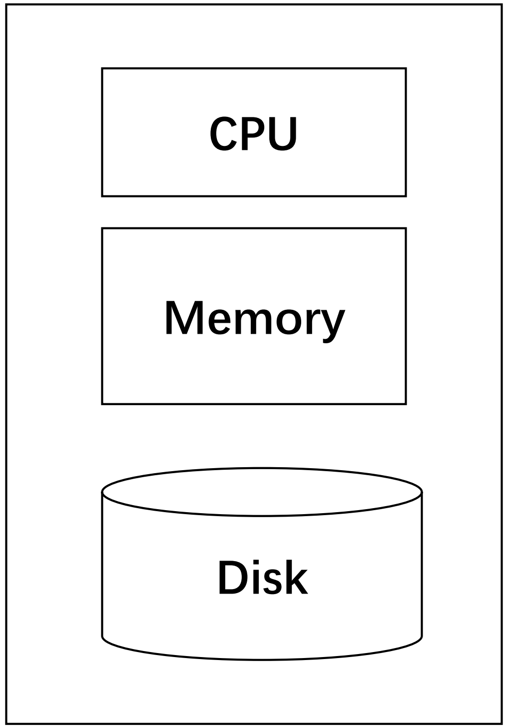
CS3319 Foundations of Data Science

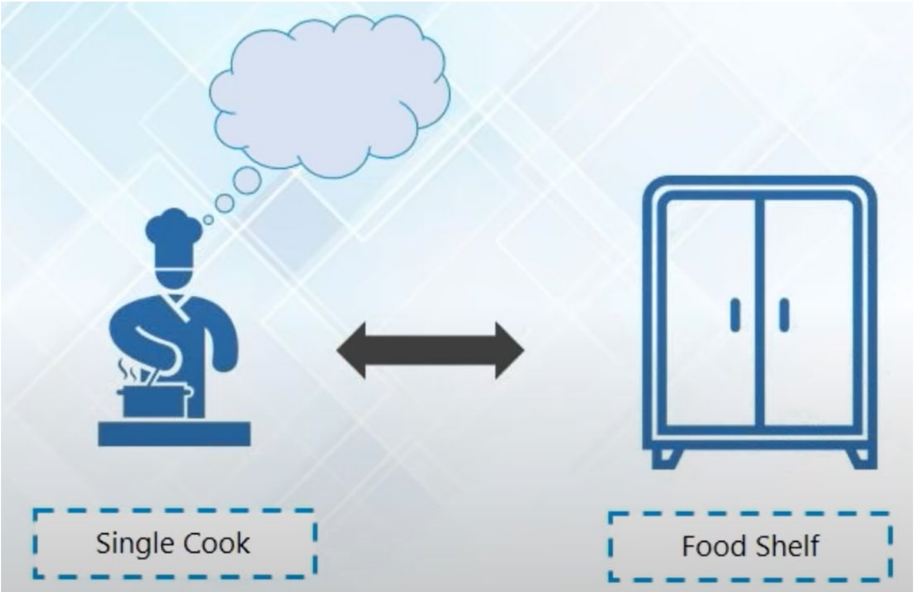
3. MapReduce

Jiaxin Ding
John Hopcroft Center

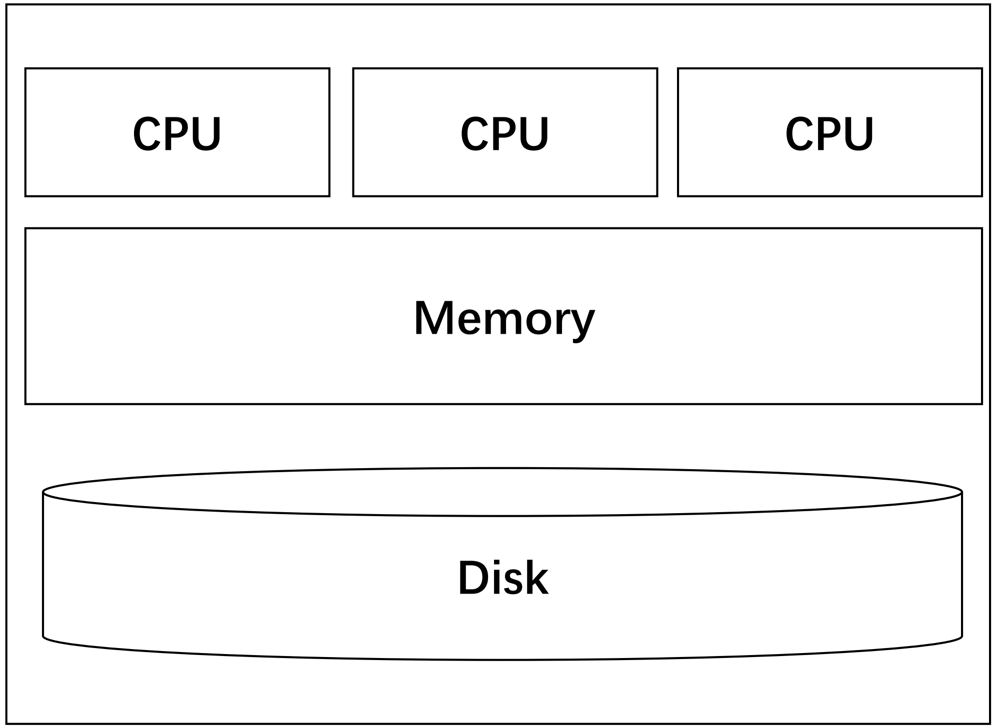
上海交通大学
约翰·霍普克罗夫特计算机科学中心
John Hopcroft Center for Computer Science

