BIG DATA PROCESSING

EGCI 466

SPARK

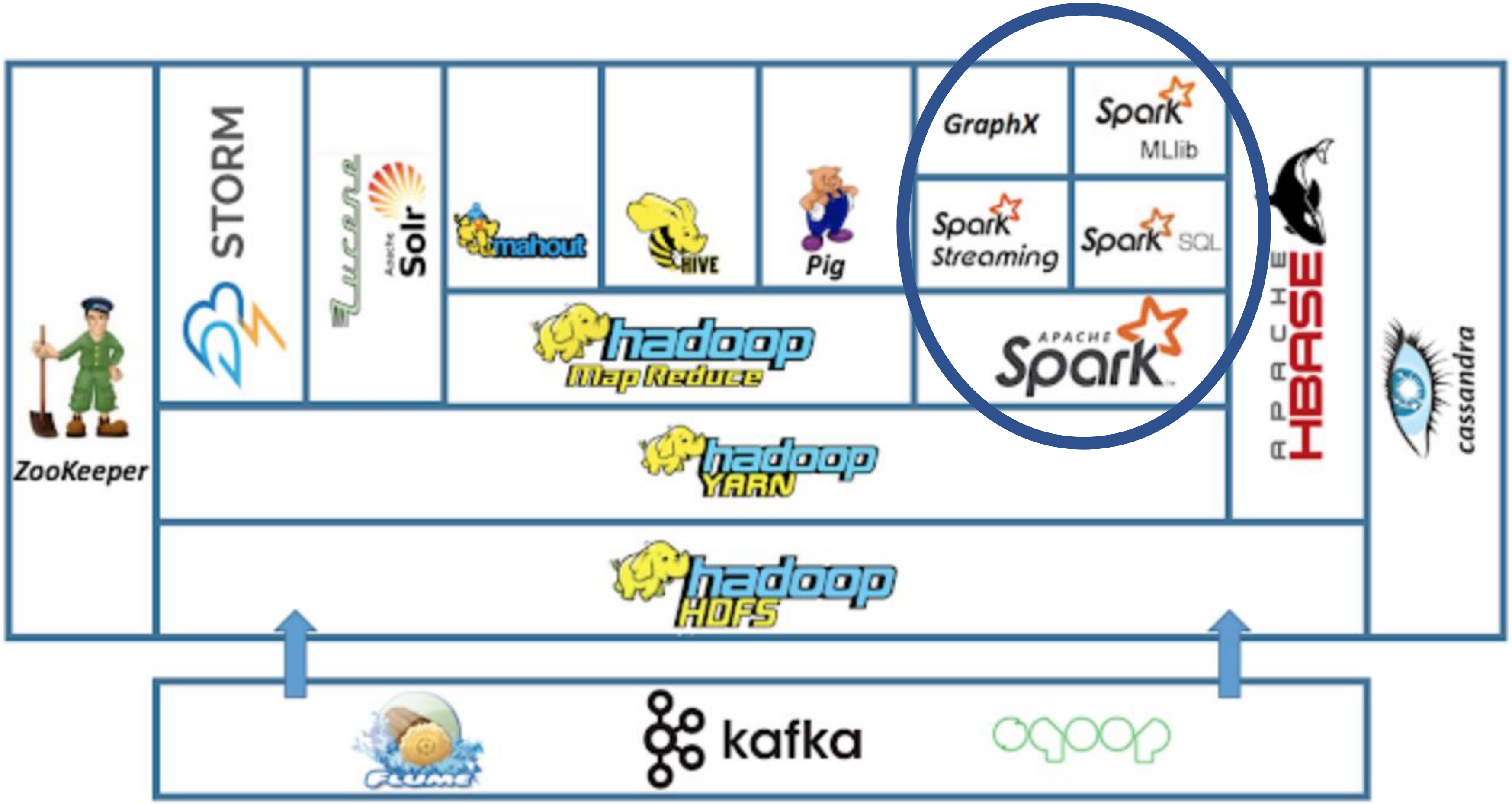
Apache SPARK

- Supports a computing framework for large scale data
- Provides parallel and distributed processing
- Provide scalability
- Provide fault tolerance on comodity hardware

Apache SPARK

- Is open sourced in-memory application framework
- Supports a computing framework for large-scale data processing and analysis.
- Provides parallel distributed data processing capabilities, scalability, and fault-tolerance on commodity hardware.
- Enable programming– Scala, python and Java APIs

Hadoop Layer



Spark vs Hadoop Maprduce

MapReduce involves a lot of I/O (slow)

- Limit only to map and reduce
- Difficult for more complicated data
- Native only for Java
- No iteractive shell support (interface)
- No support for streaming

Spark vs Hadoop Maprduce

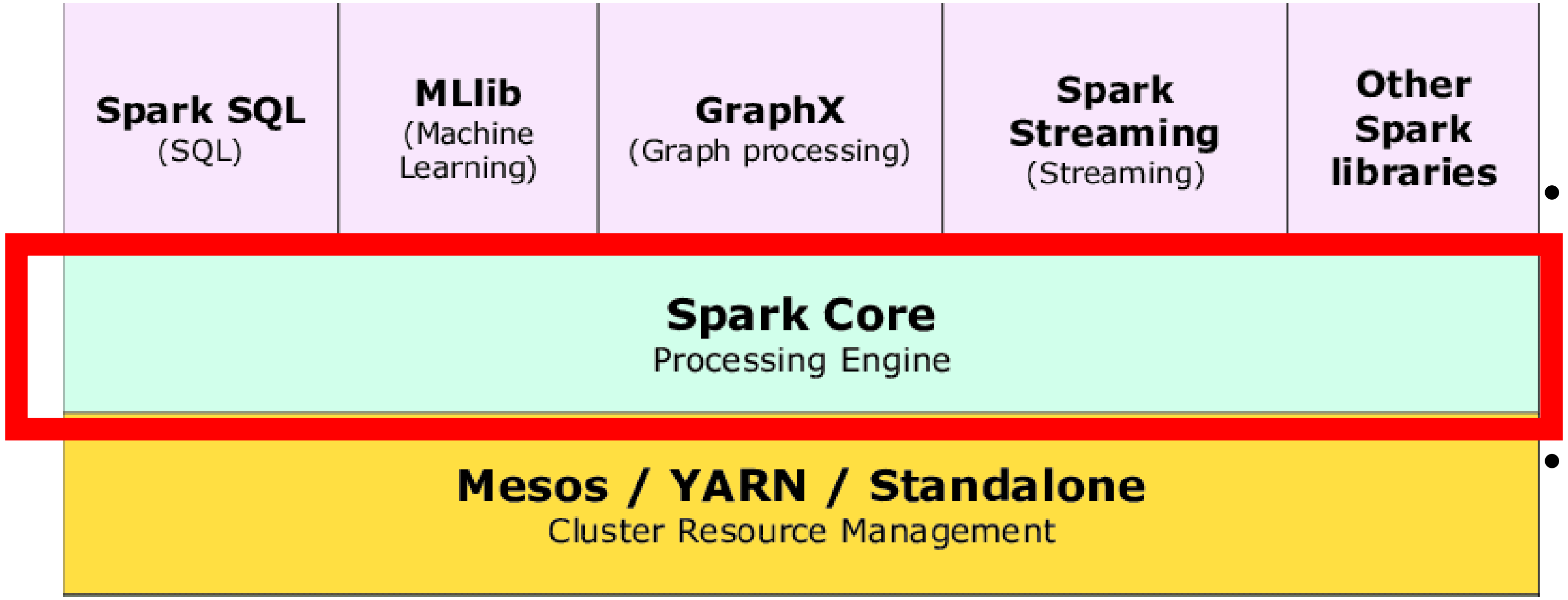
- In memory process
    - factor of 10-100 for some operations

RDD
(Resilient Data Distribution)

- Provide lots of modules for distributed system
- Simple APIs
- Interactive shell
- Support multiple workloads: batch & streaming

Mahidol University
Wisdom of the Land

Spark Stack



- Querying data
- Streaming
- Machine Learning Library
- Graph

Use RDD for programming abstraction (Resilient Data Distribution)

Carry data across many computing nodes in parallel, and transform it.

