INT 303 BIG DATA ANALYTICS

Lecture6: *Infrastructure that supports Big Data processing*

Jia WANG

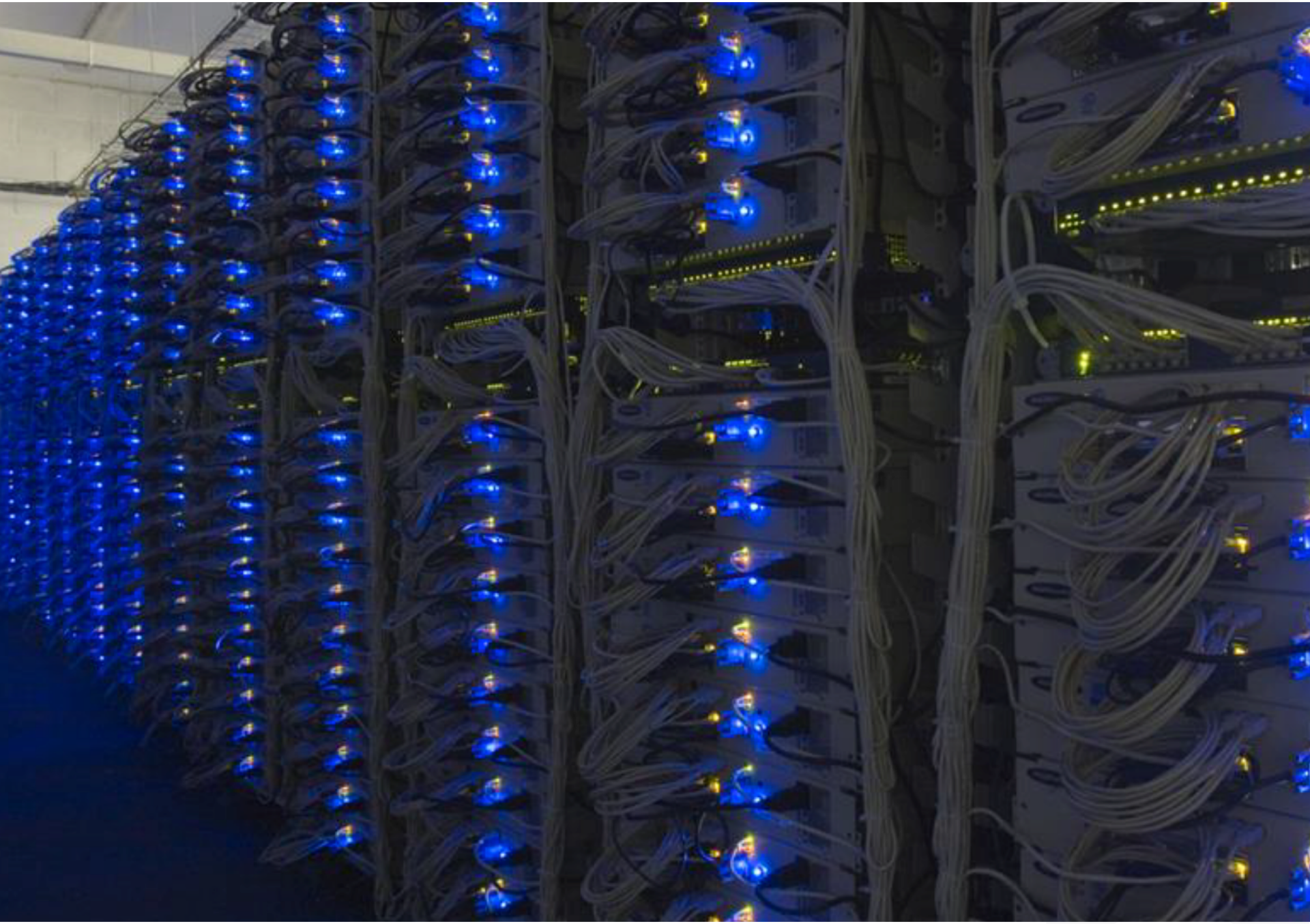
Jia.wang02@xjtlu.edu.cn

Xi'an Jiaotong-Liverpool University

西交利物浦大學

OUTLINE

- Large-scale computing
- Distributed file system
- MapReduce: Distributed computing programming model
- Spark: Extends MapReduce



LARGE-SCALE COMPUTING

Challenges:

- **How do you distribute computation?**
- **How can we make it easy to write distributed programs?**
- **Machines fail:**
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to lose 1/day
  - With 1M machines 1,000 machines fail every day!

