CSE 4/587
Data Intensive Computing

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Dr. Shamshad Parvin
shamsadp@buffalo.edu
313 Davis Hall

Introduction to MapReduce

Additional Reference for MapReduce

Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, Synthesis Lectures on Human Language Technologies, 2010, Vol. 3, No. 1, Pages 1-177, (doi: 10.2200/S00274ED1V01Y201006HLT007).

An online version of this text is also available through UB Libraries since UB subscribes to Morgan and Claypool Publishers.

Online version available at:
http://lintool.github.com/MapReduceAlgorithms/index.html

Recap

- Last week we covered the Hadoop File System (HDFS)
  - Large scale distributed storage for huge files of data
  - Fault tolerance allows for reliability at scale
  - Provides the underlying backbone for a number of different technologies

How Big is Big Data?

- Man on the moon with 4K RAM, 32KB HDD (1969); my laptop has 16GB RAM (2017)
- Google collects 270PB data in a month (2007), 20PB a day (2008), 200PB a day estimated (2020)
- 2010 census data is a huge gold mine of information
- Data mining huge amounts of data collected in a wide range of domains
    - Astronomy, Healthcare, Finance, etc.
- Data is an important asset to any organization
- National Science Foundation refers to it as “data-intensive computing” and industry calls it “big-data” and “cloud computing”

Introduction (Ch 1. Lin and Dyer)

- Text Processing at large scales
  - Simple word count, cross reference, n-grams, etc
- **A simpler technique on more data can beat a more sophisticated technique on less data.**
- Google researchers call this "Unreasonable effectiveness of data" [1]

[1] Alon Halevy, Peter Norvig, and Fernando Pereira. **The unreasonable effectiveness of data.** Communications of the ACM, 24(2):8:12, 2009.

MapReduce

- MapReduce is a programming model **and** an execution framework
  - Developed by Google for operating on its large amounts of data
  - Open Source implementation in Hadoop
- Computation specified in terms of *map* and *reduce* functions
- Underlying runtime system (RTS) automatically parallelizes and coordinates the computation across a cluster of machines
  - Also handles machine failures, communication, and performance issues
- APIs originally in Java, now also supports Python, Ruby, C++, etc…

Big Ideas

- **Scale-out not scale-up:** Use a large number of commodity servers, as opposed to smaller number of high-end specialized servers
  - Part of this comes down to economies of scale and warehouse scale computing — what costs are associated with running such a warehouse?
  - High-end SMP servers will always outperform a network of commodity servers, but once data gets big, network communication becomes unavoidable — levels the playing field.

Big Ideas

- **Failures are the norm — not an exception**
  - Typical MTBF for commodity components of 1000 days — if you have 1000s in your cluster, probability of at least 1 being down at any time nears 100%

Big Ideas

- **Failures are the norm — not an exception**
  - Typical MTBF for commodity components of 1000 days — if you have 1000s in your cluster, probability of at least 1 being down at any time nears 100%
- **Move "Processing" to the Data:** Co-locate processing of the data with the data itself rather than sending data around as in HPC.

Big Ideas

- **Failures are the norm — not an exception**
  - Typical MTBF for commodity components of 1000 days — if you have 1000s in your cluster, probability of at least 1 being down at any time nears 100%
- **Move "Processing" to the Data:** Co-locate processing of the data with the data itself rather than sending data around as in HPC.
- **Process Data Sequentially vs Random Access:** Do mass analytics on large sequential build data as opposed to search for individual items

Big Ideas

- **Hide System Details from the User Application:** Programmers are bad at details (at least compared to computers). Let the RTS manage details for you.
    - ie: where is the data located, what communication is required, what is a given machine doing, etc.

Big Ideas

- **Hide System Details from the User Application:** Programmers are bad at details (at least compared to computers). Let the RTS manage details for you.
  - ie: where is the data located, what communication is required, what is a given machine doing, etc.
- **Seamless Scalability:** Machines can be added or removed without changing the algorithms.
  - Allows scaling up to process larger data sets without rethinking the entire application

Issues to Address

- How do we decompose large problems into smaller ones?
- How do we assign tasks to workers distributed across the cluster?
  - How do the workers get the data?
  - How do we synchronize among workers?
  - How do we share partial results among workers?
- How do we do all of this in the presence of faults?

Issues to Address

- How do we decompose large problems into smaller ones?
- How do we assign tasks to workers distributed across the cluster?
  - How do the workers get the data?
  - How do we synchronize among workers?
  - How do we share partial results among workers?
- How do we do all of this in the presence of faults?

As discussed last week, MR is supported by a distributed file system that provides many of these answers.

MapReduce Basics

Fundamental Concept: key-value pairs

- Key-value pairs form the basic structure of MapReduce
- Keys can be anything from simple data types to custom types

MapReduce Basics

Fundamental Concept: key-value pairs

- Key-value pairs form the basic structure of MapReduce
- Keys can be anything from simple data types to custom types
- Examples:

<docid, doc>

<yourName, yourLifeHistory>

<graphNode, nodeCharacteristics>

<geneNum, {pathway, geneExp, proteins}>

<yourID, yourFollowers>

<studentNum, studentDetails>

<word, numberOfOccurences>

etc…

Conceptual Example

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.

Conceptual Example

Consider a large data collection:

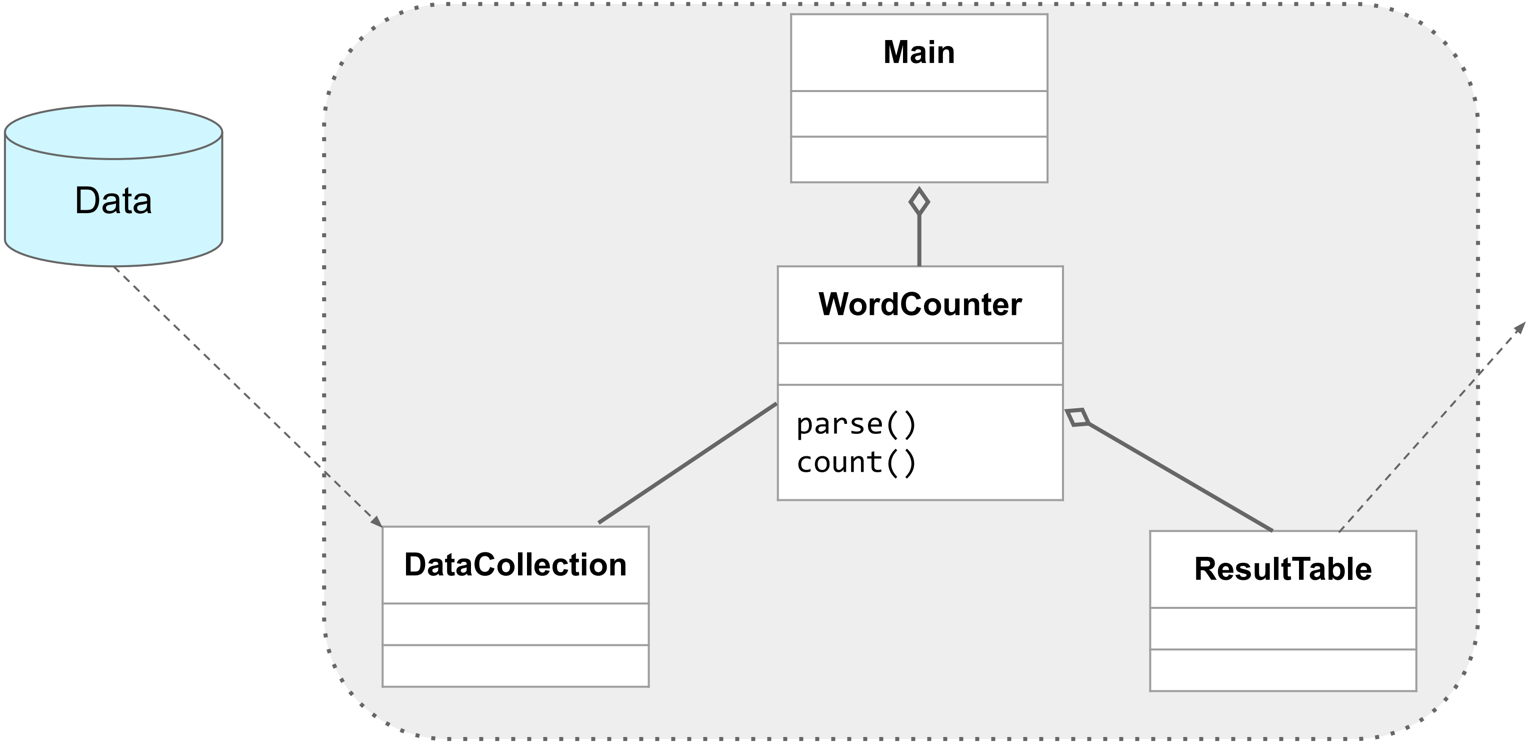
{web, weed, green, sun, moon, land, part, web, green,…}

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

Let's design a solution for this problem:

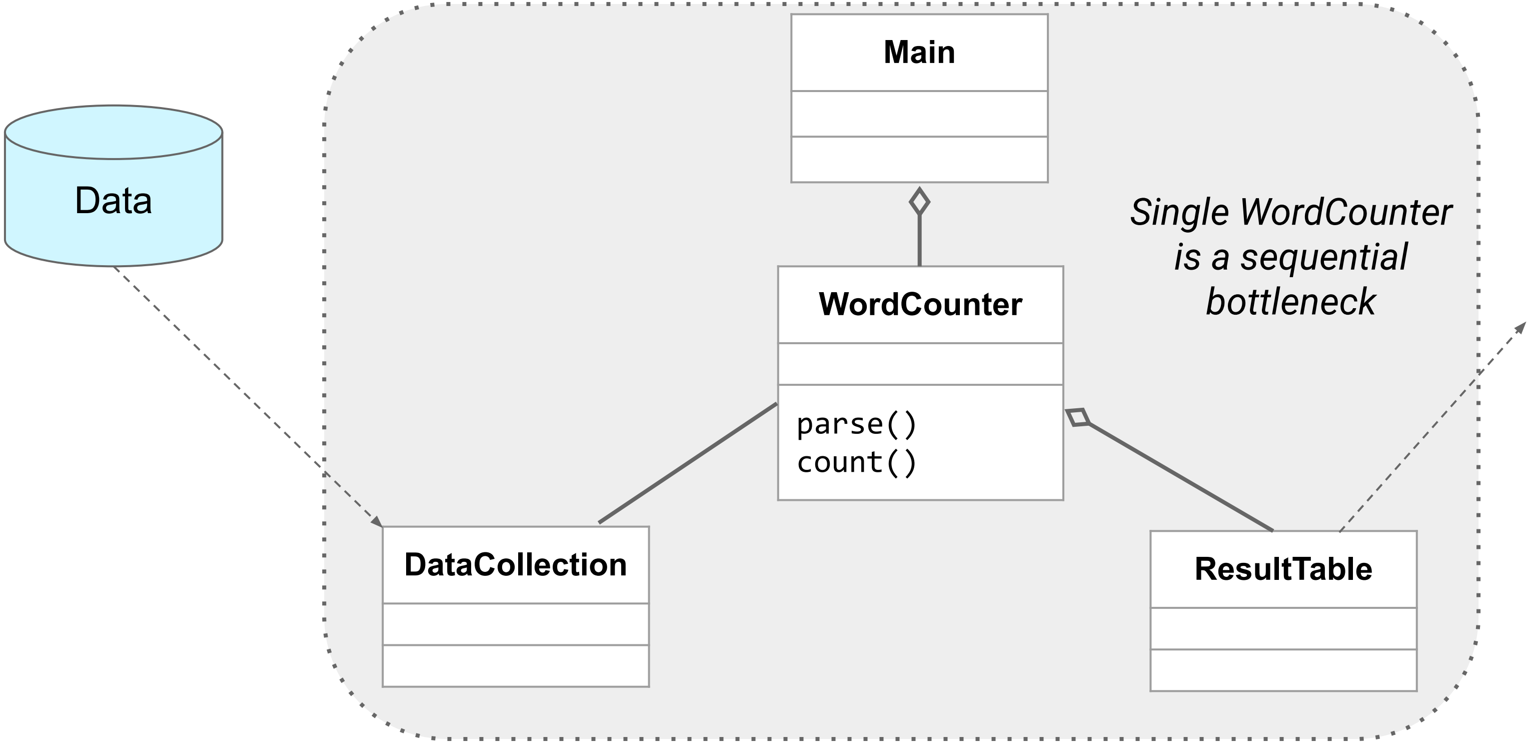
- We will start from scratch
- We will add and relax constraints
- We will do incremental design, improving solution as we go

Sequential Counter and Table



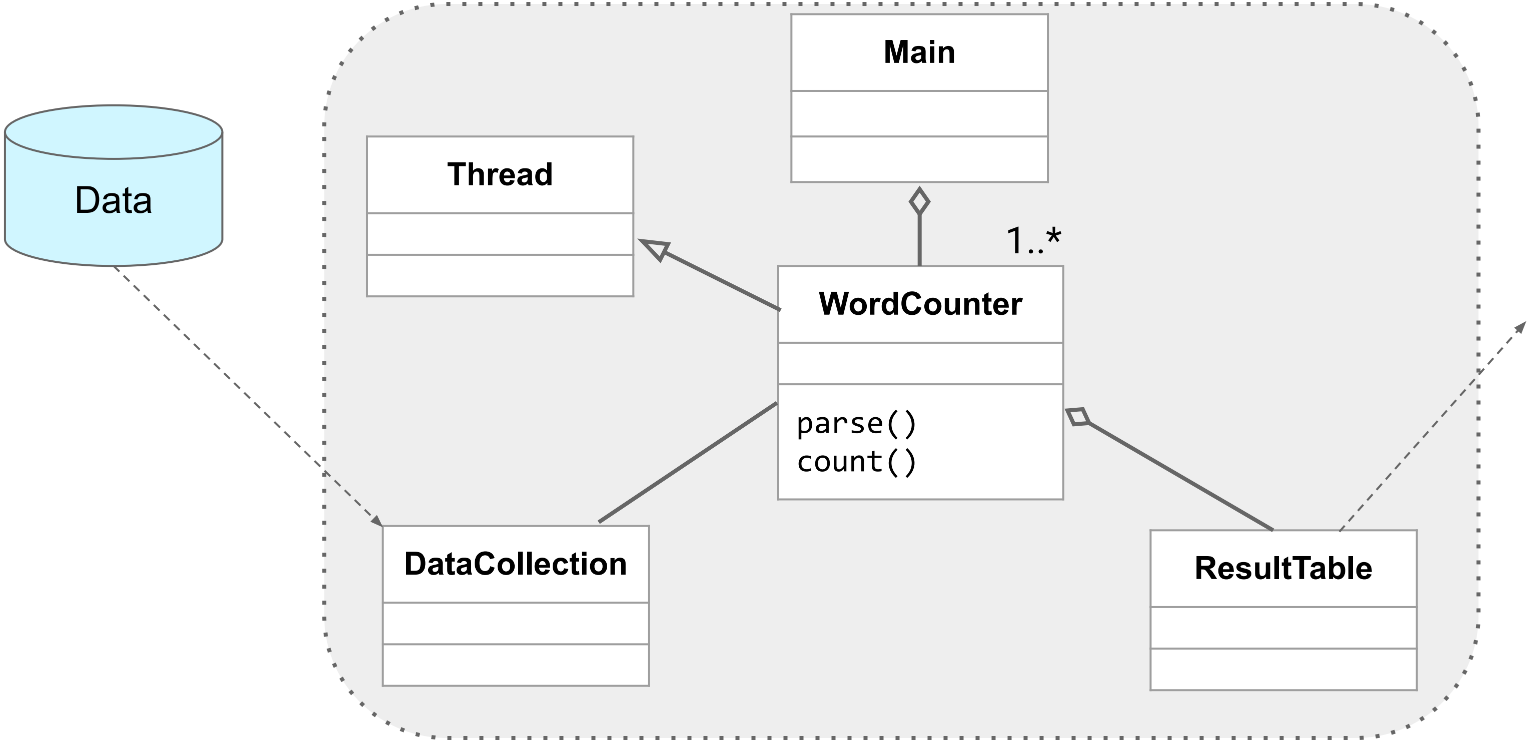
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1
…	…

Sequential Counter and Table



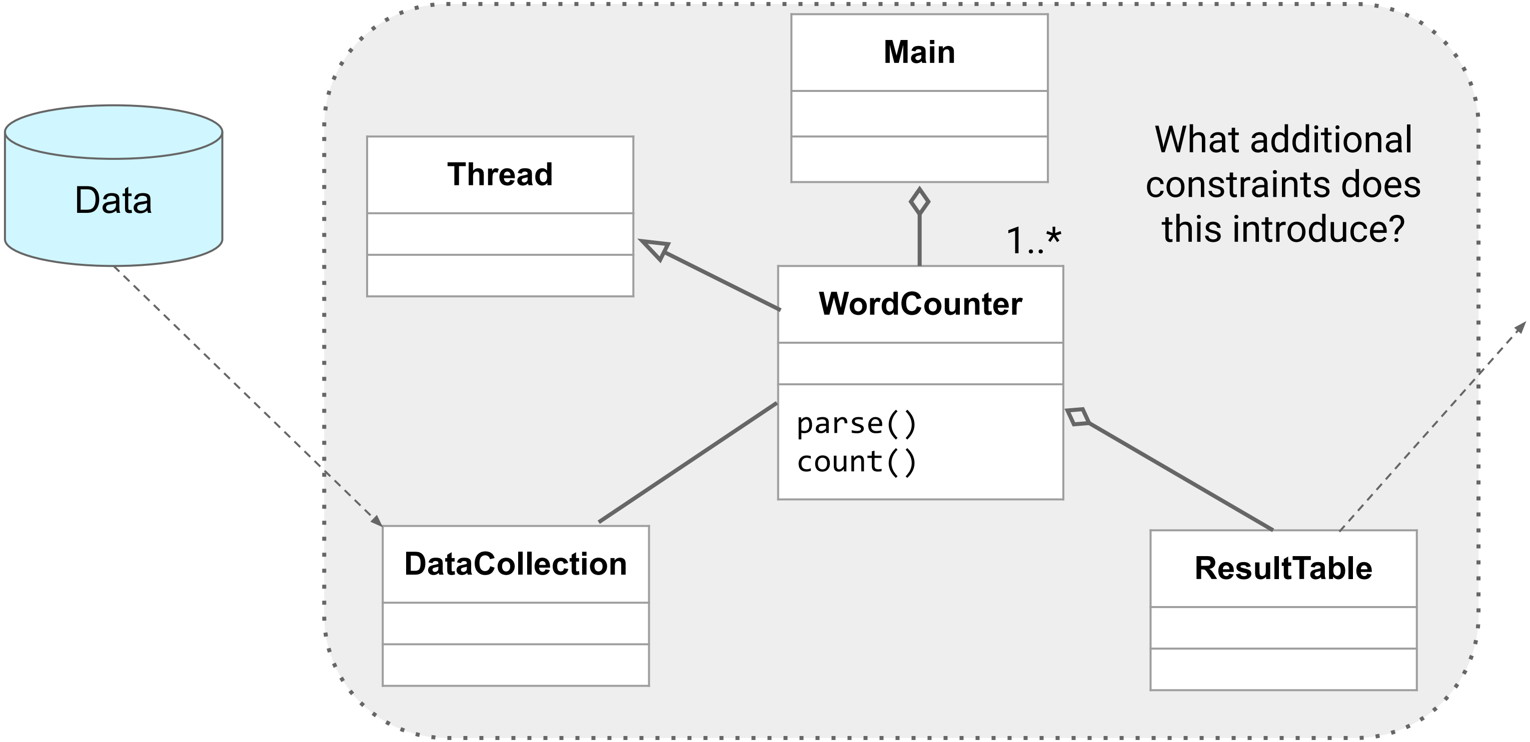
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1
…	…