Functional Programming

- FP: mathematical programing style
- Use **expressions** instead of **statements**

- First implemenation LISP (LISt programming Language)

- Most recent: **Scala → Spark**

- Others: Java, Python, R

Traditionally

```
int inc(int x[],n){
    int i;
    for(i=0;i<n;i++)
        x[i]++;
}
```

```
def inc(myList){
    N=size(myList)
    for i in range(N):
        myList[i]+=1
    return myList
}
```

Functional Programming Example

- **Function creator**

f(x) =x+1

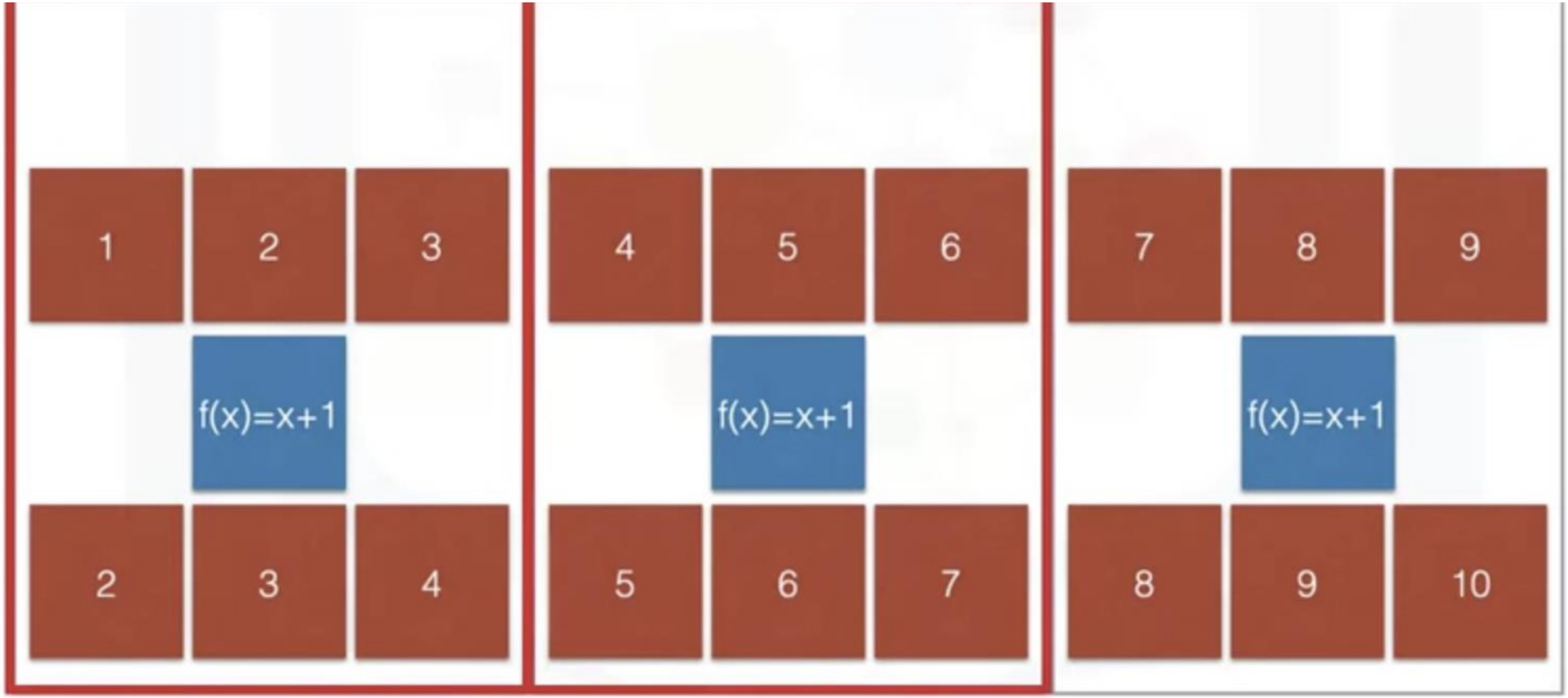
apply(f(x),

2
3
4
5
6

)=

3
4
5
6
7

Parallerlization



lambda

Lambda: function or operation with simple operations

```scala
val add=(x:Int, y:Int)
⇒x+y

println(add(1,2))
```

Scala

```python
add=lambda x,y:
    x+y

print(add(1,2))
```

Python

Spark Core

- Base engine, fault tolerance

- Perform scheduling, memory management

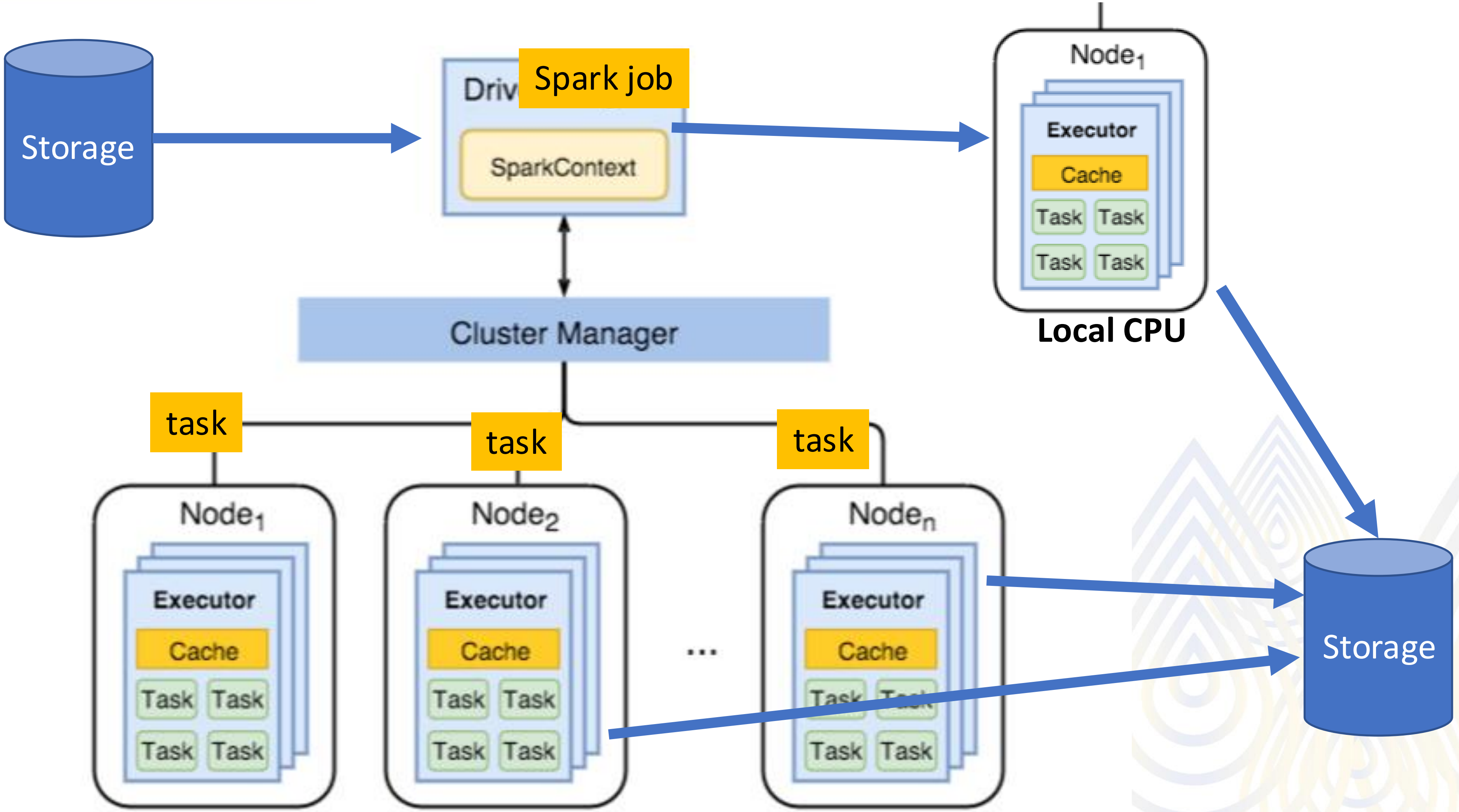
- Provide APIs to define RDD and other data type