AN IDEA AND A SOLUTION

Issue:

Copying data over a network takes time

Idea:

- Bring computation to data
- Store files multiple times for reliability

- **Spark/Hadoop address these problems**
  - **Storage Infrastructure –File system**
  - Google: GFS. Hadoop: HDFS
- **Programming model**
  - MapReduce
  - Spark

STORAGE INFRASTRUCTURE

Problem:

- If nodes fail, how to store data persistently?

Answer:

- **Distributed File System**
  - Provides global file namespace

Typical usage pattern:

- Huge files (100s of GB to TB)
- Data is rarely updated in place
- Reads and appends are common

DISTRIBUTED FILE SYSTEM

Chunk servers

- File is split into contiguous chunks
- Typically each chunk is 16-64MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

Master node

- a.k.a. Name Node in Hadoop'sHDFS
- Stores metadata about where files are stored
- Might be replicated

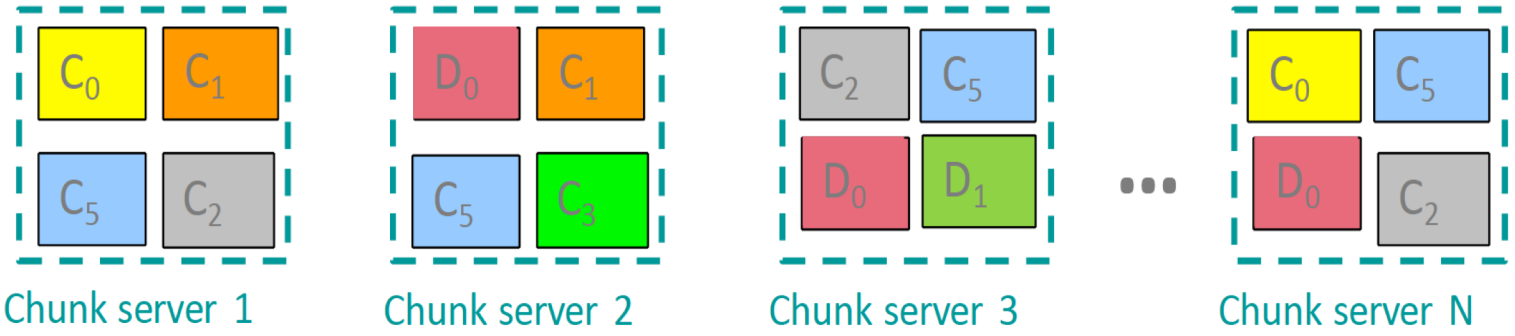
Client library for file access

- Talks to master to find chunk servers
- Connects directly to chunk servers to access data

DISTRIBUTED FILE SYSTEM

Reliable distributed file system

- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
- Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

MAPREDUCE: DISTRIBUTED COMPUTING PROGRAMMING MODEL

PROGRAMMING MODEL: MAPREDUCE

MapReduce is a style of programming designed for:

- Easy parallel programming
- Invisible management of hardware and software failures
- Easy management of very-large-scale data

It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called “MapReduce”

MAPREDUCE: OVERVIEW

3 steps of MapReduce

Map:

- Apply a user-written *Map function* to each input element
  - *Mapper* applies the Map function to a single element
  - Many mappers grouped in a *Map task*(the unit of parallelism)
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.

Group by key: Sort and shuffle

- System sorts all the key-value pairs by key, andoutputs key-(list of values) pairs

Reduce:

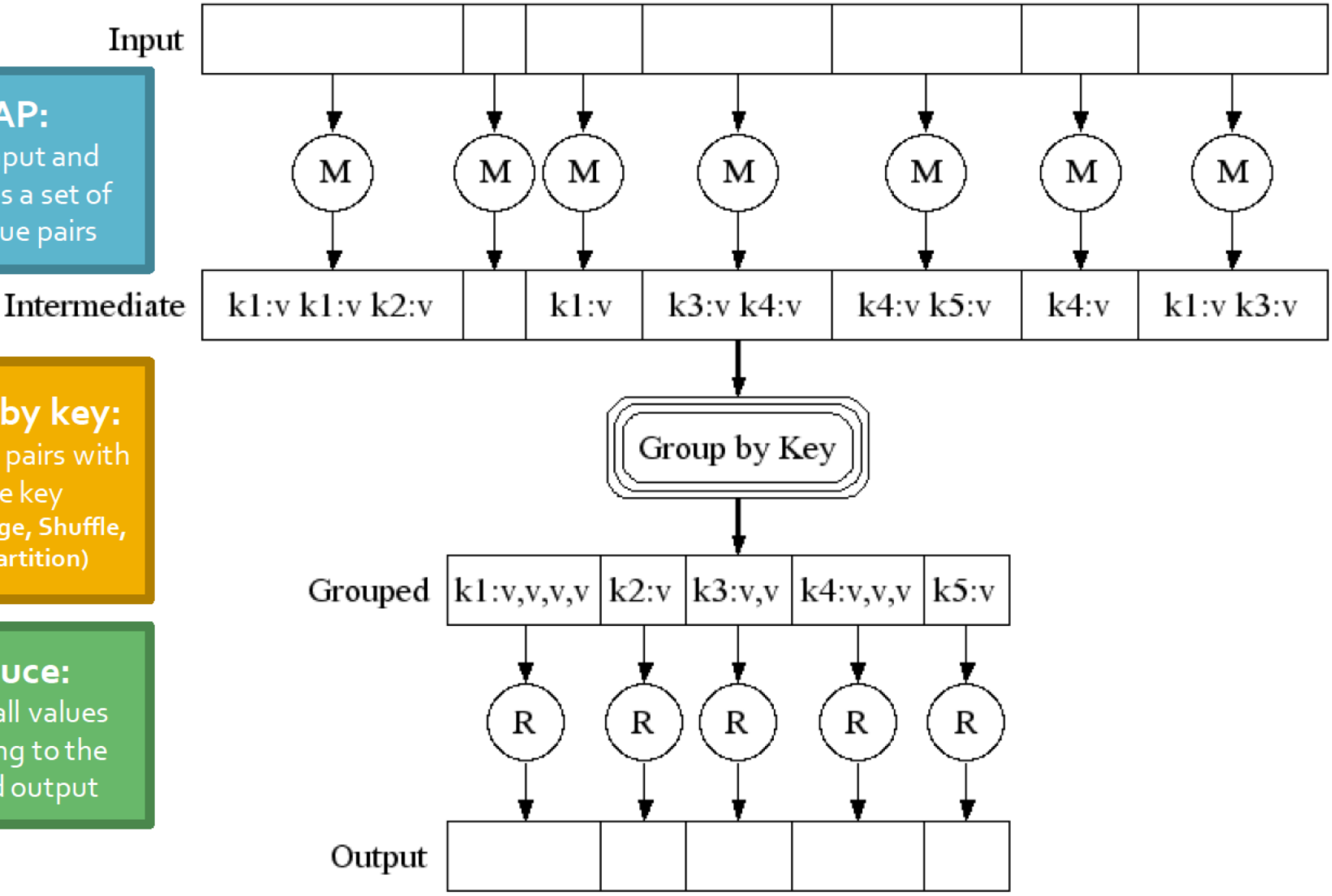
- User-written Reduce functions applied to each key-(list of values)

