BROWN
Computer Science

CS1951A: Data Science

Lecture 8: Map Reduce

Lorenzo De Stefani

Spring 2022

Outline

- MapReduce: motivation and main idea
- The MapReduce workflow
- Mappers and Reducers
- Example: counting words in documents
- DIY joins
- Mapping and reducing on multiple rounds
- Graph manipulation examples

Motivation

- Datasets can be extremely large
  - Tens to hundreds of terabytes
- Traditional programming is serial
  - Strong intrinsic limit on scalability
- Parallel programming
  - Break processing into parts that can be executed concurrently on multiple processors

Challenges

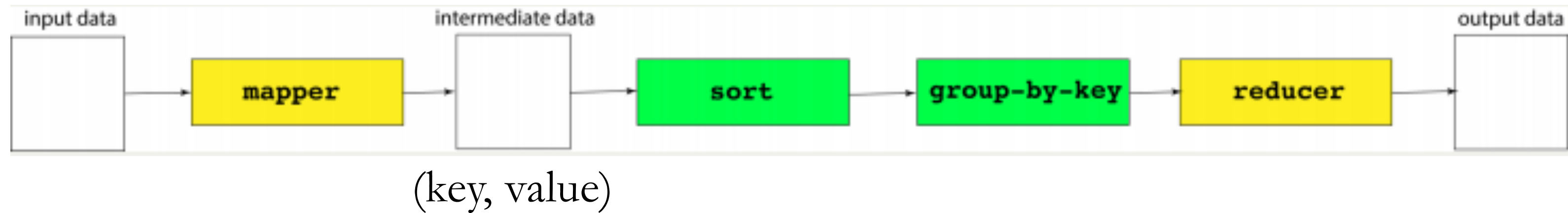
- Identify tasks that can run concurrently and/or groups of data that can be processed concurrently
- Not all problems can be parallelized
- Multiple possible parallel architectures/hardware
  - How to organize computations on this architecture?
- Different programming models
  - Message Passing
  - Shared Memory
  - Distributed memory
- The programmer shoulders the burden of managing concurrency and coordination

Map Reduce

MapReduce is a parallel, distributed programming model and implementation infrastructure used to process and manage large data sets.

- Core idea:
  - map the dataset into a collection of pairs and then
  - reduce over all pairs with the same key
- Simple Programming interface: Map + Reduce
  - The map component of a MapReduce job typically parses input data and distills it down to some intermediate result.
  - The reduce component of a MapReduce job collates these intermediate results and distills them down even further to the desired output

Map Reduce Workflow

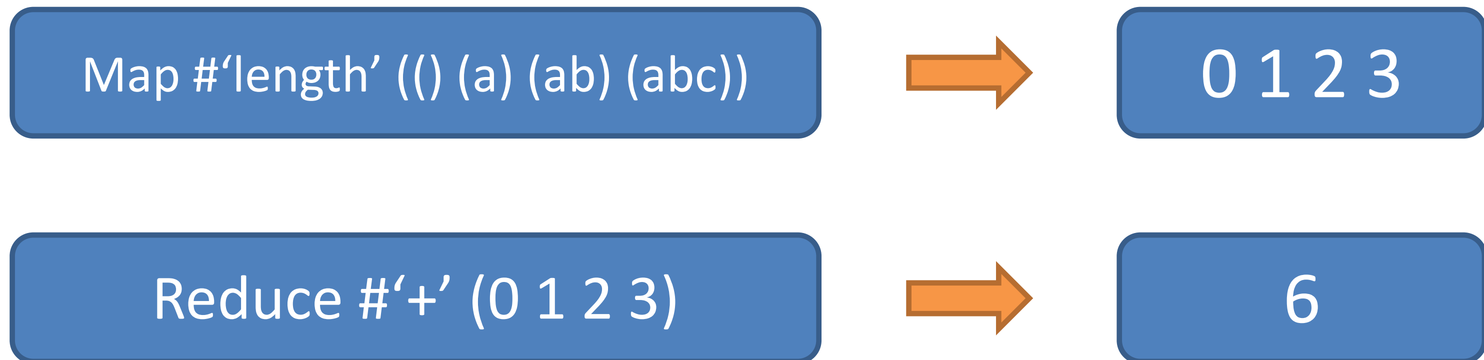


- The processes shaded in yellow are programs specific to the data set being processed
- The processes shaded in green are present in all MapReduce pipelines.

- We (the programmer) need to create the map and the reduce script
- All the rest is handled by the Amazon Elastic MapReduce framework (ERM)

Map Reduce

- Distributed implementation that hides all the messy details
  - Fault tolerance (via distributed storage)
  - I/O scheduling
  - Parallelization and coordination
- Functional programming language Inspired by map and reduce functions in Lisp



Programing Model

The programmer only needs to specify two functions:

- Map Function

map (in_key, in_value) -> list(out_key, intermediate_value)

  - Processes input key/value pair
  - Produces set of output key/intermediate value pairs

