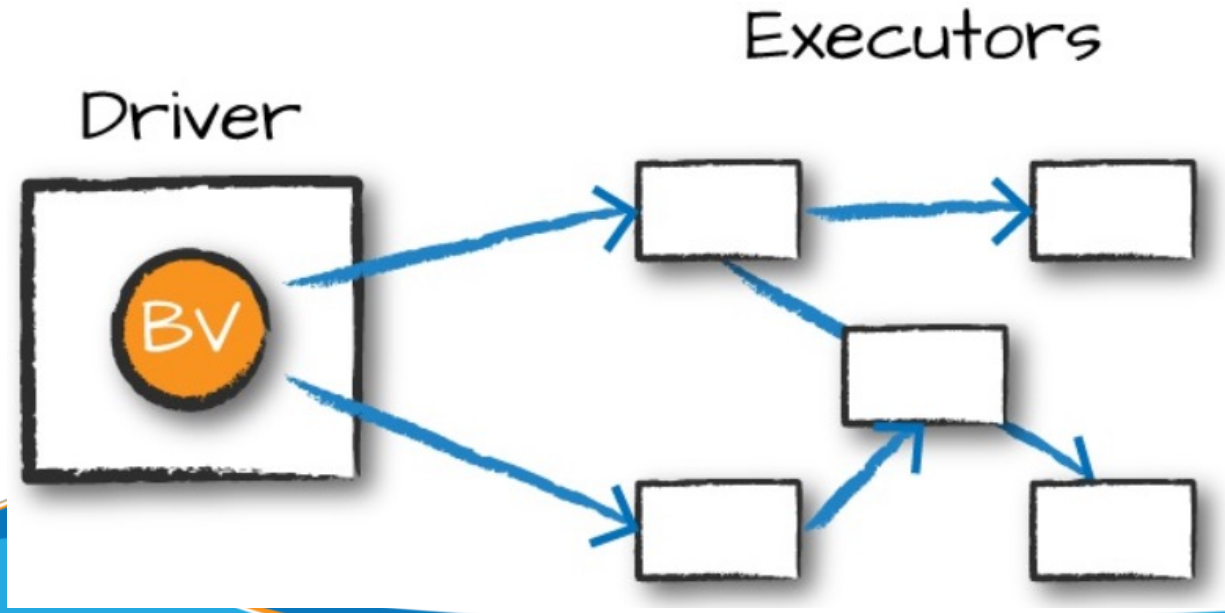


Distributed Shared Variables

A way to share an immutable value efficiently around the cluster

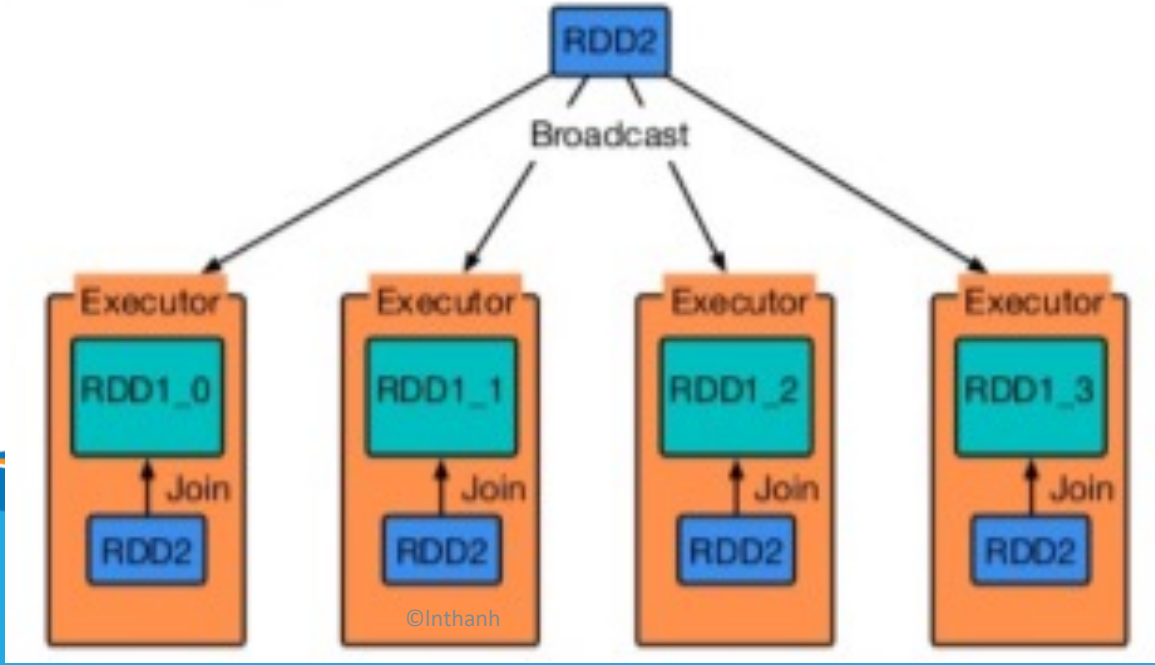
Broadcast variables

- Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task.
  - E.g., pass around a large lookup table that fits in memory on the executors and use that in a function



Broadcast variables

- An efficient way to share immutable values efficiently around the cluster without encapsulating it in a function closure
  - This depends on the amount of data and the number of executors. For very small data (low KBs) on small clusters, it might not be
- Broadcasting can be used in the context of an RDD, a UDF or a Dataset and achieve the same result.



