Big Data Analytics
Lecture 3
MapReduce

Yilu Zhou, PhD
Associate Professor
Gabelli School of Business
Fordham University

* Some slides are adopted from Professor Kunpeng Zhang's Big Data course @UMD

What we will cover

- The Idea of Map Reduce
    - Required components
        - Input, Mapper, Reducer, Driver, and Output
    - Optional components
        - Combiner and Partitioner
    - Customize components
        - Input / output
        - Data type
    - Chaining jobs

- Examples
    - Single-source shortest path for a large graph

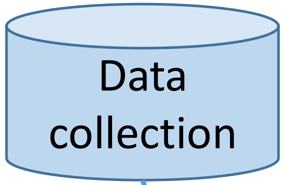
The Idea of MapReduce

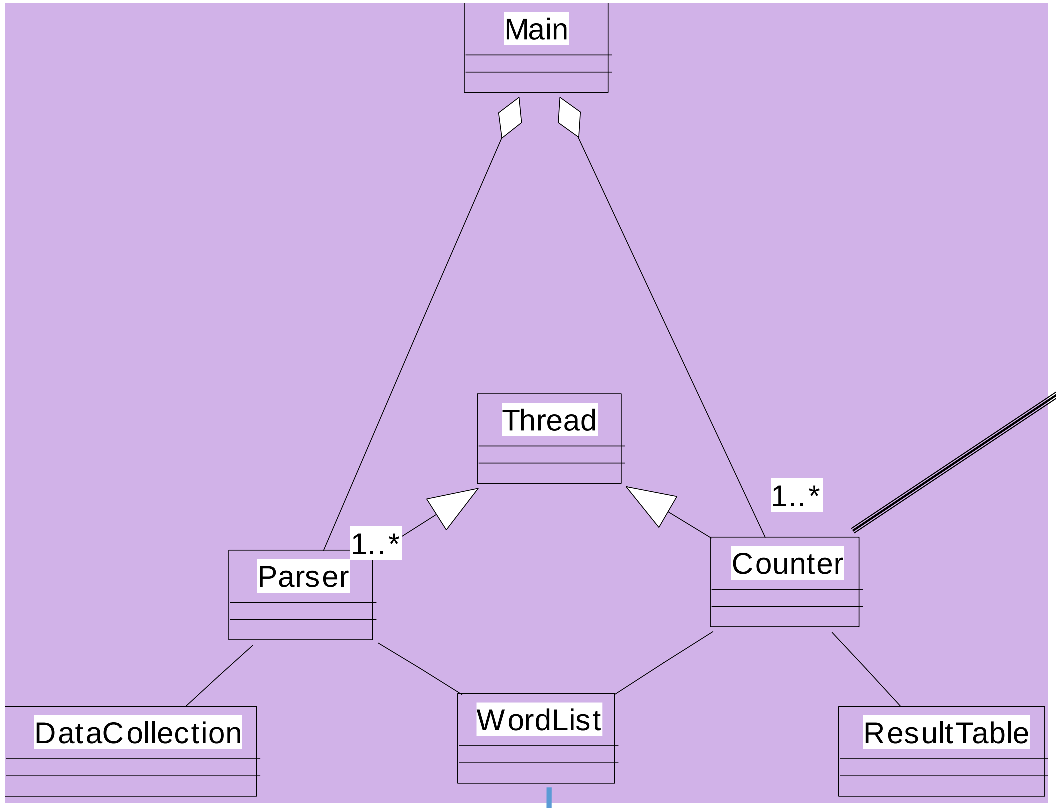
MapReduce – Data Processing

- MapReduce is a data processing job which splits the input data into independent chunks, which are then processed by the map function and then reduced by grouping similar sets of the data.

- Using Hadoop, the MapReduce framework can allow code to be executed on multiple servers — called nodes from now on — without having to worry about single machine performance. Nodes can be grouped into clusters, dispersing processing and memory constraints, for faster access to datasets.

Improve Word Counter for Performance





No need for lock

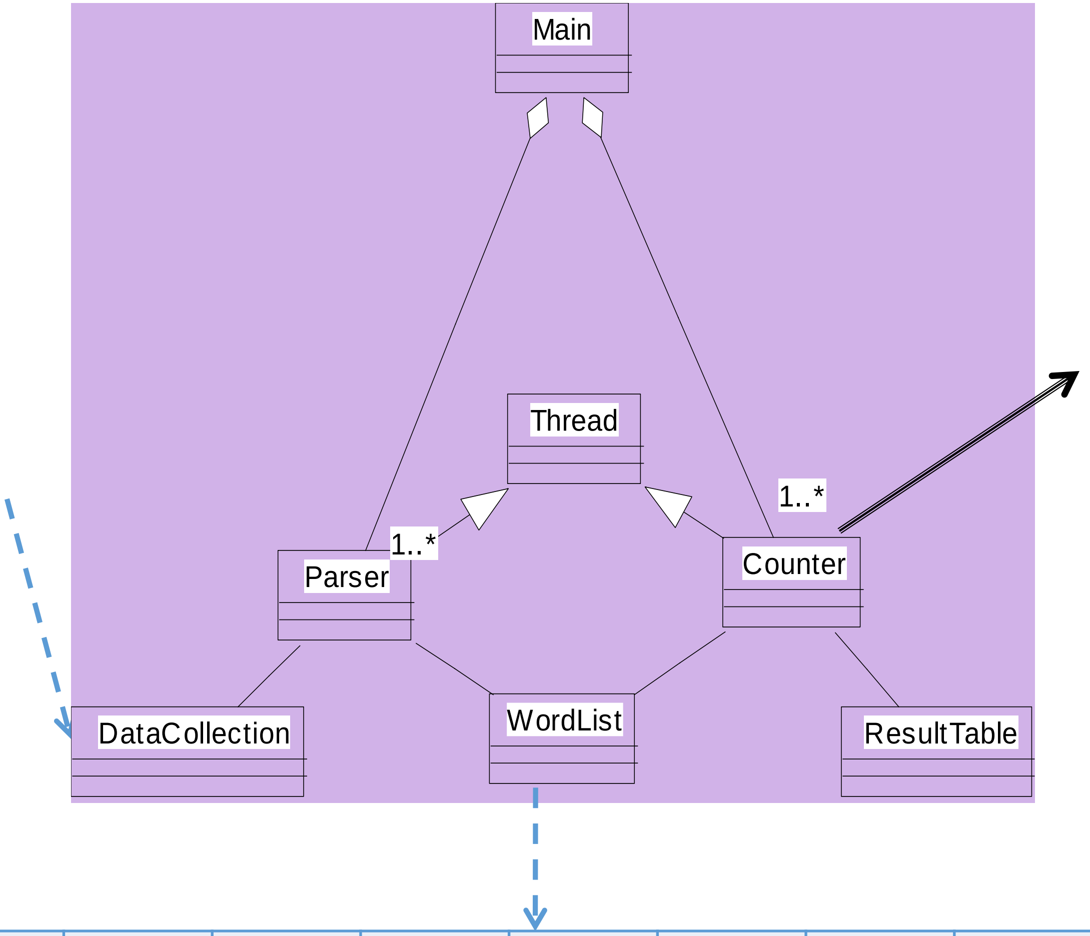
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Separate counters