- Reduce Function

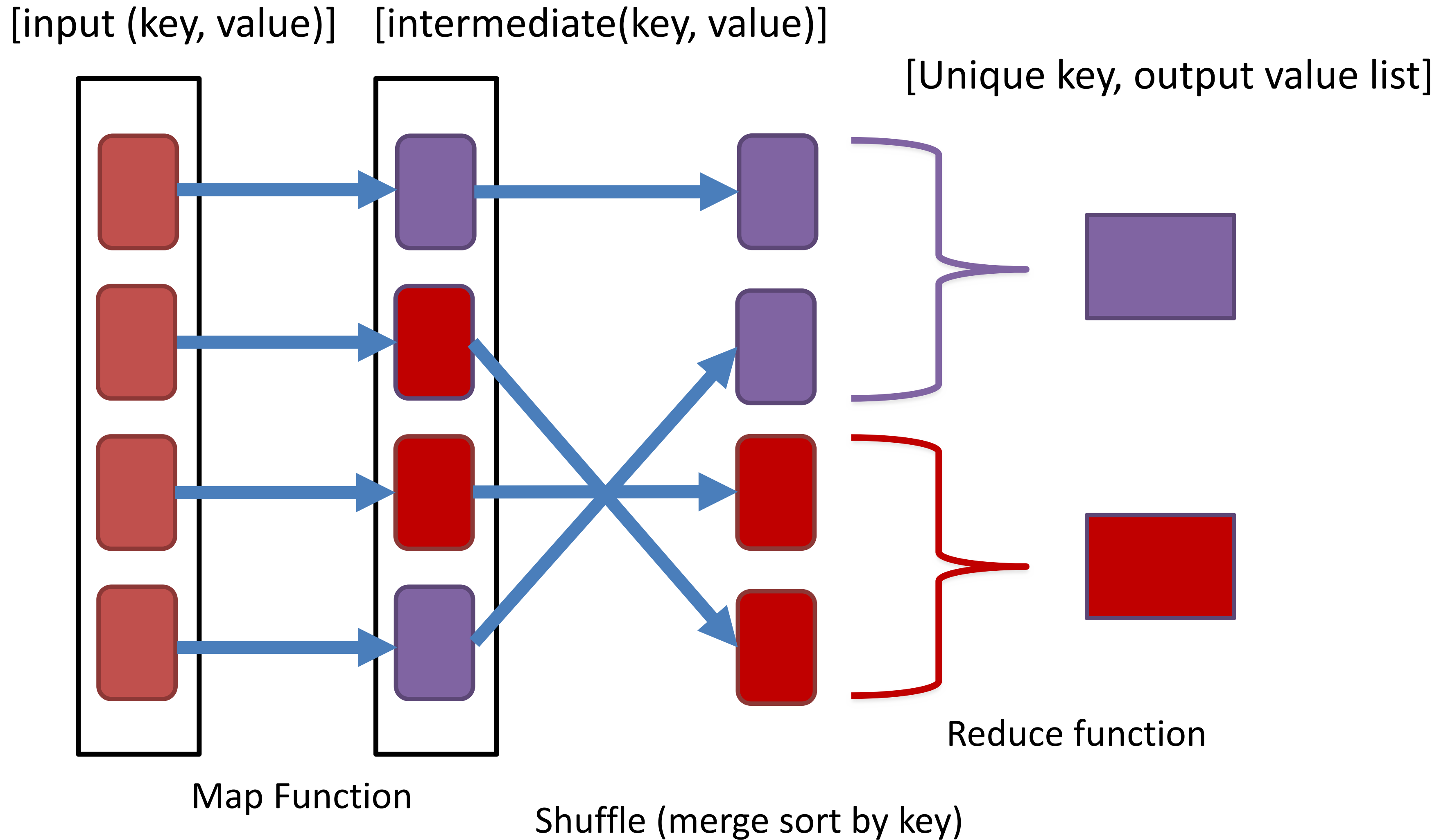
reduce (out_key, intermediate_value) -> list(out_value)

  - Process intermediate key/value pairs
  - Combines intermediate values per unique key
  - Produce a set of merged output values(usually just one)

Map Reduce paradigm

- Very general approach can be used for many applications
- One "master" scheduler which assigns tasks (mapping or reducing) to machines
- No shared state between machines
  - Massively parallelizable
- Tolerates very high failure rates on workers

MapReduce model



MapReduce workflow

- When we start a MapReduce workflow, the framework (i.e., the master) will split the input into segments passing each segment to a different machine
- Each machine then runs the map script on the data segment assigned to it
  - The map script takes the input data and maps it to a list of <key, value> pairs
  - The map script does not do any aggregation!
  - You can think of if as a parser that transforms the data into <key, value> pairs which can be processed by the reducer

MapReduce workflow

- The resulting pairs are then shuffled in the machines in the sort phase
  - Pairs with the same key are grouped into the same machine
- The reduce script takes as input a collection of <key, value> pairs and "reduces" (aggregates) them according to the specifications.
- Finally, the results of the reducers are combined in a final result.

Data Flow

- Input and final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local file system of map and reduce workers
- Output is often input to another map reduce task
- Some technical details will depend on implementation