Broadcast variables: An example

fit@hcmus

- Suppose there is a list of words or values

```
# in Python
my_collection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")
words = spark.sparkContext.parallelize(my_collection, 2)
```

- Supplement the list of words with other information (right join)
  - This may be many KBs, MBs, or potentially even GBs in size.

```
# in Python
supplementalData = {"Spark":1000, "Definitive":200,"Big":-300, "Simple":100}
```

@lnthanh

Broadcast variables: An example

fit@hcmus

- Broadcast the structure across Spark and reference it by using suppBroadcast.

```
# in Python
suppBroadcast = spark.sparkContext.broadcast(supplementalData)
```

  - This value is immutable and is lazily replicated across all nodes when trigger an action

- Reference this variable via the value method, which returns the exact value that had earlier

```
# in Python
suppBroadcast.value
```

- Transform the given RDD using this value

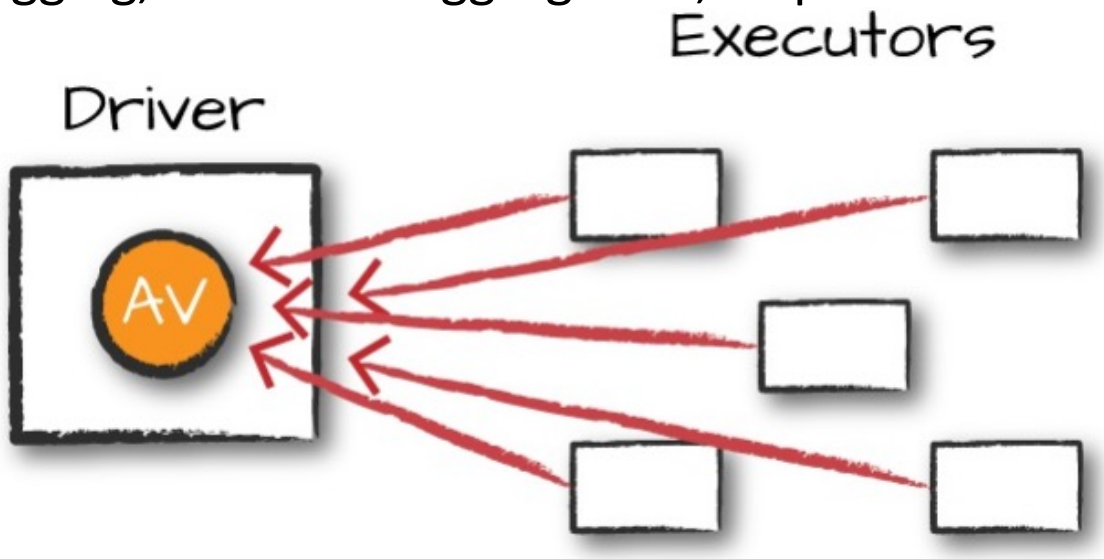
```
# in Python
words.map(lambda word: (word, suppBroadcast.value.get(
.sortBy(lambda wordPair: wordPair[1])\
.collect()
```

```
[('Big', -300),
('The', 0),
...
('Definitive', 200),
('Spark', 1000)]
```

@lnthanh

Accumulators

- Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis.
  - E.g., debugging, low-level aggregation, implement counters (as in MapRe



- Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Accumulators: An example

fit@hcmus

- Read the flight data stored in a Parquet-formatted file

```
# in Python
flights = spark.read\
.parquet("/data/flight-data/parquet/2010-summary.parquet")
```

- Create an accumulator to count the number of flights to or from China

```
# in Python
accChina = spark.sparkContext.accumulator(0)
```

- define the way to add to the accumulator

```
def accChinaFunc(flight_row):
    destination = flight_row["DEST_COUNTRY_NAME"]
    origin = flight_row["ORIGIN_COUNTRY_NAME"]
    if destination == "China":
        accChina.add(flight_row["count"])
    if origin == "China":
        accChina.add(flight_row["count"])
flights.foreach(lambda flight_row: accChinaFunc(flight_row))
```

Accumulators: An example

fit@hcmus

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	

©lnthanh