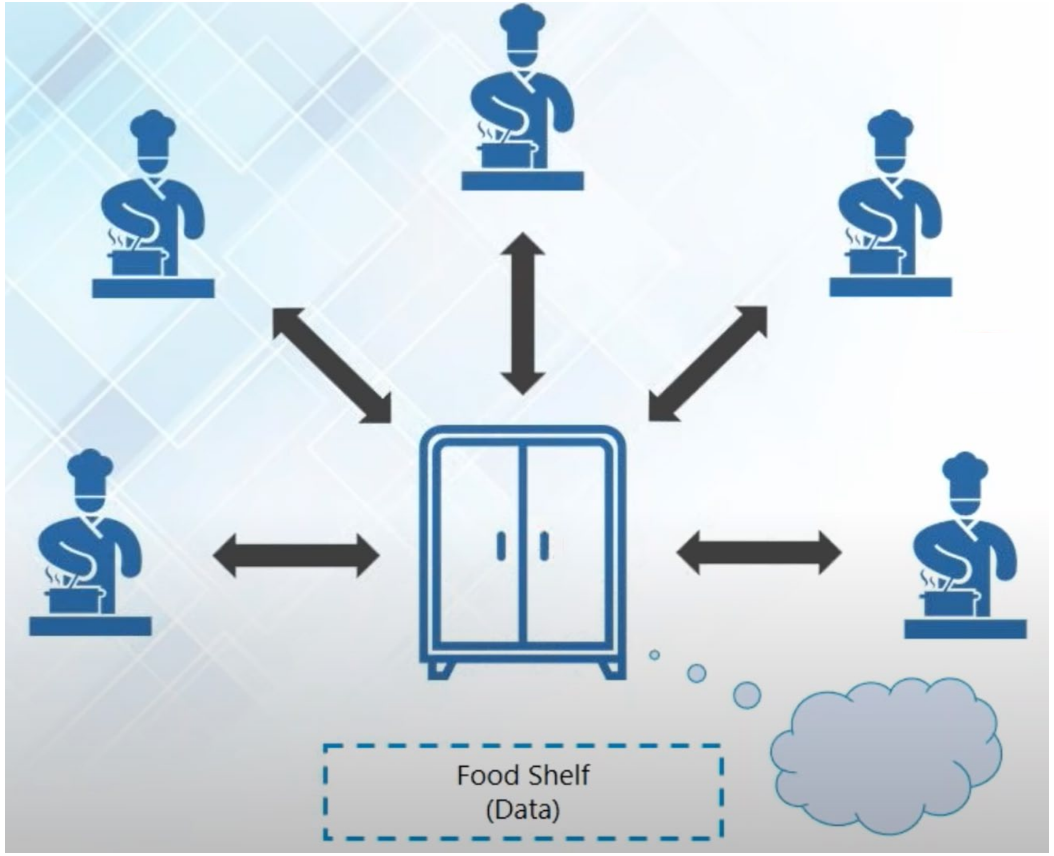
Single Node Architecture



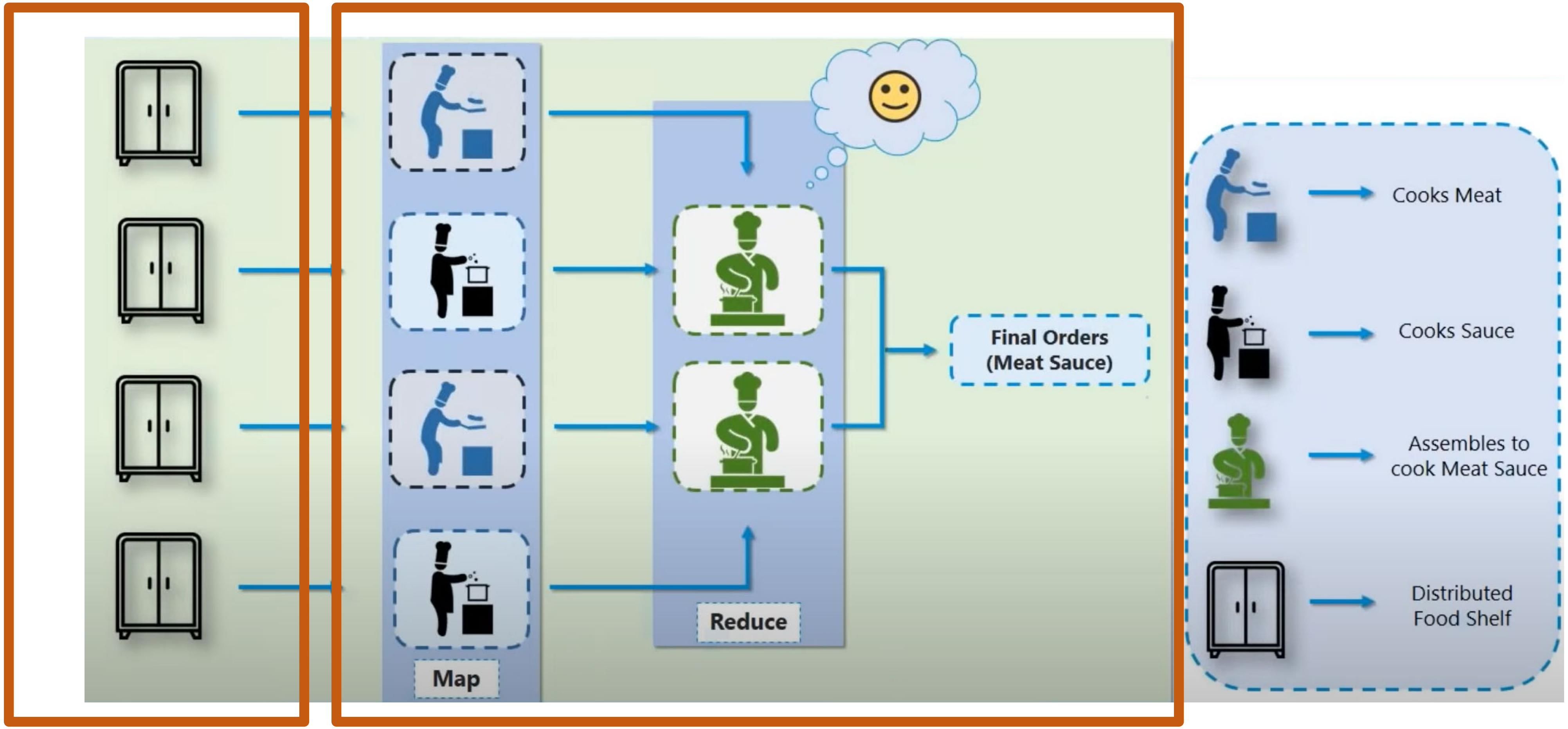


Distributed Computing





MapReduce



Distributed File System **MapReduce**

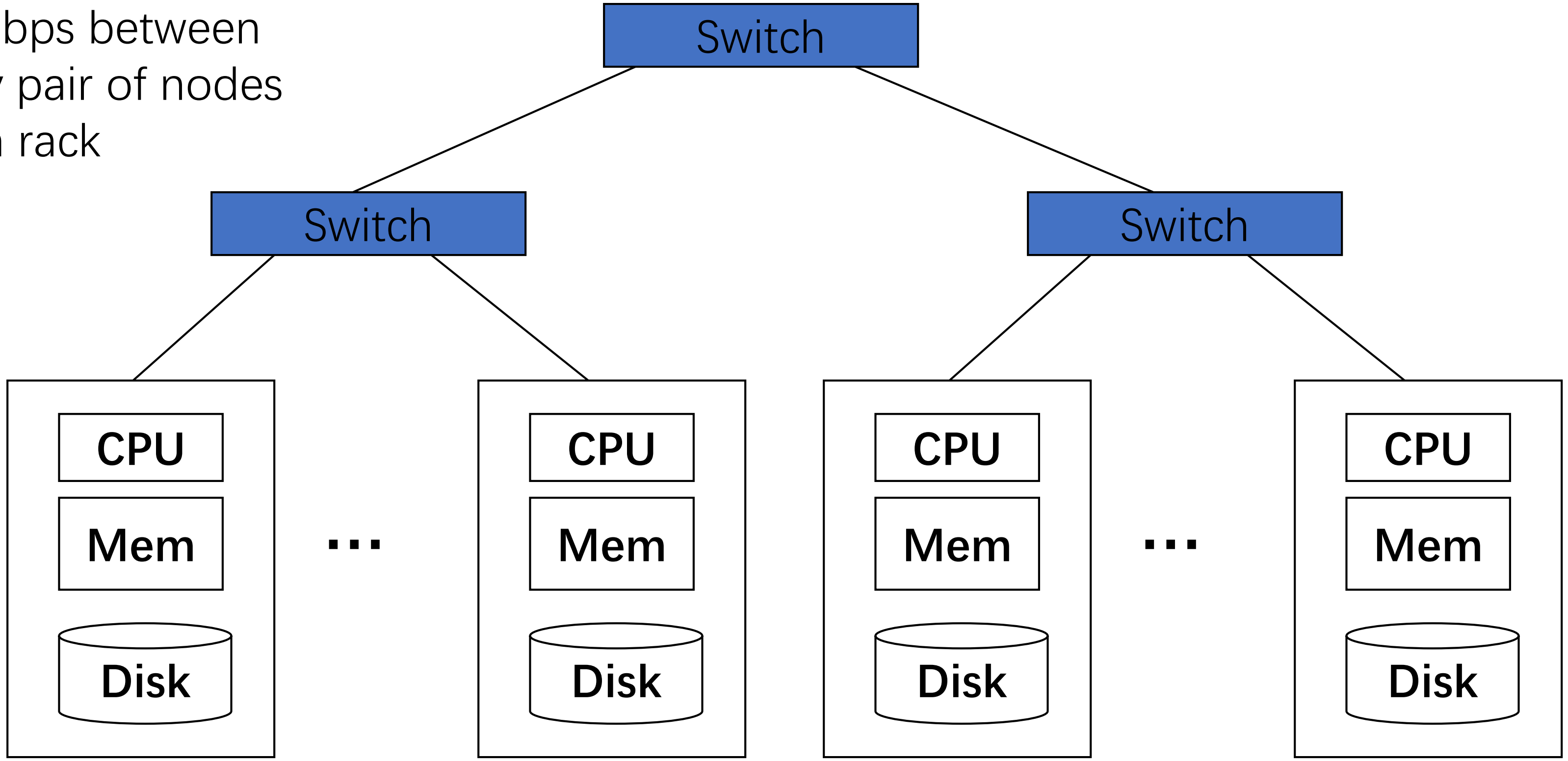
Google Example

- 50+ billion web pages x 20KB = 1000+ TB
  - 1 computer reads 300 MB/sec from disk, ~1 months to read the web
  - ~1,000 hard drives to store the web
- **Solving such problems with a standard architecture :**
  - **Cluster** of commodity Linux nodes
  - **Commodity network** to connect them