Map-Reduce: A diagram

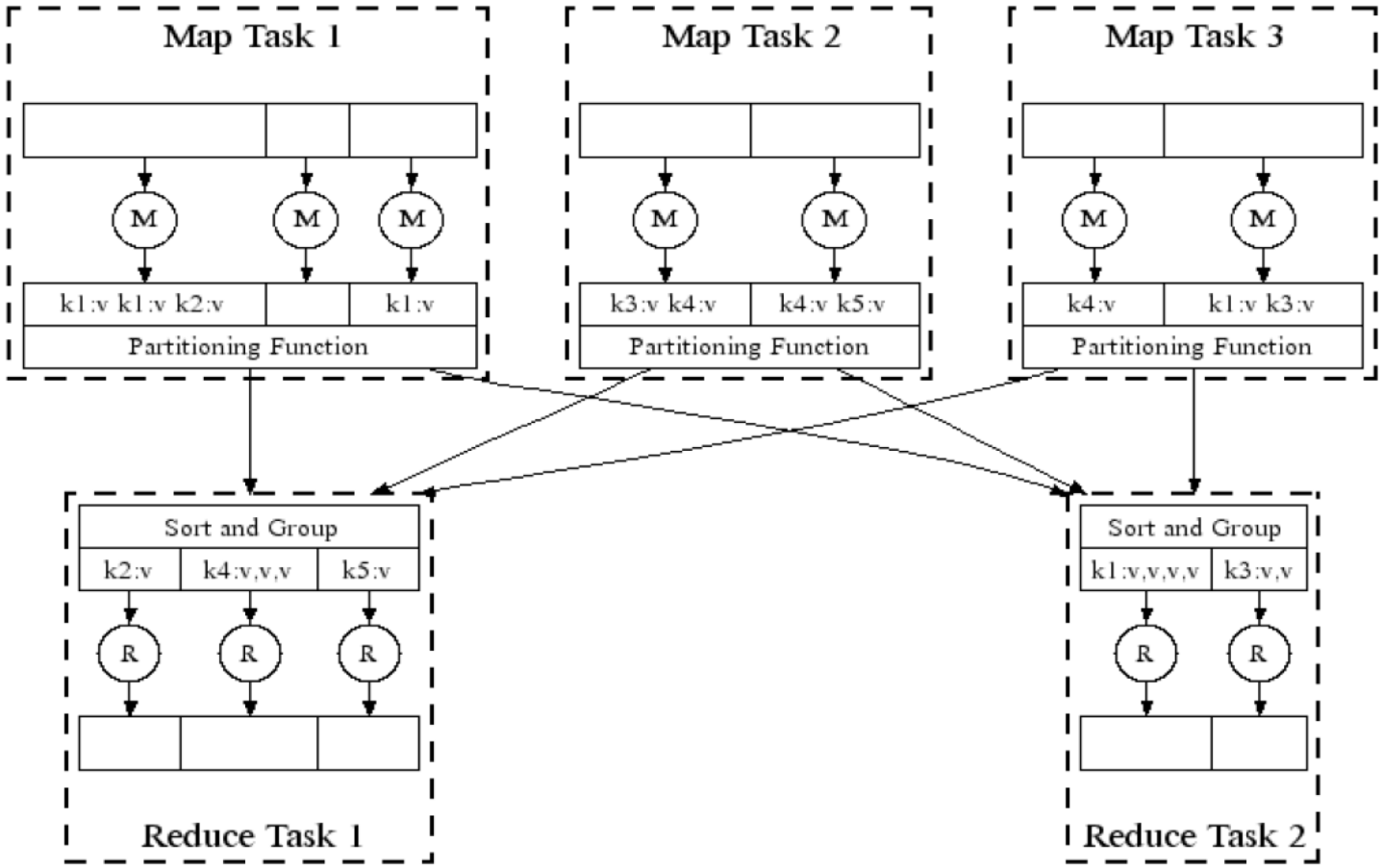
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