KEY	**web**	**weed**	**green**	**sun**	**moon**	**land**	**part**	**web**	**green**	**…….**
VALUE										

The following slides are adopted from CCSCNE 2009 Palttsburg

Peta-scale Data



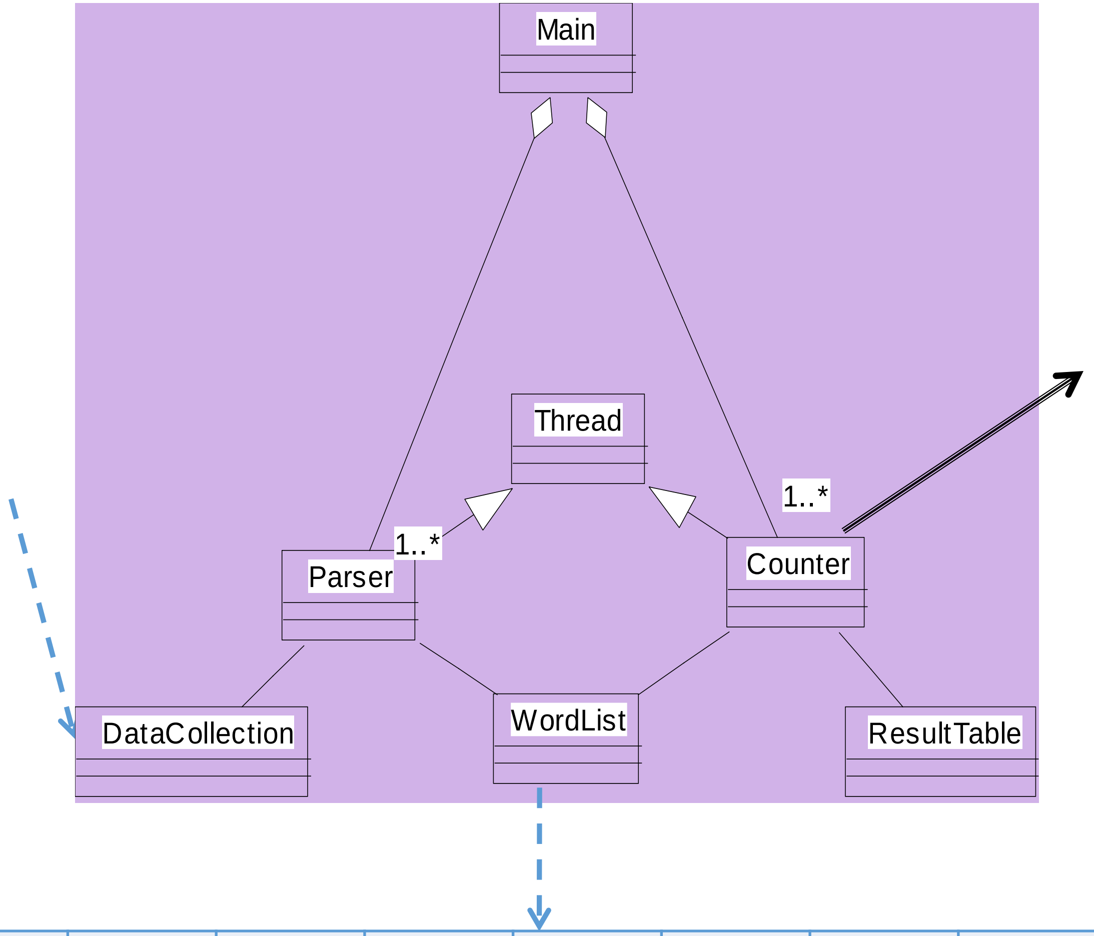
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

KEY	**web**	**weed**	**green**	**sun**	**moon**	**land**	**part**	**web**	**green**	**…….**
VALUE										

Addressing the Scale Issue

- Single machine cannot serve all the data: you need a distributed special (file) system
- Large number of commodity hardware disks: say, 1000 disks 1TB each
    - Issue: With Mean time between failures (MTBF) or failure rate of 1/1000, then at least 1 of the above 1000 disks would be down at a given time.
    - Thus failure is norm and not an exception.
    - File system has to be fault-tolerant: replication, checksum
    - Data transfer bandwidth is critical (location of data)
- Critical aspects: fault tolerance + replication + load balancing, monitoring
- Exploit parallelism afforded by splitting parsing and counting
- Provision and locate computing at data locations