Multiple Word Counters



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

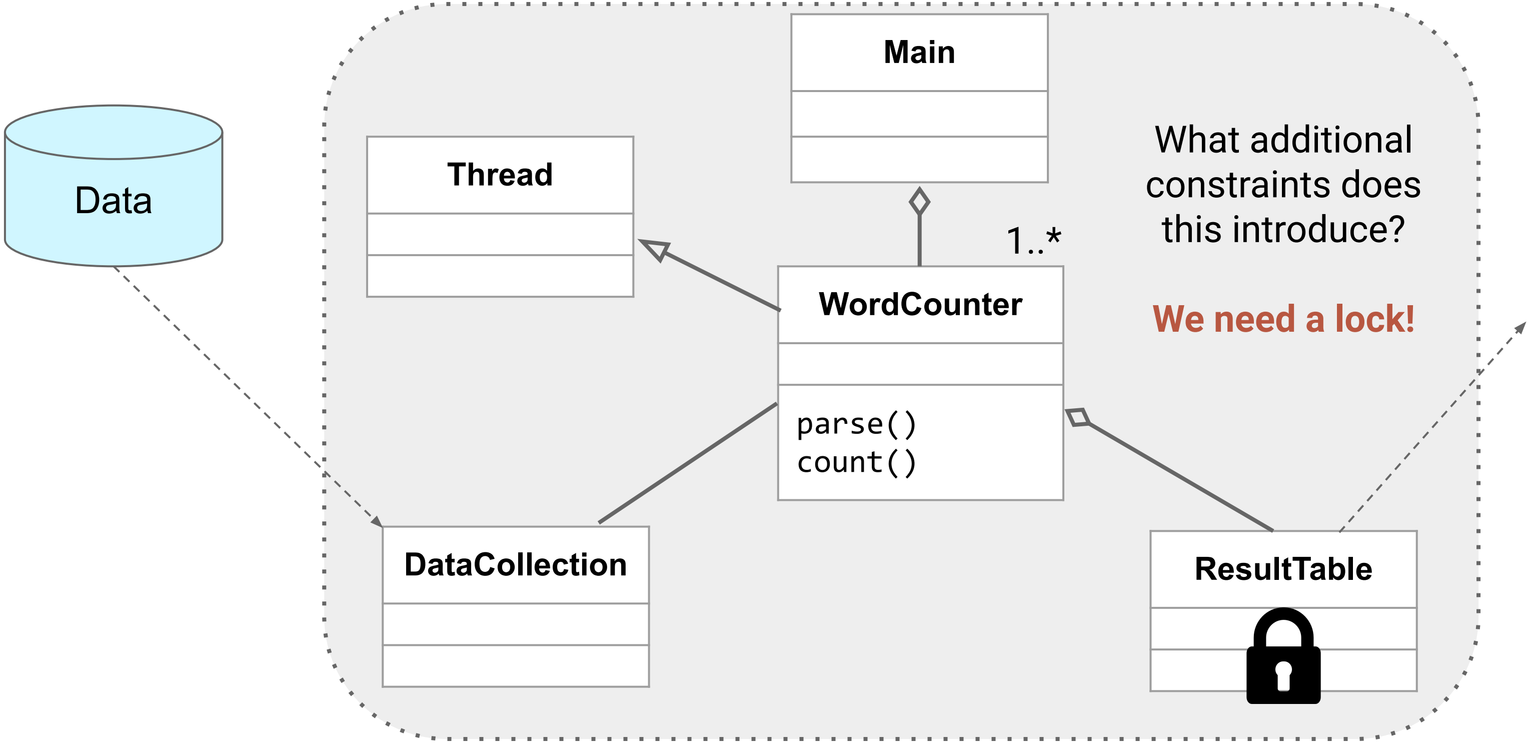
Multiple Word Counters



What additional constraints does this introduce?

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

Multiple Word Counters

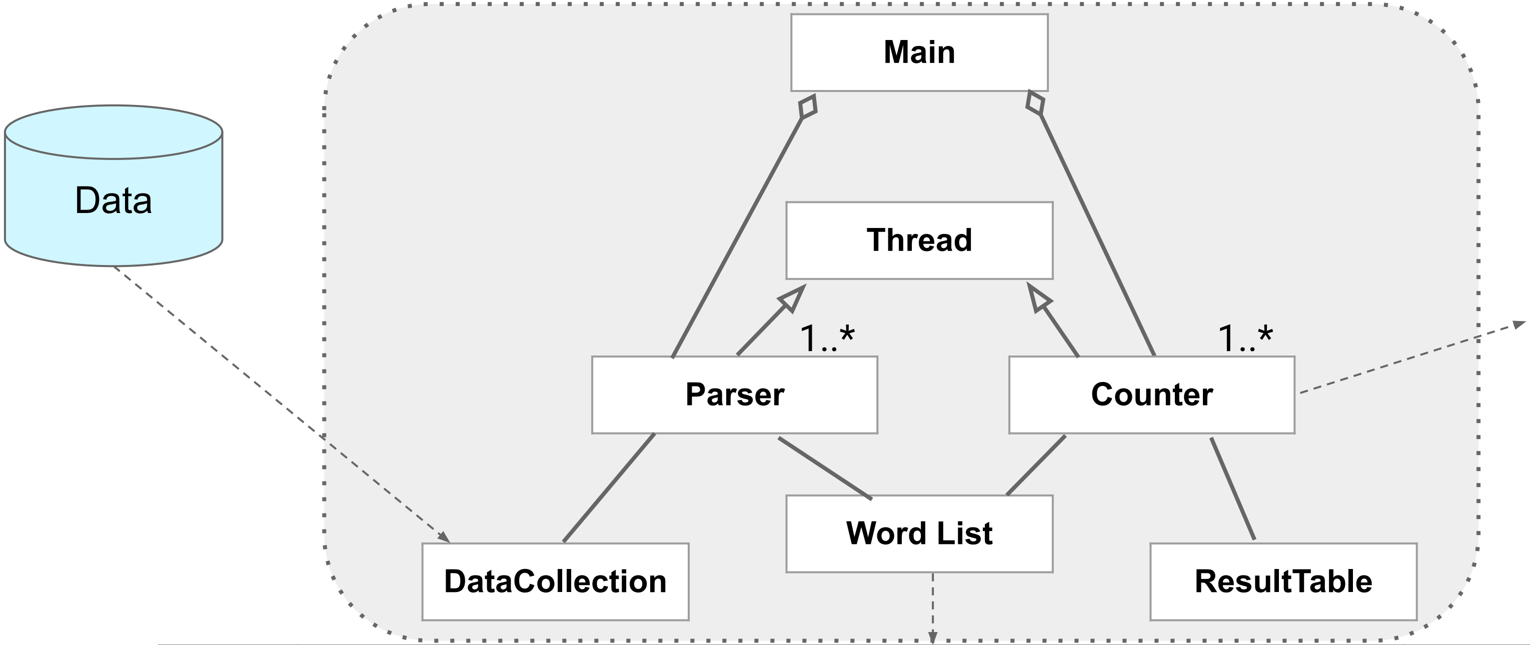


What additional constraints does this introduce?

We need a lock!

