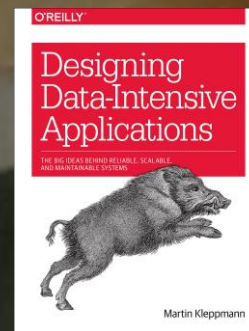


MSFT Sys Meetup



<http://>

Policy Against Harassment at ACM Activities

<https://www.acm.org/about-acm/policy-against-harassment>

MSFT Sys Meetup wants to encourage and preserve this open exchange of ideas, which requires an environment that enables all to participate without fear of personal harassment. We define harassment to include specific unacceptable factors and behaviors listed in the ACM's policy against harassment. Unacceptable behavior will not be tolerated.

Freely accessible resources



[Code](#)
[Zoom](#)
[Course](#)

[DDIA \(O'Reilly\)](#)

[Distributed System 3rd edition](#)

Calendar:

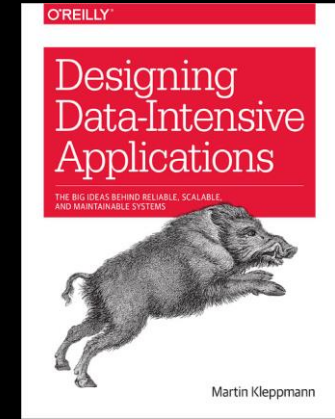
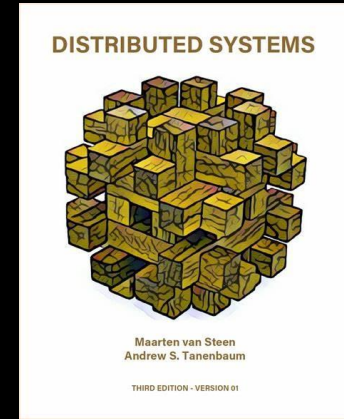
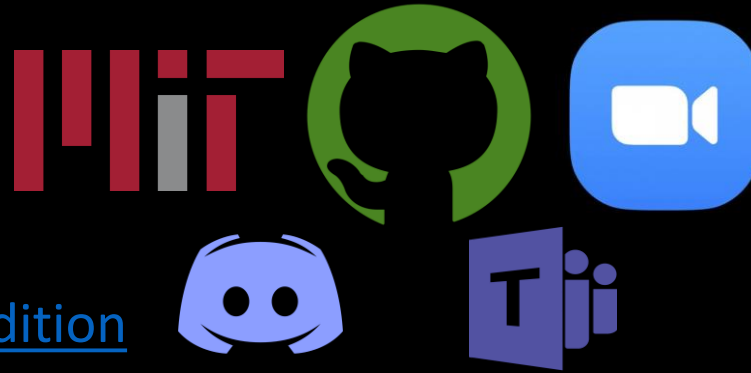
<https://docs.google.com/spreadsheets/d/1RsbGpq1cwNSmYn5hcmT8Hv5O4qssl2HXsTcG82RHVQk/edit?usp=sharing>

(Internal) [Teams](#): g078pwd

(Public) [Discord](#)

(Public) WeChat: add mossaka or Lin1991Wen

YouTube: <https://www.youtube.com/playlist?list=PL1voNxn5MODMJxAZVvgFHZ0jZ-fuSut68>



The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free?

--Alan Kay, in interview with *Dr Dobb's Journal* (2012)




6.824



Lecture 1 & MapReduce

What is distributed system?

- (Steen) A distributed system is **a collection of autonomous computing elements** that appears to its users as **a single coherent system**
- (Morris) ...the core of it, (distributed system) is a **set of cooperating computers** that are communicating with each other over network to **get some coherent task done.**
- Examples:
 - Storage for big websites
 - Big data computations (MapReduce)
 - P2P file sharing
 - Bitcoin



“You should try
everything else before
you try building
distributed systems,
because they are not
simpler.”

- Robert

~~Strong~~
Weak



Why do people build distributed systems?

