An Innovative Approach to Parallel Processing Data

BINA RAMAMURTHY
PARTIALLY SUPPORTED BY
NSF DUE GRANT: 0737243, 0920335

The Context: Big-data

- Man on the moon with 32KB (1969); my laptop had 2GB RAM (2009)
- Google collects 270PB data in a month (2007), 20PB a day (2008) [1]
- 2010 census data is expected to be a huge gold mine of information
- Data mining huge amounts of data collected in a wide range of domains from astronomy to healthcare has become essential for planning and performance.
- We are in a knowledge economy.
  - Data is an important asset to any organization
  - Discovery of knowledge; Enabling discovery; annotation of data
- We are looking at newer
  - programming models, and
  - Supporting algorithms and data structures.
- NSF refers to it as “data-intensive computing” and industry calls it “big-data” and “cloud computing”

MapReduce

What is MapReduce?

- MapReduce is a programming model Google has used successfully in processing its "big-data" sets (~ 20 peta bytes per day in 2008)
  - Users specify the computation in terms of a *map* and a *reduce* function,
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues.

-- Reference: Dean, J. and Ghemawat, S. 2008. **MapReduce: simplified data processing on large clusters.** *Communication of ACM* 51, 1 (Jan. 2008), 107-113.

Big idea behind MR

- **Scale-out** and not scale-up: Large number of commodity servers as opposed large number of high end specialized servers
  - Economies of scale, ware-house scale computing
  - MR is designed to work with clusters of commodity servers
  - Research issues: Read Barroso and Holzle's work
- **Failures are norm** or common:
  - With typical reliability, MTBF of 1000 days (about 3 years), if you have a cluster of 1000, probability of at least 1 server failure at any time is nearly 100%

Issues to be addressed

- How to break large problem into smaller problems? Decomposition for parallel processing
- How to assign tasks to workers distributed around the cluster?
- How do the workers get the data?
- How to synchronize among the workers?
- How to share partial results among workers?
- How to do all these in the presence of errors and hardware failures?
- MR is supported by a distributed file system that addresses many of these aspects.

MapReduce Basics

- Fundamental concept:
- Key-value pairs form the basic structure of MapReduce <key, value>
- Key can be anything from a simple data types (int, float, etc) to file names to custom types.
- Examples:
  - <docid, docitself>
  - <yourName, yourLifeHistory>
  - <graphNode, nodeCharacteristicsComplexData>
  - <yourId, yourFollowers>
  - <word, itsNumofOccurrences>
  - <planetName, planetInfo>
  - <geneNum, <{pathway, geneExp, proteins}>
  - <Student, stuDetails>

From CS Foundations to MapReduce (Example#1)

Consider a large data collection:

{web, weed, green, sun, moon, land, part, web, green,…}

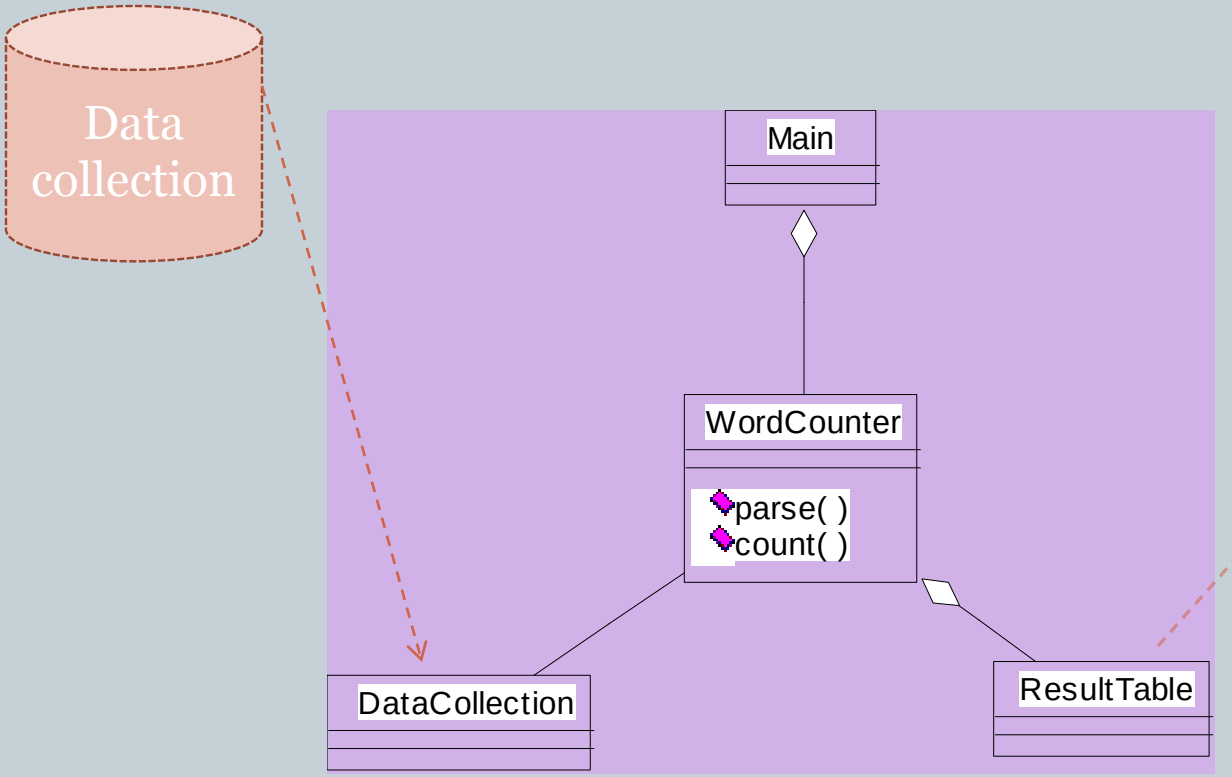
Problem: Count the occurrences of the different words in the collection.

Lets design a solution for this problem;

- We will start from scratch
- We will add and relax constraints
- We will do incremental design, improving the solution for performance and scalability