Map-Reduce: In Parallel

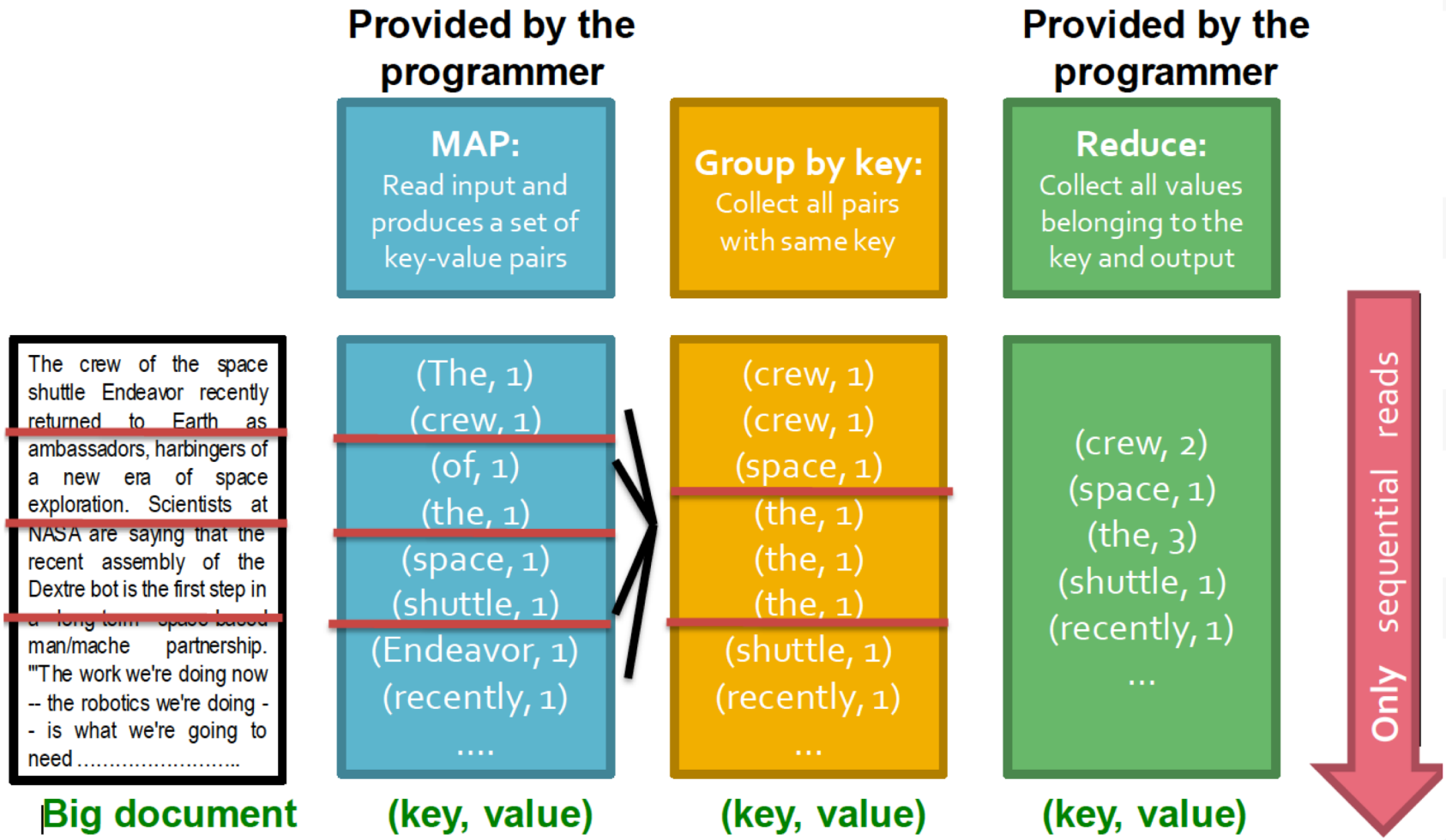


All phases are distributed with many tasks doing the work

Map-Reduce: In Parallel

- Each mapper/reducer must generate the same number of output key/value pairs as it receives on the input. (Wrong)
- The output type of keys/values of mappers/reducers must be of the same type as their input. (Wrong)
- The inputs to reducers are grouped by key. (True)
- It is possible to start reducers while some mappers are still running . (Wrong)