- Use resilient distributed data sets (RDD) as the main programming abstraction

- Carry data across many the cluster nodes in parallel

(distribute and parallelize)

Spark Core


Storage
Driv
Spark job
SparkContext
Node1
Executor
Cache
Task Task
Task Task
Local CPU
Cluster Manager
task
task
task
Node1
Executor
Cache
Task Task
Task Task
Node2
Executor
Cache
Task Task
Task Task
...
Noden
Executor
Cache
Task Task
Task Task
Storage

Resilient Distributed Dataset (RDD)

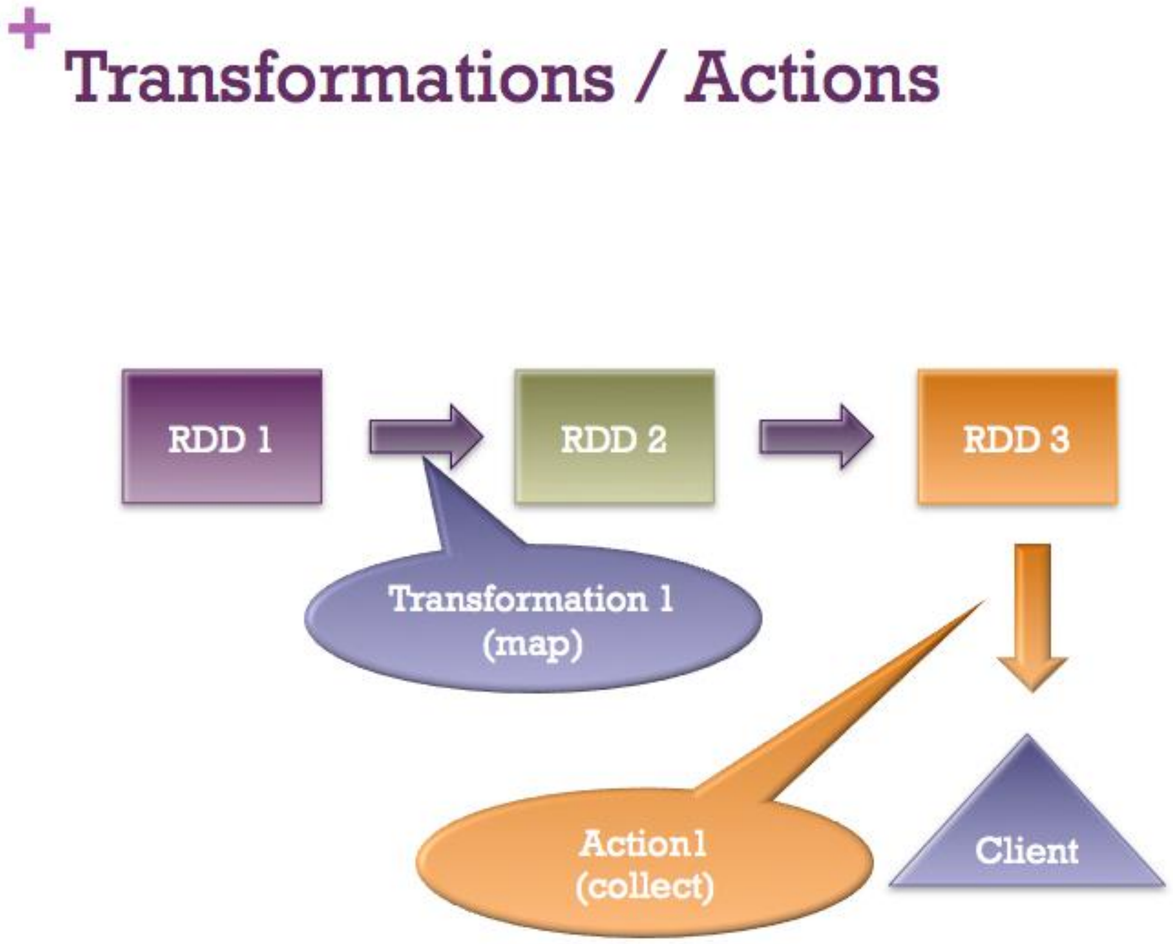
- Spark’s primary data abstraction
- Collections of fault tolerant elements partitioned across the cluster's nodes
- Partitioned across the nodes of the cluster
- Capable of accepting parallel operations
- Immutable, cannot be changed once created

Spark Applications

- Main components
  - Driver program → run user's main function + control worker node
  - worker nodes → Executor program → Running JVM driven by the driver
- Spark supports three main interfaces for cluster management
  - Spark's standalone cluster manager
  - the Apache Mesos,
  - Hadoop YARN
- Spark operation can Create, transform and Action

Processing RDD

- Create
  - From Many Type of data
- Transformations
  - Create a new version of RDD
  - Lazy evaluation
  - Not immediately executed
  - wait for an action to be performed
- Actions
  - Converted and saved in a persistent storage -- HDFS or local drive
  - Run time error usually happen here



Creating RDD

- Import from input files
  - line = sc.textfile(“hdfs://filename”)