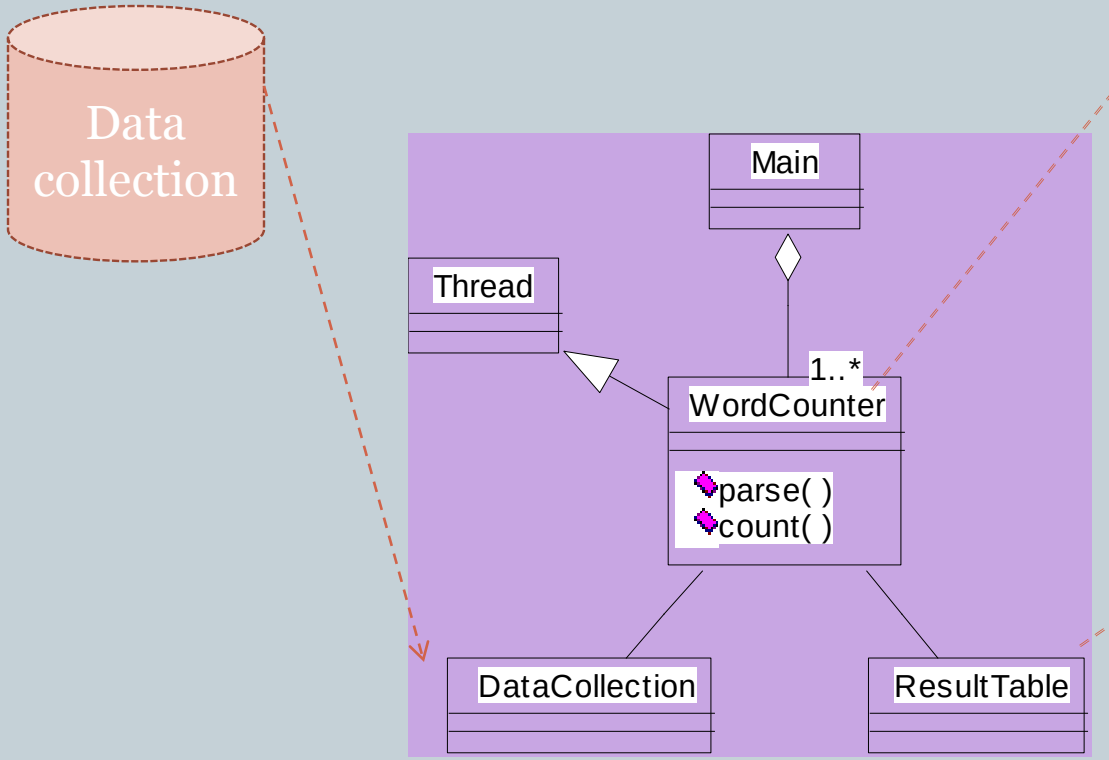
Word Counter and Result Table

{web, weed, green, sun, moon, land, part, web, green,...}



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

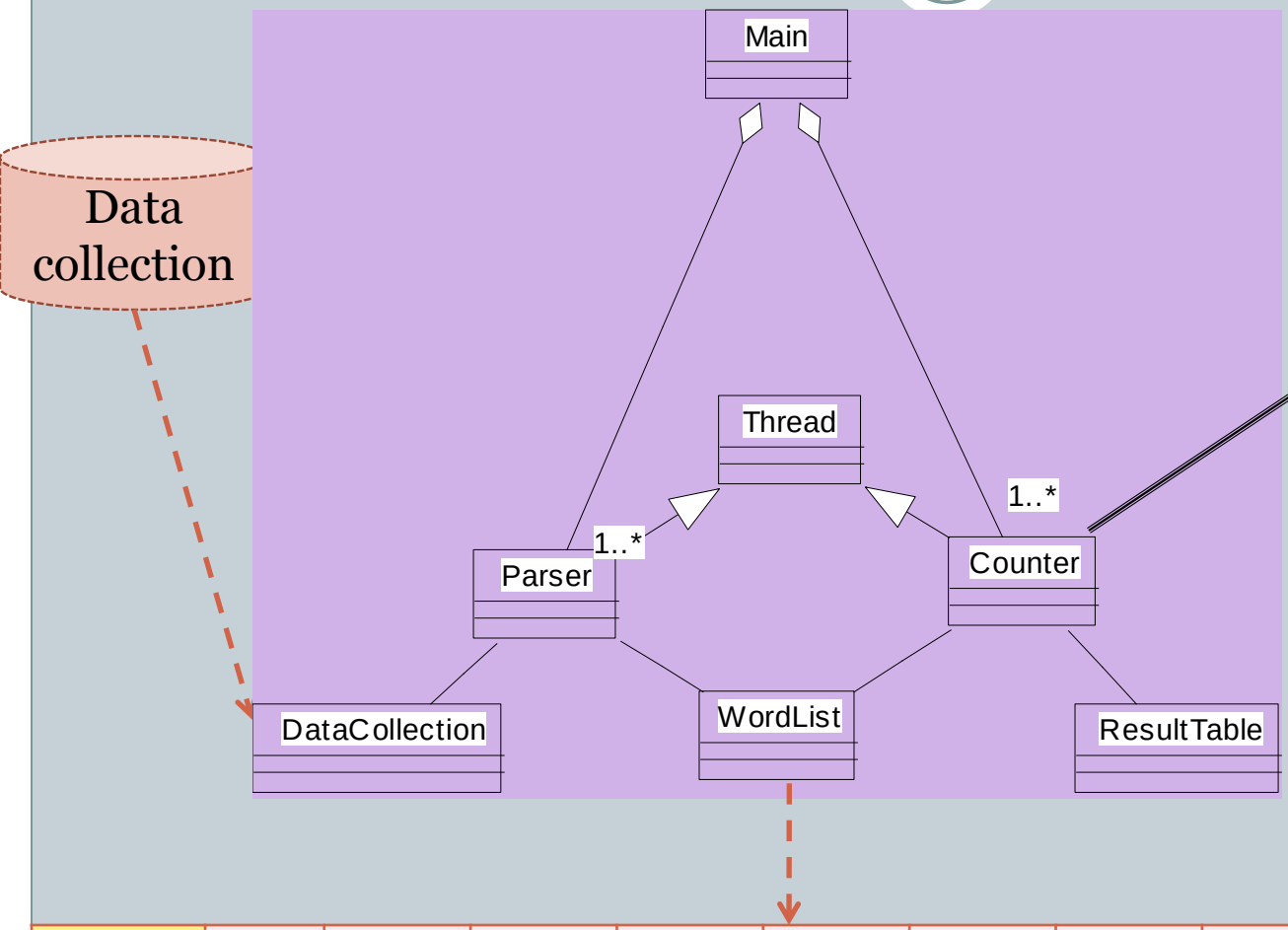
Multiple Instances of Word Counter



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

Observe:
Multi-thread
Lock on shared data

Improve Word Counter for Performance



No need for lock

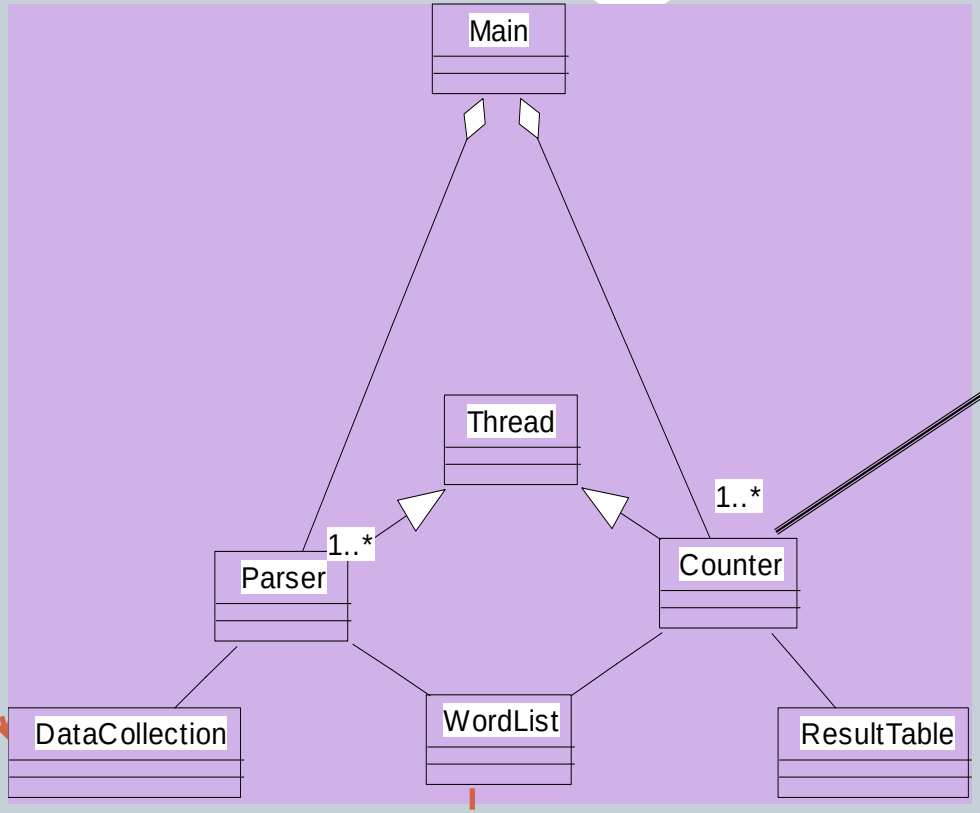
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Separate counters