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

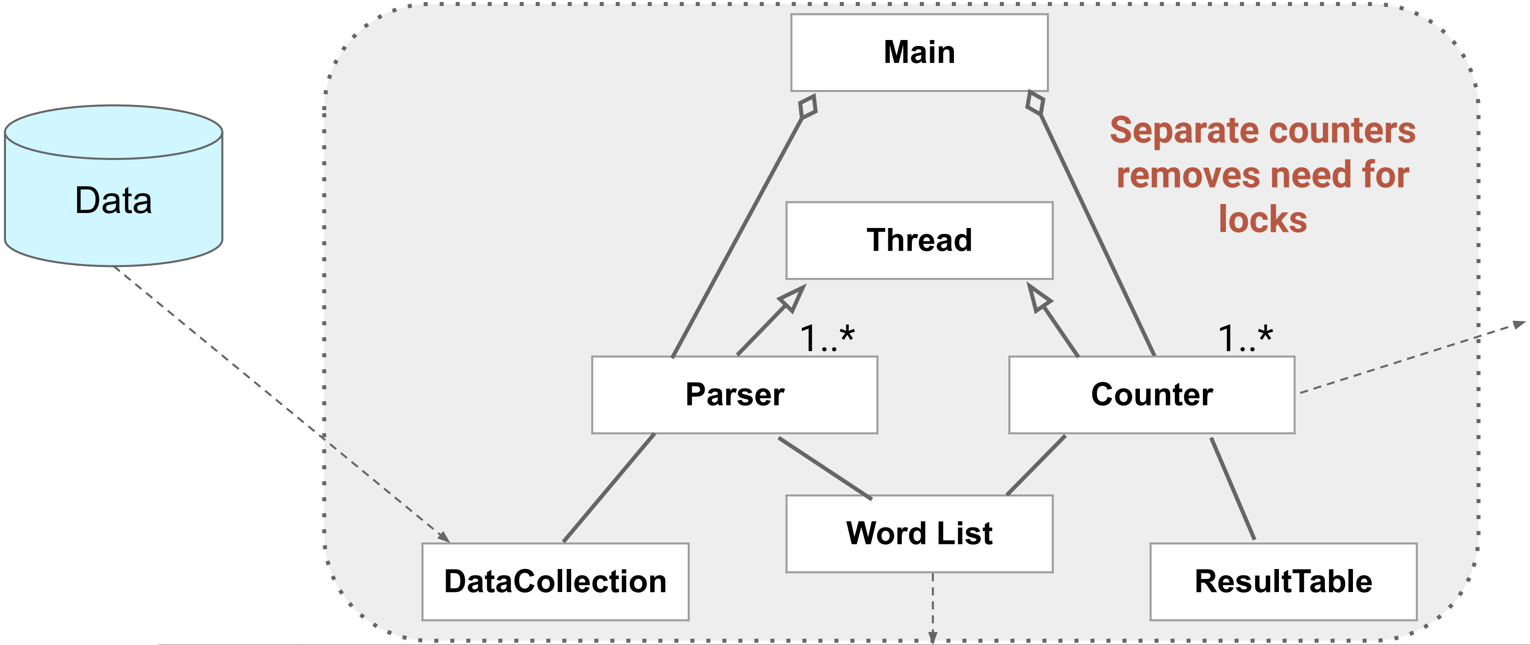
Splitting Up Our Tasks



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										

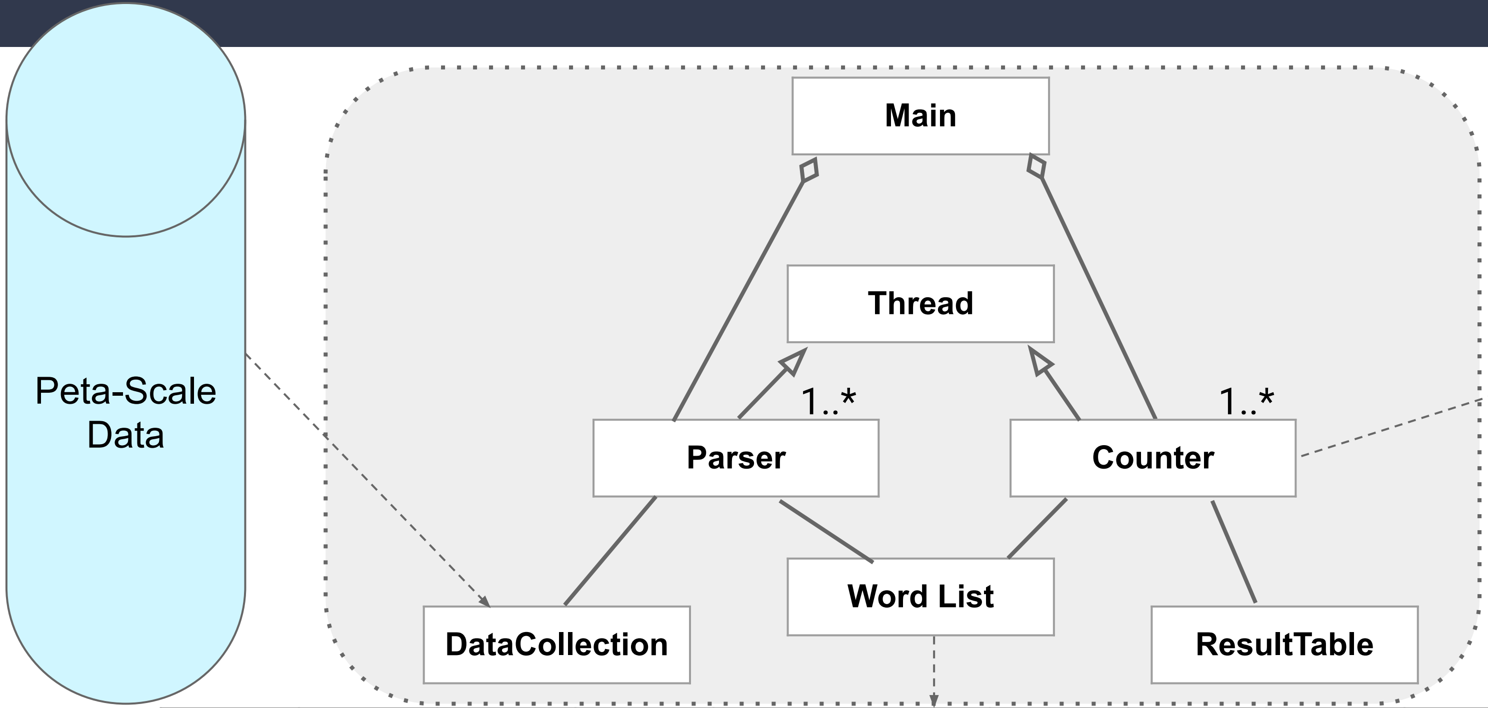
Splitting Up Our Tasks



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										

What if our Data is "Big"?



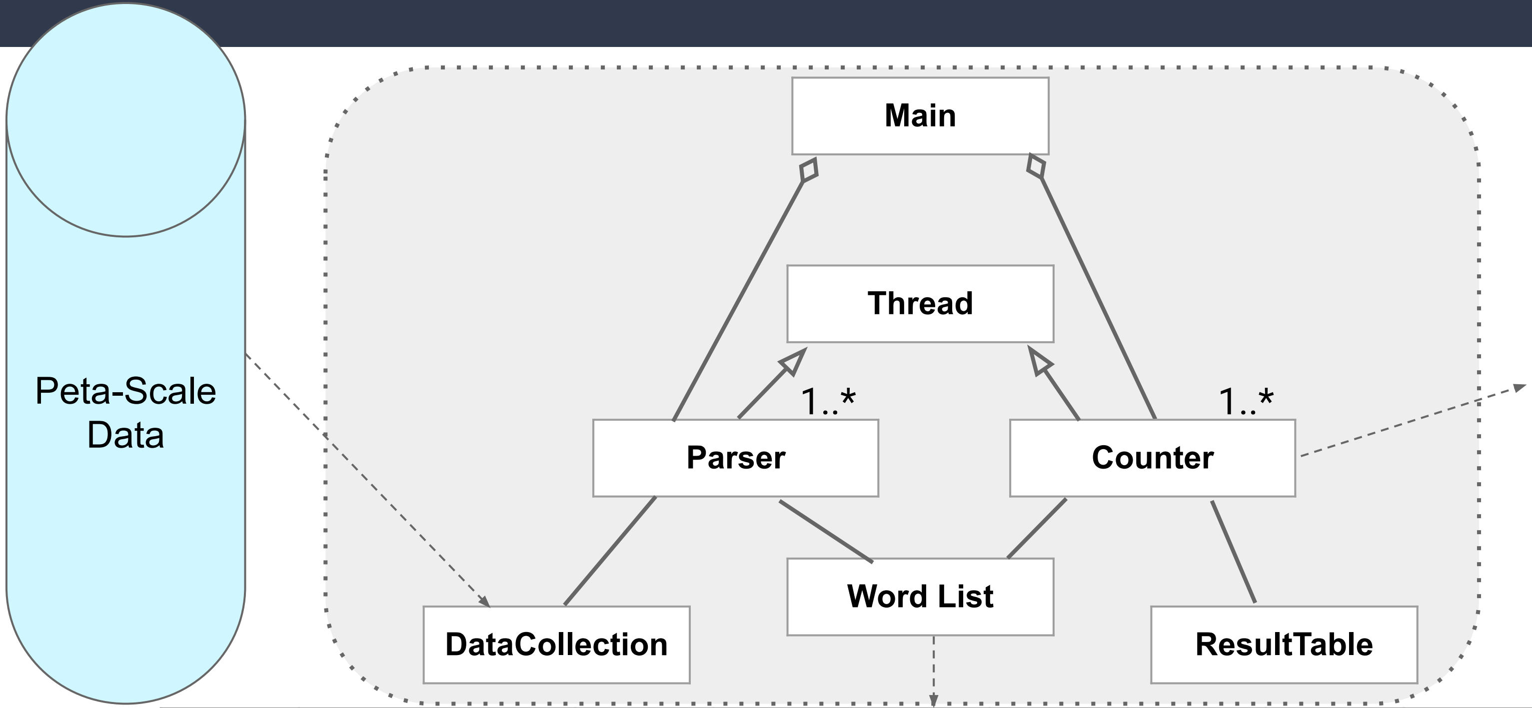
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1
…	…

Key	web	weed	green	sun	moon	land	part	web	green	...
Value										

Addressing the Scale Issue

- Eventually a single machine can't hold all of our data
  - We need a distributed file system! (HDFS)
- Large number of commodity disks; ie 1000s of disks @ 1TB each
  - **Issue:** with a failure rate of 1/1000, then at least 1 of the above disks would be down at any given time
  - Failure is the norm; need reliability
    - Replication, checksum, etc
  - Bandwidth of data transfer also becomes critical at this point
- We need to exploit parallelism afforded by splitting parsing and counting
- Move these computations to where the data is

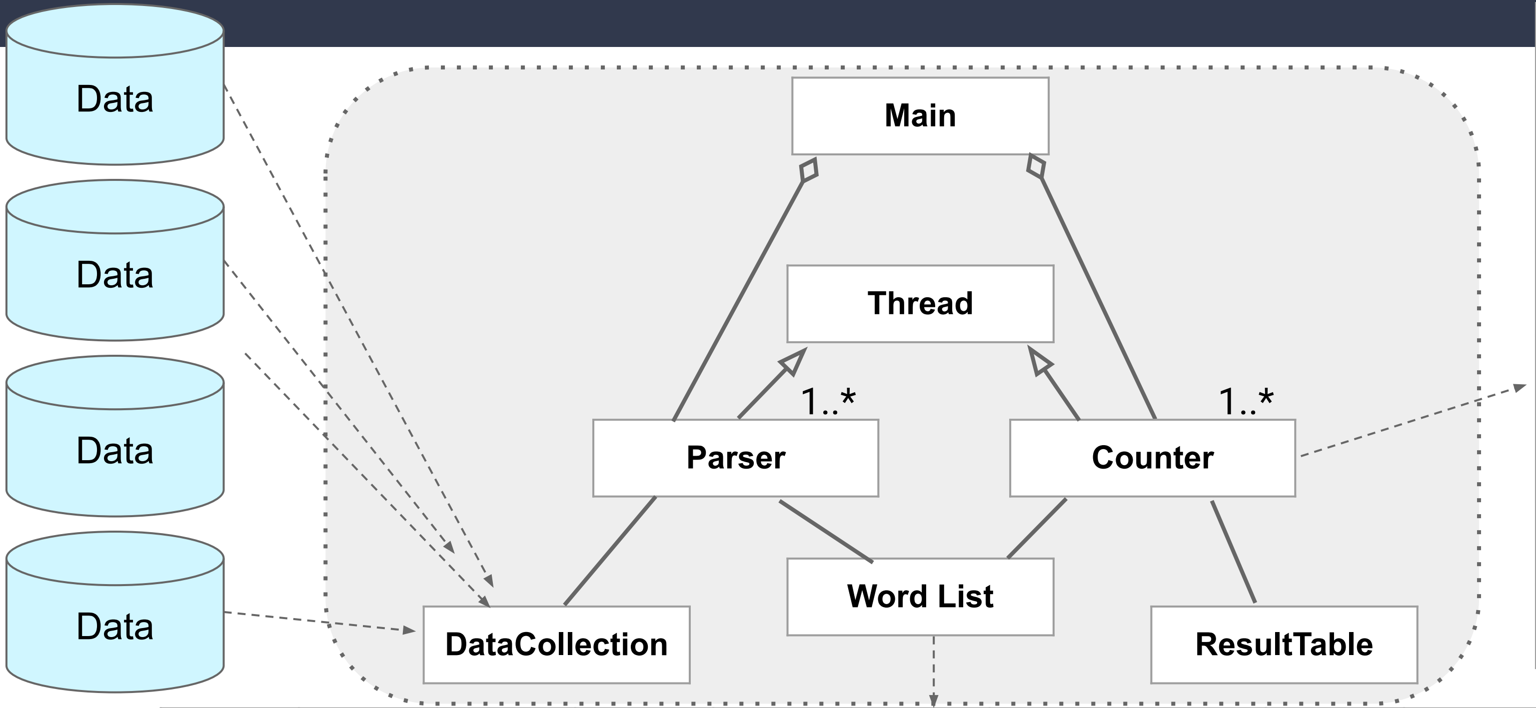
What if our Data is "Big"?