- Create RDD from a list

```
data =[1,2,3,4,5] //python example
distData= sc.parallerize(data,3)
```

parallelize into 3 partitions

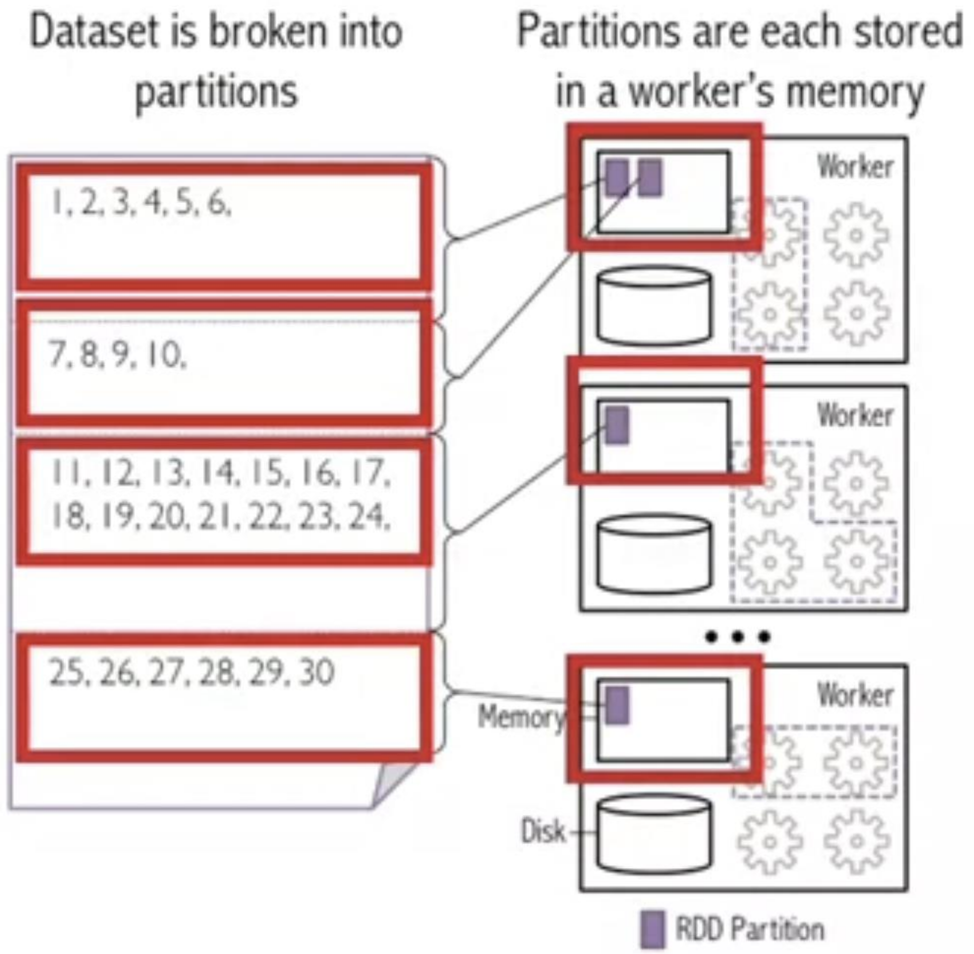
[1,3], [2,5], [4]

```
data =Array(1,2,3,4,5)//Scala example
val distData= sc.parallerize(data,3 )
```

- Transform from existing RDD

RDD supoorted file types

- Text
- SequenceFiles
- Avro
- Parquet
- Hadoop input format



RDD supported file formats

- Local
- Cassandra
- Habase
- HDFS
- Amazon S3
- etc

Resilience in RDD

- The persisting or caching of a data set in memory across operations.
- The cache is fault tolerant and always recoverable
- RDDs are immutable and Hadoop provide fault tolerant.
- Each node stores the partitions and subsequent action
- Persisting or caching is used as a key tool for iterative algorithms and fast interactive use

Spark Cheat Sheet

Method	Desctiption	Usage
`appName()`	A name for your job to display on the cluster web UI.	`from pyspark.sql import SparkSession spark = SparkSession.builder.appName("MyApp").getOrCreate()`
`cache()`	caches the specified RDD in the memory of your cluster's workers. Caching operation takes place only when a Spark **action**	`df = spark.read.csv("customer.csv") df.cache()`
`pip install()`	Find the lastes version and install	`pip install pyspark`
`sc.parallalize()`	Creates a parallelized collection.	rdd = sc.parallelize([1, 2, 3, 4, 5])

Advantages

- **Increases Manageability**
  - small operations
  - group operations
- **Reduction of Complexity**
  - ***Time*** is saved. The action gets triggered only when data is required
  - **Space** is used only when necessary, which saves space.
- **Saves Computational Power and increases Speed:**
  - Spark functions get triggered through the driver and run on a cluster.
  - Lazy evaluation only triggers a computation when necessary
  - Saves trip between Driver and Cluster – speed the execution

Optimization:

- The number of queries being run is very low at a given time -- optimized

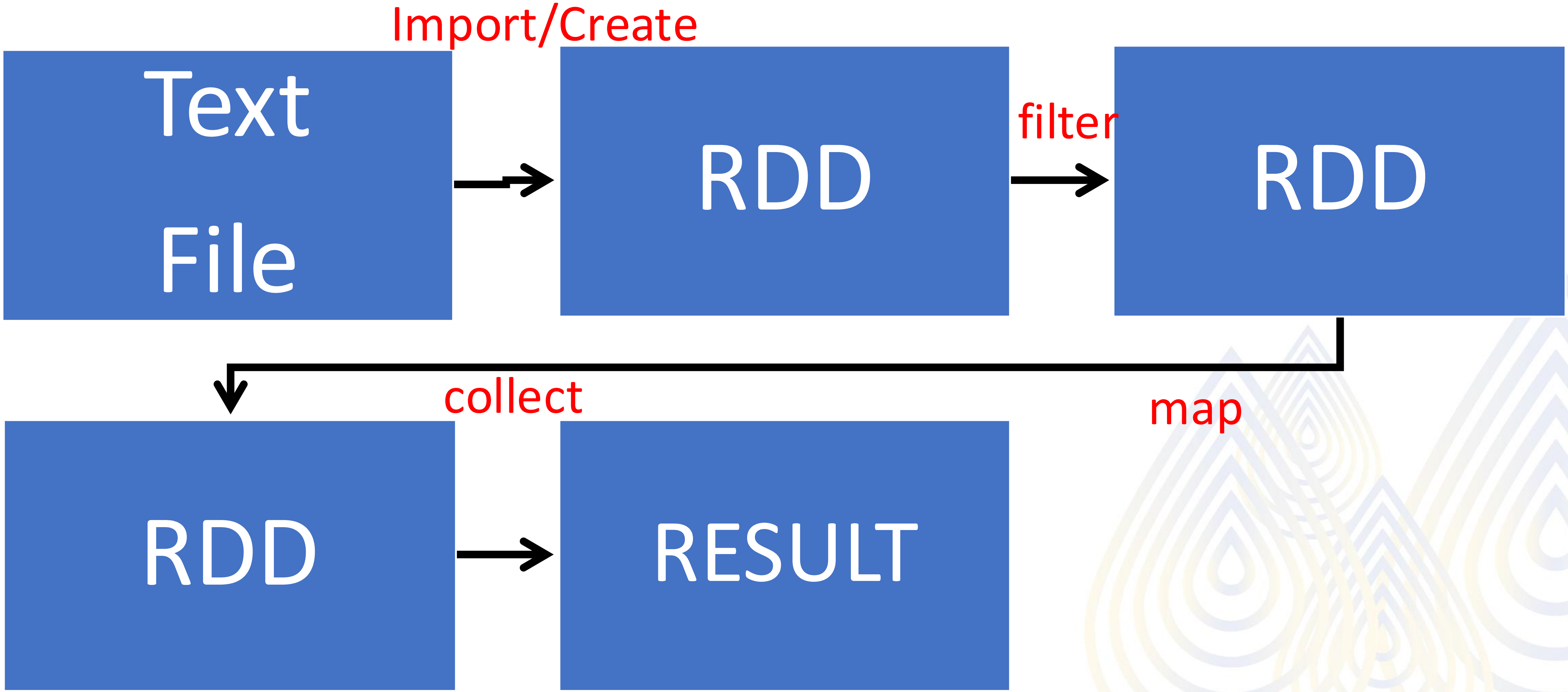
Spark Core: Transformations

- Transformations do not execute immediately (lazy)
- A transformation is a single pipeline for execution later (Action)
- Keep track of all process
- The proess is linear