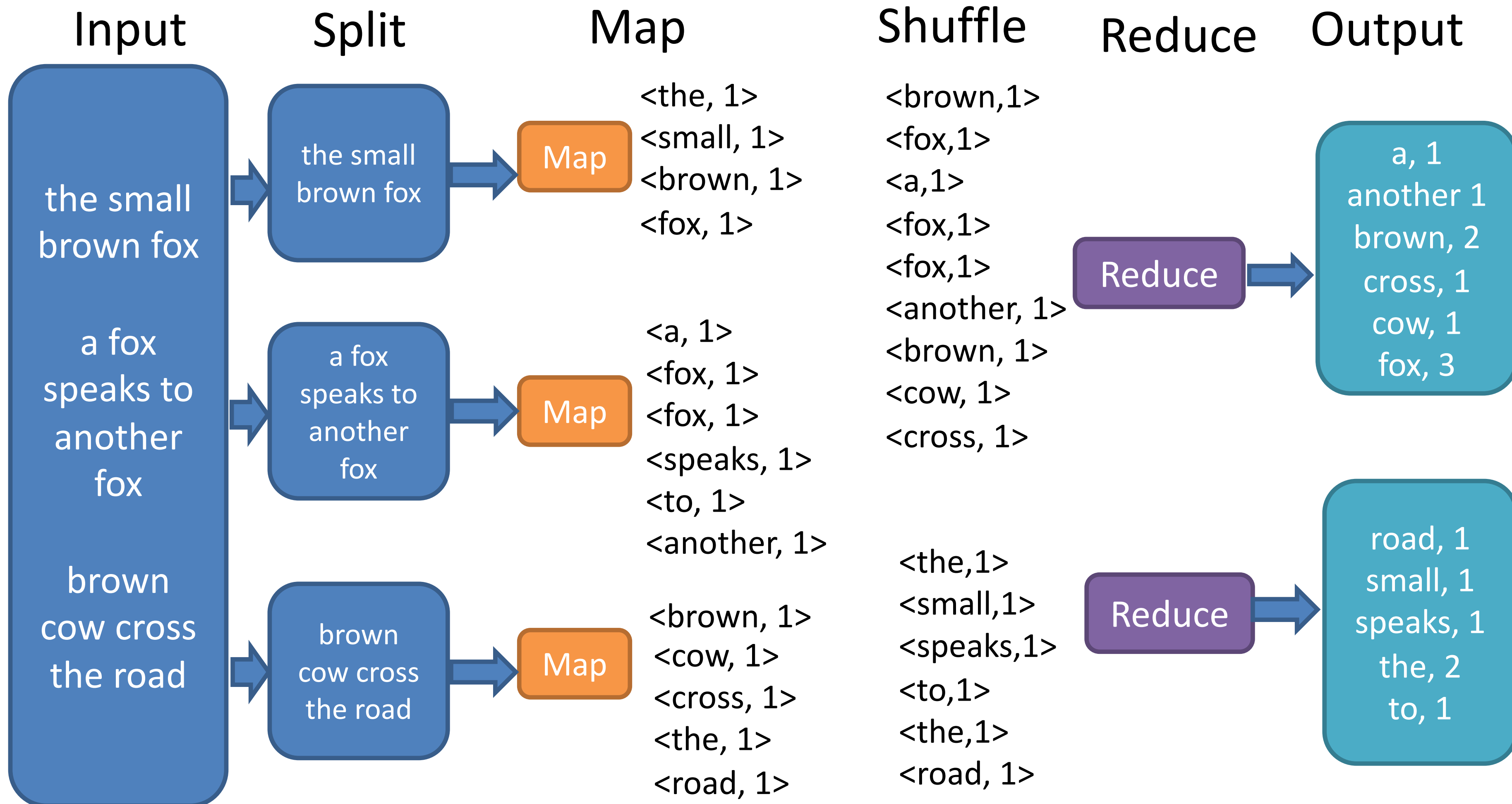
Example: Word Count

- Suppose we have a large corpus of text documents
- We want to count the number of times each distinct word appears in the each document and/or in the corpus
- Sample application: analyze web server logs to find popular URLs

Example: Word Count

- Simple idea
  - We split the documents into multiple machines,
  - For each document we enumerate the words in the documents
  - We shuffle the words so that all instances of the same word are stored in the same machine
  - We aggregate the words to obtain the count
- The above captures the essence of MapReduce
  - Great thing is it is easily parallelizable
  - Naïve parallelism in the breaking down and aggregation

Example: Word Count



Word Count using MapReduce

- The code presented here should be interpreted as a guideline pseudocode

```
map(key, value):
// key: document name; value: text of 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(result)
```

Setting up the workflow

- We need a ”main” function to initialized the Map Reduce operations and pass the input

```
//define your pipeline
Table<String, String> table = read(table_path)
Table<String, Int> output =
    table.MapFn().ReduceFn();
write(output)
```

output

Word	Count
hello	2
world	4
oh	1
hi	1
there	2

Input: table of docs

DocID	Text
1	hello world
2	oh hi there world
3	why hello there , world
4	world ! how the hell are ya ?

Word count example

```
// enumerate occurrences of each word with count of 1
def MapFn: (String, String) -> (String, Int) {
         for w in input.split(){
                  emit(w, 1);
         }
}

// sum the total counts of each word
def ReduceFn:(String, List(Int)) -> (String, Int){
         sum = 0;
         for c in input.value(){
                  sum += c;
         }
         emit(input.key(), sum);
}

// define your pipeline
def main() {
         Table<String, String> table =read(table_path);
         Table<String, Int> output = table.MapFn().ReduceFn();
         write(output)
}
```

Constraints on the mapper and reducer

- The mapper must be equivalent to applying a deterministic pure function (i.e., a mathematical function) to each input independently
- The reducer must be equivalent to applying a deterministic pure function to the sequence of values for each key

Benefits of the approach

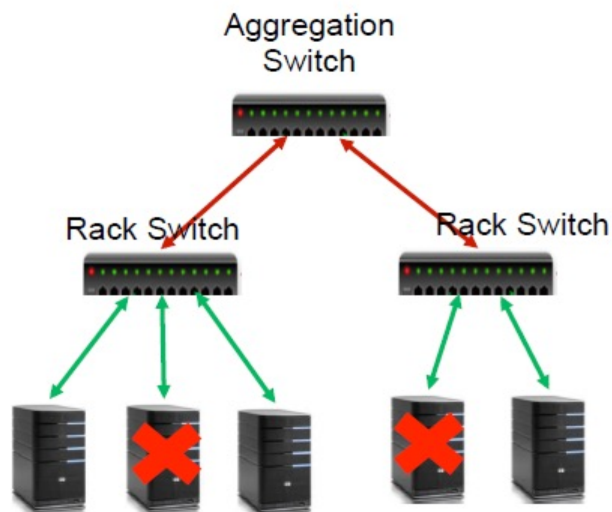
- When a program contains only pure functions, expressions can be evaluated in any order, lazily, and in parallel
  - Consistent results regardles of how computation is partitioned
- Referential transparency: a call expression can be replaced by its value (or vice versa) without changing the program
  - Re-computation and caching of results, as needed.