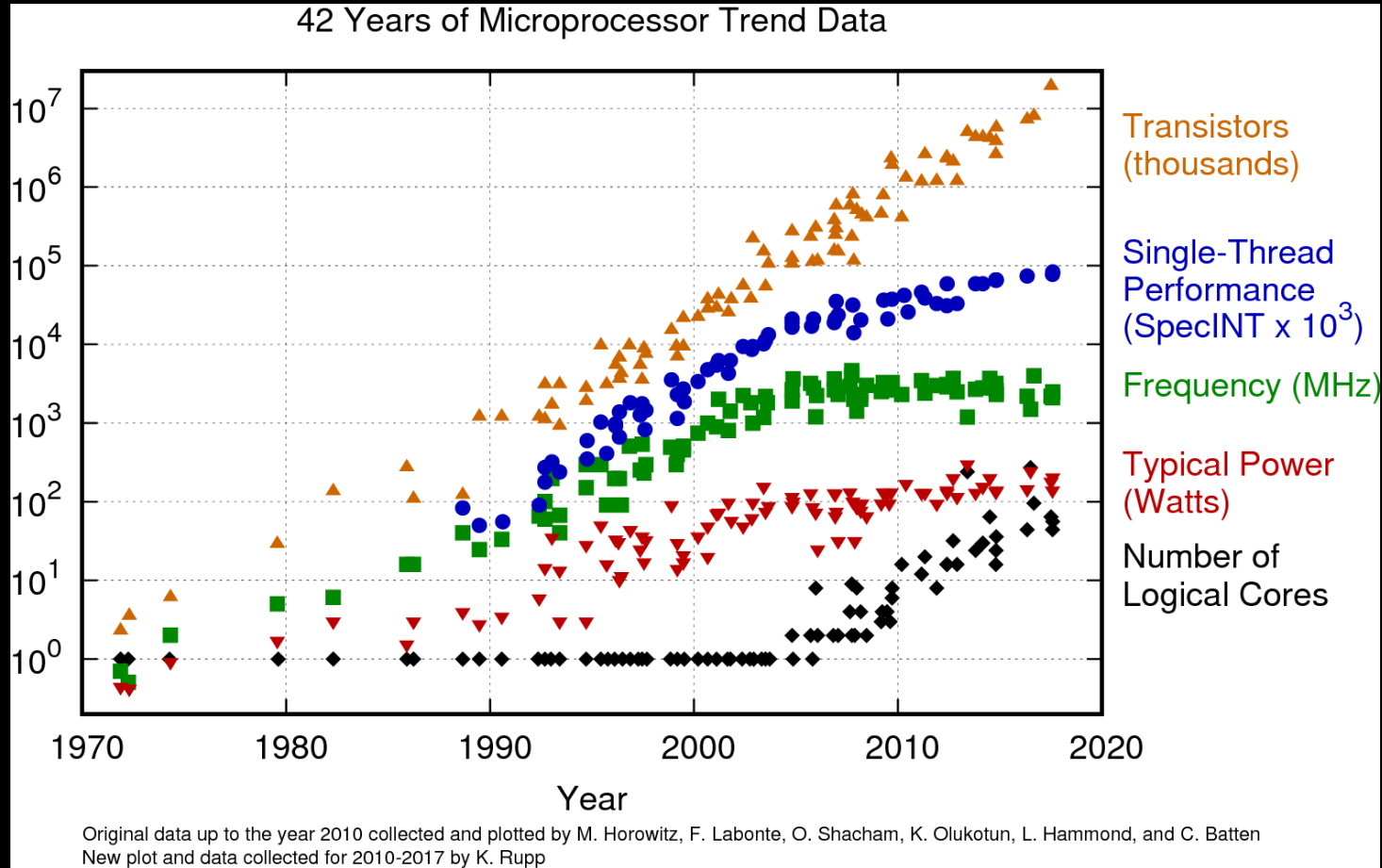
Performance: CPU clock bottleneck, need to increase scalable throughput. Parallelism

Consistency: fault tolerance

Physical reasons – bank storage across different regions need to communicate with each other

Security and isolation

Performance



- CPU Clock hits bottleneck.
- Where should we use the extra transistors?
- Increase cores
- Parallelism

Fault Tolerance (properties)

- DDIA: “on a storage cluster with 10,000 disks, we should expect average one disk die per day”
- DDIA: Fault Tolerance on “certain types of faults”
- We need to hide the failures from the application (means of abstraction).
- **Availability**: app can make progress despite failures
- **Recoverability**: app will come back to life when failures are repaired.
- Design: Replication

Consistency (Safety)

Strong consistency:

Get(k) yields the value from the most recent **Put(k, v)**

Weak consistency

Often weak consistency gives stronger performance.