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										

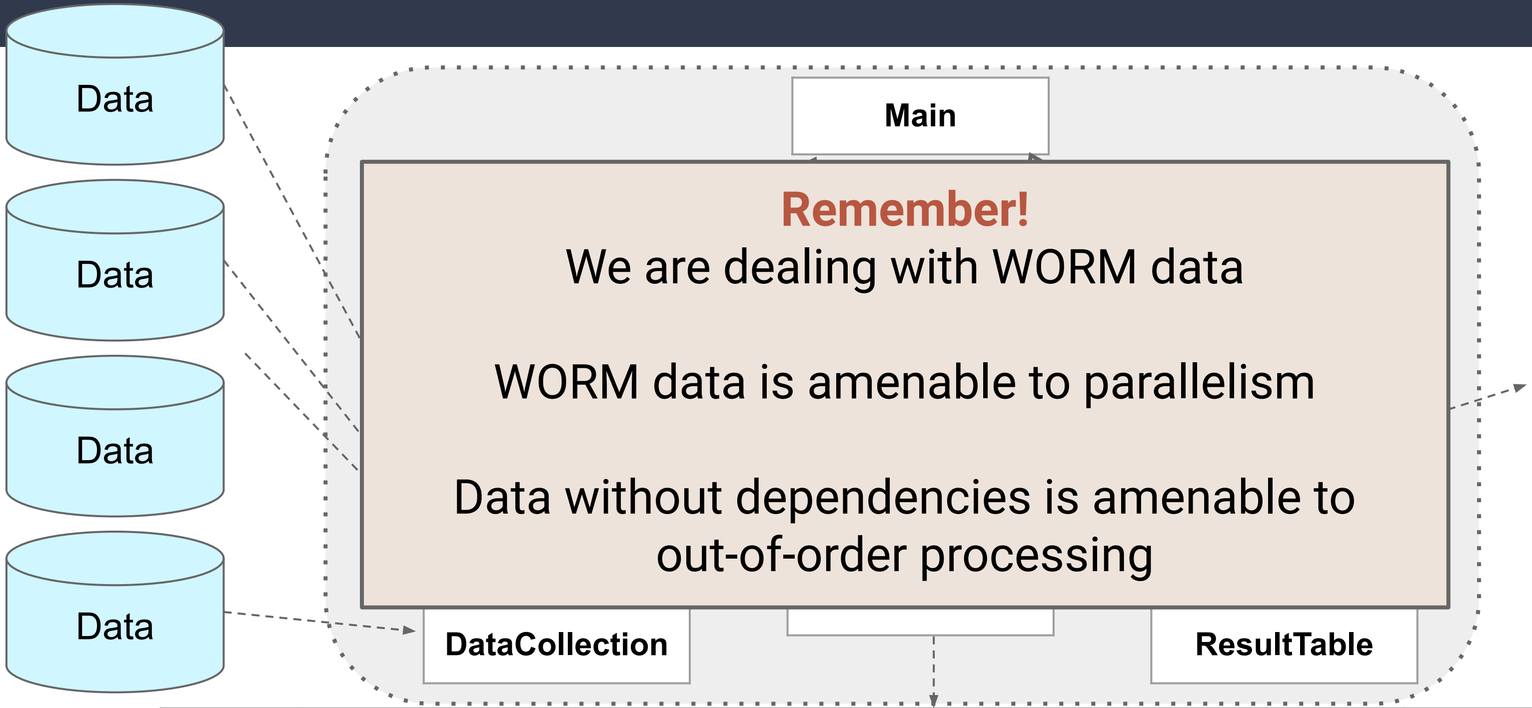
Distribute the Data



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										

Distribute the Data



Remember!

We are dealing with WORM data

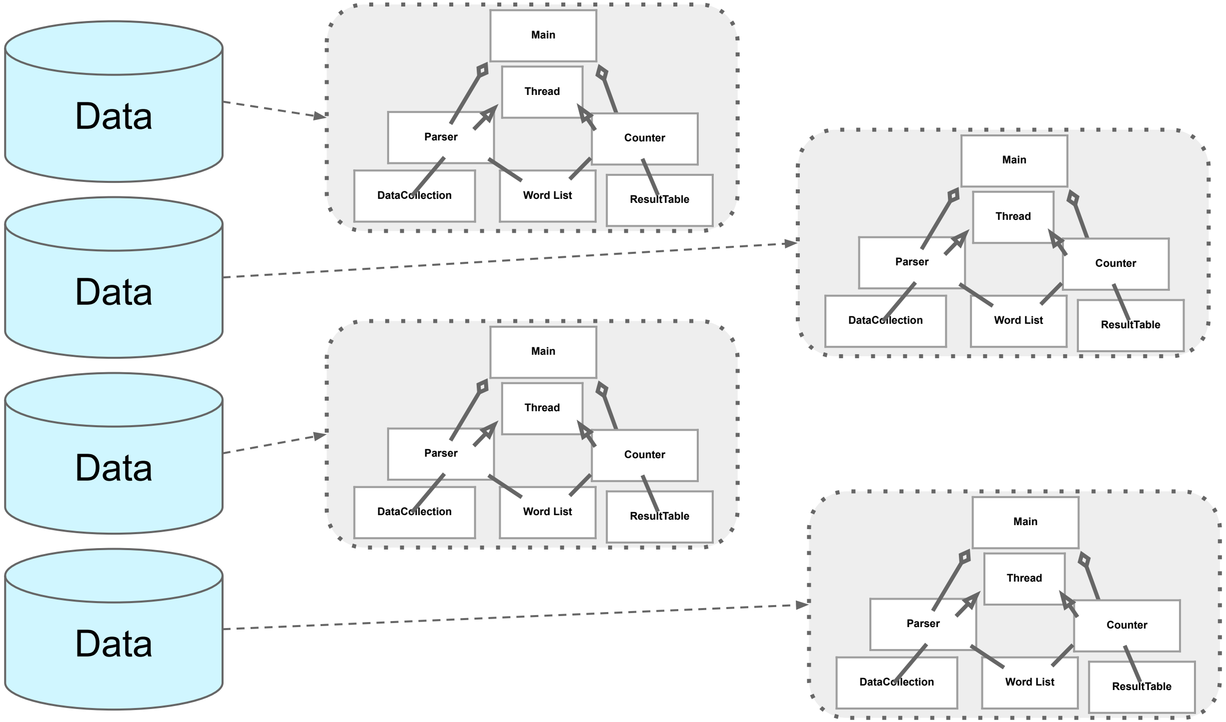
WORM data is amenable to parallelism

Data without dependencies is amenable to out-of-order processing

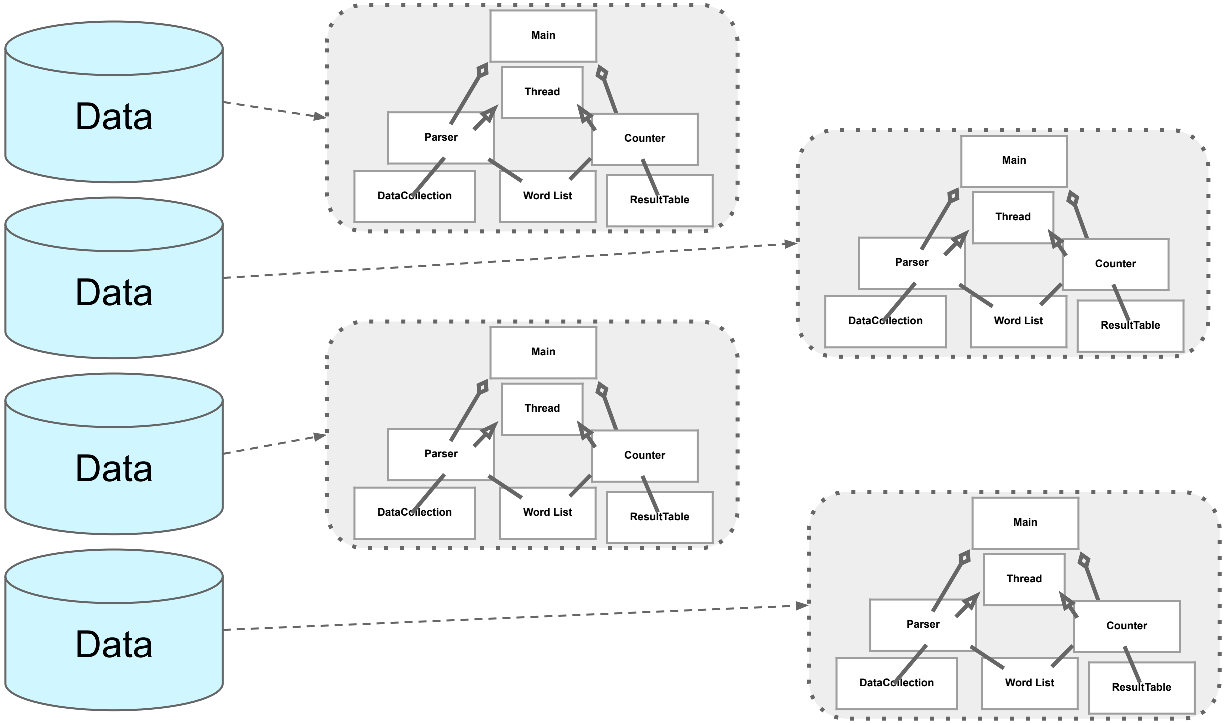
web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1
…	…