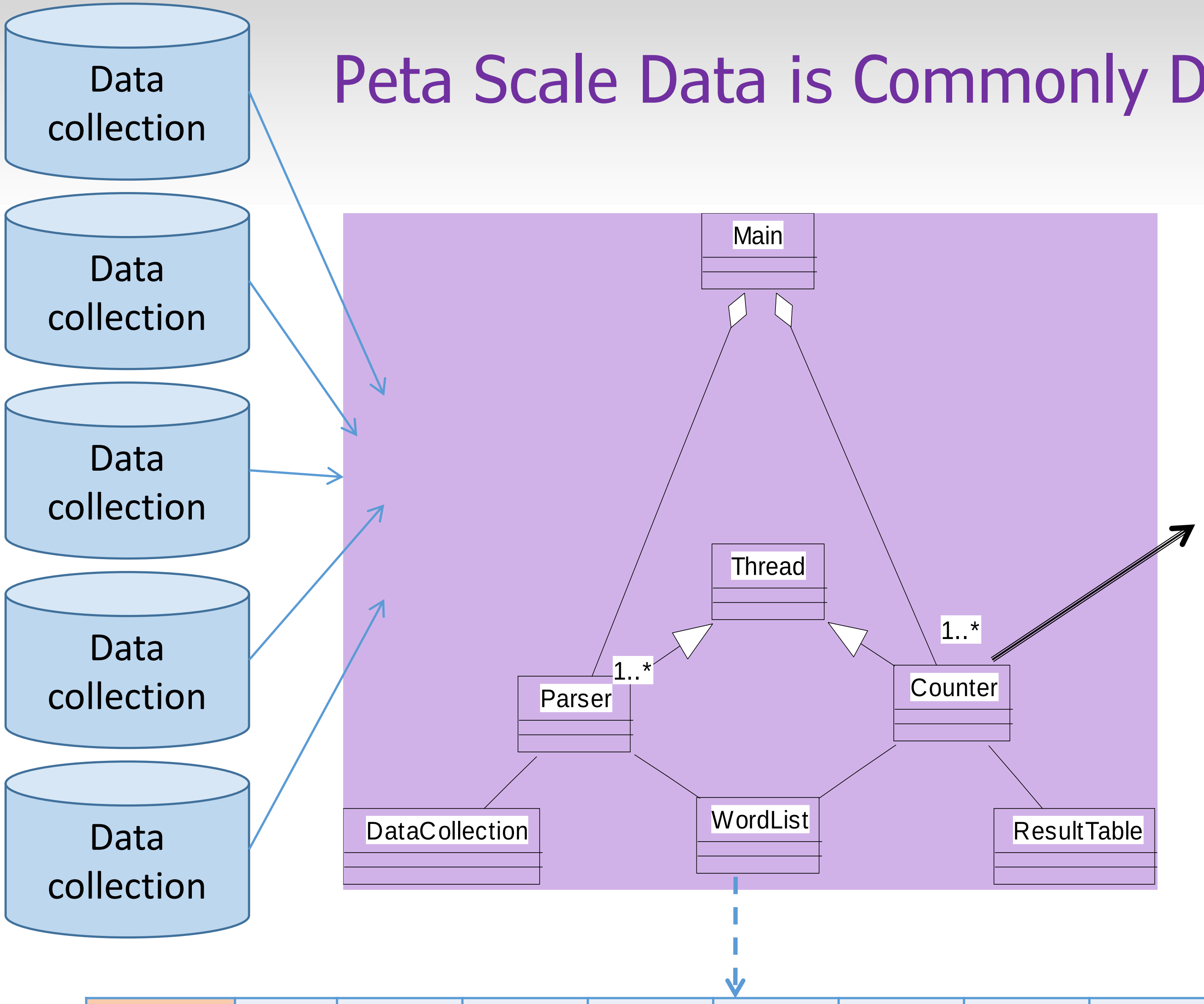
Peta-scale Data



web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

KEY	**web**	**weed**	**green**	**sun**	**moon**	**land**	**part**	**web**	**green**	**.......**
VALUE										

Peta Scale Data is Commonly Distributed (Hadoop)

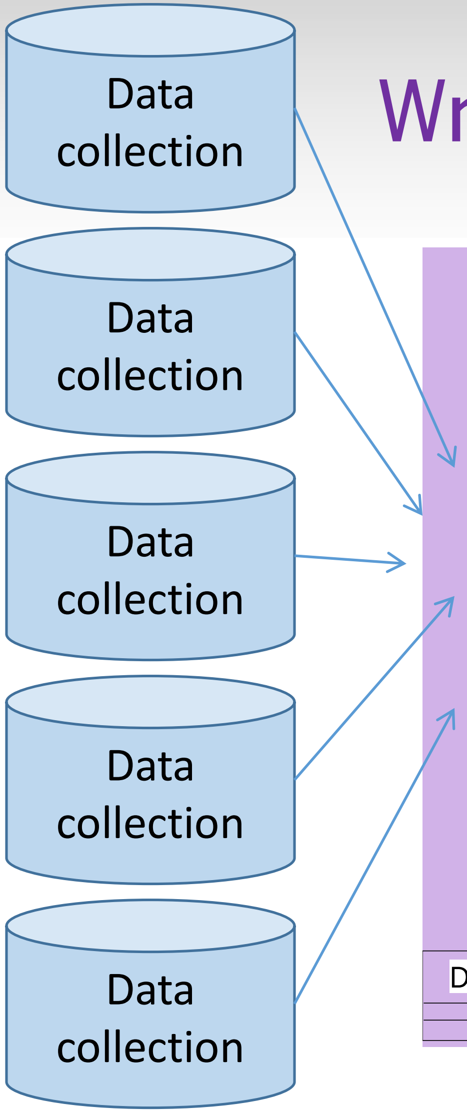


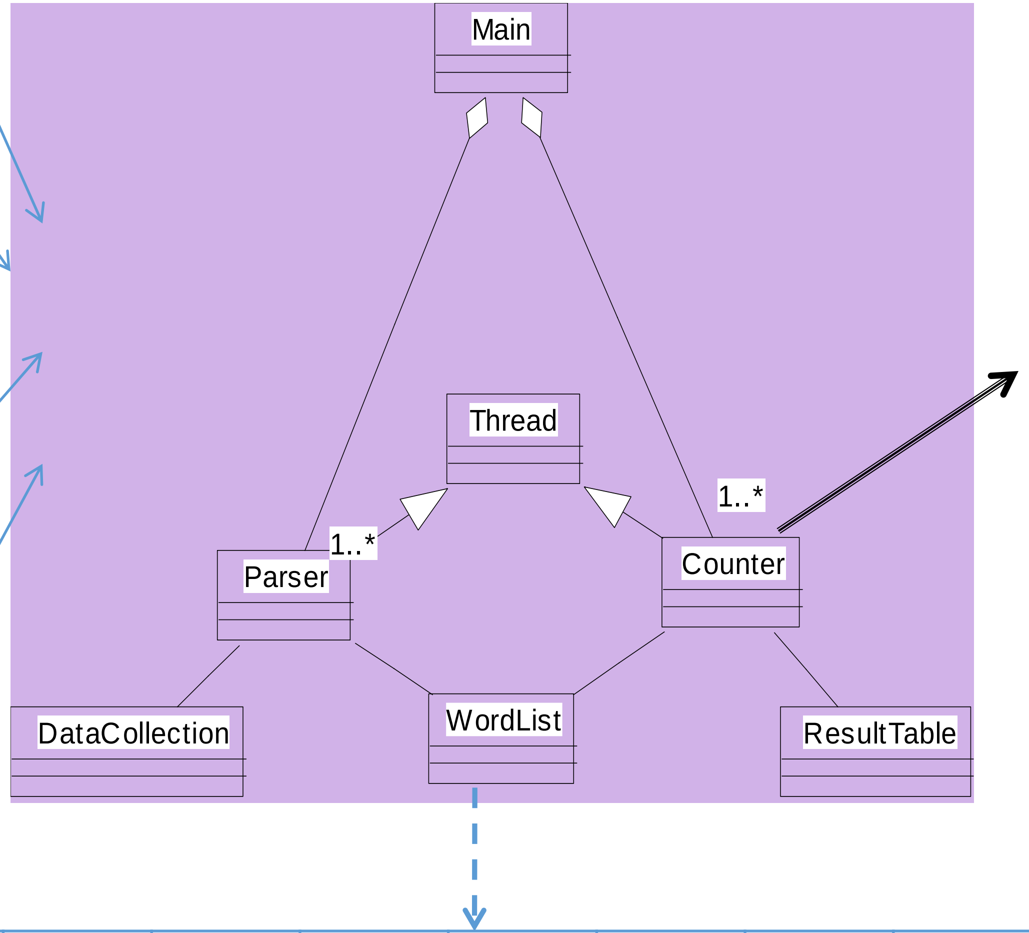
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Issue: managing the large scale data