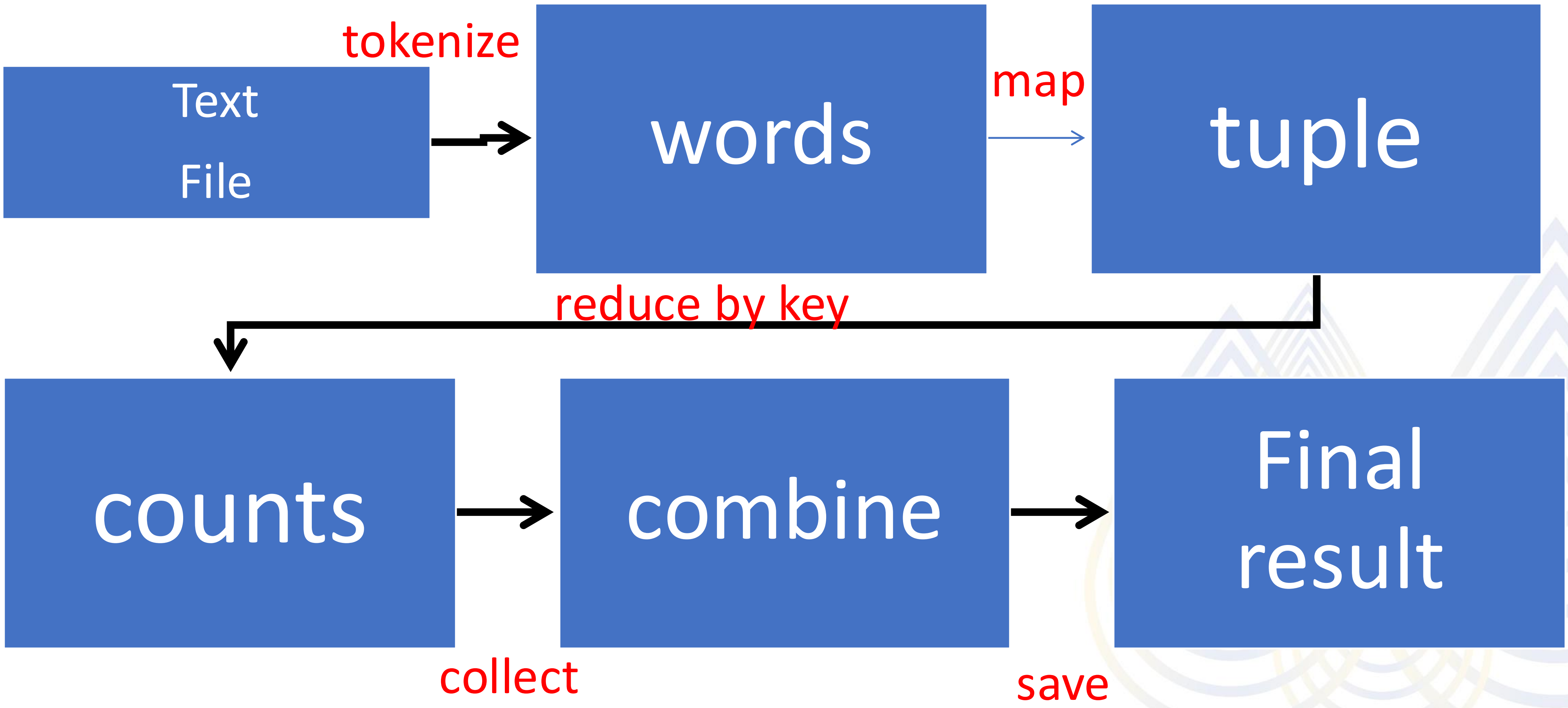
Transformation types

- **Narrow Transformations**
  - Do not require data shuffling across partitions.
  - **map()**, **flatmap(), filter() , coalesce()**
- **Wide Transformations**
  - **Require** data shuffling through partitions
  - **GroupByKey, ReduceByKey**

Processing RDD : Example



Processing RDD : word count



Word coupt Example

- Import from input files
  - **line** = sc.textfile(“hdfs:/filename”)
- Transformation (Tokenized)
  - words = lines.flatMap(lambda line : line.split(" "))
- Transformation (count)
  - tuples = words.map(lambda word : (word, 1))
- Transformation (Reduce)
  - counts = tuples.reduceByKey(lambda a, b: (a + b))
- Save
  - counts.coalesce(1).saveAsTextFile('hdfs:/user/cloudera/wordcount/outDir')

map transformation

- 1:1 operation
- apply to each element of RDD

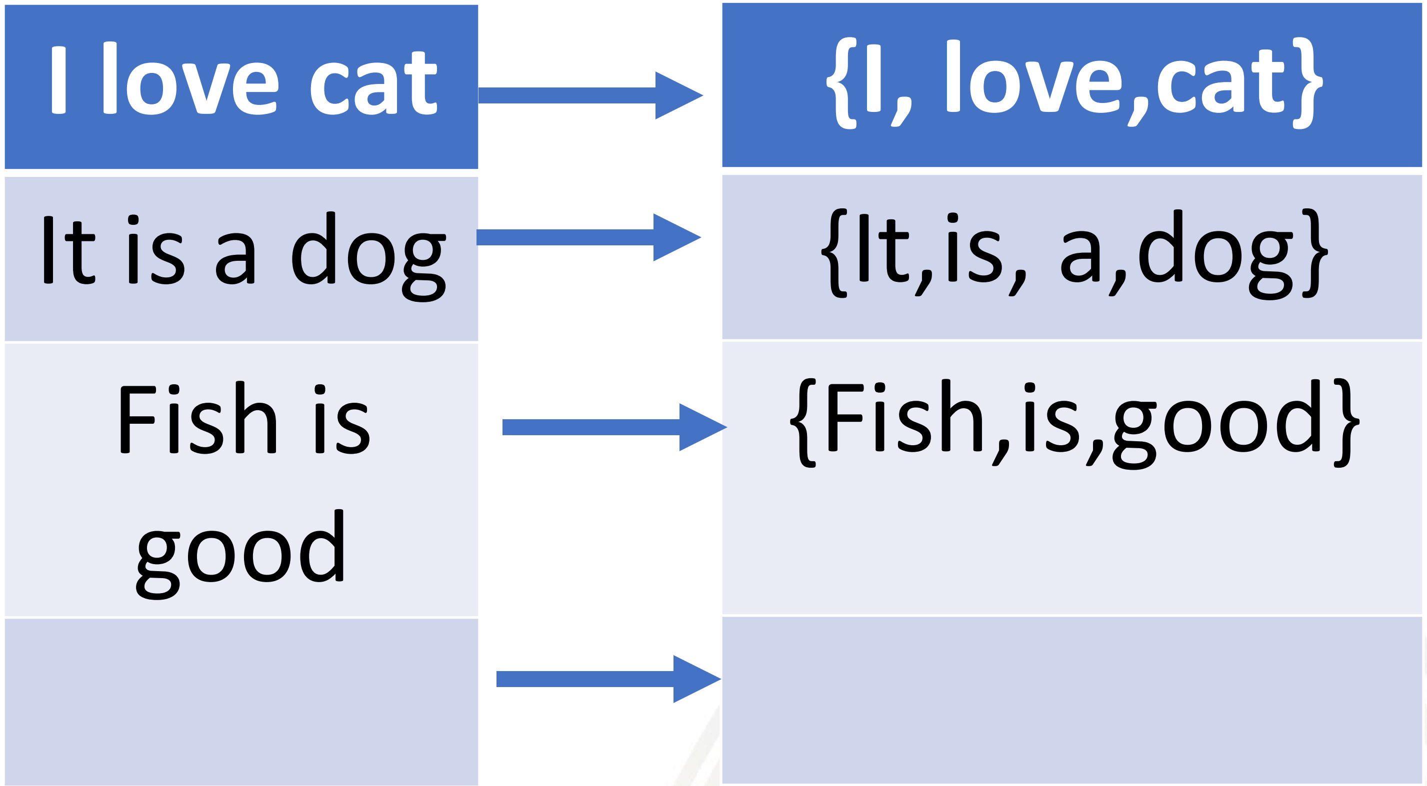
```
def lower(line):
     return line.lower()

lower_text_RDD=text_RDD.map(lower)
```