EXAMPLE: WORD COUNTING



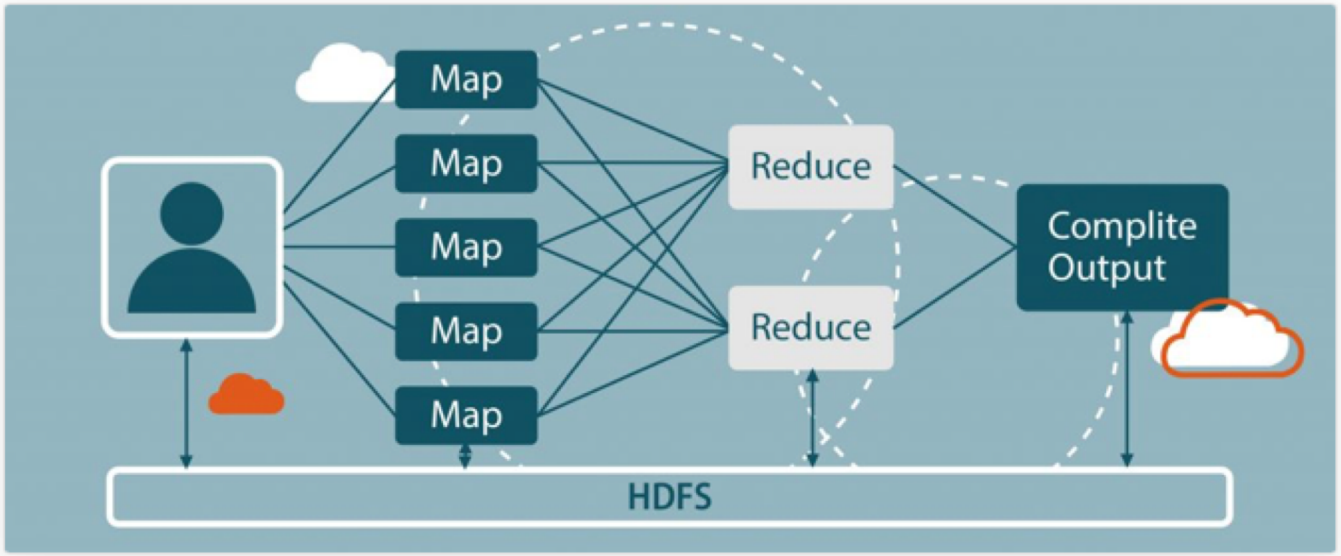
Word Count Using MapReduce

```
map(key, value):
# key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

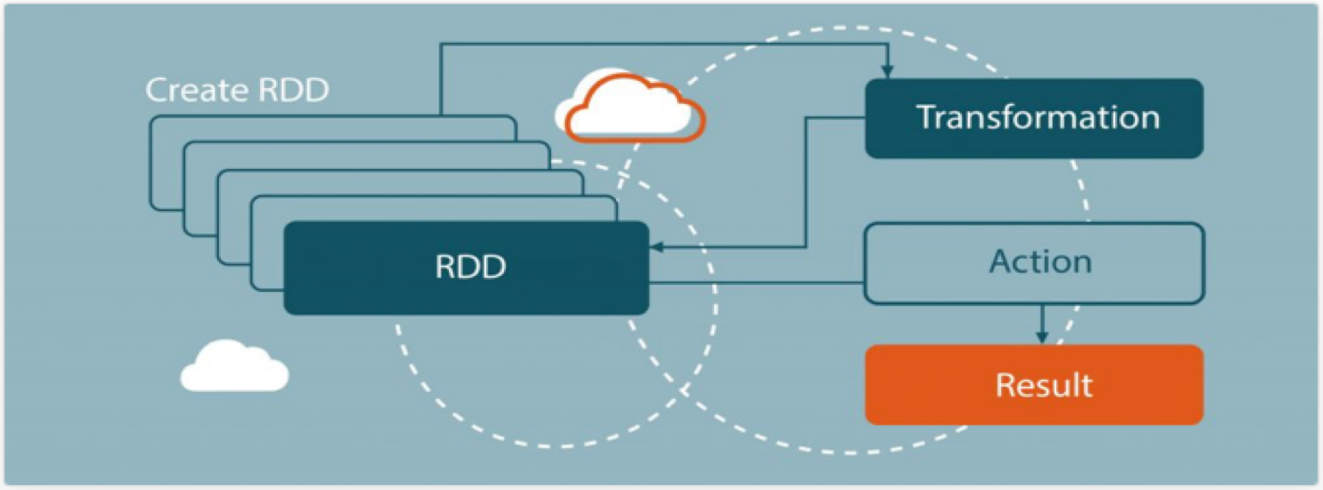
```
reduce(key, values):
# key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