Key	web	weed	green	sun	moon	land	part	web	green	…
Value										

Divide and Conquer



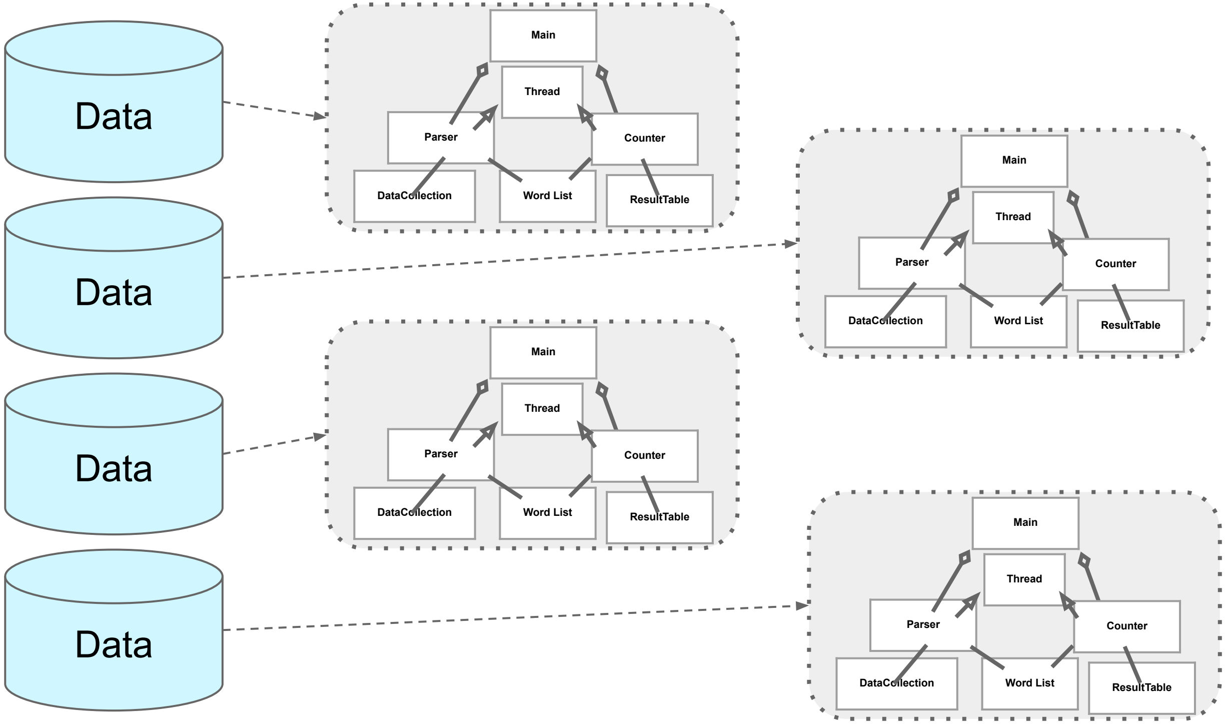
Divide and Conquer



For our example:

1. We schedule parse tasks
2. We then schedule count

Divide and Conquer



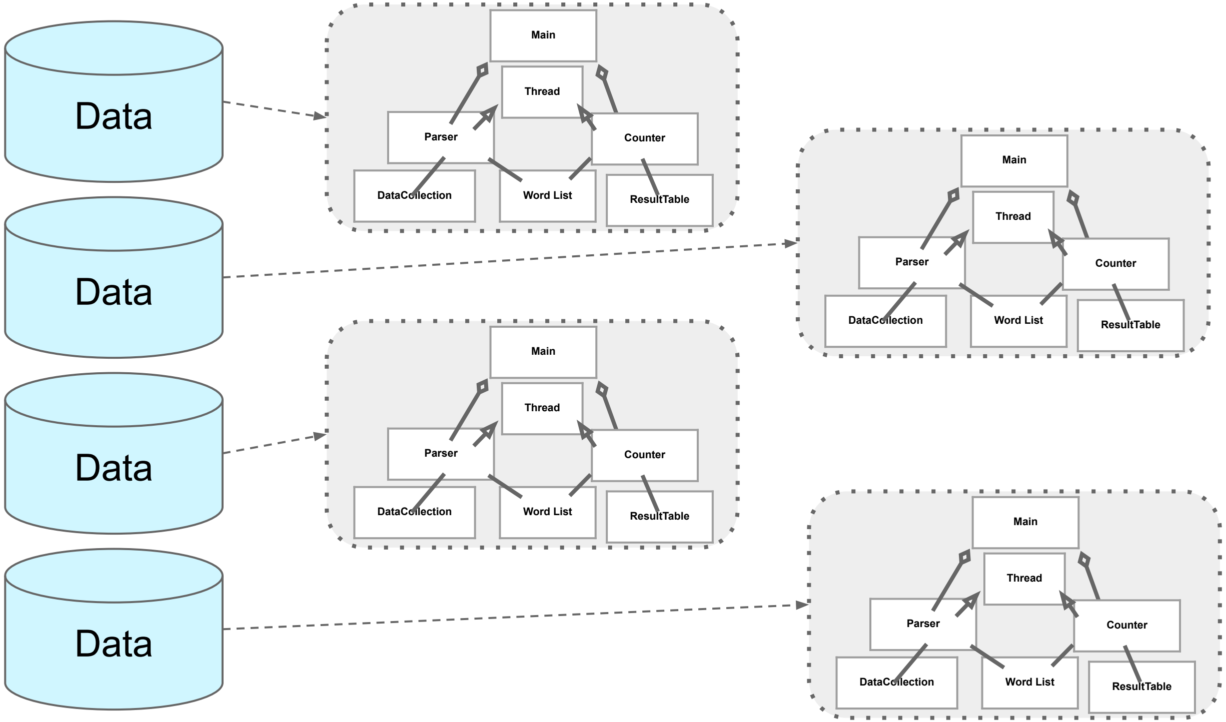
For our example:

1. We schedule parse tasks
2. We then schedule count

Let's generalize this:

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

Divide and Conquer



For our example:

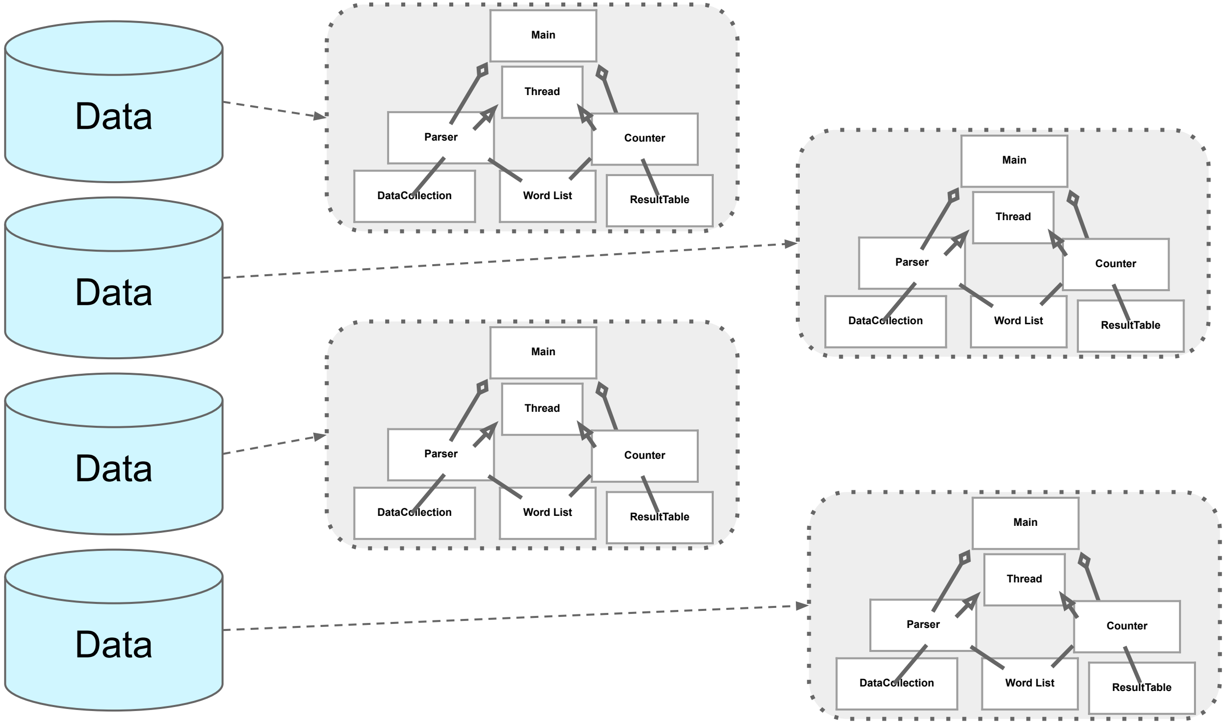
1. We schedule parse tasks
2. We then schedule count

Let's generalize this:

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

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

Divide and Conquer



For our example:

1. We schedule parse tasks
2. We then schedule count

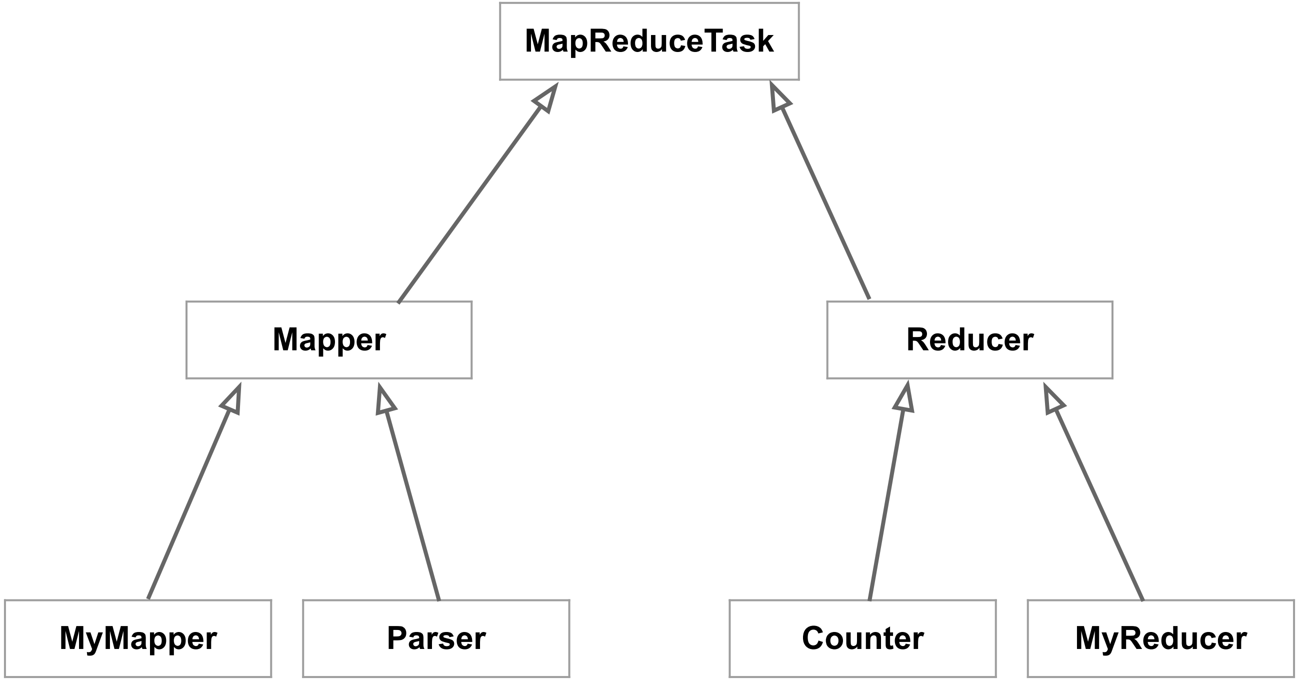
Let's generalize this:

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

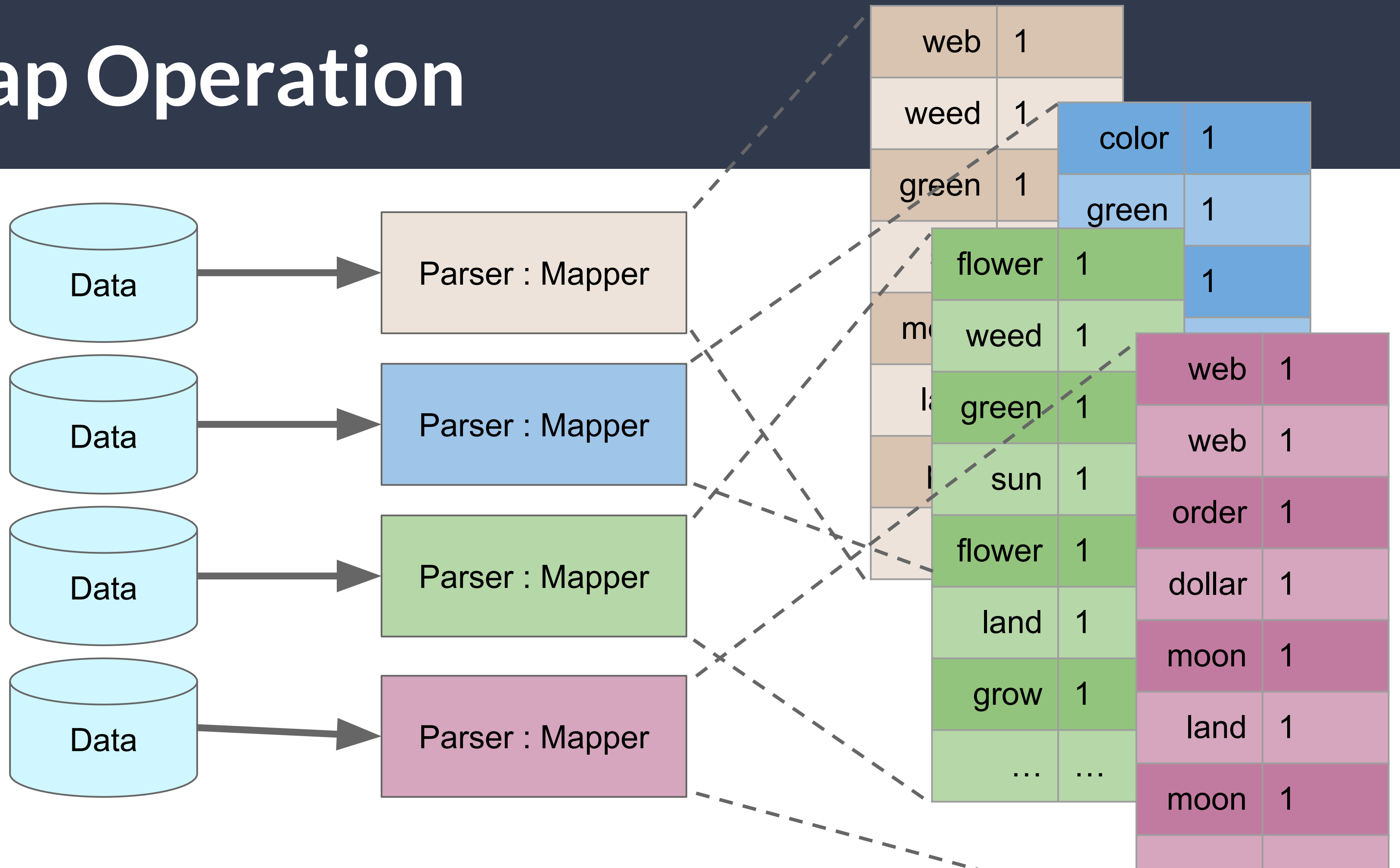
Our "count" is a reduce operation
REDUCE: <key, value> pairs reduced

RTS adds distribution, fault tolerance, replication, monitoring, load balancing, etc…

