CS162
Operating Systems and Systems Programming
Lecture 23

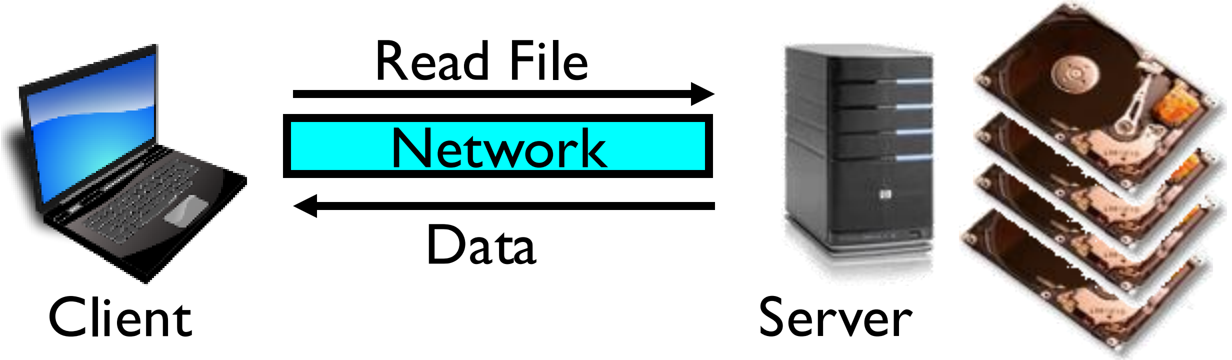
Internet & Data Processing Systems

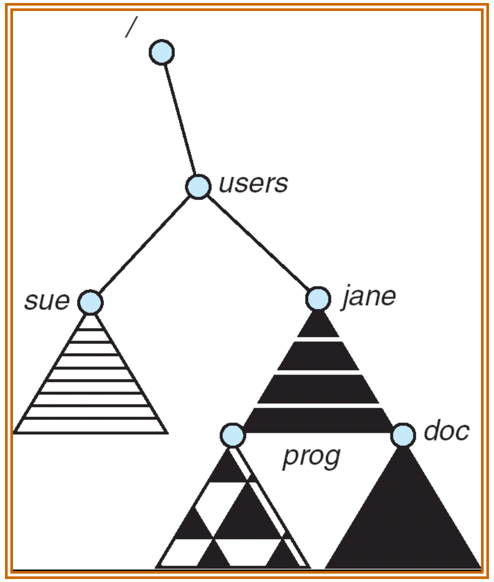
Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: Distributed File Systems