KEY	web	weed	green	sun	moon	land	part	web	green	…….
VALUE										

Write Once Read Many (WORM) data

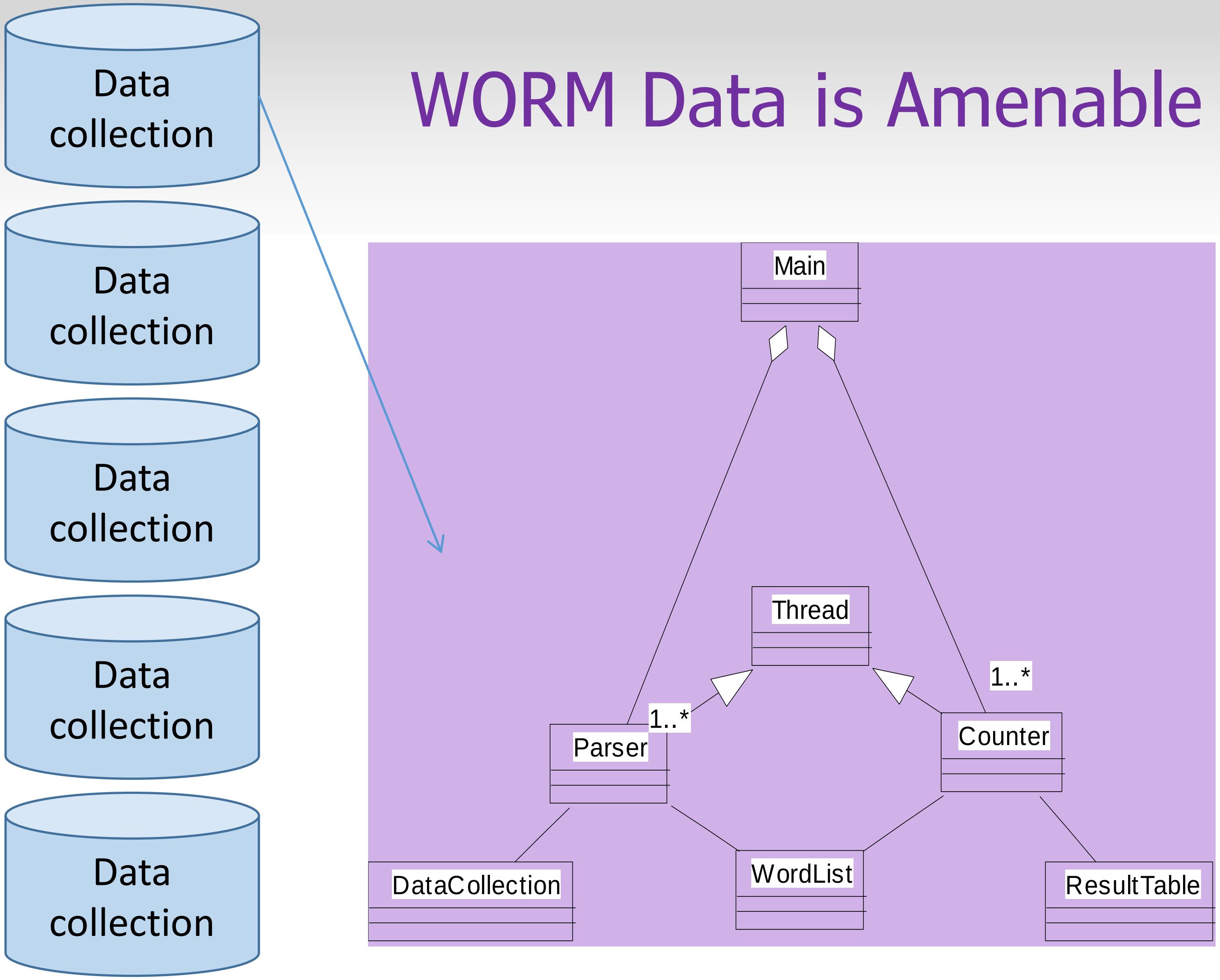




web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

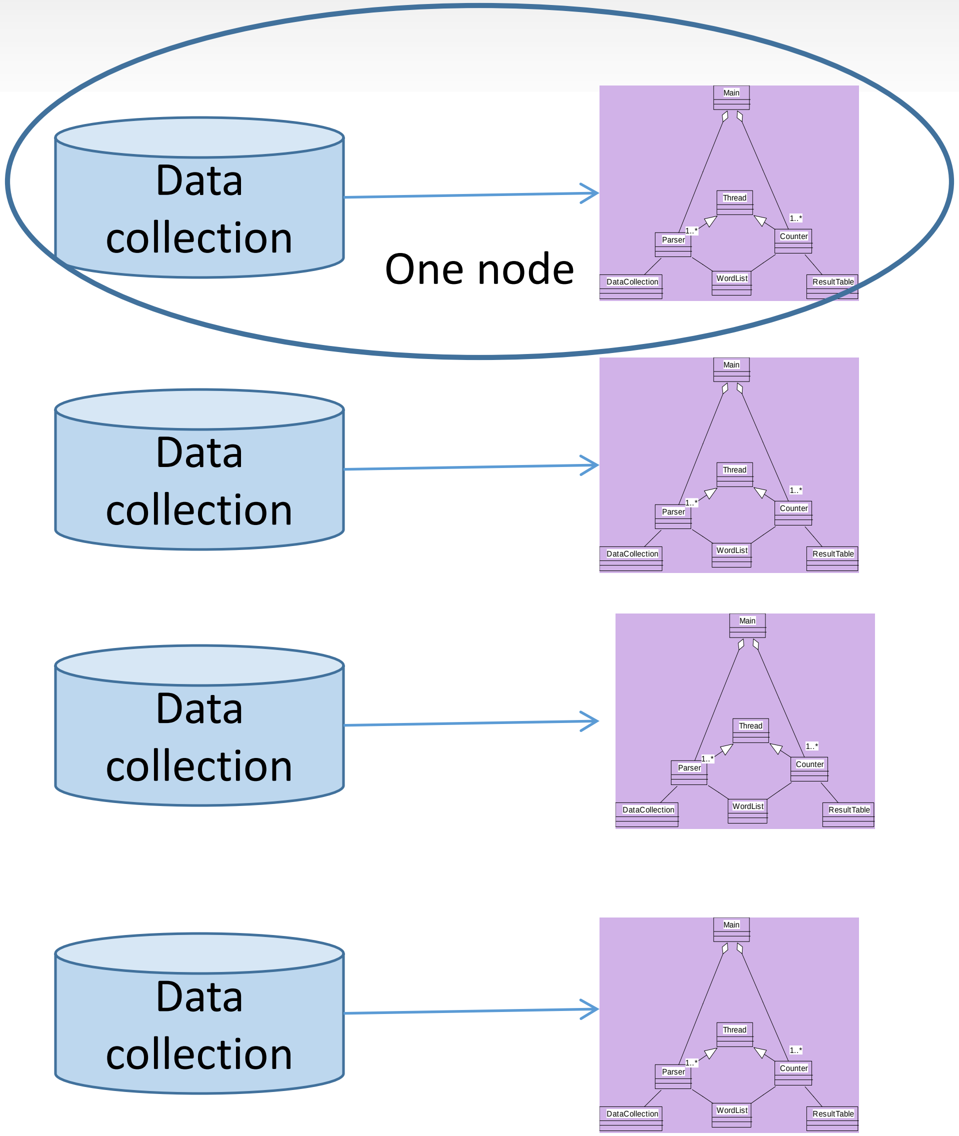
KEY	**web**	**weed**	**green**	**sun**	**moon**	**land**	**part**	**web**	**green**	**.......**
VALUE										