Cat	→	cat
Bee	→	bee
dOg	→	dog
Fish	→	fish
pine	→	pine

flatMap transformation

- map then flatten output

```
def split_words(line):
    return line.split()
```




```
words_RDD=text_RDD.flatmap(split_words)
words_RDD.collect()
```

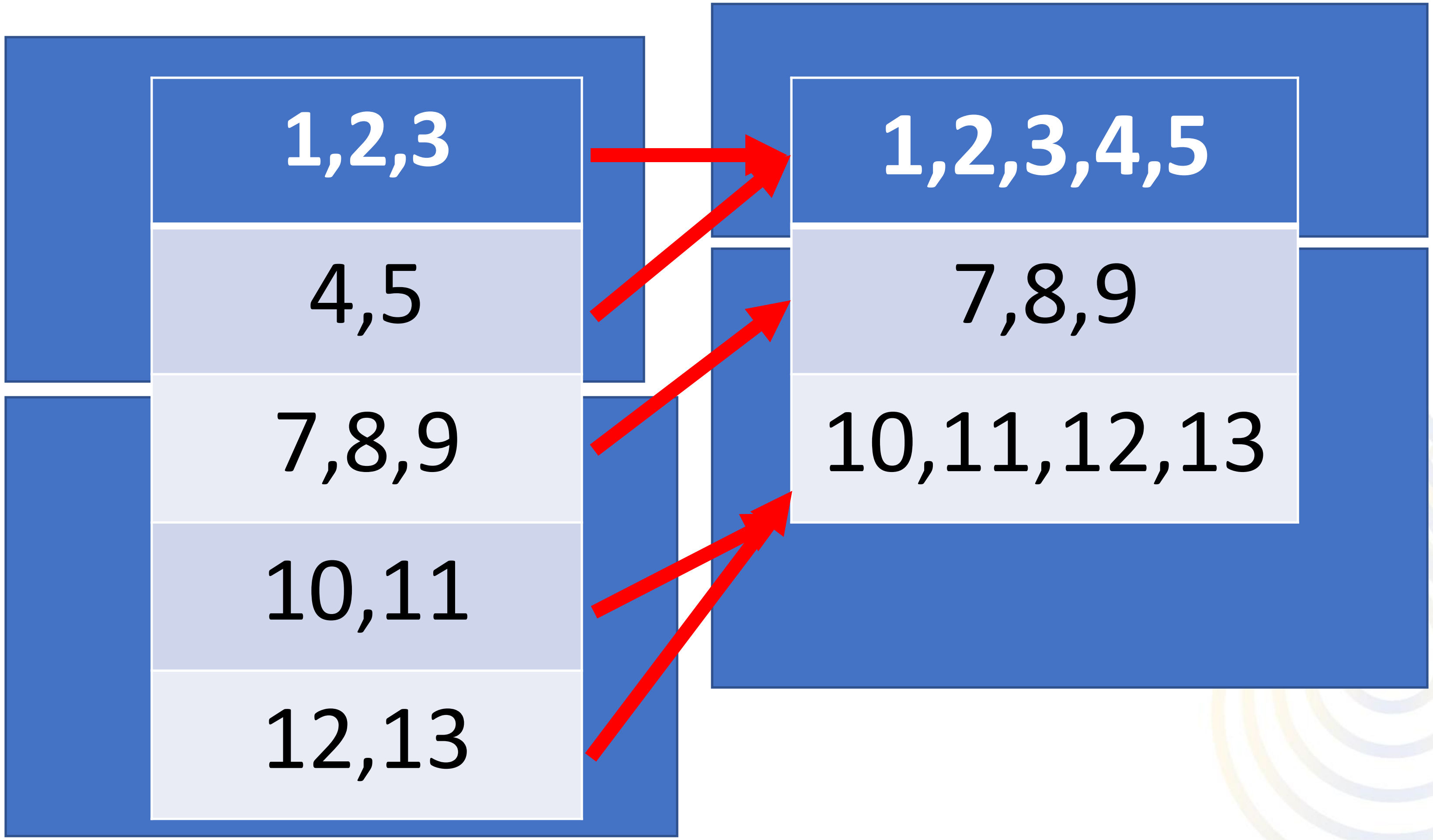
Transformation:filter

Example functions: filter()

```
using python:
  hadoopbooksRDD=booksRDD.filter(lambda x: “hadoop” in x)
  kafkabookRDD=booksRDD.filter(lambda x: “kafka” in x)

 def starts_with_a(word)
        return word.lower().startswith(“a”)
 words_RDD.filter(strats_with_a).collect
```