Cluster Architecture

2-10 Gbps backbone between racks

1 Gbps between
any pair of nodes
in a rack

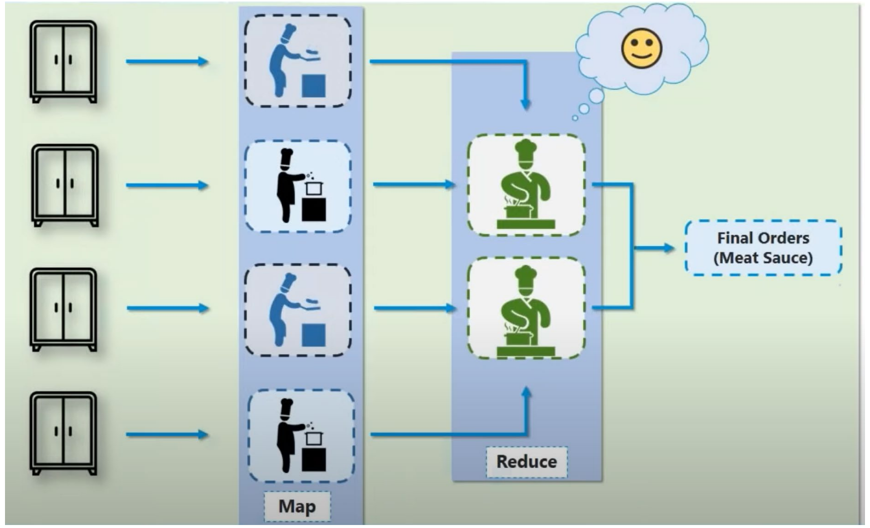


Each rack contains 16-64 nodes

In 2019 it was guestimated that Google had 2.5M machines

Large-scale Computing Challenges

- **Large-scale computing** on **commodity hardware**
- **Challenges:**
  - **Latency issues:**
    - Copying data over a network takes time
  - **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 loose 1/day
    - People estimated Google had ~2.5 M machines in 2019
      - 2,500 machines fail every day!



Adapted from J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org
Part of pictures are from Edureka.co

Solutions

- **Idea:**
  - Latency: bring computation close to the data
  - Computation: a new computation and programming framework
  - Machine failure: store files multiple times for reliability

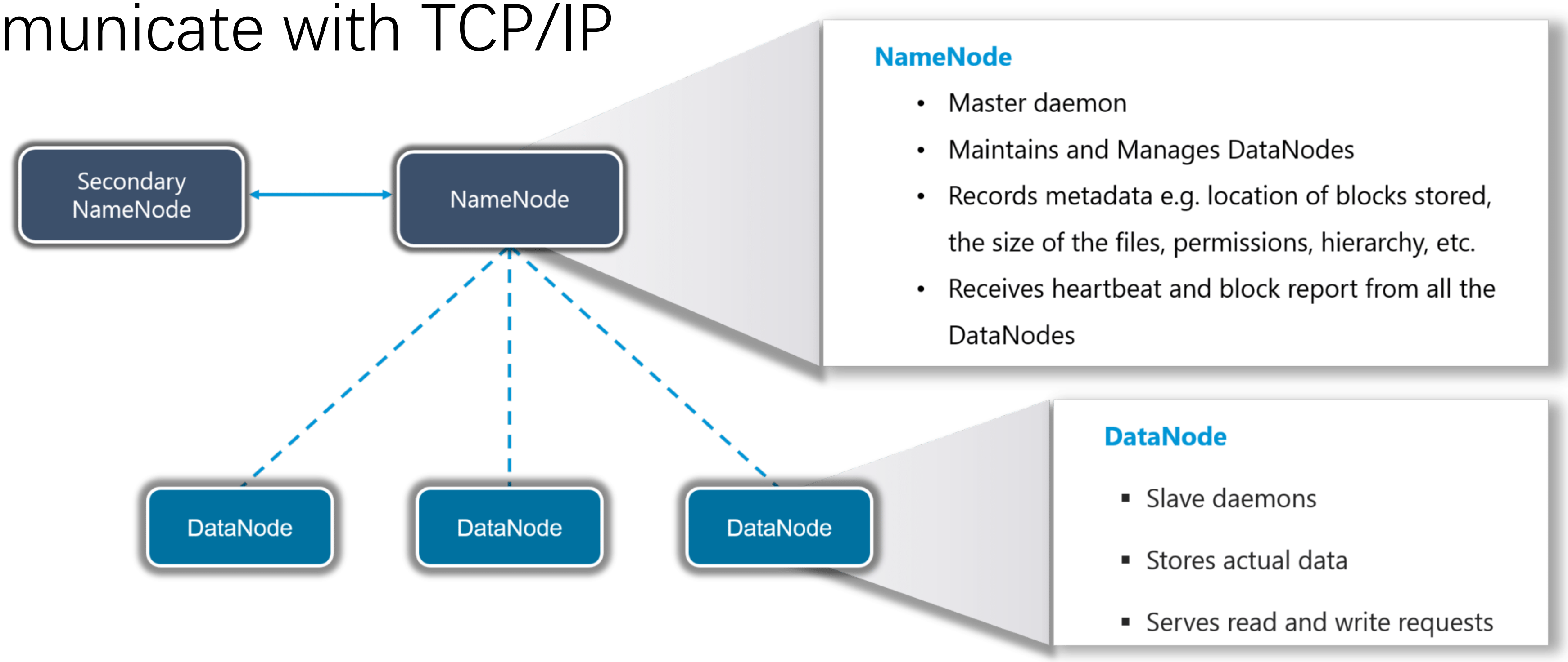
- **Solutions**
  - **Storage: File system**
    - Google: GFS. Hadoop: HDFS
  - **Computing: Programming model**
    - MapReduce: Google and Hadoop

Storage

- **Distributed File System:**
  - Provides global file namespace
  - Google GFS; Hadoop Distributed File System (HDFS);
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - **Write Once – Read Many Philosophy**
    - Data is rarely updated in place
    - Reads and appends are common

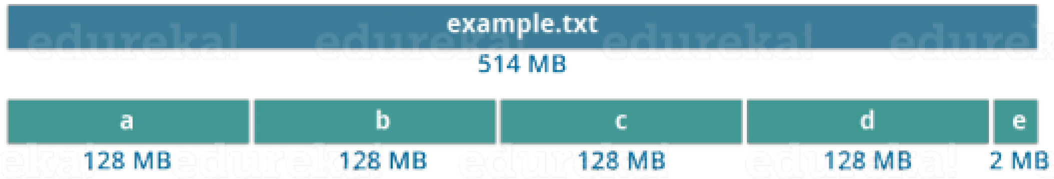
Hadoop Distributed File System

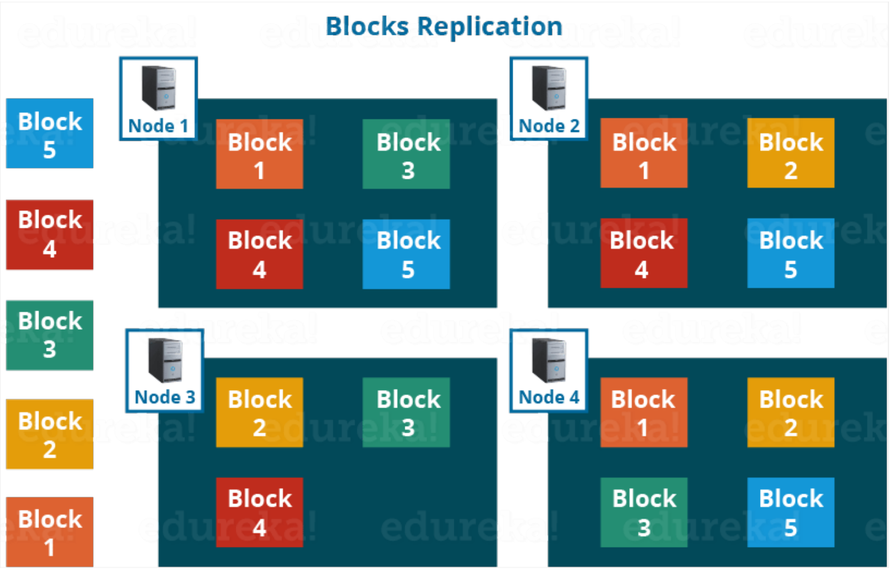
- Name Node
- Data Node
- Communicate with TCP/IP