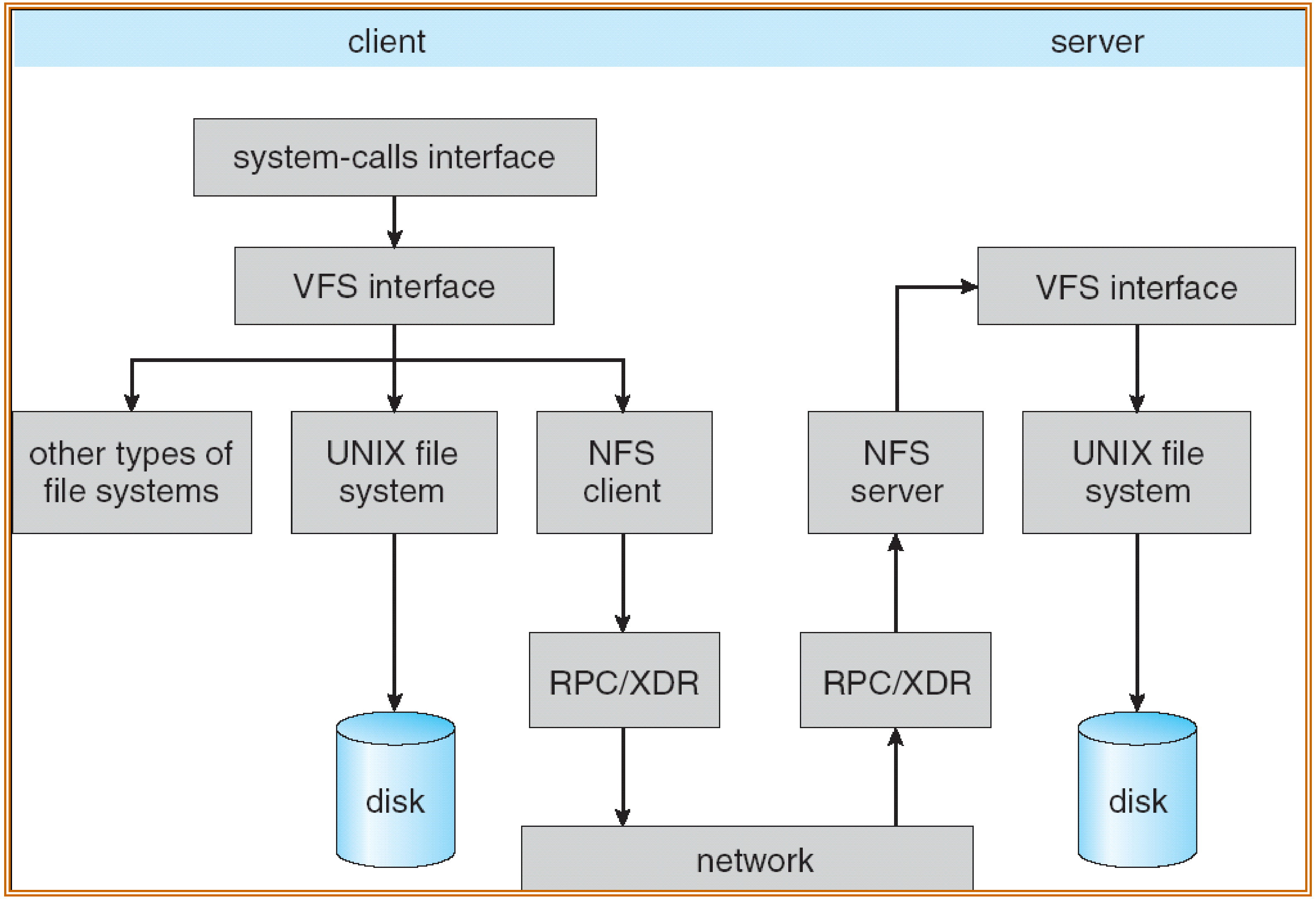


Transparent access to files stored on a remote disk

Mount remote files into your local file system

- Directory in local file system refers to remote files
- e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `fs.cs.berkeley.edu`

Recall: Network File System (NFS)



Recall: Stateless Protocol

Stateless Protocol: A protocol in which all information required to service a request is included with the request

Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)

E.g. “ReadAt(file_id, offset, length)” instead of “Read(fd, length)”

Recall: NFS Cache Consistency

NFS protocol: weak consistency

- Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.

What if multiple clients write to same file?

- » In NFS, can get either version (or parts of both)
- » Completely arbitrary!

Recall: The Internet

Many different applications

- Email, web, P2P, etc.

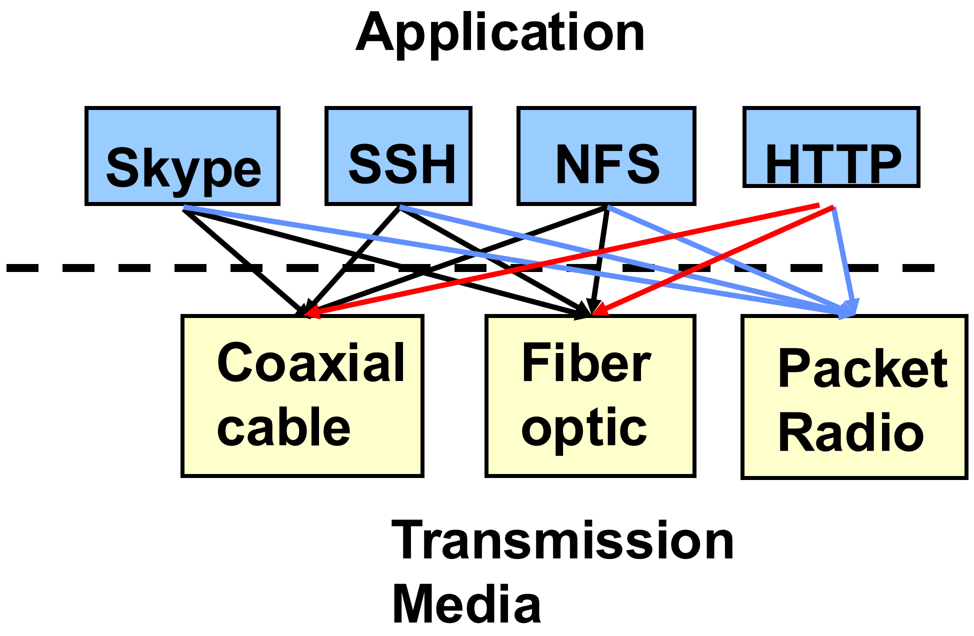
Many different operating systems and devices