WORM Data is Amenable to Parallelism



1. Data with WORM characteristics : yields to parallel processing;
2. Data without dependencies: yields to out of order processing

Divide and Conquer: Provision Computing at Data Location



For our example,
#1: Schedule parallel parse tasks
#2: Schedule parallel count tasks

This is a particular solution;
Lets generalize it:

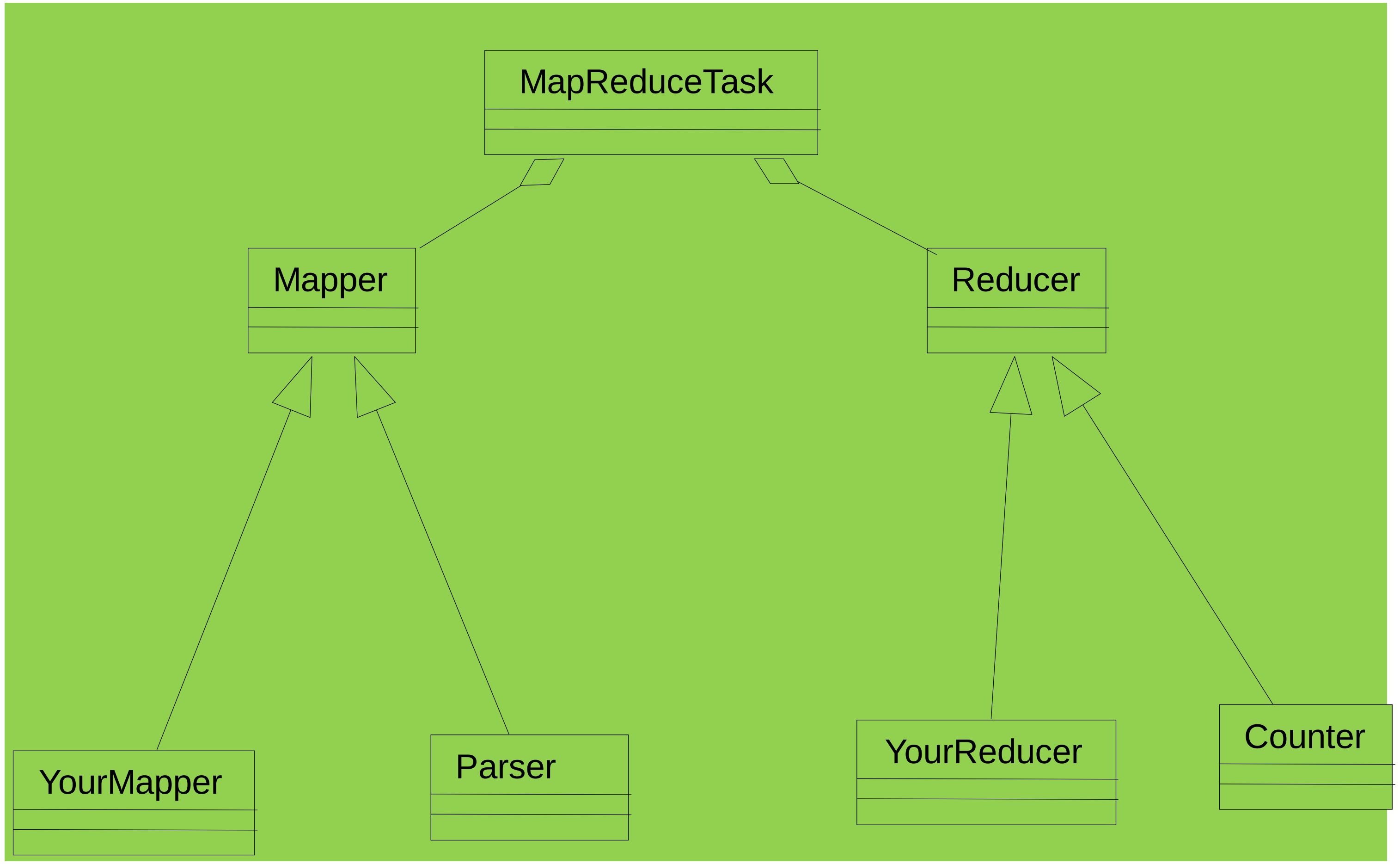
Our parse is a mapping operation:
MAP: input → <key, value> pairs

Our count is a reduce operation:
REDUCE: <key, value> pairs reduced

Map/Reduce originated from Lisp
But have different meaning here

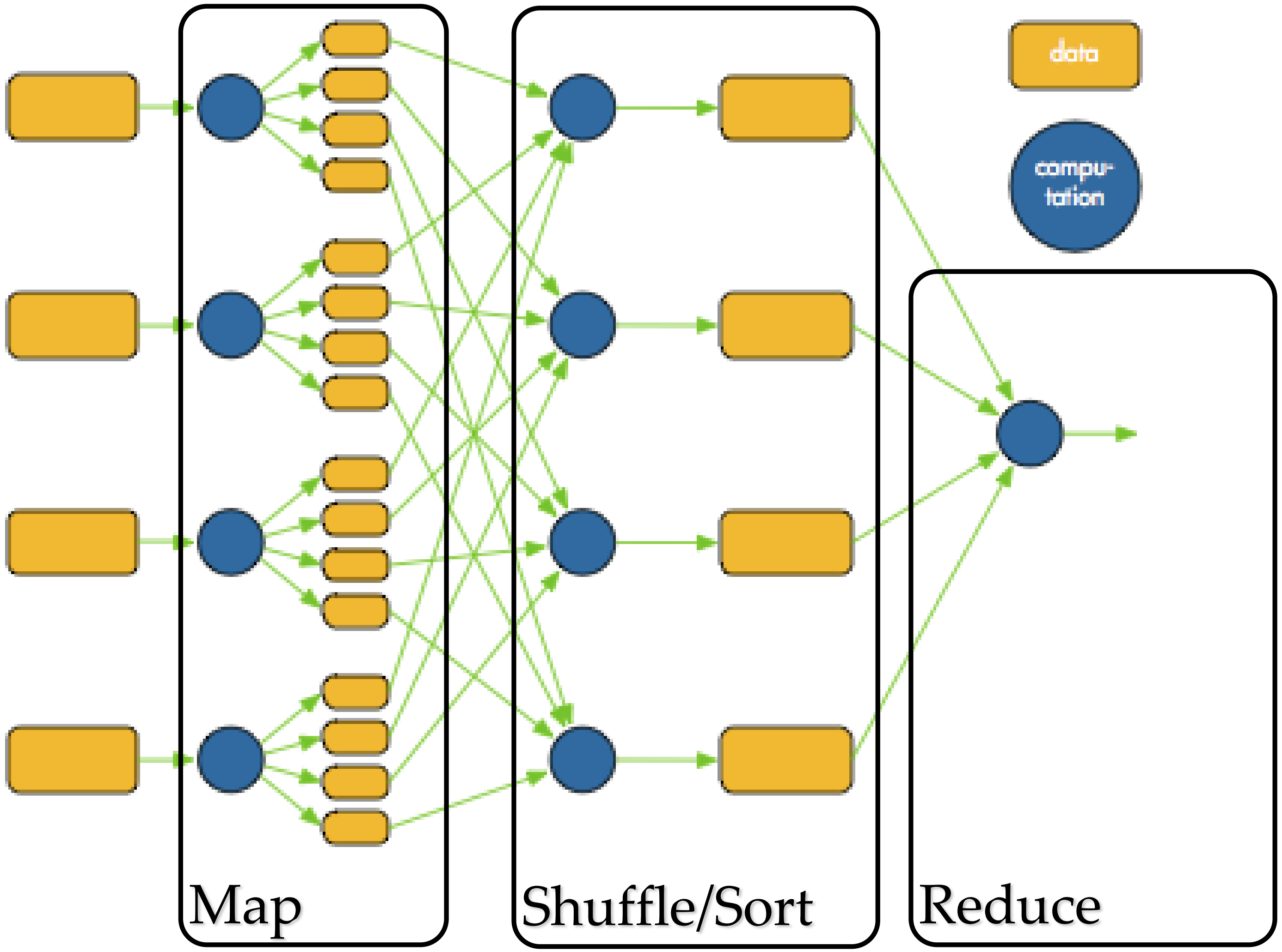
Runtime adds distribution + fault tolerance + replication + monitoring + load balancing to your base application!