Transformation:coalease

- Reduce the number of partition



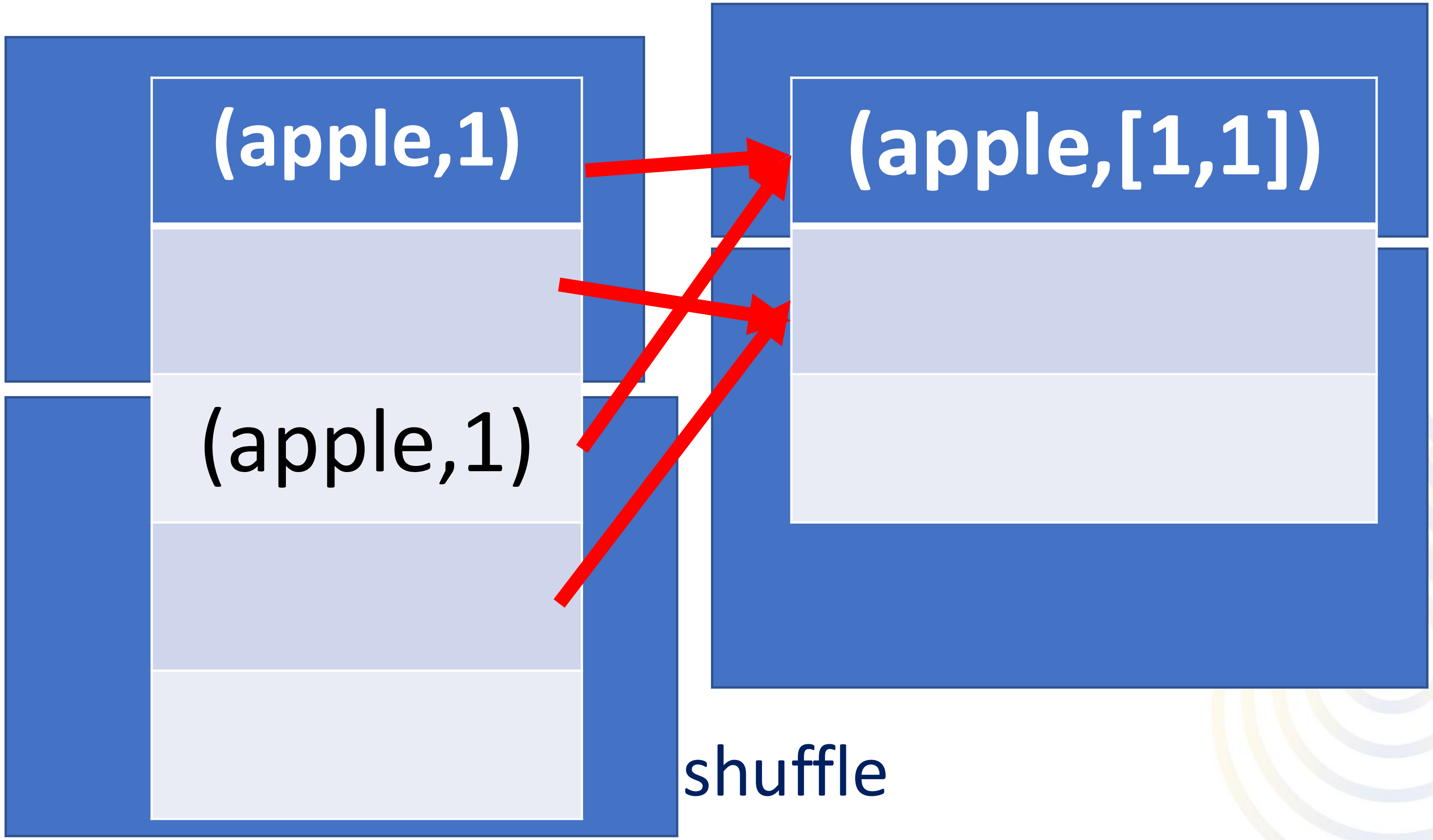
Wide transformation

- Require shuffle
- Group by partition
- reduceByKey, groupByKey

Mahidol University
Wisdom of the Land

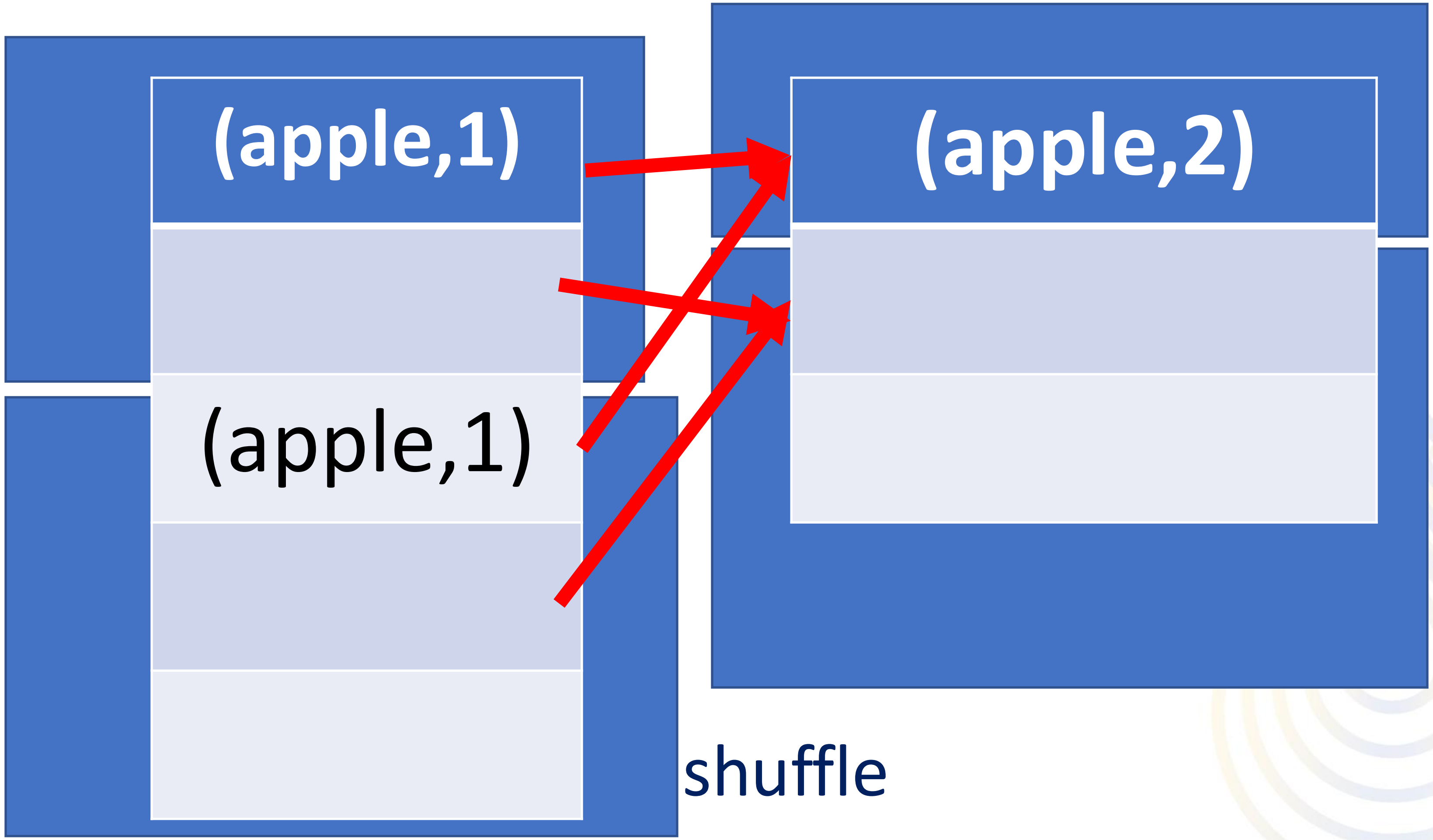
groupByKey

(Key,value) ➔ (key , [list of all value])