The Google MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data,

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 de-

map and reduce in functional programming

















- The paper is referring to LISP, a programming language dialect, dates to 1958 (exists 62 years)
- Here are an example usage in Python

```
from functools import reduce
import operator

nums = range(10)
nums = map(lambda num: num + 2, nums)
num = reduce(operator.add, nums)
num # 65
```

Hoogle Translate

fold

	APL	/ (reduce)	-	Doc
	CUDA	reduce	Thrust	Doc
	D	reduce	algorithm.iteration	Doc
	Ruby	reduce	Enumerable	Doc
	Python	reduce	itertools	Doc
	Elixir	reduce	Enum	Doc
	Kotlin	reduce	collections	Doc
	Clojure	reduce	core	Doc
	C++	reduce	<numeric>	Doc
	Haskell	foldl	Data.List	Doc
	Racket	foldl	base	Doc
	Rust	fold	trait.Iterator	Doc
	q	over	-	Doc
	C#	Aggregate	Enumerable	Doc
	J	/ (insert)	-	Doc
	C++	accumulate	<numeric>	Doc

Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets.

Abstraction: users specify a **map function** that processes a list of key/value pairs to generate a set of intermediate key/value pairs, and a **reduce function** that merges all intermediate values associated with the same intermediate key. It hides the details of parallelization, fault-tolerance, data distribution and load balancing in a library.

Programs are executed on a large cluster of commodity machines

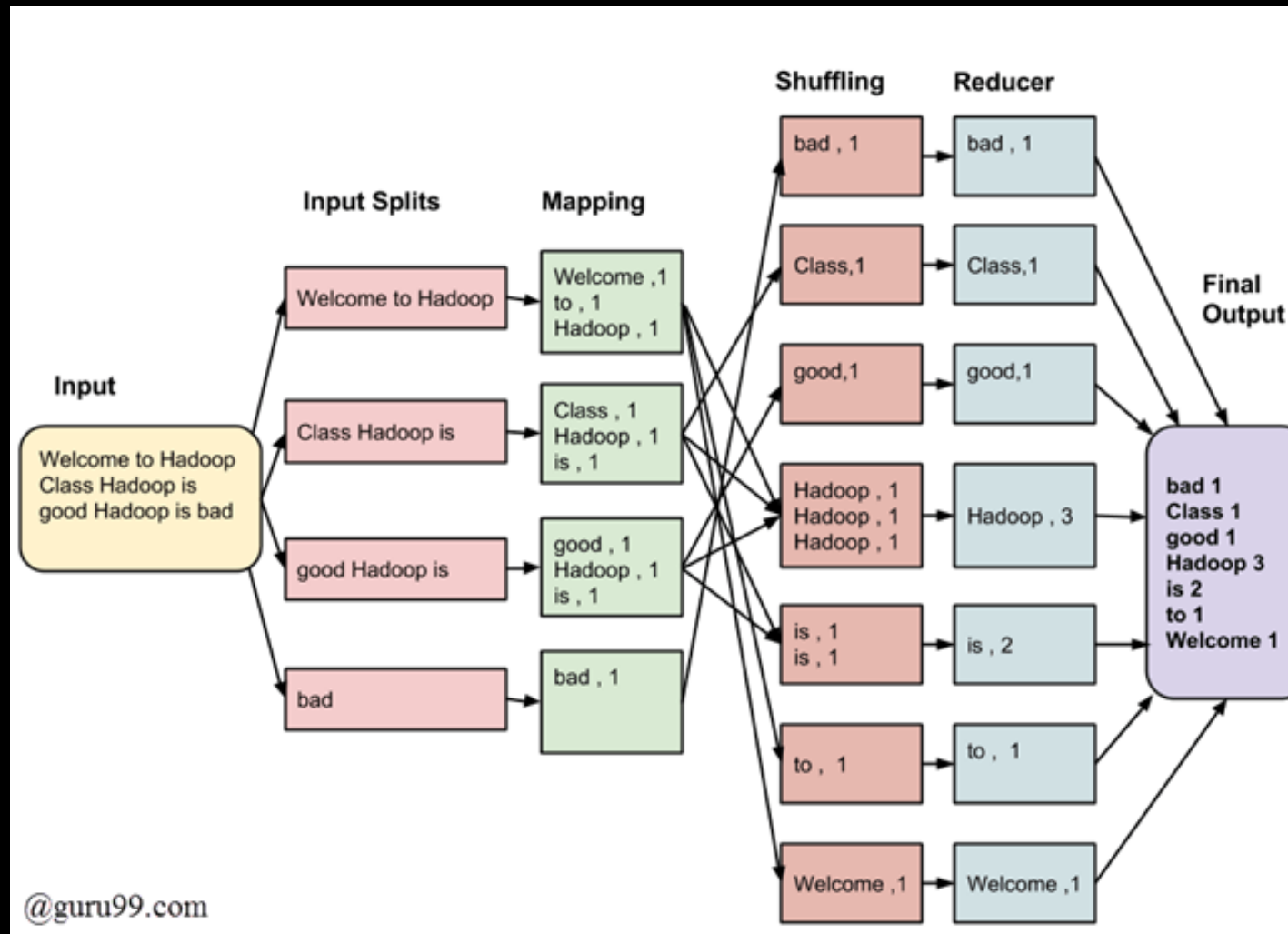
Contributions

A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations.