Coordination

- Master assign tasks to the available mappers and reducers
- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its intermediate files, one for each reducer
  - Master ***incrementally*** pushes this info to reducers

Fault Tolerance

Master pings workers periodically to detect failures



- Map worker failure
  - Completed or in-progress tasks are reset to idle
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce Task is aborted and client is notified

- Reset tasks are rescheduled on another machine

Other MapReduce Functions

- Sort
- Unique
- Sample
- First
- Filter
- Join

- Joins are usually computed "under the hood" by most MapReduce implementations (like in SQL)
- But you can imagine having to do them yourself…

Joins

FACTS

Subject	Predicate	Object
Barack Obama	won	the electoral vote
Kamala Lopez	wrote	an op-ed for HuffPo
Charles Mingus	wrote	jazz
Barack Obama	opposed	the appropriations bill
Barack Obama	listens to	jazz

CATEGORIES

Category	Entity
Person	Barack Obama
Person	Kamala Lopez
Person	Charles Mingus
Huffington Post Columnists	Barack Obama
Huffington Post Columnists	Kamala Lopez
US Presidents	Barack Obama
Jazz Composers	Charles Mingus
Harvard Law School Graduate	Barack Obama

Joins

FACTS

Subject	Predicate	Object
Barack Obama	won	the electoral vote
Kamala Lopez	wrote	an op-ed for HuffPo
Charles Mingus	wrote	jazz
Barack Obama	opposed	the appropriations bill
Barack Obama	listens to	jazz

CATEGORIES

Category	Entity
Person	Barack Obama
Person	Kamala Lopez
Person	Charles Mingus
Huffington Post Columnists	Barack Obama
Huffington Post Columnists	Kamala Lopez
US Presidents	Barack Obama
Jazz Composers	Charles Mingus
Harvard Law School Graduate	Barack Obama

```
select * from Facts, Categories
where Subject == Entity
GroupBy Subject, Predicate, Object
```

Desired output:

Subject	Predicate	Object	Categories
Barack Obama	won	the electoral vote	Person, US Presidents, Huffington Post Columnists, Harvard Law School Graduate
Kamala Lopez	wrote	an op-ed for HuffPost	Person, Huffington_Post_Columnists, Actor
...	....	.....	....

Ideas?

- For the mapper: Break down the tables!
  - Generate items corresponding to the lines of the table
  - Only include the attributes selected by the join!
  - The parameter used as join condition will become the “key” of of the elements (subject/entity in this case)
  - We need to account for different possible relations and break them down accordingly

- For the reducer:
  - We group up categories with the same entity
    - This applies to the elements generated from “mapping ” the CATEGORIES table
  - For each element generated from the FACT table, we associated the categories corresponding to the same entity

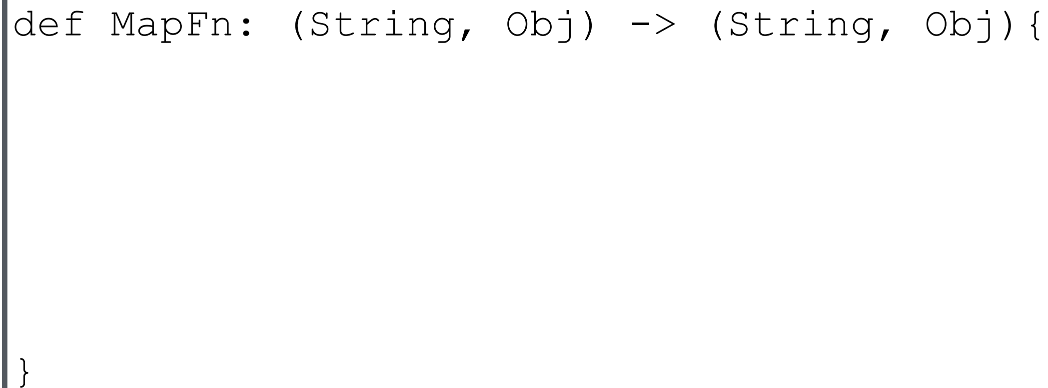
DIY Joins

Facts

Subject	Predicate	Object

Categories

Category	Entity




```
def ReduceFn: (String, Obj) -> (Fact, List(String)){



}
```

DIY Joins

Facts

Subject	Predicate	Object

Categories

Category	Entity

```
def MapFn: (String, Obj) -> (String, Obj){
    v = input.value()
    if (typeof(v) == Fact) {
        emit(v.Subject, v)
    } else {
        emit(v.Entity, v)
    }
}
```

```
def ReduceFn: (String, Obj) -> (Fact, List(String)){

}
```