Mapper and Reducer



Remember: MapReduce is simplified processing for larger data sets:
MapReduce Version of [WordCount Source code]

Mapreduce overview



Mapreduce: slow motion

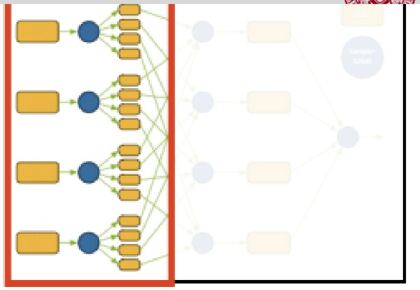
- The canonical mapreduce example is word count
- Example corpus:

  Joe likes toast

  Jane likes toast with jam

  Joe burnt the toast

MR: slow motion: Map



key is the line number

key is word

Input

Joe likes toast
Map 1

Jane likes toast with jam
Map 2

Joe burnt the toast
Map 3

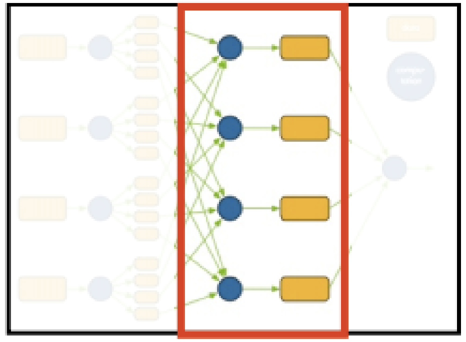
Output

Joe	1
likes	1
toast	1

Jane	1
likes	1
toast	1
with	1
jam	1

Joe	1
burnt	1
the	1
toast	1

MR: slow motion: Sort



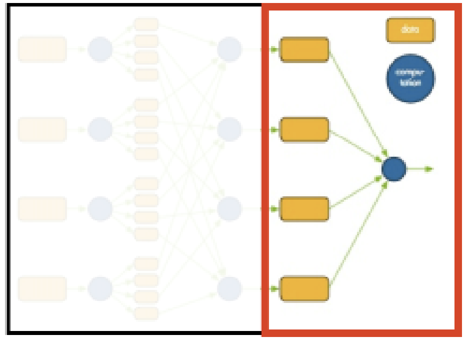
Input

Joe	1
likes	1
toast	1
Jane	1
likes	1
toast	1
with	1
jam	1
Joe	1
burnt	1
the	1
toast	1

Output

Joe	1
Joe	1
Jane	1
likes	1
likes	1
toast	1
toast	1
toast	1
with	1
jam	1
burnt	1
the	1

MR: slow mo: Reduce



Input

Joe	1	Reduce 1
Joe	1	
Jane	1	Reduce 2
likes	1	Reduce 3
likes	1	
toast	1	Reduce 4
toast	1	
toast	1	
with	1	Reduce 5
jam	1	Reduce 6
burnt	1	Reduce 7
the	1	Reduce 8