Inspired the open-source project and a field of research paradigm



Programming Model



An example

- Word count from the homework `wc.go`
- User facing

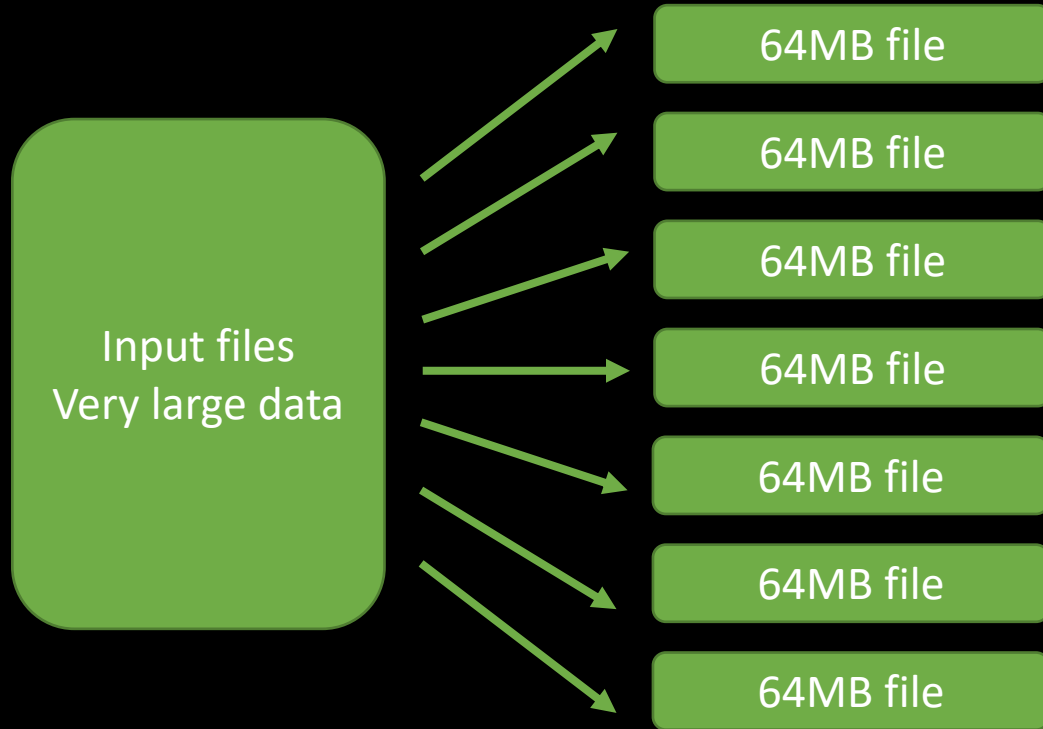
```
//  
// The map function is called once for each file of input. The first  
// argument is the name of the input file, and the second is the  
// file's complete contents. You should ignore the input file name,  
// and look only at the contents argument. The return value is a slice  
// of key/value pairs.  
//  
func Map(filename string, contents string) []mr.KeyValue {  
    // function to detect word separators.  
    ff := func(r rune) bool { return !unicode.IsLetter(r) }  
  
    // split contents into an array of words.  
    words := strings.FieldsFunc(contents, ff)  
  
    kva := []mr.KeyValue{}  
    for _, w := range words {  
        kv := mr.KeyValue{w, "1"}  
        kva = append(kva, kv)  
    }  
    return kva  
}  
  
//  
// The reduce function is called once for each key generated by the  
// map tasks, with a list of all the values created for that key by  
// any map task.  
//  
func Reduce(key string, values []string) string {  
    // return the number of occurrences of this word.  
    return strconv.Itoa(len(values))  
}
```