DIY Joins

Facts

Subject	Predicate	Object

Categories

Category	Entity

```
def MapFn: (String, Obj) -> (String, Obj){
    v = input.value()
    if (typeof(v) == Fact) {
        emit(v.Subject, v)
    } else {
        emit(v.Entity, v)
    }
}
```

```
def ReduceFn: (String, Obj) -> (Fact, List(String)){
    all_cats = []; all_facts = []
    for v in input.value(){




    }

}
```

DIY Joins

Facts

Subject	Predicate	Object

Categories

Category	Entity

```
def MapFn: (String, Obj) -> (String, Obj){
    v = input.value()
    if (typeof(v) == Fact) {
         emit(v.Subject, v)
    } else {
         emit(v.Entity, v)
    }
}
```

```
def ReduceFn: (String, Obj) -> (Fact, List(String)){
    all_cats = []; all_facts = []
    for v in input.value(){
        if (typeof(v) == Fact) {
             all_facts.append(v)
        } else {
            all_cats.append(v.Category)
        }
    }

}
```

DIY Joins

Facts

Subject	Predicate	Object

Categories

Category	Entity

```
def MapFn: (String, Obj) -> (String, Obj){
    v = input.value()
    if (typeof(v) == Fact) {
        emit(v.Subject, v)
    } else {
        emit(v.Entity, v)
    }
}
```

```
def ReduceFn: (String, Obj) -> (Fact, List(String)){
    all_cats = []; all_facts = []
    for v in input.value(){
        if (typeof(v) == Fact) {
            all_facts.append(v)
        } else {
            all_cats.append(v.Category)
        }
    }
    for f in all_facts { emit(f, all_cats); }
}
```

Multiple Reduce rounds

- Sometimes we may want to apply the shuffle and reduce more than once
- This are referred as rounds
- The number of rounds is generally used to characterize the complexity of a MapReduce algorithm
  - Local computations are fast
  - Communications between machines are the bottleneck

Exercise:

- Given a collection of documents Find the ***number of unique documents*** that each word occurs in

```
// enumerate occurrences of each word with count of 1
def MapFn1: String -> (String, Int) {
        ???
}
def ReduceFn1: (String, List(Int)) -> (String, Int) {
        ???
}
// sum the total counts of each word
def ReduceFn2: (String, List(Int)) -> (String, Int) {
        ???
}
// define your pipeline
def main() {
        Table<String, String> table = read(table_path)
        Table<String, Int> output =
        table.MapFn1().ReduceFn1().ReduceFn2();
        write(output)
}
```

Solution

```
// enumerate occurrences of each word with count of 1
def MapFn1: (String, String) -> ((String, String), Int) {
        for w in input.value().split(){
                emit((input.key(), w), 1)
        }
}

// eliminates multiple copies of the pair for each word-document pair
def ReduceFn1: ((String,String), List(Int)) -> (String, Int) {
        emit(input.key()[1], 1)
}

// sum the total counts of each word
def ReduceFn2: (String, List(Int)) -> (String, Int) {
        sum = 0
        for (w, c) in input{
                sum += c
        }
        emit(w, sum)
}

// define your pipeline
def main() {
        Table<String, String> table = read(table_path)
        Table<String, Int> output =    table.MapFn1().MapFn2().ReduceFn()
        write(output)
}
```

Document id

We just select one item from the list

In the second round each

Bonus Question

Do these two produce the same output?

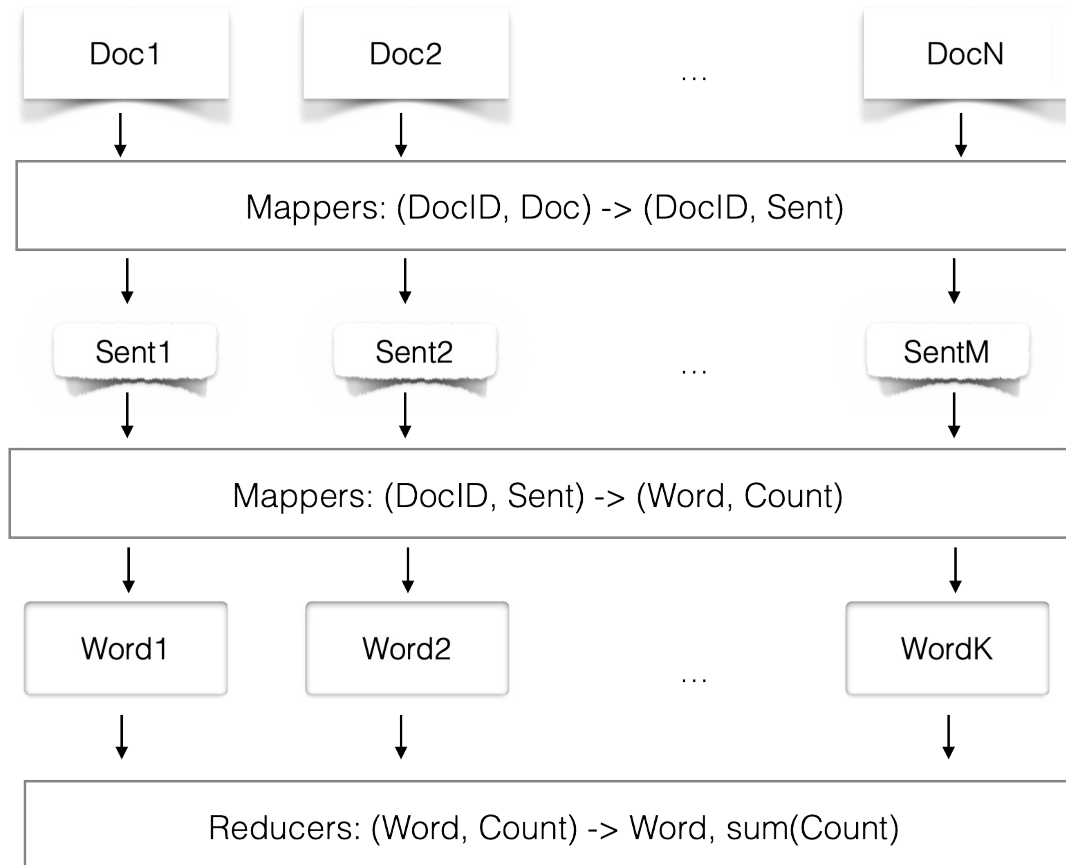
```
// enumerate occurrences
// of each word with count of 1
def MapFn1: {
 for w in input.value().split(){
  emit((input.key(), w), 1)
 }
}
def ReduceFn1: {
 emit(input.key()[1], 1)
}
// sum the total counts
// of each word
def ReduceFn2:{
 sum = 0;
 for (w, c) in input{ sum += c; }
 emit(w, sum);
}
```

```
// enumerate occurrences
// of each word with count of 1
def MapFn1: {
 for w in input.value().split(){
  emit(input.key(), w)
 }
}
def ReduceFn1: {
 for w in input.value(){emit(w, 1)}
}
// sum the total counts
// of each word
def ReduceFn2:(S, I) -> (S, I){
 sum = 0;
 for (w, c) in input{ sum += c; }
 emit(w, sum);
}
```