Output

Joe	2
Jane	1
likes	2
toast	3
with	1
jam	1
burnt	1
the	1

MapReduce in Hadoop

MapReduce in Hadoop

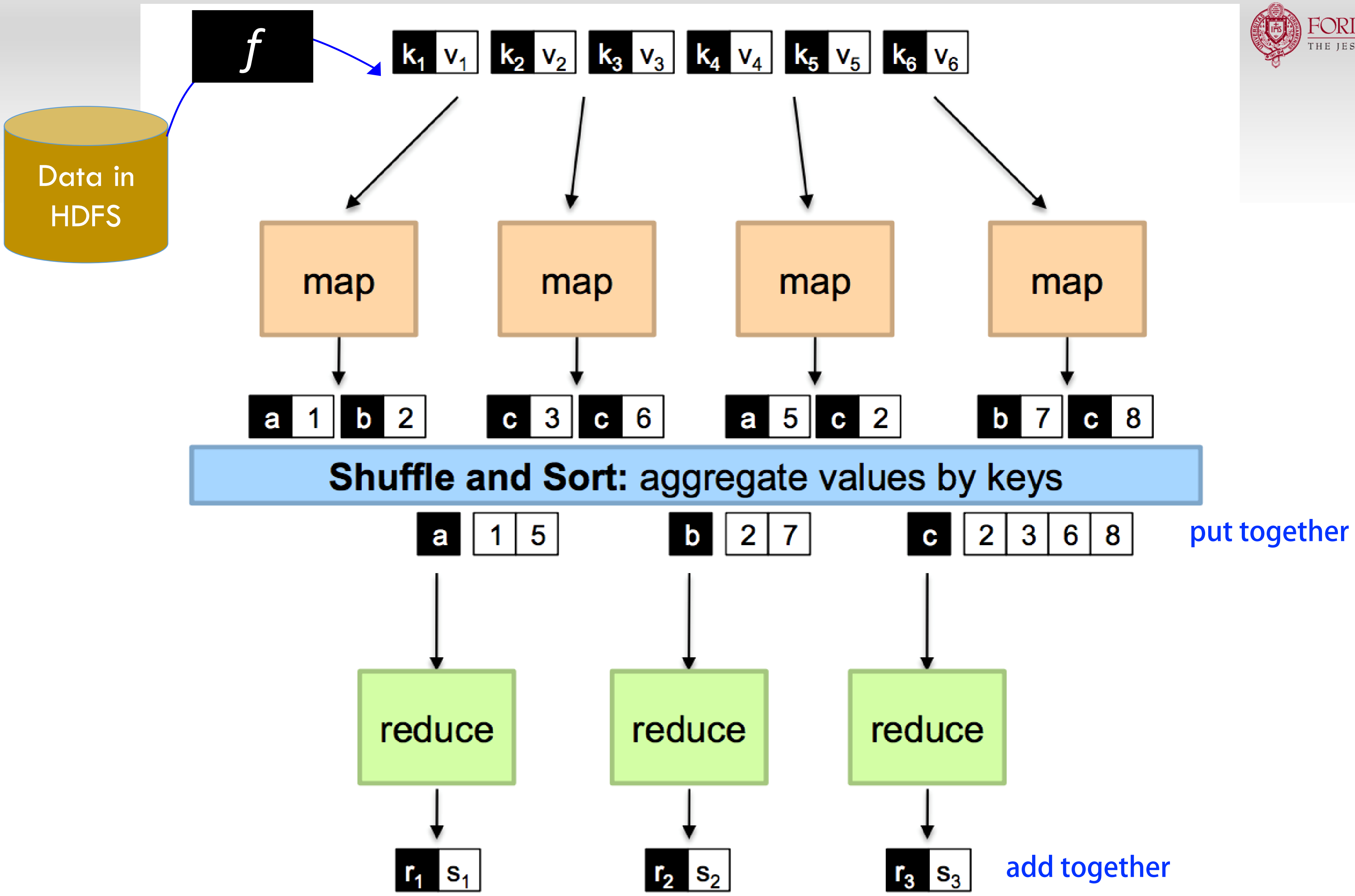
- **Everything** in MapReduce is **<key, value>** pair
- Programmers create/implement three functions/methods:

line number Text

map(key1, value1) → **[**<key2, value2>**]**

reduce(key2, **[**value2**]**) → <key3, value3>

driver() / main()


Data in HDFS
f
k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
put together
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3
add together

(Lin, WWW 2013)

MapReduce: map

- map(key1, value1) → [<key2, value2>]

```
public static class MyMapper extends Mapper<InKey, InVal, OutKey, OutVal>
{
        // other methods, e.g., setup(), cleanup()

        public void map(InKey key, InVal value, Context context){
                // method body…
                …
                context.write(outkey_value, outval_value);
        }

}
```

MapReduce: map

- Input:
  - Key: byte offset of the line
  - Value: the content of the line
  - E.g., “I am taking big data course at IFI summer school 2015. I really enjoy summer time here at IFI.”
- Output:
  - (I, 1), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 1), (IFI, 1), (summer, 1), (school, 1), (2015, 1), (., 1), (I, 1), (really, 1), (enjoy, 1), (summer, 1), (time, 1), (here, 1), (at, 1), (IFI, 1), (., 1)

MapReduce: reduce

- reduce(key2, [value2]) → <key3, value3>

```
public static class MyReducer extends Reducer<InKey, InVal, OutKey, OutVal>
{
        // other methods, e.g., setup(), cleanup()

        public void reduce(InKey key, Iterable<InVal> value, Context context){
                // method body…
                …
                context.write(outkey_value, outval_value);
        }

}
```

MapReduce: reduce

- Input:
  - (I, 1), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 1), (IFI, 1), (summer, 1), (school, 1), (2015, 1), (., 1), (I, 1), (really, 1), (enjoy, 1), (summer, 1), (time, 1), (here, 1), (at, 1), (IFI, 1), (., 1)
- Output:
  - (I, 2), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 2), (IFI, 2), (summer, 2), (school, 1), (2015, 1), (., 2), (really, 1), (enjoy, 1), (time, 1), (here, 1)

MapReduce: driver

- Driver in Hadoop (job configuration)
  - Specify where to load input and where to put output
  - Specify input format and output format
  - Specify mapper, combiner, partitioner, and reducer
  - Specify number of mappers, reducers, etc.
  - …

MapReduce

- Programmers create three functions/methods:

  map(key1, value1) → [<key2, value2>]

  reduce(key2, [value2]) → <key3, value3>

  driver() / main()

- The execution framework in Hadoop handles everything else…

What's "everything else"?

MapReduce “Runtime”

- Handles scheduling
  - Assigns mappers and reducers to do tasks
- Handles data distribution
  - Moves processes to data
- Handles synchronization
  - Collects, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects failures and restarts
- Everything happens on top of a distributed file system

Optional functions

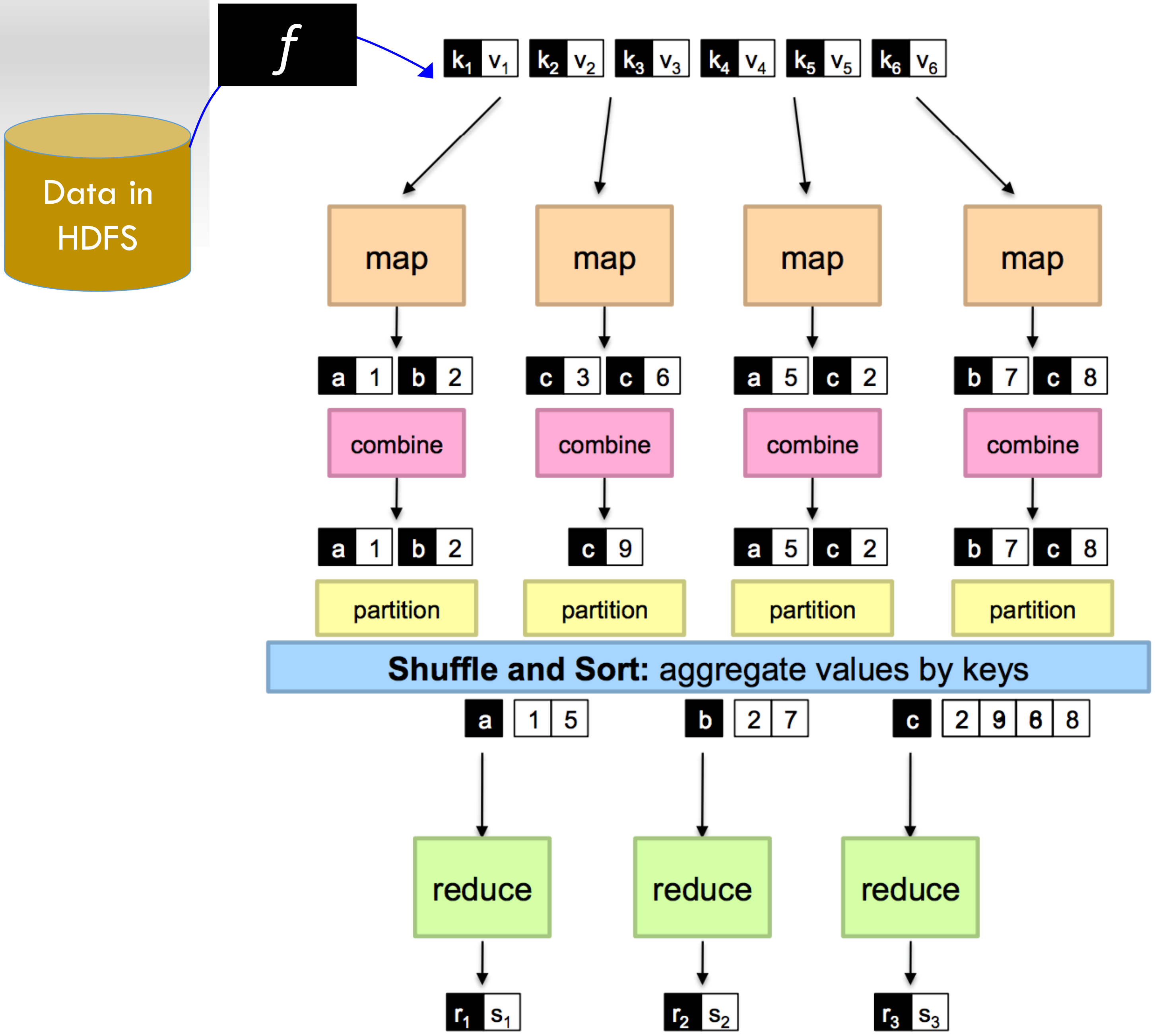
- Programmers create three functions/methods
- The execution framework in Hadoop handles everything else…
- Usually, programmers also include the following two functions to optimize performance:

  combine (k, v) → <k, v'>

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., *hash(k') mod n*
  - Divides up key space for parallel reduce operations


Data in HDFS
f
k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

(Lin, WWW 2013)

Combiner

small reduce within the map

- In MapReduce framework, usually the output from the map tasks is large and data transfer between map and reduce tasks will be high
- Combiner functions summarize the map output records with the same key and output of combiner will be sent over network to actual reduce task as input
- The combiner does not have its own interface and **it must implement Reducer interface** and reduce() method of combiner will be called on each map output key
- reduce() method **must have the same input and output key-value types as the reducer class**
- **Hadoop doesn’t guarantee on how many times a combiner will be called** for each map output key
- In most cases, the reducer is used as the combiner

Partitioner

- The mechanism sending specific key-value pairs to specific reducers is called partitioning (the key-value pairs space is partitioned among the reducers)
- The default partitioner is *HashPartitioner*
  - Hashes a record's key to determine which partition (and thus which reducer) the record belongs in
  - The number of partition = the number of reduce tasks for the job

Examples

Example

- Suppose that your job's input is a (huge) set of tokens and their number of occurrences and that you want to sort them by number of occurrences

#occ	word
1	abandonedly
1	abasement
...	
1	zoroaster
3	abandon
...	
6502	and
14620	the

Reducer 1

#occ	word
2	aback
2	abaft
...	
2	zoology
4	abide
...	
4776	a
6732	of

Reducer 2

Case 1

- Suppose that your job's input is a (huge) set of tokens and their number of occurrences and that that you want to sort them by number of occurrences
  - 95% tokens occur once
  - Using 2 reducers

- Customized partitioning function
  - Partition <#occ, token> pair instead of just #occ to balance load for two reducers

Case 2

- Suppose that your job's input is a (huge) set of tokens and their number of occurrences and that you want to sort them by number of occurrences

#occ	word
1	abandonedly
1	abasement
…	
1	zoroaster
3	abandon
…	
6502	and
14620	the

Reducer 1

#occ	word
2	aback
2	abaft
…	
2	zoology
4	abide
…	
4776	a
6732	of

Reducer 2

#occ	word
1	abandonedly
1	abasement
…	
2	aback
2	abaft
…	
30	touch
30	vain

Reducer 1

#occ	word
31	across
31	battle
…	
4776	a
6502	and
6732	of
14620	the

Reducer 2

- Customized partitioning function
  - All the tokens having a number of occurrences inferior to N (here 30) are sent to reducer 1 and the others are sent to reducer 2, resulting in two partitions

Customized partitioner

- The input data format:
  - Name<tab>age<tab>gender<tab>salary
  - E.g.,

    Rajee<tab>23<tab>female<tab>5000
    Rama<tab>34<tab>male<tab>7000
    Arjun<tab>67<tab>male<tab>900000
    Keerthi<tab>38<tab>female<tab>100000
    Kishore<tab>25<tab>male<tab>23000
    Daniel<tab>78<tab>male<tab>7600
    James<tab>34<tab>male<tab>86000
    Alex<tab>52<tab>male<tab>6900
    Nancy<tab>7<tab>female<tab>9800
    Adam<tab>9<tab>male<tab>3700
    Jacob<tab>7<tab>male<tab>2390
    Mary<tab>6<tab>female<tab>9300
    Clara<tab>87<tab>female<tab>72000
    Monica<tab>56<tab>female<tab>92000

- Output: find the maximum salary in each gender and three age categories: less than 20, 20 to 50, greater than 50

- Mapper
  - Input: each line
  - Output
    - Key: gender<tab>somestringhere, value: nameAgeSalary
- Partitioner
  - Input is from the output of mapper
  - Output
    - Key: gender<tab>somestringhere, value: nameAgeSalary
    - Partition 1: if age <=20
    - Partition 2: if 20<age<=50
    - Partition 3: if age>50
- Reducer
  - Input is from the output of partitioner
  - Output
    - The same format as the original input but having three files with each including two lines indicating maximum salary for male and female, respectively

Additional Marks

- Hadoop is essentially two parts: storing the input and output in Hadoop Distributed File System (HDFS), with the power of parallel processing of MapReduce for our data.

- With parallel processing, as large volumes of data can be worked through in hours not days. It also allows for better scalability of your jobs. By adding additional nodes (called horizontal scaling) you get a bump in processing power, rather than having to increase the performances of your nodes (vertical scaling). If you need more processing, add more nodes, don't increase the power of the nodes.

MapReduce in Tech Companies

- At Google:

– Index building for Google Search
– Article clustering for Google News
– Statistical machine translation

- At Yahoo!:

– Index building for Yahoo! Search
– Spam detection for Yahoo! Mail

- At Facebook:

– Data mining
– Ad optimization
– Spam detection Example

- At Amazon:

– Product clustering
– Statistical machine translation

Final Marks

- During the processing Hadoop coordinates tasks between various nodes — should you have more than one running — to ensure the load is balanced correctly. It takes care of the design issues of the system, by moving reliability and fault tolerance into the background, letting you focus the data and what you want to extract from it.

Lab 3 Cloudera Installation

Questions?