SPARK: EXTENDS MAPREDUCE

Hadoop MapReduce



Apache's RDD



SPARK: OVERVIEW

Open source software (Apache Foundation)
Supports **Java, Scala and Python**

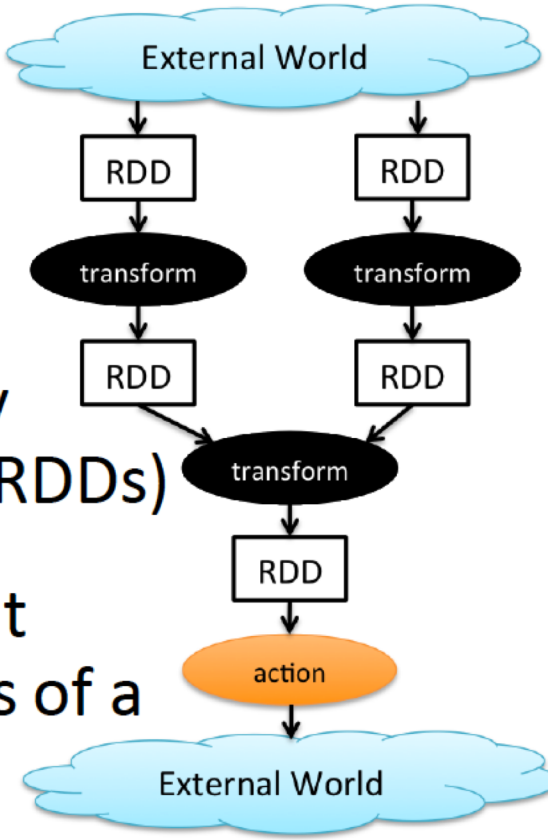
Key construct/idea: Resilient Distributed Dataset (RDD)
Higher-level APIs:

- DataFrames& DataSets
- Introduced in more recent versions of Spark
- Different APIs for aggregate data, which allowed to introduce SQL support

Spark: RDD

Key concept *Resilient Distributed Dataset* (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible
- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset



Spark RDD Operations

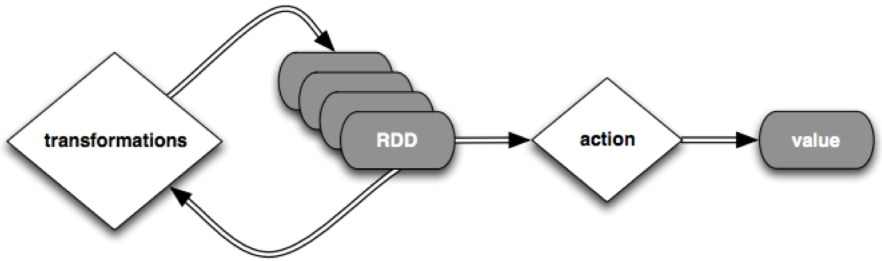
- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include *map, filter, join, union, intersection, distinct*
  - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
  - Actions include *count, collect, reduce, save*
  - Actions can be applied to RDDs; actions force calculations and return values

WORD COUNT!

```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 3)
wordCountsCollected = (wordsRDD
                       .map(lambda w: (w, 1))
                       .reduceByKey(lambda x,y: x+y)
                       .collect())
[('rat', 2), ('elephant', 1), ('cat', 2)]
```

FILE RDD —flatMap—>[list of words]—map—>[(word,1),..]—reduceByKey (map combiner + reduce transformation) —>[(word, count),..] --save Action-->File

```
text_file = spark.textFile("hdfs://...")
counts = (text_file.flatMap(lambda line: line.split(" "))
        .map(lambda word: (word, 1))
        .reduceByKey(lambda a, b: a + b)
)
counts.saveAsTextFile("hdfs://...")
```



BROADCASTERS AND ACCUMULATORS

- Broadcaster: read only-cached on each machine, spread data
- broadcastVar = sc.broadcast(list(range(1, 4)))
- Accumulator: only seen on driver

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
 global accum
 accum += x
rdd.foreach(f)
accum.value
```

CREDITS AND READING

Databricks and Berkeley Spark MOOC: https://www.edx.org/course/introduction-big-dataapache-spark-uc-berkeleyx-cs100-1x (the animals examples are stolen right from there).
» http://spark.apache.org/examples.html
» https://www.mapr.com/blog/5-minute-guide-understanding-significance-apache-spark
» http://spark.apache.org/docs/latest/programming-guide.html (many quotes taken from here)
» http://training.databricks.com/workshop/itas_workshop.pdf (a lot of figures are stolen from here)
» http://www.slideshare.net/pacoid/crash-introduction-to-apache-spark
» http://nbviewer.ipython.org/github/tdhopper/rta-pyspark presentation/blob/master/slides.ipynb (some code examples taken from here)
»https://speakerd.s3.amazonaws.com/presentations/7ee131d78a2b43338de693076ed4ecc1PySparkBestPractices.pdf

THANK YOU

VISIT US

WWW.XJTLU.EDU.CN

FOLLOW US

@XJTLU

Xi'an Jiaotong-Liverpool University
西交利物浦大学