This code just counts the number of occurrences of the words in the documents

Multiple mapping rounds

We can also have multiple mapping rounds

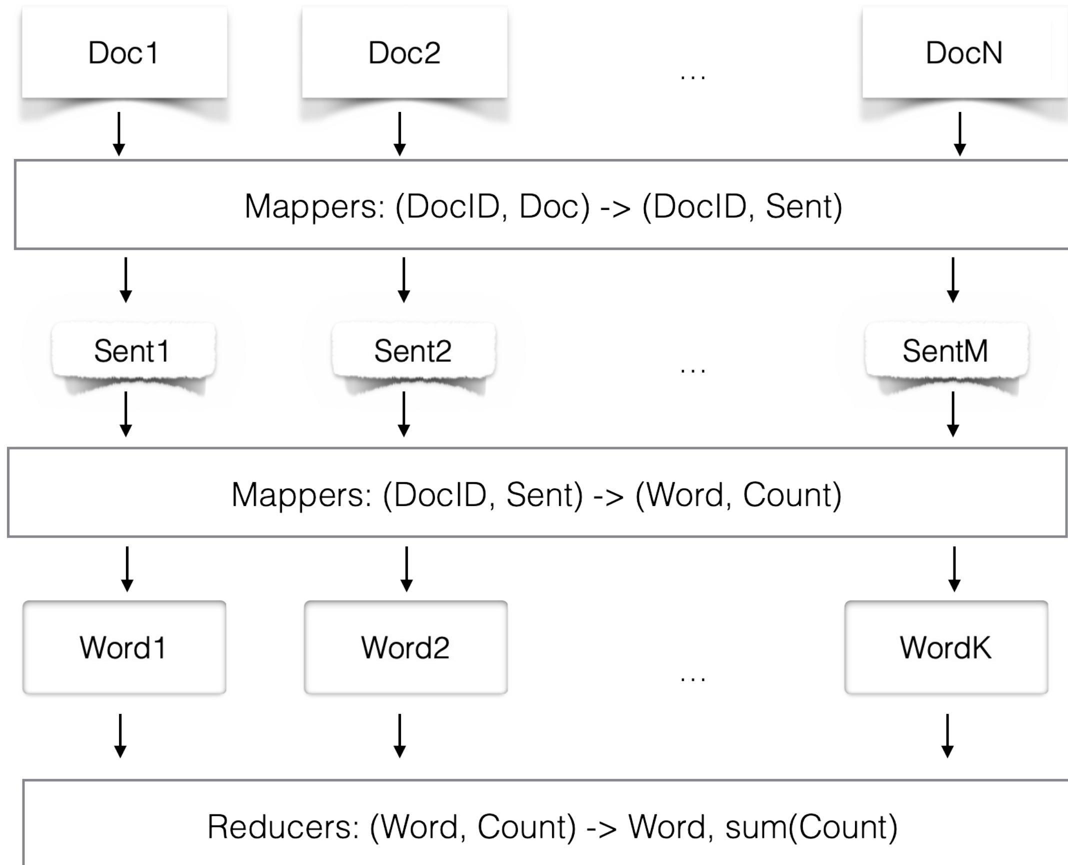


Consider again the word count problem

- We may first want to break down the documents in sentences (first map round)
- We then break down the sentences in words (second map round)
- The we aggregate in the reduce round

No aggregation occurs during MAP rounds!

How much can we parallelize?