Hadoop Distributed File System

- **Blocks**
  - HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster. The default size of each block is 128 MB (Compared to Linux 4KB).
- **Replication** management to recovery failures.
  - How many replicas are needed?
  - How to store replicas?
- **Bring computation to data.**



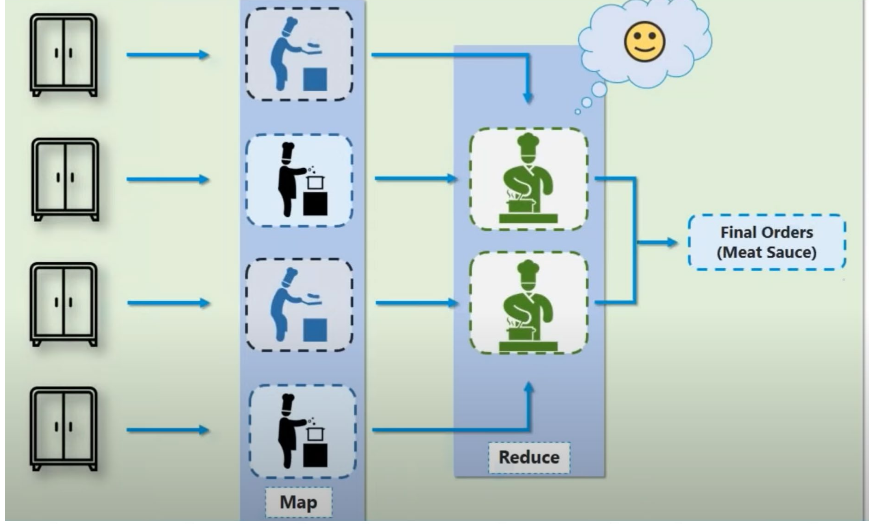


Programming Model: MapReduce

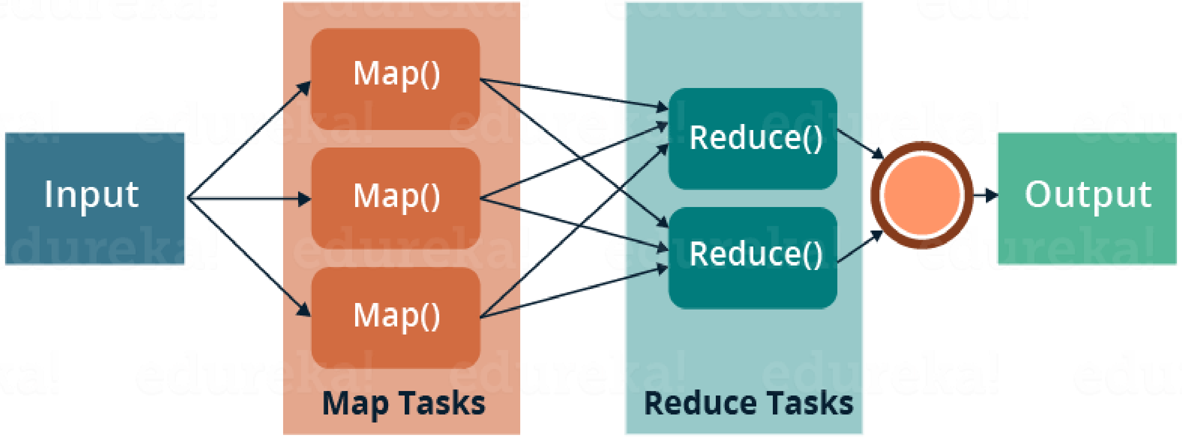
- **MapReduce** is a style of programming design for
  - **Easy** parallel **programming**
  - **Invisible** management of **hardware and software failures**
  - **Easy** management of **large-scale data**

- Implementations
  - Google MapReduce
  - Hadoop
  - Spark (improved)

MapReduce: Overview



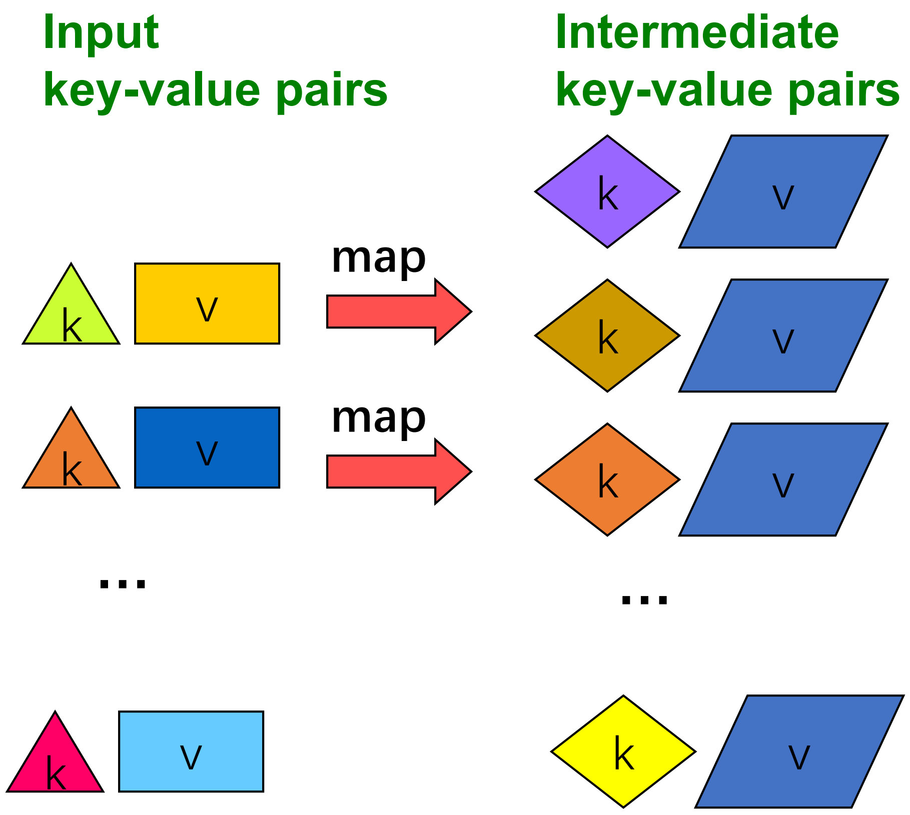
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- Group by key:
  - Sort and shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result to disks