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

Peta-scale Data

Data collection



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										

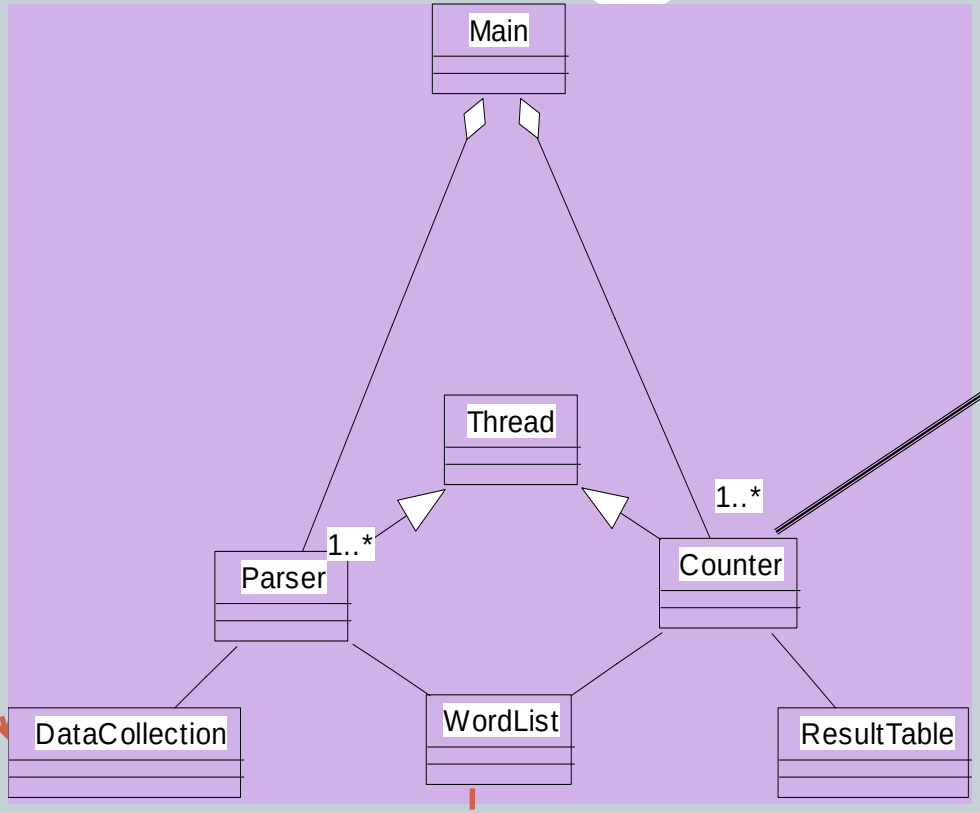
Addressing the Scale Issue

- 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

Data collection



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

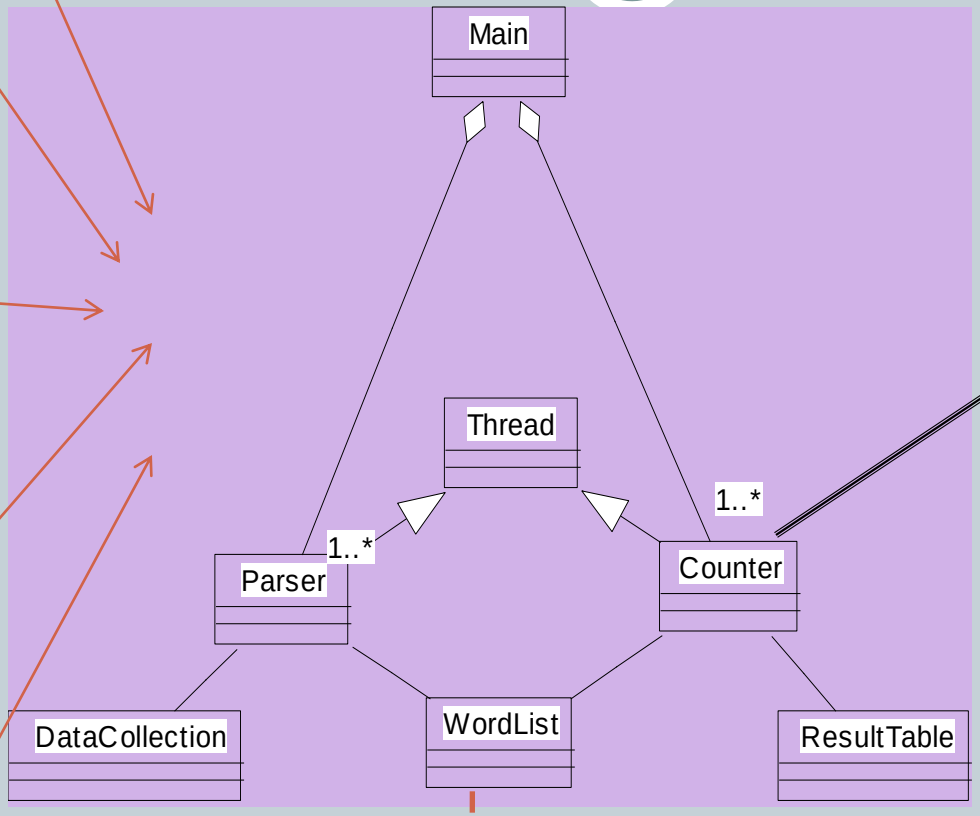
Data collection

Data collection

Data collection

Data collection

Data collection

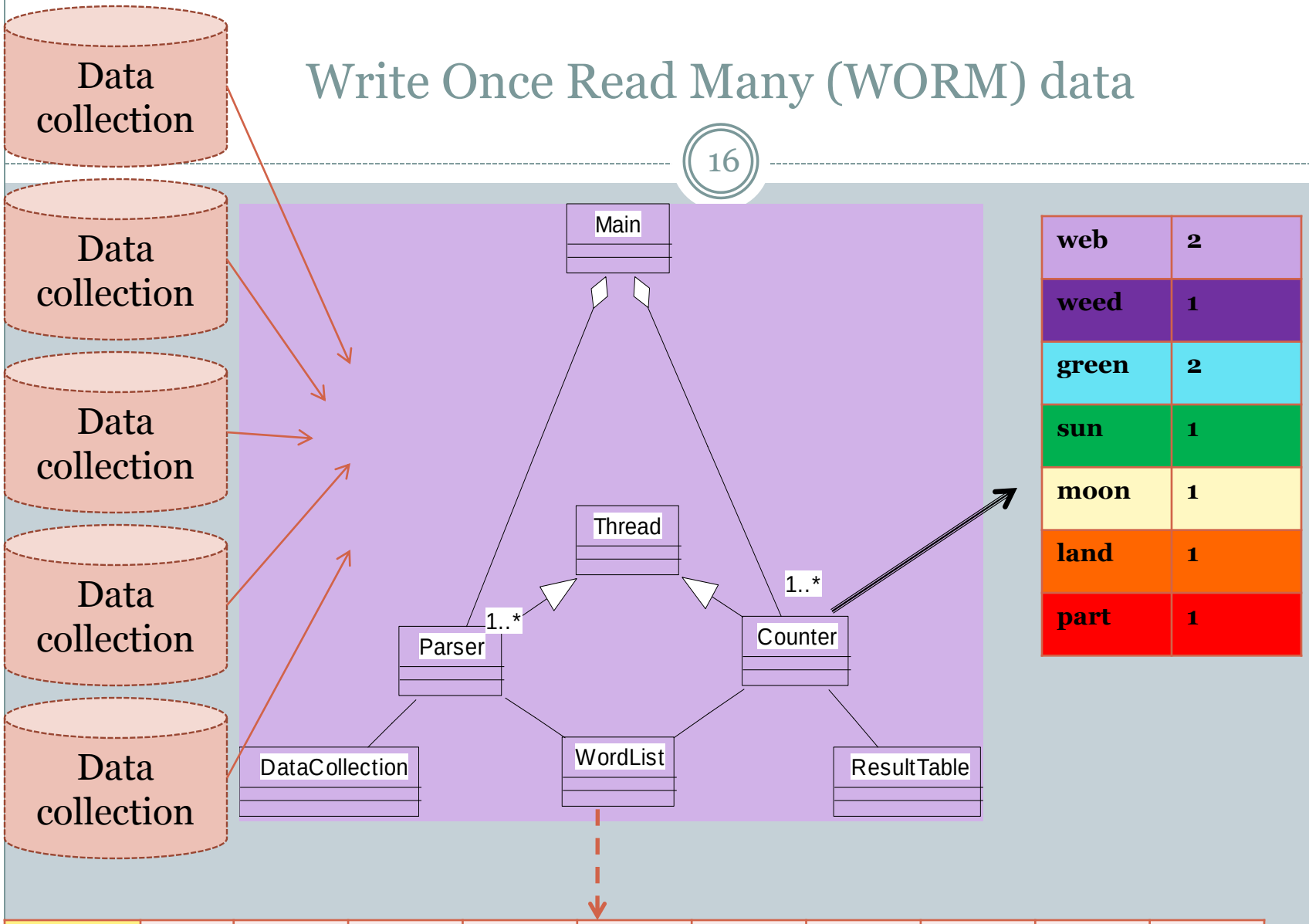


web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Issue: managing the large scale data