Abstract view



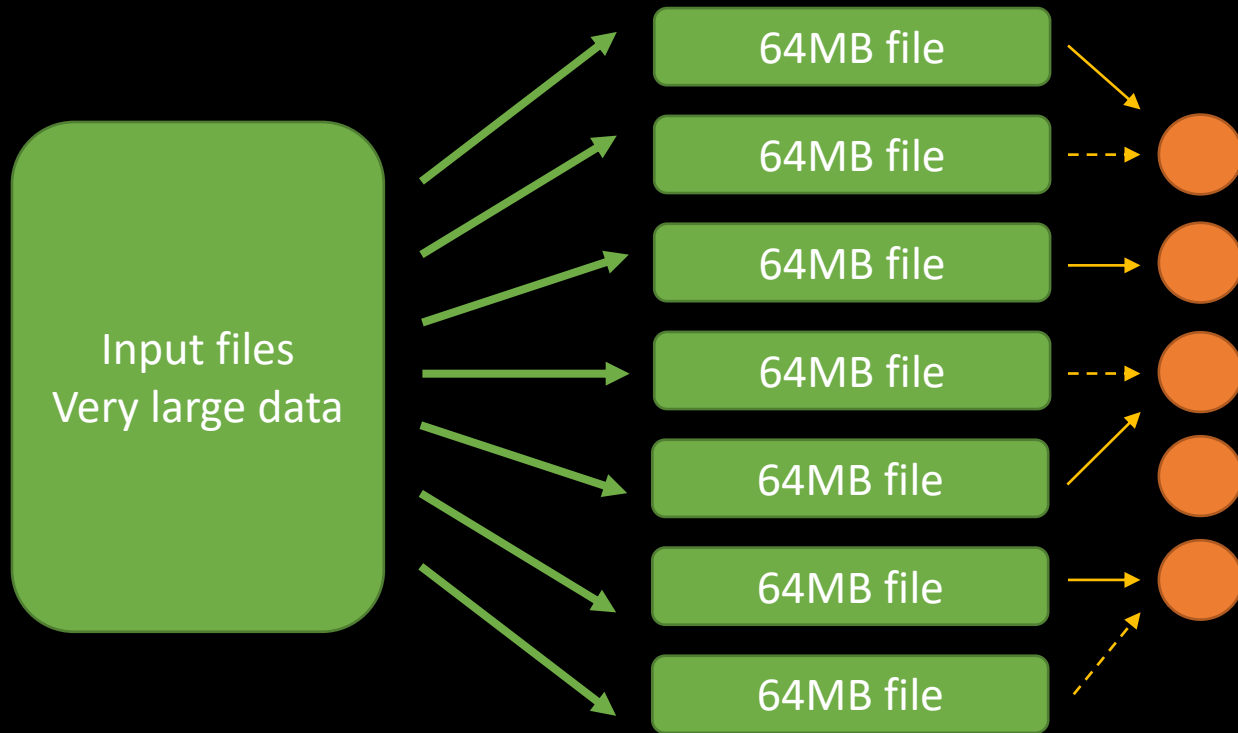
Input files
Very large data

Abstract view step 1



Splits to M pieces

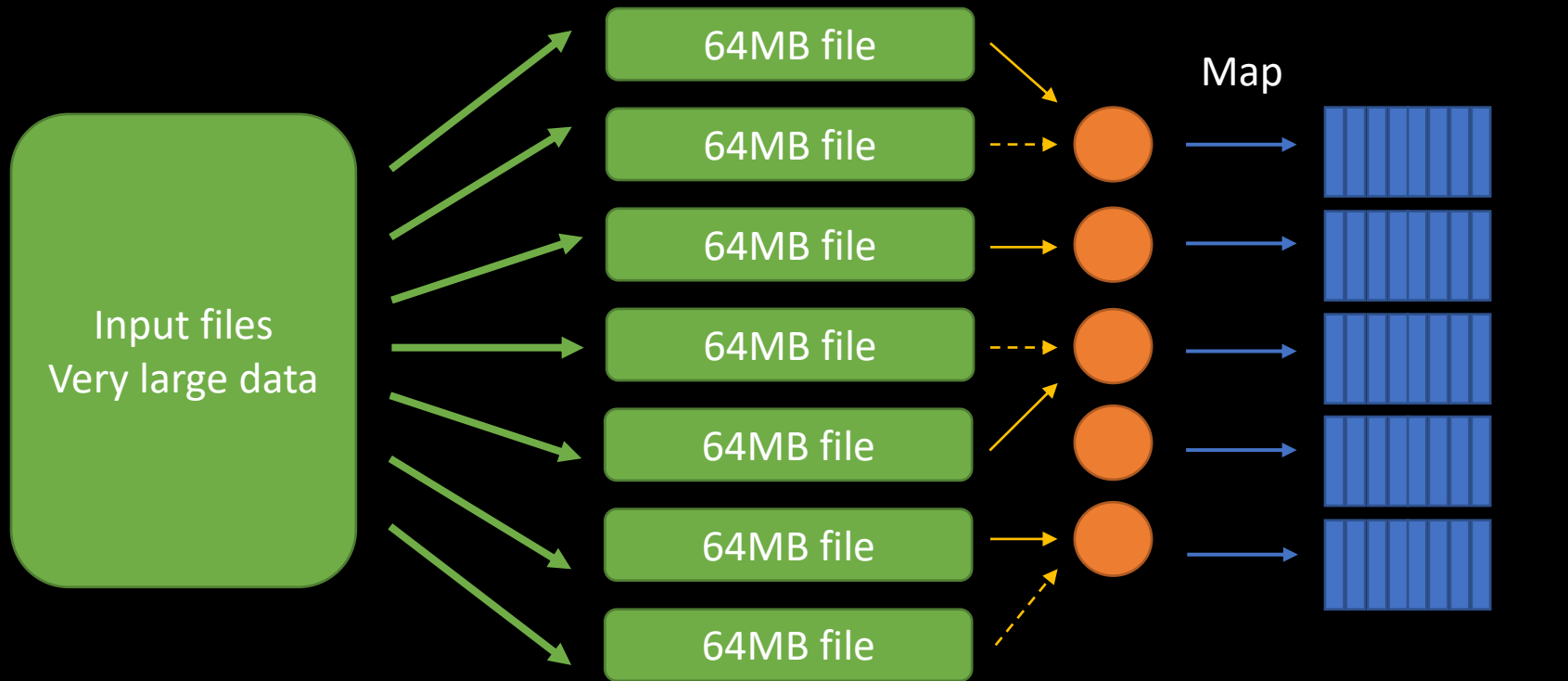
Abstract view step 2



Splits to M pieces