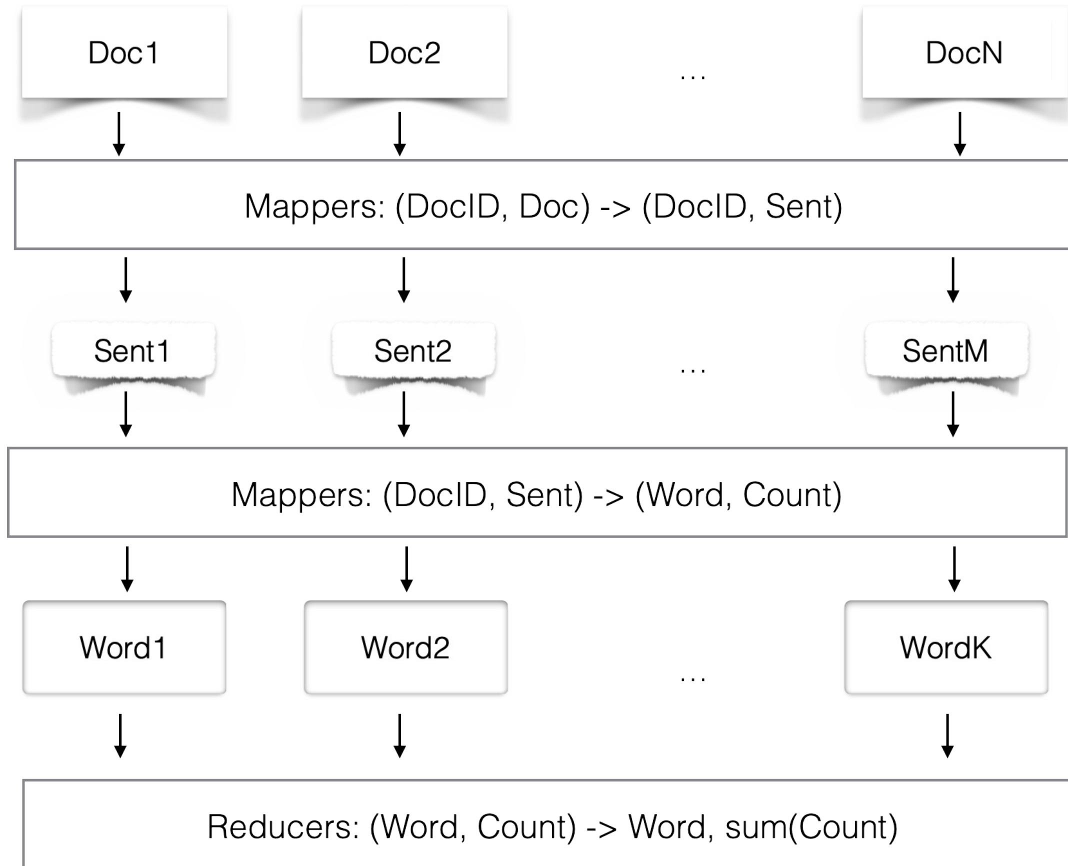


Assume we have N documents.

- How much can we parallelize the first map round?
- I.e., how many mappers can we use at most?
  a) N ✓
  b) $\sqrt{N}$
  c) Depends on the length of the documents

- Potentially we could assign every document to a single machine!

How much can we parallelize?



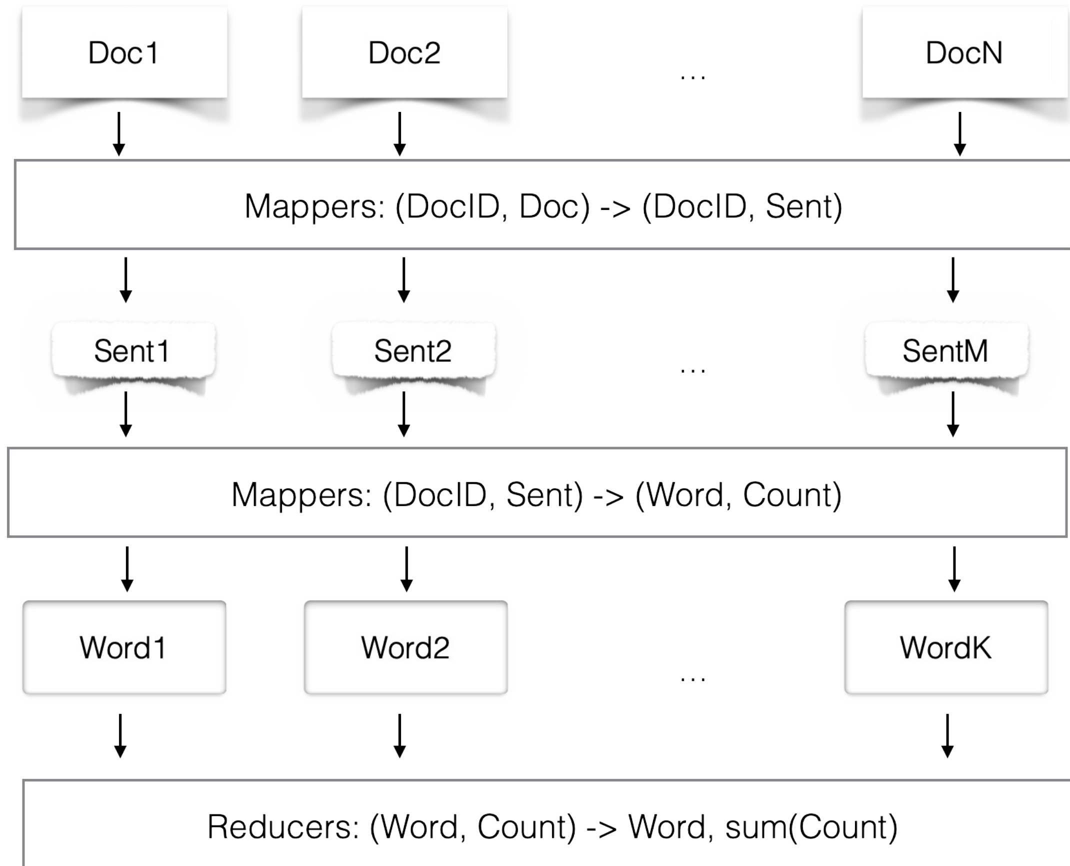
Assume we in the first round we generated M (DocID, Sent) pairs of whom D are distincy

- How much can we parallelize the second map round?
  a) N
  b) M ✓
  c) D
  d) $\min\{N, M\}$
  e) M/D
  f) $\max\{N, M\}$

- Potentially we could assign (DocID, Sent) pair to a single machine!