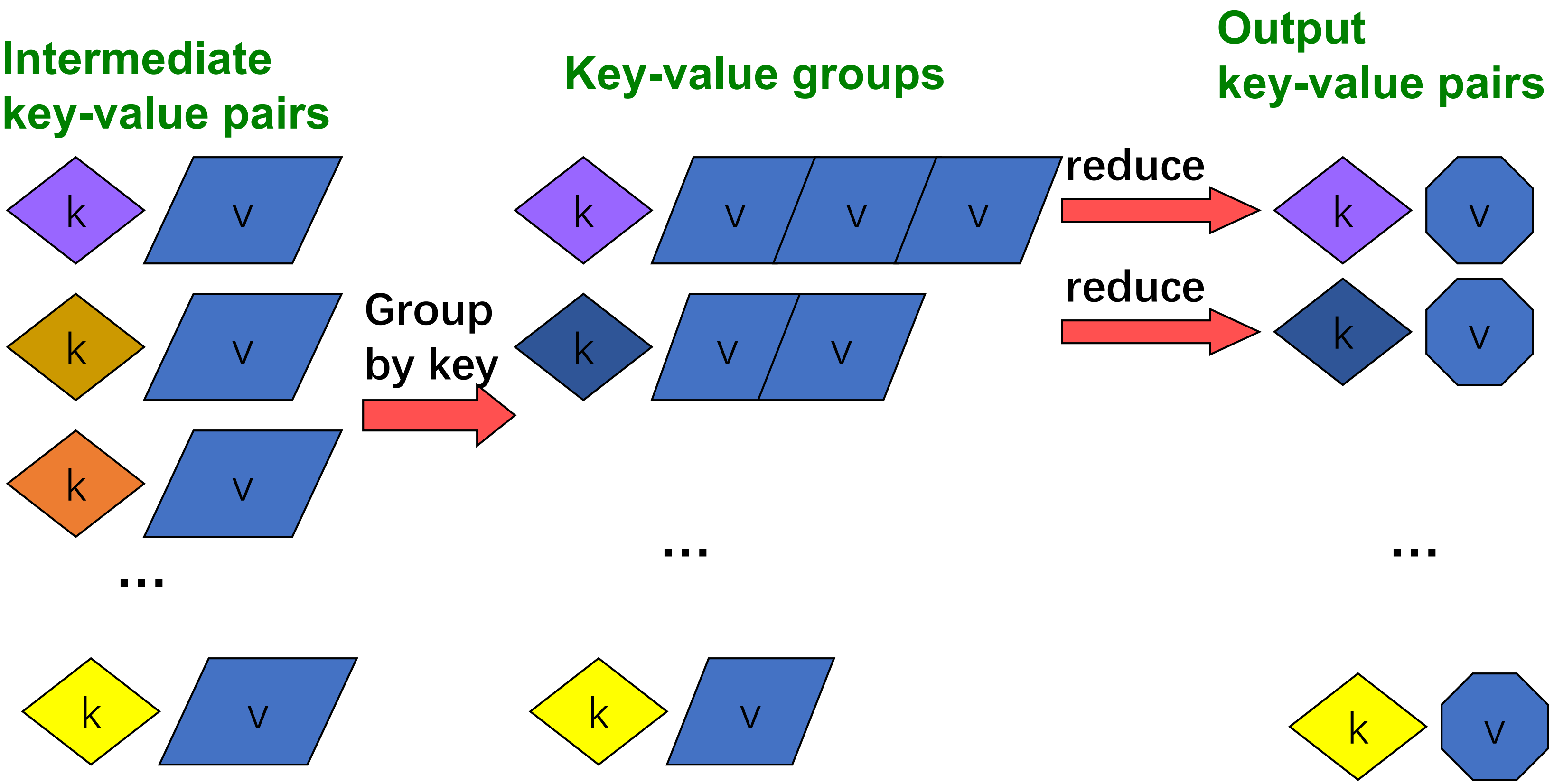
Outline stays the same, **Map** and **Reduce** change to fit the problem.

<Key,value> pair abstraction of data input

MapReduce: The Map Step



MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** → <k’, v’>*
    - **Takes a key-value pair and outputs a set of key-value pairs**
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every (k,v) pair
  - **Reduce(k’, <v’>*)** → <k’, v’’>*
    - **All values v’ with same key k’ are reduced together and processed in v’ order**
    - There is one Reduce function call per unique key k’

MapReduce: Word Counting

- We have huge text document
- Count the number of times each distinct word appears in a file

Provided by the programmer

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

Group by key:
Collect all pairs with same key

Provided by the programmer

Reduce:
Collect all values belonging to the key and output

Sequentially collect to get the result

Big document	(key, value)	(key, value)	(key, value)
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. '"The work we're doing now -- the robotics we're doing -- is what we're going to need ……………………..	(The, 1) (crew, 1) (of, 1) (the, 1) (space, 1) (shuttle, 1) (Endeavor, 1) (recently, 1) ….	(crew, 1) (crew, 1) (space, 1) (the, 1) (the, 1) (the, 1) (shuttle, 1) (recently, 1) …	(crew, 2) (space, 1) (the, 3) (shuttle, 1) (recently, 1) …

Word Count Using MapReduce

Your programs:

Provided by the programmer

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

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
…

(key, value)

```
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)
```

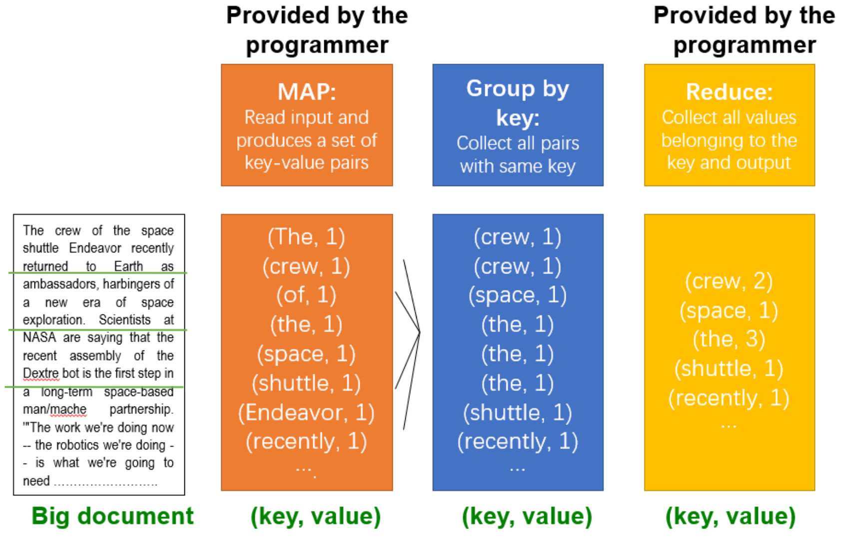
Provided by the programmer

Reduce:
Collect all values belonging to the key and output

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
…

(key, value)

Map-Reduce: Environment



Map-Reduce environment takes care of:

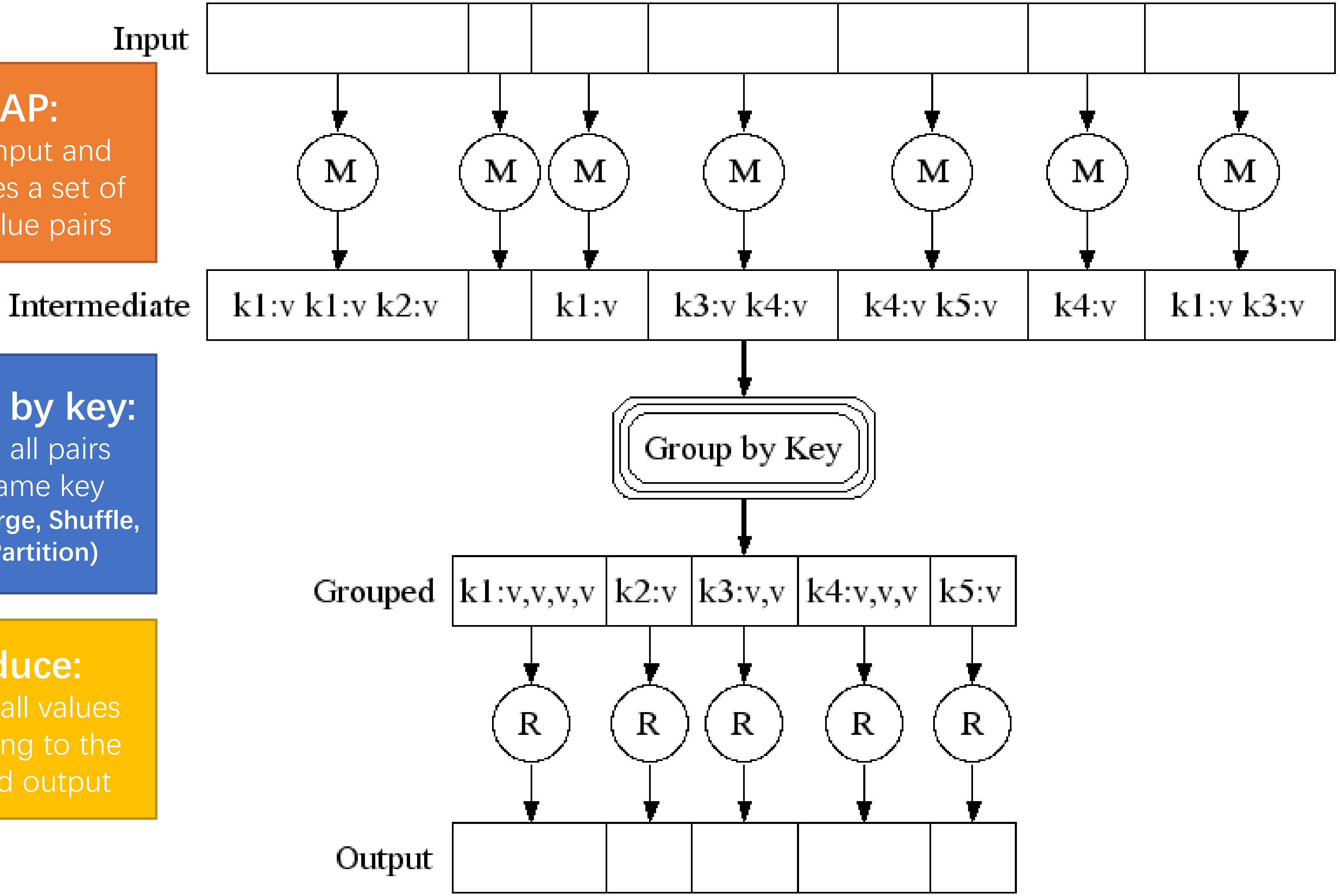
- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
- **Refine** tasks by intermediate combiners
- Handling machine **failures**
- Managing required inter-machine **communication**