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

Write Once Read Many (WORM) data



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

WORM Data is Amenable to Parallelism

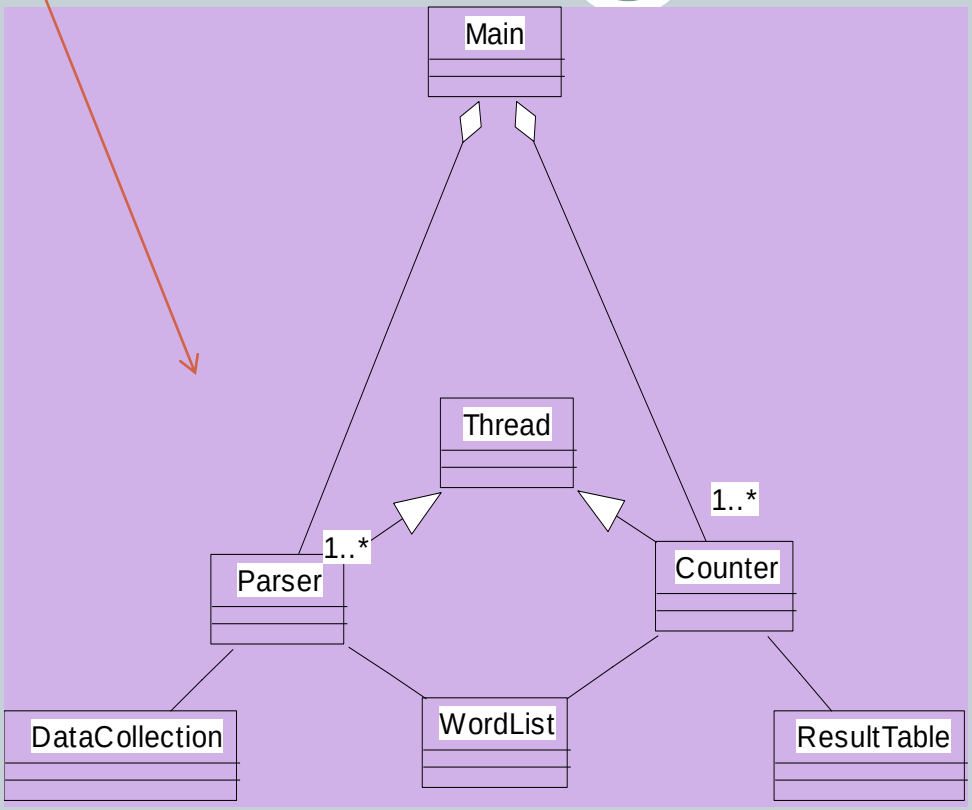
Data collection

Data collection

Data collection