**Master picks idle worker
and assigns a map task**

Abstract view step 3

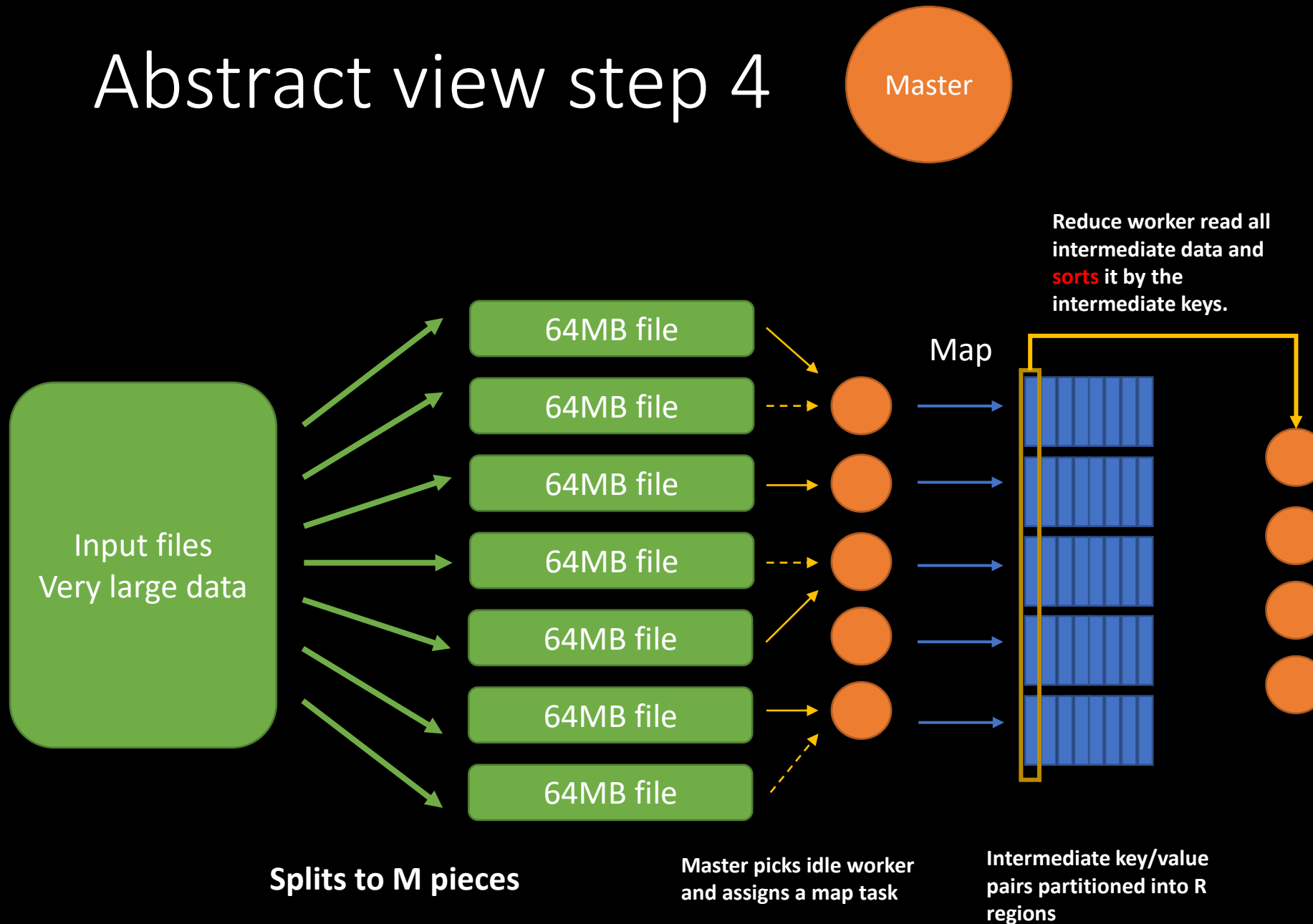


Splits to M pieces

Master picks idle worker
and assigns a map task

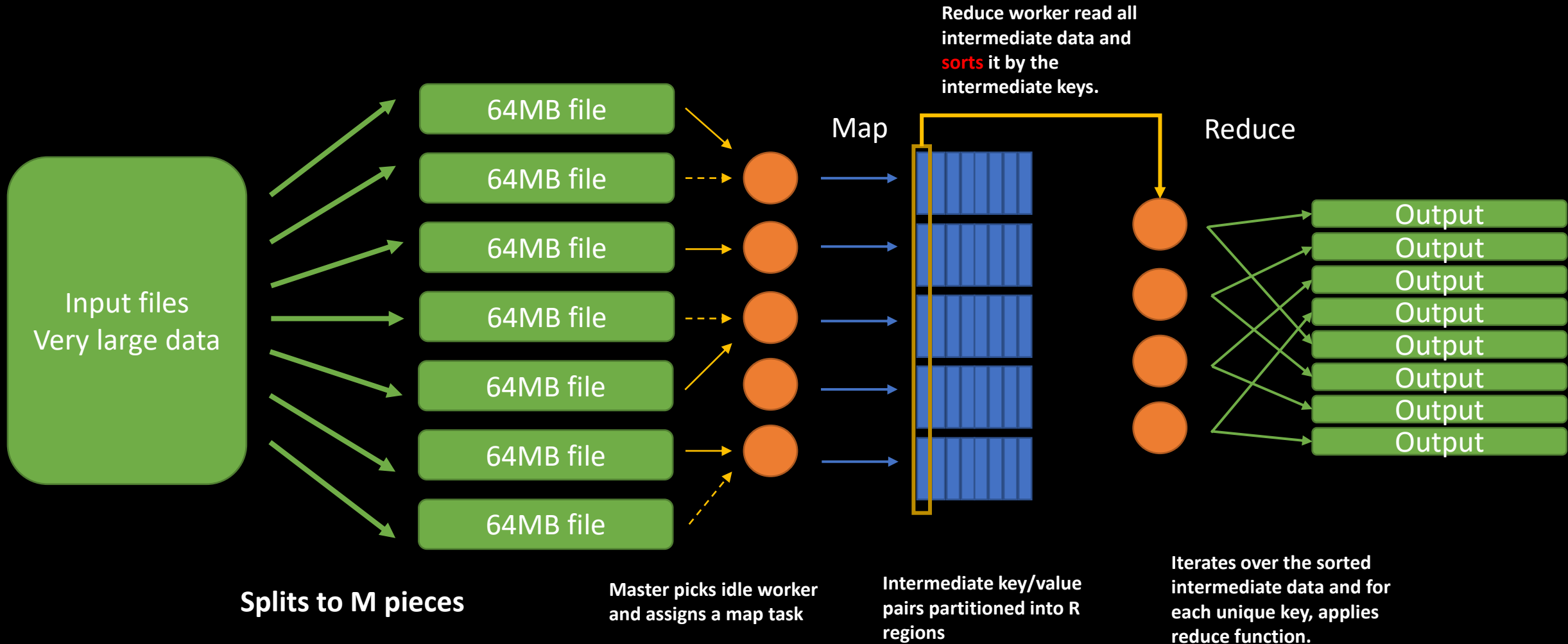
Intermediate key/value
pairs partitioned into R
regions

Abstract view step 4



Abstract view step 5

Master



Final step: completion

- When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns to the user code.