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



Example: Join By Map-Reduce

- **Compute the natural join** $R(A,B) \bowtie S(B,C)$
- R and S are each stored in files
- Tuples are pairs (a,b) or (b,c), join R and S by the same b, and output the results of (a,c)

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

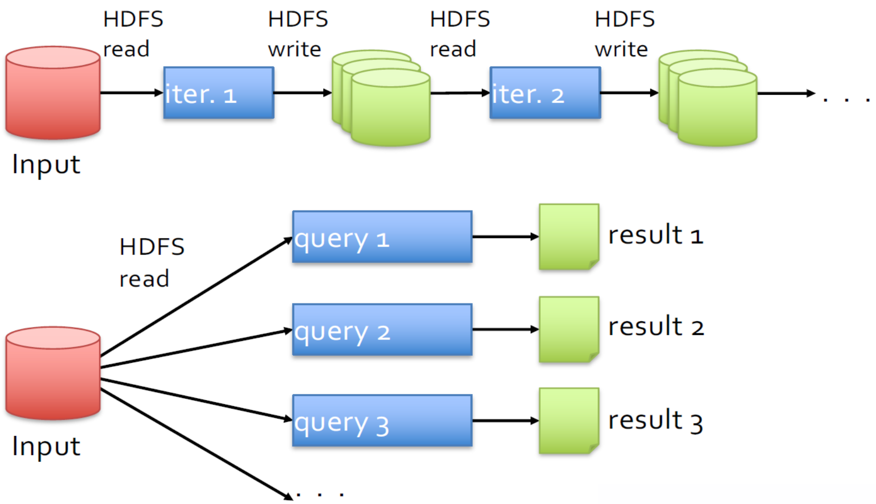
Join by MapReduce

- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*

- Group by keys:
  - Use a hash function *h* from B-values to *1...k*, Map processes send each key-value pair with key *b* to Reduce process *h(b)*
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,c)*.

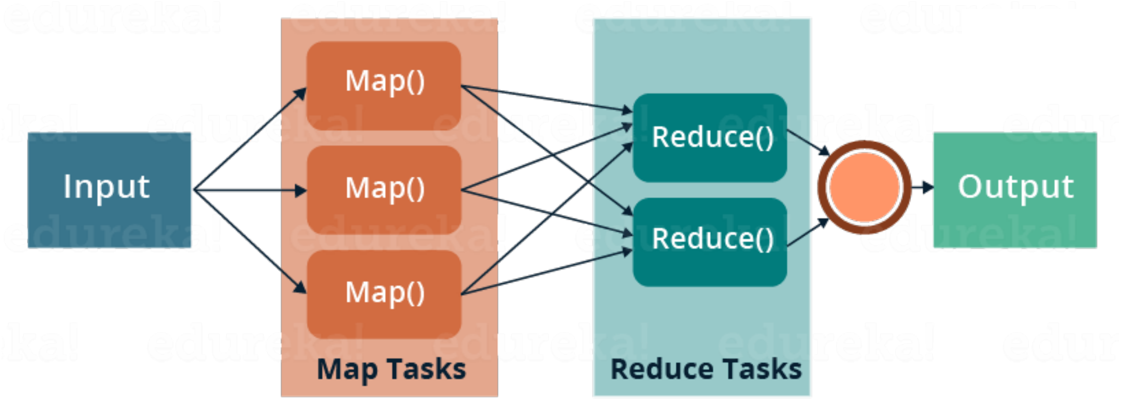
Problems with MapReduce

- Hadoop MapReduce is **inefficient** for applications that repeatedly reuse a working set of data:
  - **Iterative** algorithms (machine learning, graphs): incurs substantial **overheads** due to data replication, disk I/O
  - **Interactive** data mining tools： all Java codes; R, Python not supported



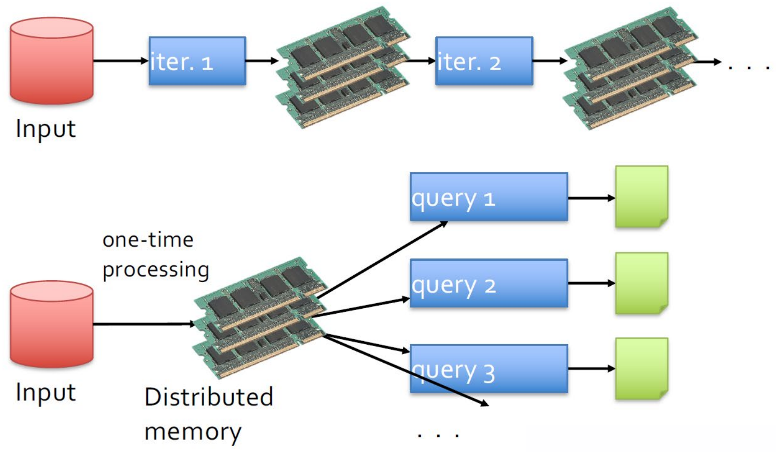
Problems with MapReduce

- **Data flow** is not flexible enough
  - MapReduce uses only two types of tasks: Map and Reduce; data flows are always from Map to Reduce.



Solution: Spark

- Allow apps to keep working sets in **memory** for efficient reuse
- Retain the attractive properties of MapReduce
  - Fault tolerance, data locality, scalability
- Additions to MapReduce model:
  - Richer functions than just map and reduce
  - Better data flow scheduler



Spark Overview

- Spark is a **unified analytics** engine for large-scale **data processing**.
- **100x Faster**
  - **RDD**: resilient distributed datasets(弹性分布式数据集), core building block.
  - **DAG**: directed acyclic graph(有向无环图), general execution graph scheduler.
- **Ease of use**
  - Spark provides **data focused API** which makes writing large-scale programs easy, such as DataFrames & DataSets
  - Compatible with Scala, Java, R, Python

Core Concept: RDD

Resilient distributed datasets (RDDs): Primary abstraction

- **Immutable**, **partitioned** collections of objects
  - Generalized key-value pairs
- Caching in **memory**

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

- There are currently two types:
  - *parallelized collections*

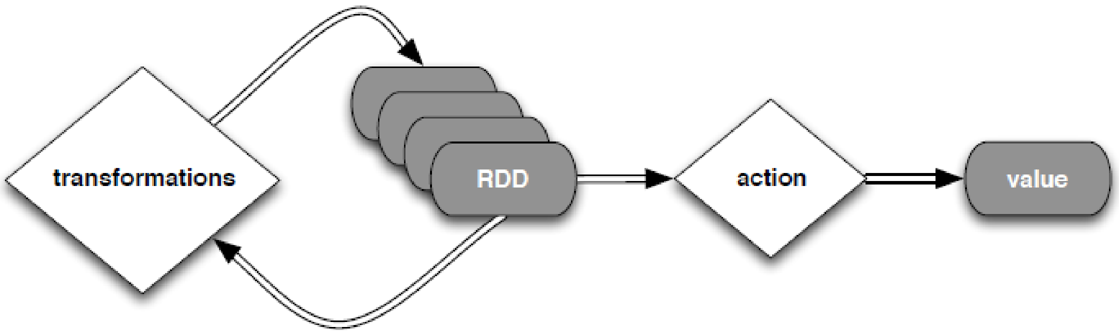
    – take an existing collection and run functions on it in parallel

  - *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop

```
>>> distFile = sc.textFile("README.md")
```

Spark RDD Operations

- **Operations** on RDDs:
  - **Transformations:** build RDDs from other RDDs
    - Transformations create a new RDD from an existing one
    - Transformations are **lazy**: nothing computed until an action requires it.
    - map, filter, groupBy, join, union, intersection, …
  - **Actions:** get results
    - A transformed RDD gets recomputed when an action is run on it
    - reduce, count, collect, save, …