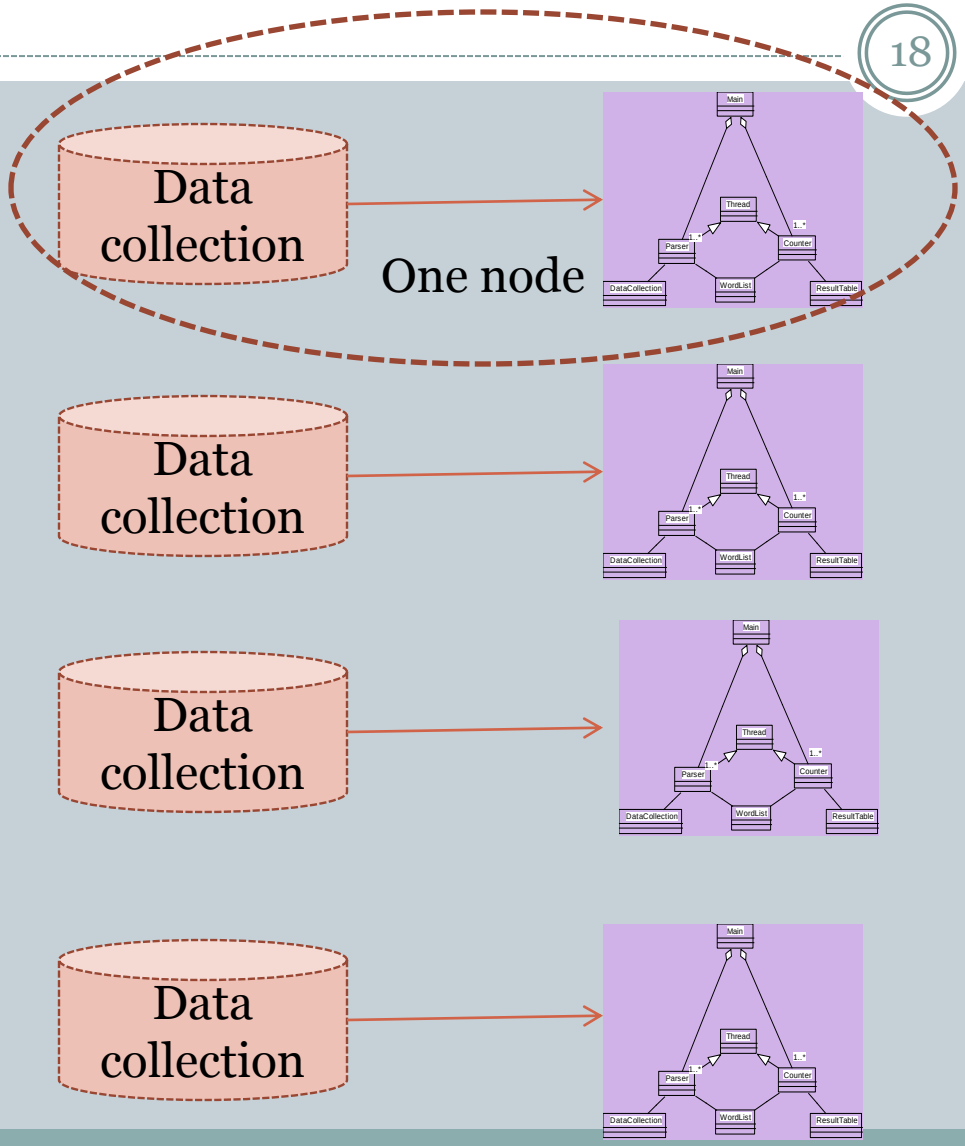
Data collection

Data collection



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:

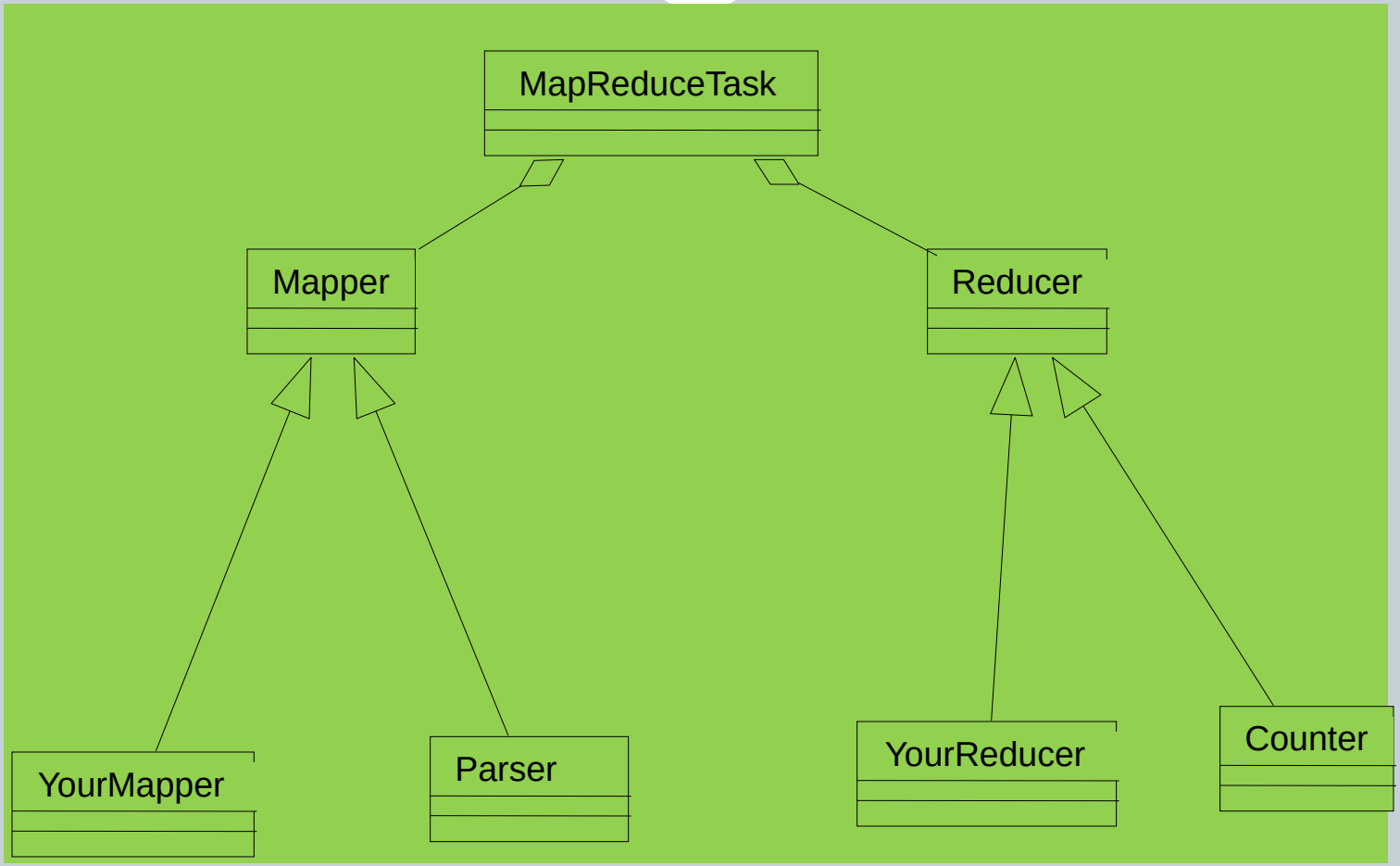
Our parse is a mapping operation:
MAP: input → <key, value> pairs

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!