Some details

1. M is automatically calculated based on the input data size
2. R is provided by the user. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a **partition function** ($\text{hash}(\text{key}) \% R$).
3. The number of workers is irrelevant to M or R.
4. When the map worker writes the key/value pair to local disk, the locations are **passed back to the master**, who can forward these locations to the reduce workers.
5. Reduce worker read and **sort** all intermediate data (why sorting?)

Implementation (skipped)

Well you will need to implement it in Lab 1 so I will skip the details of the implementation here. Let's talk about it once Lab 1 is due.

Start Early Start Often

MapReduce abstraction

- It hides many details
 - Sending app code to servers (RPC)
 - Tracking which tasks are done (communication)
 - Moving data from Maps to Reduce
 - Balancing load over servers
 - Recovering from failures (except the case when the master failed ☹)
- No real-time or streaming processing
- No iteration, no multi-stage pipelines
- No interaction or state

Parallelism benefits from the File System

- We will be talking about GFS on week 3
- Maps read in parallel
- Reduces write in parallel
- Replication of each file on 2-3 servers
- All the above are for free!

Fault Tolerance

- **Worker failure**

- Master pings worker periodically, and when timeout marks worker as failed.
- Maps or Reduces who failed need to re-run the whole job from the beginning
- Map, Reduce must be deterministic

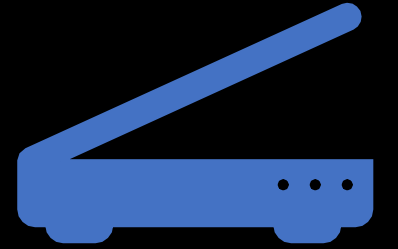
- **Master failure**

- Master writes periodic checkpoints of Master data structure
- If Master dies, a new copy can be started from the last check pointed state
- Google's implementation aborts the whole program if Master fails...

How is deterministic property being achieved

- Google's implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program
- MapReduce rely on atomic commits of map and reduce task outputs to achieve this property.
- Map produces R temporary files and notify the master the names of files. If the master receives a completion message for an already completed map task, **it ignores the message**.
- Reduce produce 1 temporary file and notify the master, and **atomically renames** its output file to the final output file. This makes sure there is only 1 output file on GFS.

Straggler



- Definition:
 - A machine that takes an unusually long time to complete of the last few map or reduce tasks in the computation.
- Solution:
 - When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks.]
 - The computational resources used by the operation increase by no more than a few percent
 - Reduce the time to complete a MapReduce operation a lot!

Refinements

- Partition Function: user can provide custom partitioning function for special cases. For example, if keys are URL and you want entries for a single host to end up in the same output file.
 - `Hash(Hostname(URL)) % R`
- Ordering guarantees
 - The intermediate key/value pairs are processed in increasing key order.
- Combiner Function: partial merging of the data from Map.
- Skipping Bad Records:
 - Bugs in user code cause the Map or Reduce functions to crash deterministically on certain records
 - Optional mode to skip these records.

Refinements

- Counters
 - For example user code may want to count total number of words processed or the number of German documents indexed.
 - Counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code.

Current Status and Conclusion

- Very influential:
 - Made big cluster computation popular
- No longer used at Google
- GFS replaced by Colossus and BigTable
- Scales well
- Easy to program
- Not the most efficient or flexible



Reference

MapReduce practice questions

<https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/>



Next week

1. **Sign up your talk in calendar**
2. Watch <https://pdos.csail.mit.edu/6.824/video/2.html>
3. Read <https://pdos.csail.mit.edu/6.824/notes/l-rpc.txt>
4. Do <http://tour.golang.org/>
5. Start working on lab 1 <https://pdos.csail.mit.edu/6.824/labs/lab-mr.html>
6. Read Chap 2 (Steen) OR Chap 2 (Kleppmann)