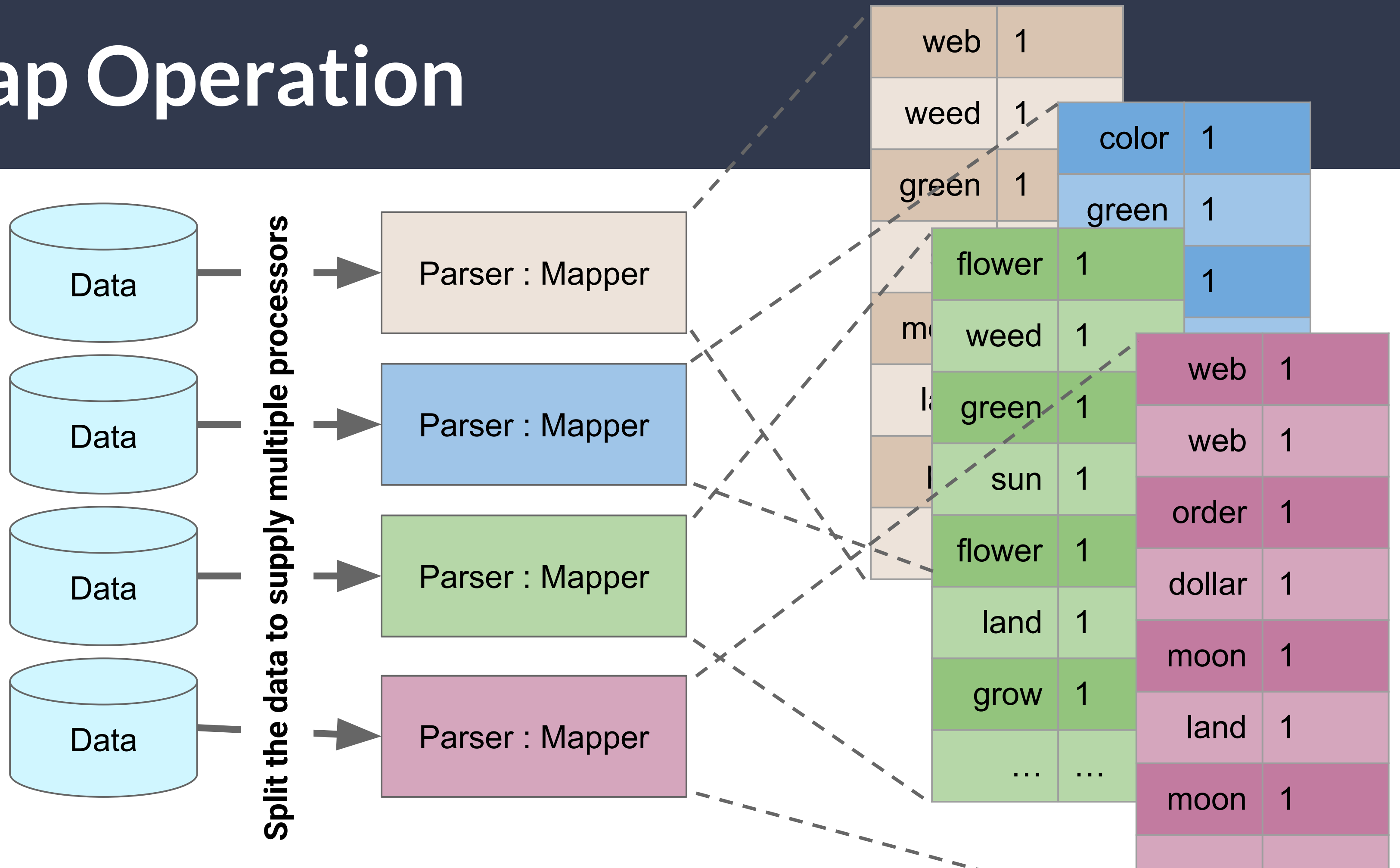
Mapper and Reducer



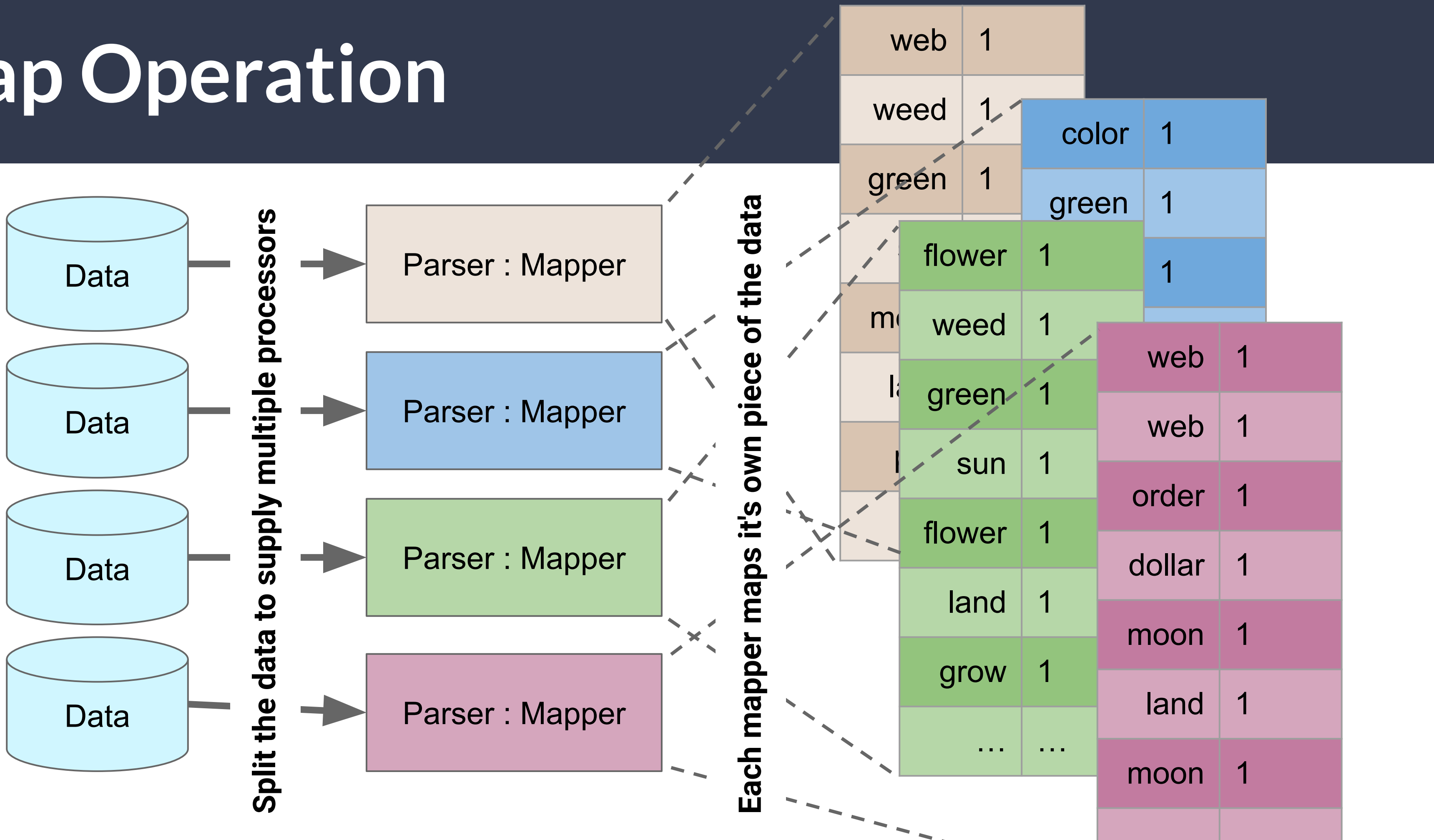
Map Operation



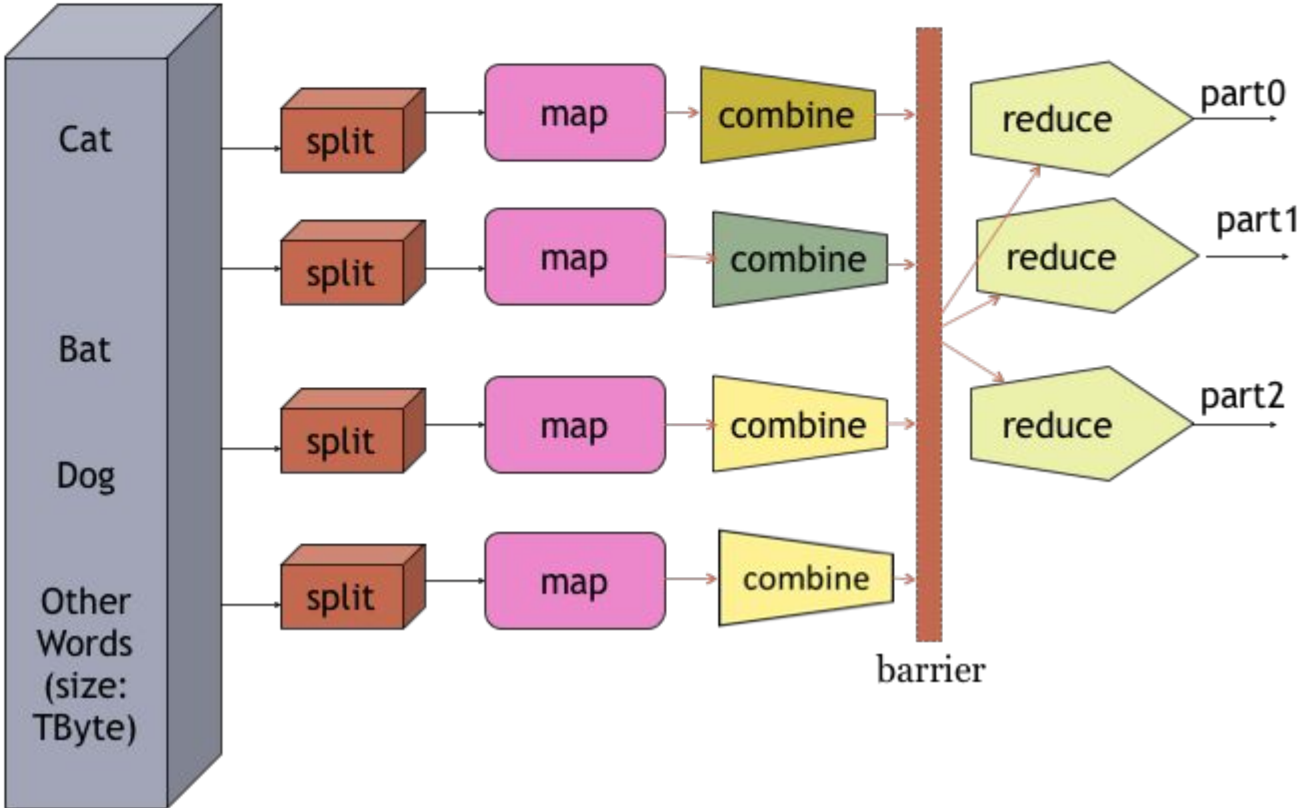
Map Operation



Map Operation



The Big Picture



MapReduce Design

- Your focus is on map, reduce, and other associated functions like combiner
  - Mapper and Reducer are classes in Java
- Configure the MR "Job" for location of these functions, location of input and output (paths), scale or size of the cluster in terms of #maps #reduces etc.
- Full job is code for the mapper, reducer, combiner, partitioner, plus job configuration. Execution framework handles everything else.
- Configuration methodology has been evolving with different versions of Hadoop

Pseudo Code

```
class Mapper
  method Map(doc d):
    for term t in doc d:
      emit(t, count = 1)
```

```
class Reducer
  method Reduce(term t, counts):
    sum = 0
    for count c in counts:
      sum = sum + c
    emit(t, count = sum)
```

Word Count Problem Revisited

This is a cat
Cat sits on a roof

The roof is a tin roof
There is a tin can on the roof

Cat kicks the can
It rolls on the roof and falls on the next roof

The cat rolls too
It sits on the can

Word Count Problem: Mappers

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

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

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

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

Word Count Problem: Shuffle to Reducers

Output of Mappers:
<this 1> <is 1> <a 1> <cat 1> <cat 1> <sits 1> <on 1> <a 1> <roof 1> <the 1> <roof 1> <is 1> <a 1> <tin 1> <roof 1> <there 1> <is 1> <a 1> <can 1> <on 1> <the 1> <roof 1> <cat 1> <kicks 1> <the 1> <can 1> <it 1> <rolls 1> <on 1> <the 1> <roof 1> <and 1> <falls 1> <on 1> <the 1> <next 1> <roof 1> <the 1> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <the 1> <can 1>

Word Count Problem: Shuffle to Reducers

Output of Mappers:
<this 1> <is 1> <a 1> <cat 1> <cat 1> <sits 1> <on 1> <a 1> <roof 1> <the 1> <roof 1> <is 1> <a 1> <tin 1> <roof 1> <there 1> <is 1> <a 1> <can 1> <on 1> <the 1> <roof 1> <cat 1> <kicks 1> <the 1> <can 1> <it 1> <rolls 1> <on 1> <the 1> <roof 1> <and 1> <falls 1> <on 1> <the 1> <next 1> <roof 1> <the 1> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <the 1> <can 1>

Input to the Reducers: delivered sorted, by key
…
<can <1,1>>
<cat <1,1,1,1>>
…
<roof <1,1,1,1,1,1>>
…

Word Count Problem: Reduce

Reduce (sum in this case) the values:

…

<can 2>

<cat 4>

…

<roof 6>

…

More on MapReduce

- All mappers work in parallel
- Barriers enforce that all mappers complete before reducers start
- Mappers and Reducers execute on same machine
- Jobs can be configured to have other combinations besides mapper/reducer.
- Mappers and reducers can have side effects
  - Allows sharing between iterations

What is it used for?

- Googe uses it (we think) for wordcount, adwords, pagerank, indexing
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining
- Facebook uses it for various things, ie demographic information
- Financial services use it for analytics
- Astronomy: Gaussian analysis for location extra-terrestrial objects
- Expected to play a critical role in semantic web and web3.0

Summary

- Very large scale WORM data (allows for parallelism)
- Map and Reduce are the main operations → simple code
- There are other supporting operations we'll look at later
- Operations are executed near the data
- Commodity hardware and storage
- RTS takes care of splitting and moving data
- Requires a distributed file system (HDFS) and runtime (Hadoop runtime)