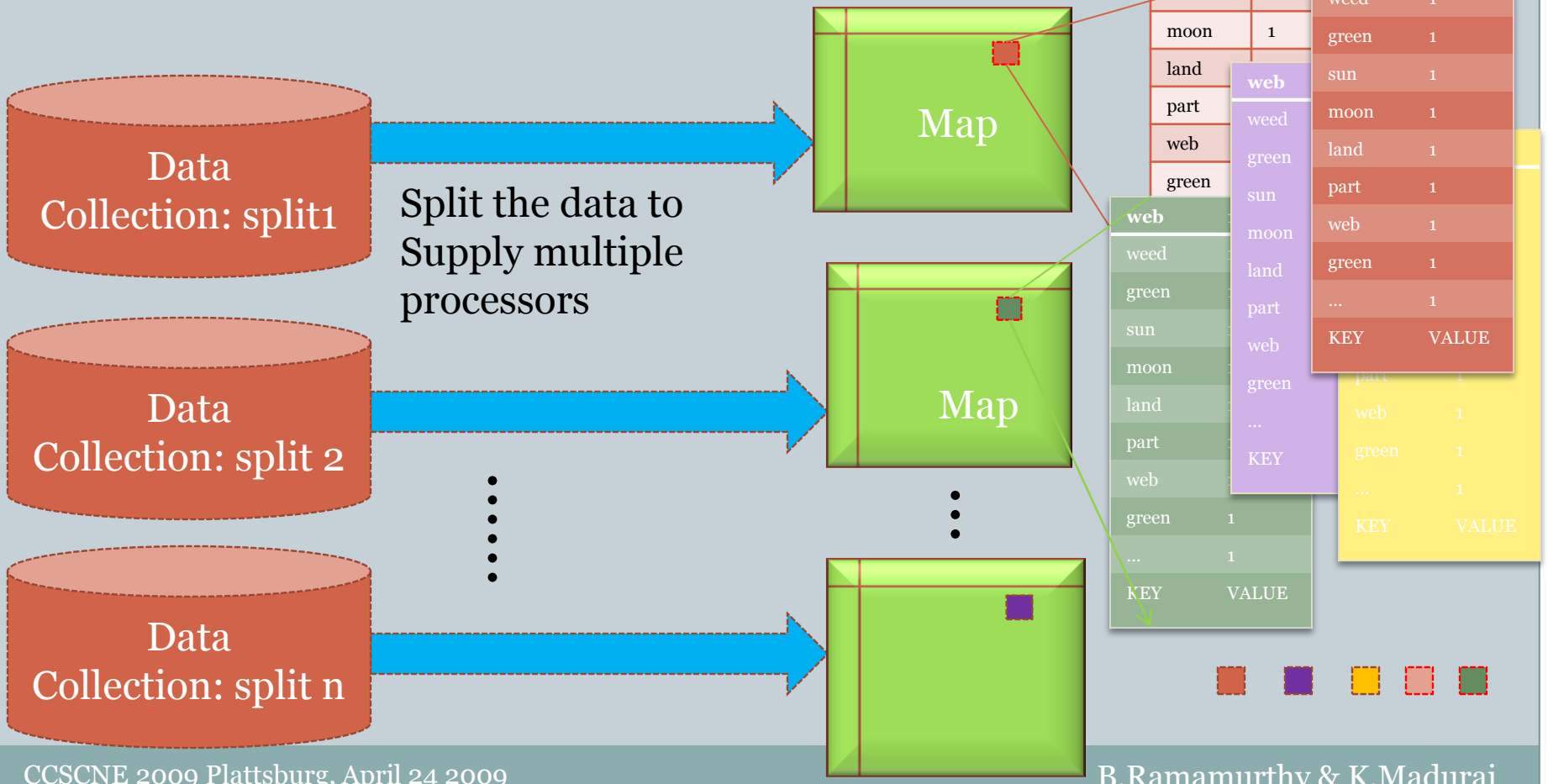
Mapper and Reducer



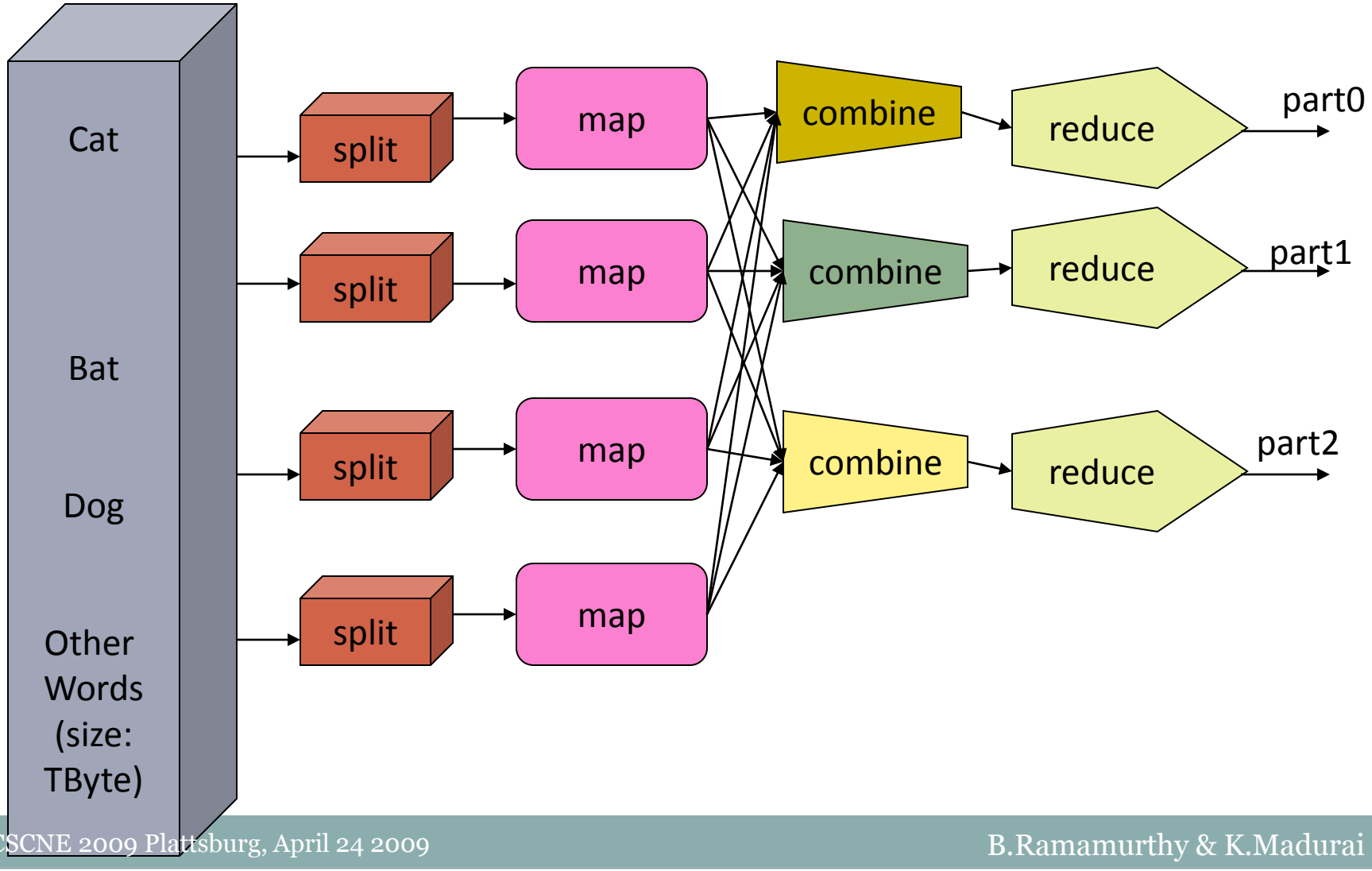
Remember: MapReduce is simplified processing for larger data sets

Map Operation

MAP: Input data ➔ <key, value> pair



MapReduce Example #2



MapReduce Design

- You focus on Map function, Reduce function and other related functions like combiner etc.
- Mapper and Reducer are designed as classes and the function defined as a method.
- Configure the MR "Job" for location of these functions, location of input and output (paths within the local server), scale or size of the cluster in terms of #maps, # reduce etc., run the job.
- Thus a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The **execution framework** handles everything else.