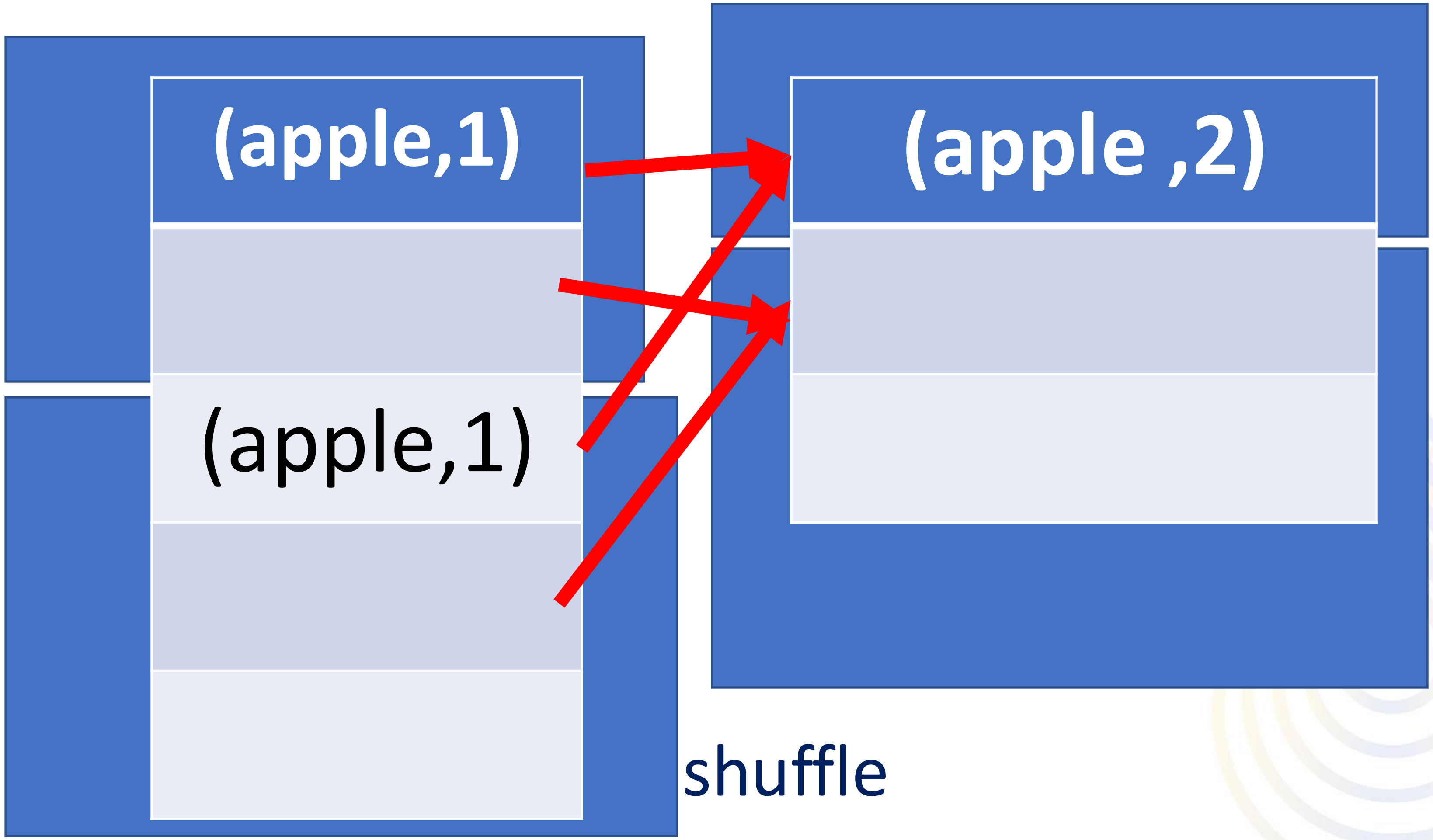
Mahidol University
Wisdom of the Land

groupByKey +reduce



reduceByKey

- Reduce = sum for this example



Spark core: Actions

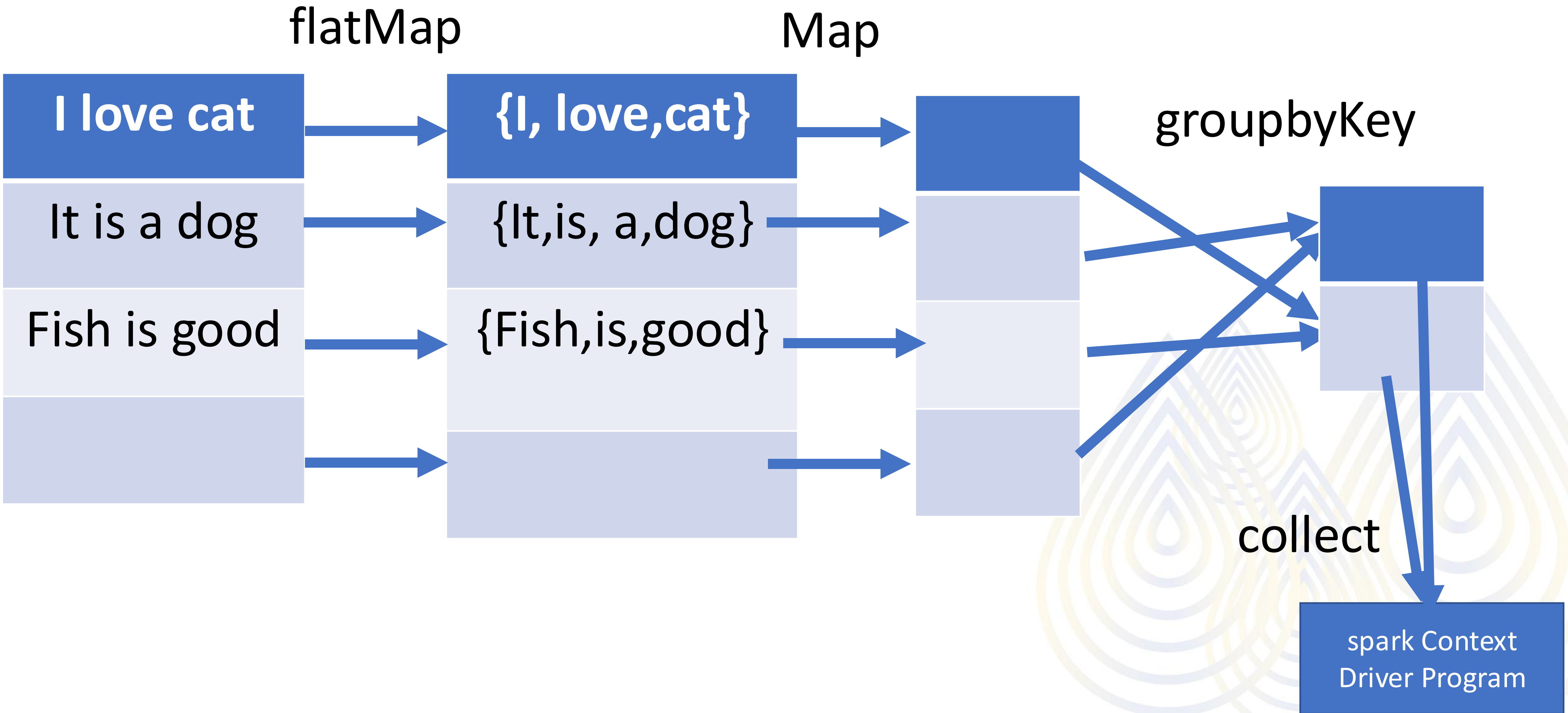
- Last step in a Spark pipeline

- Actions operations trigger the evaluation of transformation pipeline

- Return the final result to the driver program or save the results to a persistent storage.

- Collect is the most common action

Action: word count



The most common Actions

Action	Usage
`Collect()`	Copy all element to driver
`Take(n)`	Copty first ***n*** element
`Reduce(func)`	Aggregate elements with ***func*** (take 2 elements, return1)
`Count()`	Count number of data
`saveAsTextFile(Filename)`	Save to a local or HDFS file

Example 2

- [Letter Count]

More references

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Exeperiment on cloud

- Use pySpark as submit job
- Use Notebook to run the job

Reference: https://spark.apache.org/docs/latest/rdd-programming-guide.html