RDD Transformation

- Transform a file into RDD

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

README.md:
Spark
is easy

map:
[[“Spark”, “is”],
[“easy”]]

flatmap:
[“spark”, “is”, “easy]

transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement*, *fraction*, *seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

RDD Action

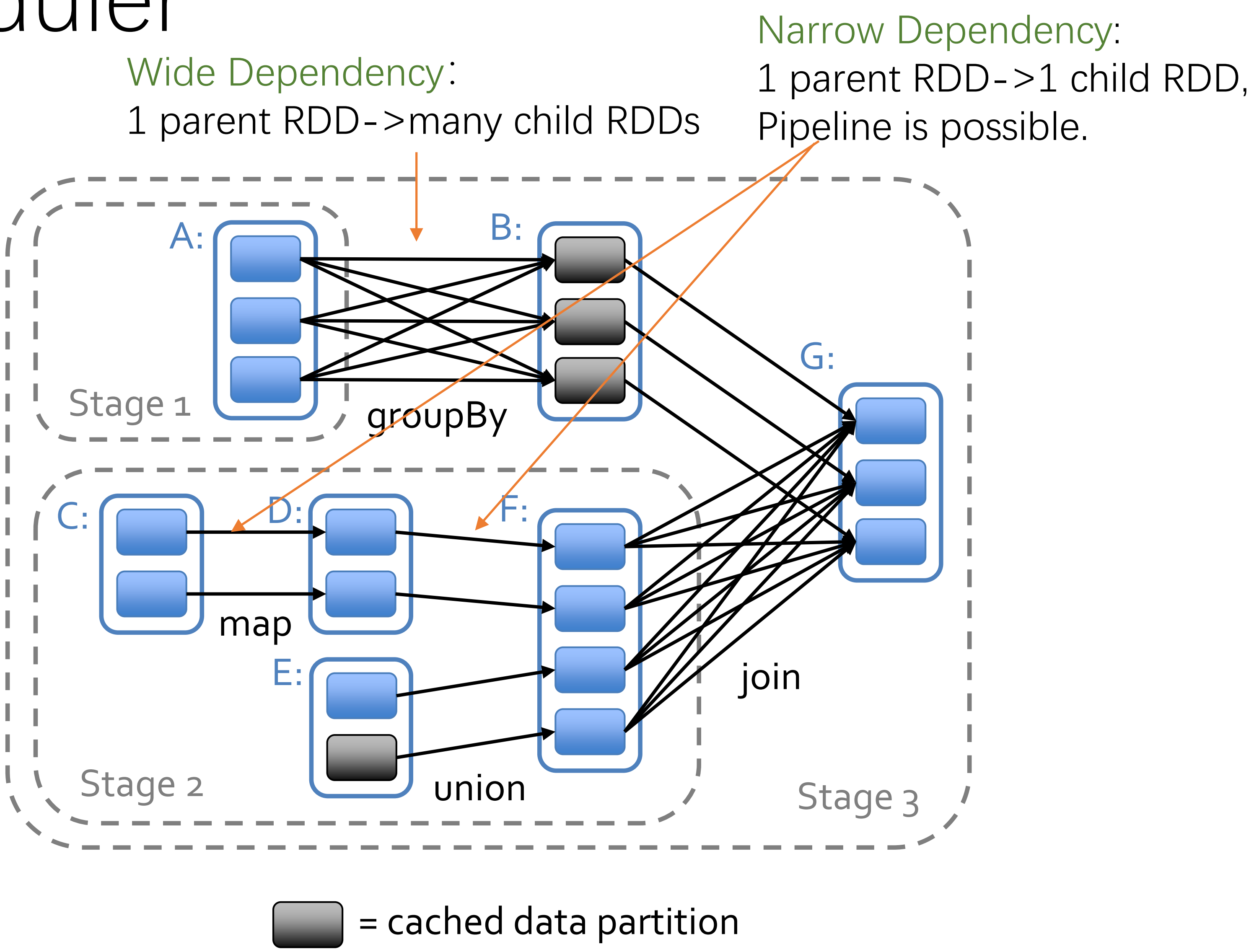
action	*description*
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

- Word Count

```
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

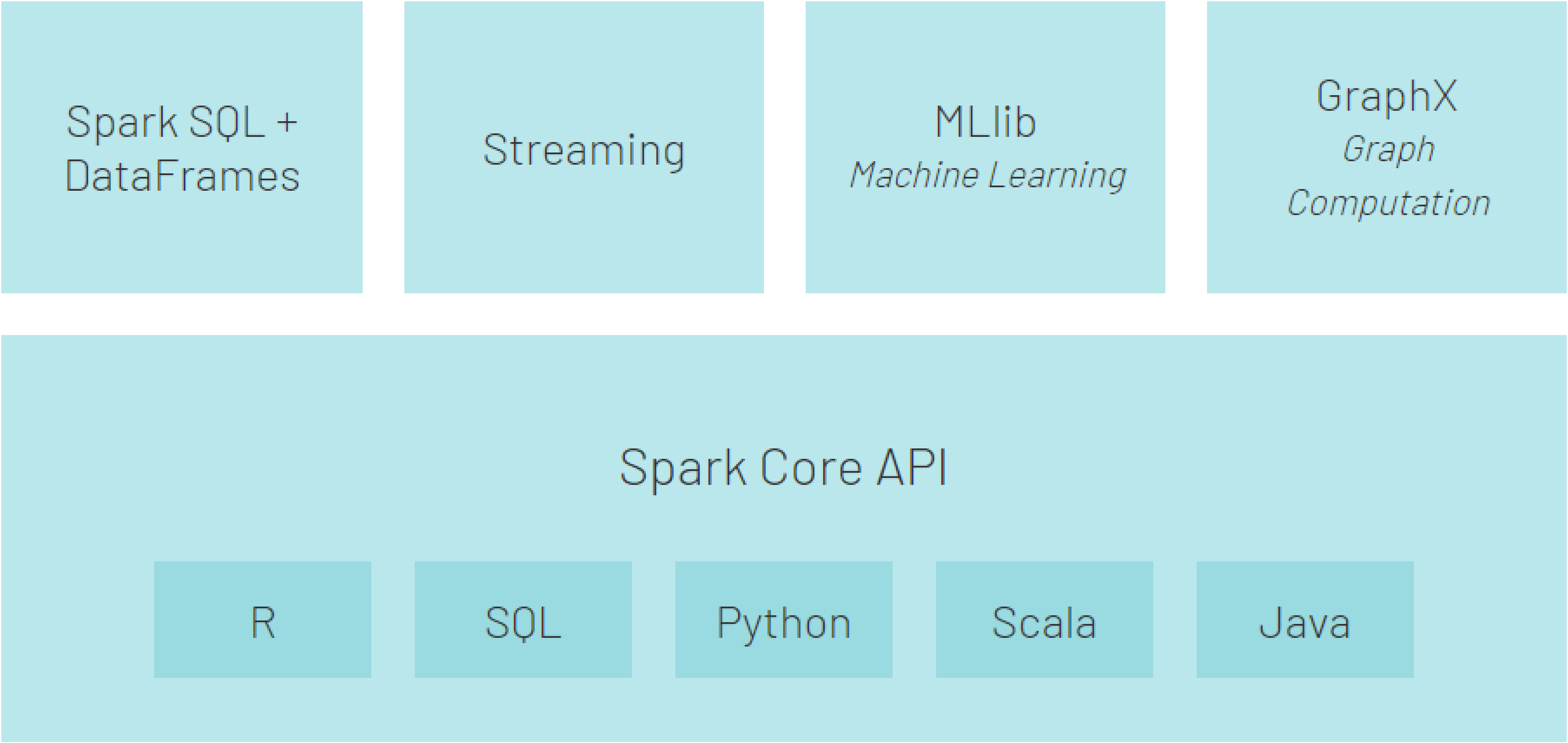
Spark DAG Scheduler

- Supports general task **graph** scheduling
- Pipelines functions within a **stage**
  - Narrow vs Wide dependency
  - Divide into stages where there is a wide dependency (can not use pipeline)
- **Cache-aware** work reuse & locality



Spark ecosystem

- Useful libraries



Summary

Big data processing:

- MapReduce: distributed programming/computing framework
    - HDFS
    - Map and Reduce
    - System handles all other processes
    - Save results to file systems
- Spark: improved over MapReduce
    - RDD: distributed in memory
    - DAG scheduling
    - Programming friendly
- Reading "Mining of Massive Datasets", Chapter 2.

Spark Programming

Spark Programming Essentials

- Basic concepts in https://spark.apache.org/docs/latest/rdd-programming-guide.html
- After installing spark
  - ./bin/pyspark for Python
  - You can configure on your jupyter notebook

Spark Essentials: SparkContext

- First thing that a Spark program does is create a **SparkContext** object, which tells Spark how to access a cluster
  - In the shell for Python, this is the **sc** variable, which is created automatically
  - Other programs must use a constructor to instantiate a new **SparkContext. SparkContext** gets used to create other variables

Python:

```
>>> sc
<pyspark.context.SparkContext object at 0x7f7570783350>
```

Spark Essentials: Master

- The master parameter for a SparkContext determines which cluster to use

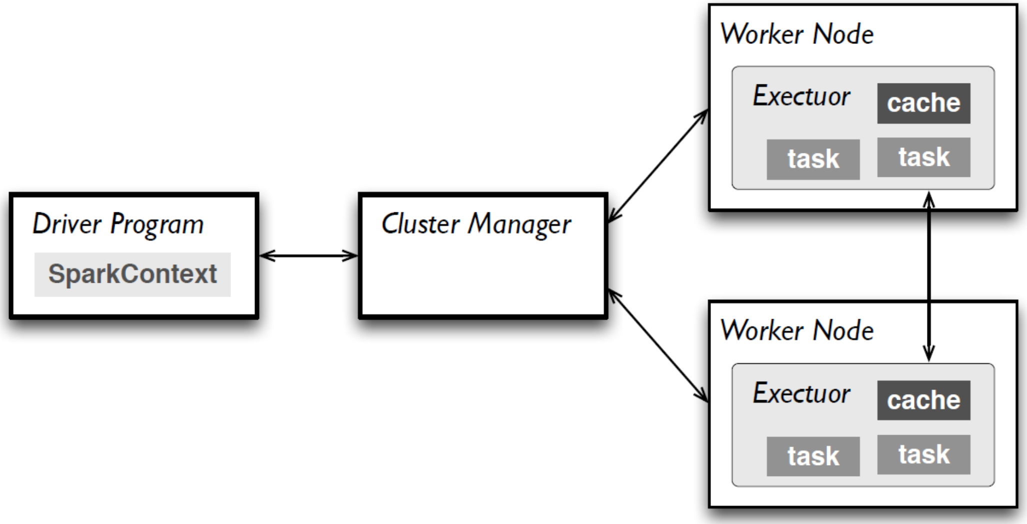
Python:

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

master	*description*
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)

Spark Essentials: Master

- Connects to a cluster manager which allocate resources across applications
- Acquires executors on cluster nodes – worker processes to run computations and store data
- Sends app code to the executors
- Sends tasks for the executors



RDD

Create RDDs from a collection

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

Create RDDs from a document

```
>>> distFile = sc.textFile("README.md")
```

RDD Transformation

- All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset
  - optimize the required calculations
  - recover from lost data partitions

transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement*, *fraction*, *seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset*)	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

RDD Transformation

- Transform a file into RDD

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

README.md:
Spark
is easy

map:
[[“Spark”, “is”],
[“easy”]]

flatmap:
[“spark”, “is”, “easy]

RDD Action

A transformed RDD gets recomputed when an action is run on it

action	*description*
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

action	*description*
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a \`Map\` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD Action

- Word Count

```
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

RDD Persistence

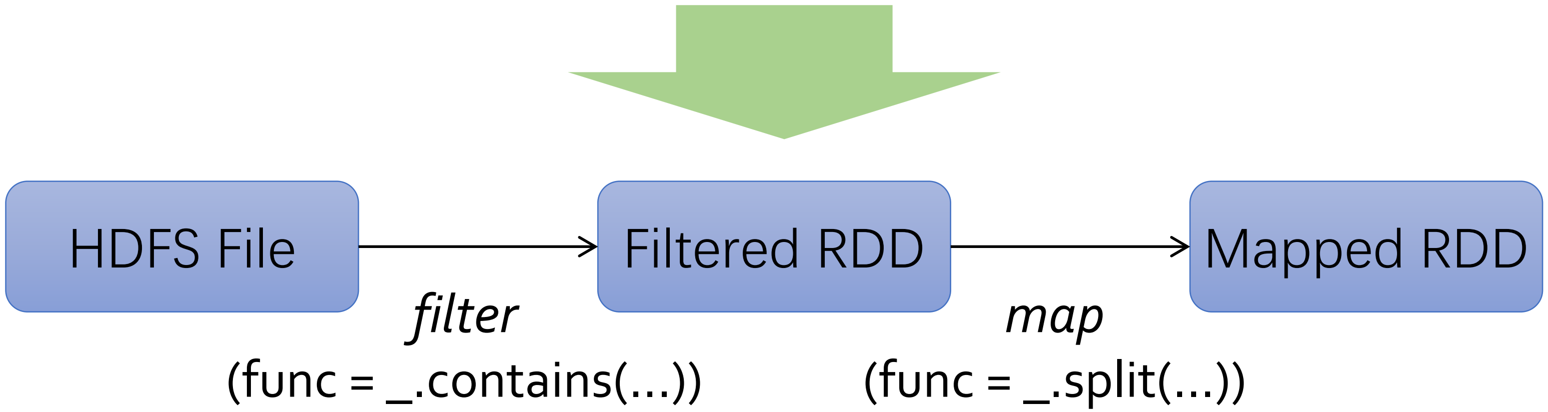
- Spark can *persist* (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

RDD Fault Tolerance

- The cache is *fault-tolerant*
  - if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it
  - RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Example:
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```

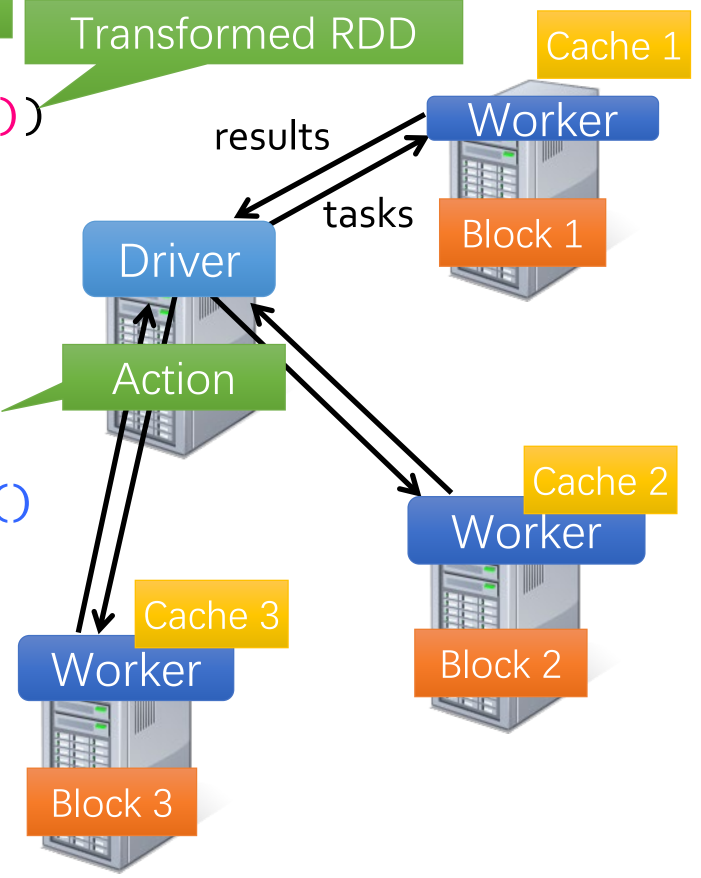


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter (lambda s:s.startsWith("ERROR"))
messages = errors.map (lambda s:s.split(' ')[2])
cachedMsgs = messages.cache()


cachedMsgs.filter (lambda s:.contains("foo")). count()
cachedMsgs.filter  (lambda s:s.contains("bar")).count()
          . . .
```



Summary

- MapReduce
- Spark
  - RDD
    - Transformation
    - Action
  - DAG scheduler
  - Libraries