The code


```
1: class Mapper
2:       method Map(docid a; doc d)
3:       for all term t in doc d do
4:                Emit(term t; count 1)

1: class Reducer
2:       method Reduce(term t; counts [c1; c2; : : :])
3:        sum =  0
4:       for all count c in counts [c1; c2; : : :] do
5:                sum = sum + c
6:       Emit(term t; count sum)
```

MapReduce Example: Mapper

This is a cat
Cat sits on a roof
<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

The roof is a tin roof
There is a tin can on the roof
<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

Cat kicks the can
It rolls on the roof and falls on the next roof
<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1>
<falls 1> <next 1>

The cat rolls too
It sits on the can
<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

MapReduce Example: Combiner, Reducer, Shuffle, Sort

<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1> <falls 1> <next 1>

<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

Combine the counts of all the same words:

<cat <1,1,1,1>>

<roof <1,1,1,1,1,1>>

<can <1, 1,1>>

…

Reduce (sum in this case) the counts:

<cat 4>

<can 3>

<roof 6>

More on MR

- All Mappers work in parallel.
- Barriers enforce all mappers completion before Reducers start.
- Mappers and Reducers typically execute on the same server
- You can configure job to have other combinations besides Mapper/Reducer: ex: identify mappers/reducers for realizing “sort” (that happens to be a Benchmark)
- Mappers and reducers can have side effects; this allows for sharing information between iterations.

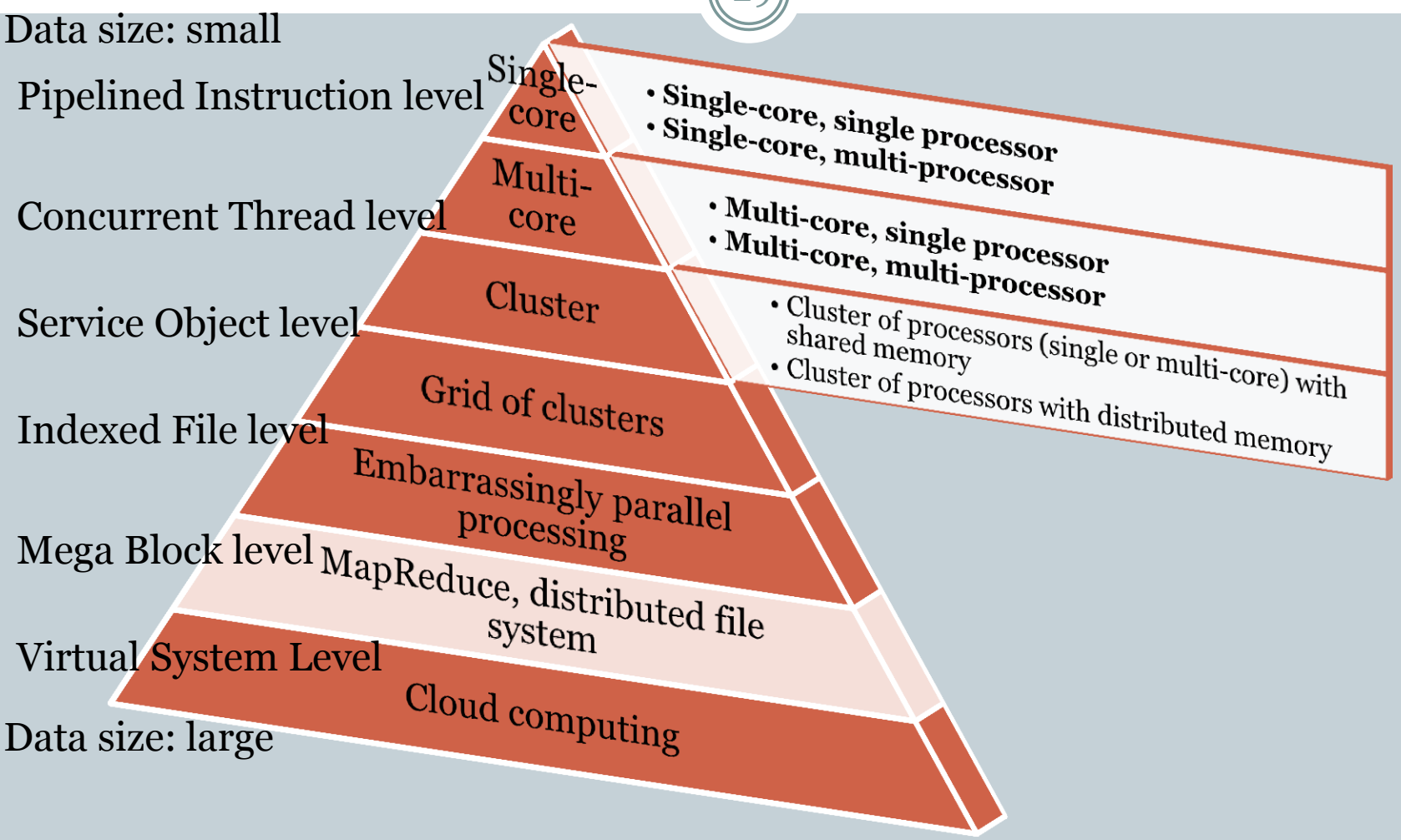
MapReduce Characteristics

- Very large scale data: peta, exa bytes
- Write once and read many data: allows for parallelism without mutexes
- Map and Reduce are the main operations: simple code
- There are other supporting operations such as combine and partition: we will look at those later.
- All the map should be completed before reduce operation starts.
- Map and reduce operations are typically performed by the same physical processor.
- Number of map tasks and reduce tasks are configurable.
- Operations are provisioned near the data.
- Commodity hardware and storage.
- Runtime takes care of splitting and moving data for operations.
- Special distributed file system: Hadoop Distributed File System and Hadoop Runtime.

Classes of problems "mapreducable"

- Benchmark for comparing: Jim Gray's challenge on data-intensive computing. Ex: "Sort"
- Google uses it (we think) for wordcount, adwords, pagerank, indexing data.
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining domain
- Facebook uses it for various operations: demographics
- Financial services use it for analytics
- Astronomy: Gaussian analysis for locating extra-terrestrial objects.
- Expected to play a critical role in semantic web and web3.0