Mapping rounds implement naïve parallelism which is extremely scalable

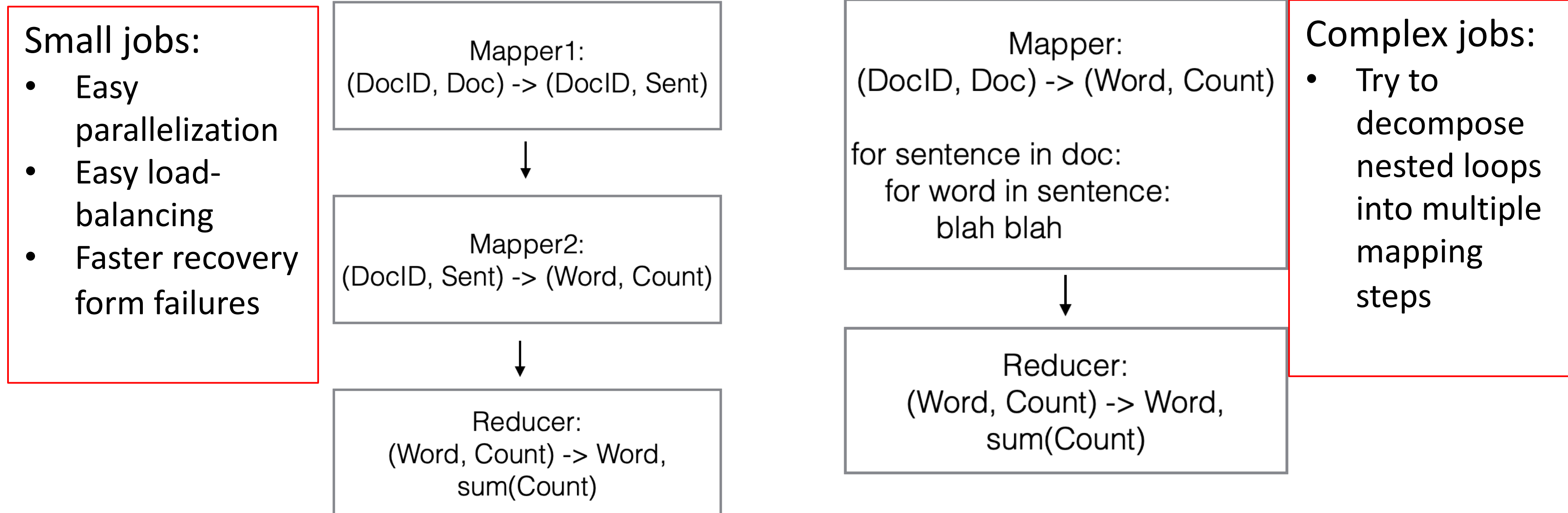
How much can we parallelize?



Assume now there are W distinct words in all of the documents

- How much can we parallelize the second map round?
- I.e., how many reducers can we use at most?
  - *a)* N
  - *b)* M
  - c) W ✓
  - d) D
  - *e)* $\min\{D, W\}$
  - f) $\max\{W, D\}$
- All pairs with the same key (i.e., the same word) must be processed by the same reducer!

Quiz!



- Do these two workflows realize the same functionality?
  - Yes!
- Which one is more likely to execute faster?
  - In general, Workflow 1 is more parallelizable and likely to be faster!
  - Still...not a clear answer!
    - Communication between machines plays a role!
    - More rounds more latency!

Example on Graph Manipulation

- Consider a directed graph described as an adjacency list
  - $(s_1: d_1, d_2, \dots, d_i)$
  - $(s_2: d_1, d_3, \dots, d_j)$
  - …
  - $(s_2: d_1, d_3, \dots, d_j)$
- Use MapReduce to construct the reverse graph
  - Give the adjacency list of the graph in which all edges are reversed

Excercise 2

- Let a market basket be a list of item purchased
  - For simplicity assume there is at most one of an item in the basket
- Given a large set of market baskets, find all frequent pairs
  - A pair is frequent if it appears in at least half of the baskets
    - Assume for simplicity that the number of baskets n is fixed and known to the reducers
- Try to find a fast implementation ☺

Ideas?

- Recall what we said about avoiding nested loops and using multiple mapping rounds
- For the mapper:
- For the reducer:

Reading

- Jeffrey Dean and Sanjay Ghemawat,

  MapReduce: Simplified Data Processing on Large Clusters

http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, **The Google File System**

http://labs.google.com/papers/gfs.html

Conclusion

MapReduce allows to achieve:

- Fault tolerance: A machine or hard drive might crash.
  - The MapReduce framework automatically re-runs failed tasks.
- Speed: Some machine might be slow because it's overloaded.
  - The framework can run multiple copies of a task and keep the result of the one that finishes first.
- Network locality: Data transfer is expensive.
  - The framework tries to schedule map tasks on the machines that hold the data to be processed.
- Monitoring: Will my job finish before dinner?!?
  - The framework provides a web-based interface describing jobs.