Scope of MapReduce



Hadoop

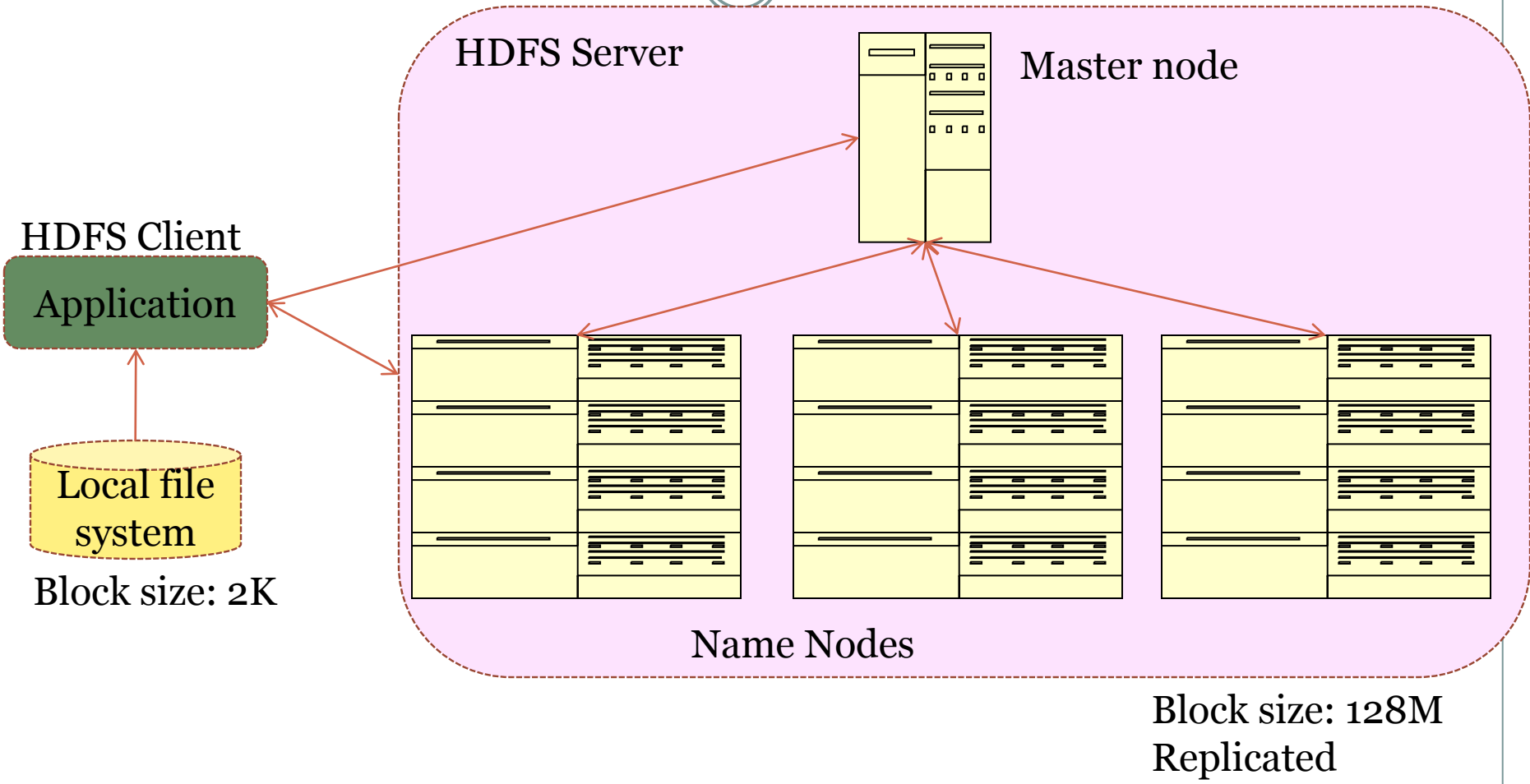
What is Hadoop?

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose.
- GFS is not open source.
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS).
- The software framework that supports HDFS, MapReduce and other related entities is called the project Hadoop or simply Hadoop.
- This is open source and distributed by Apache.

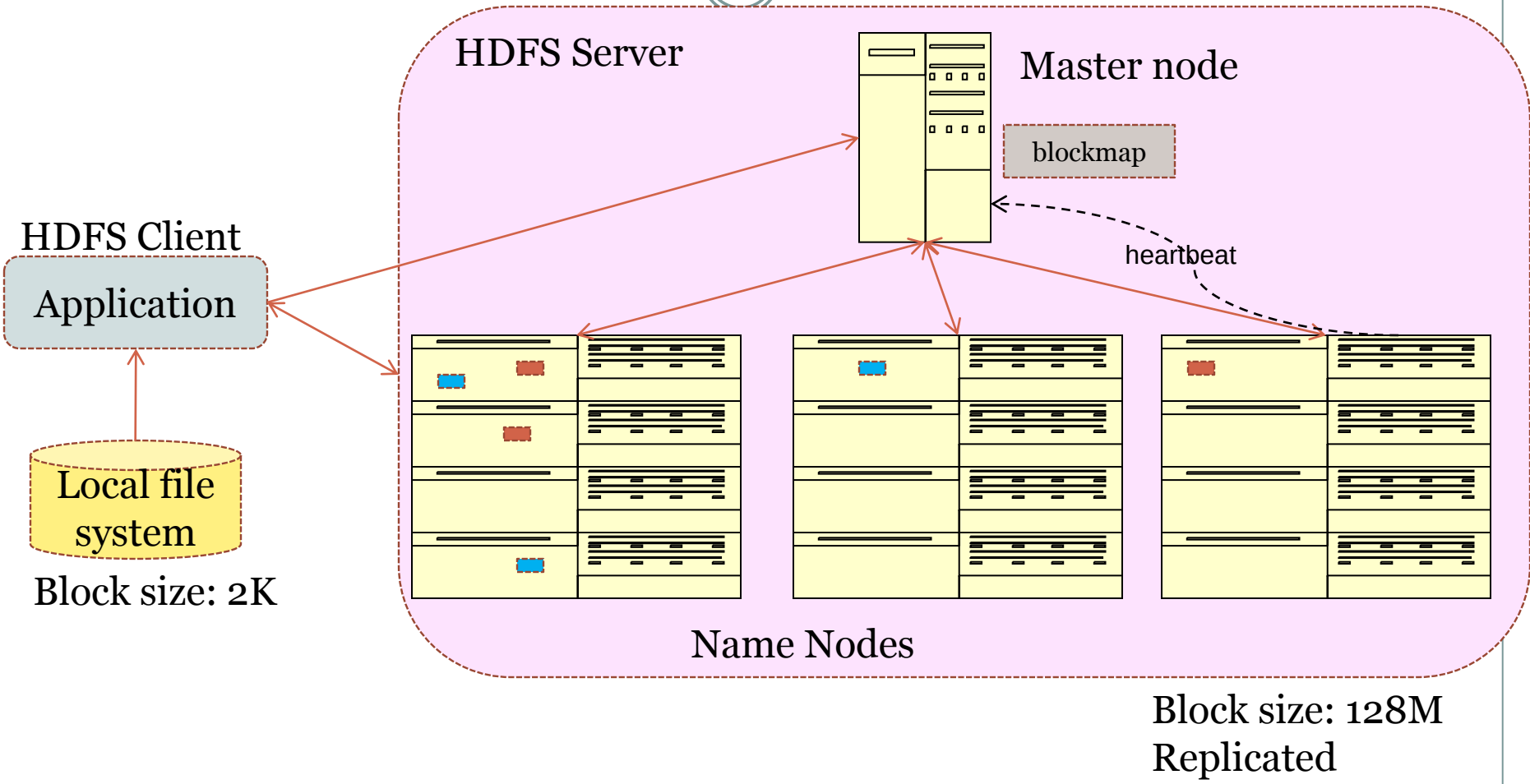
Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Hadoop Distributed File System



Hadoop Distributed File System



Hadoop (contd.)

- We will take an in-depth look at Hadoop infrastructure next lecture (Next Tuesday)

- On the demo….
- We will now setup and execute a sample MR run on a Hadoop cluster on amazon web services (cloud) infrastructure.