Many different network styles and technologies

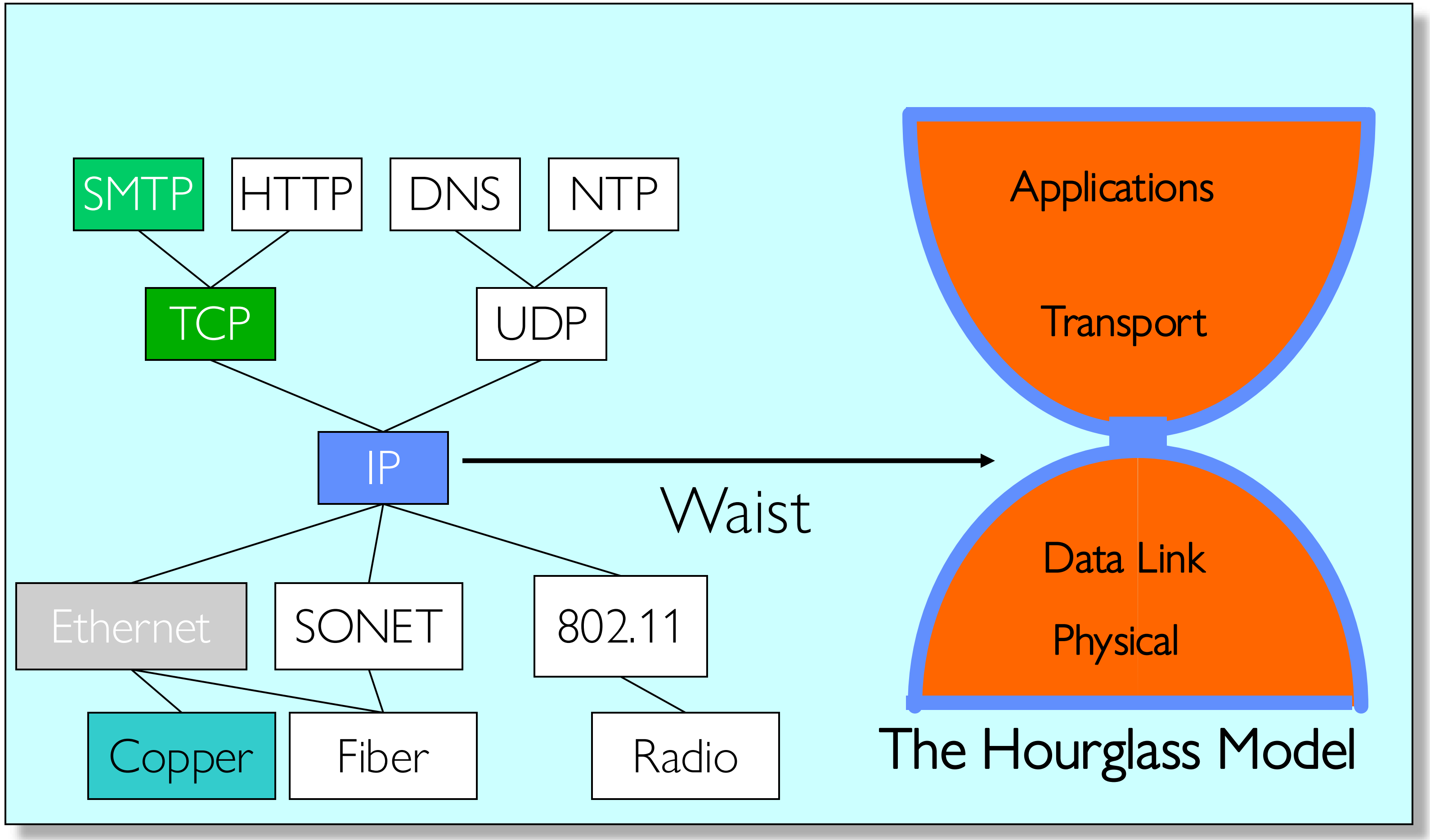
- Wireless, wired, optical

How do we organize this mess

- Layering and end-to-end principle



Recall: Internet Layers and Hourglass Model



"Narrow waist" facilitates interoperability

Layers "abstract" away hardware so that upper layers are agnostic to lower layers

=> Sound familiar?

Implications of Hourglass Model

Single Internet-layer module (**IP**):

Allows arbitrary networks to interoperate

– Any network technology that supports IP can exchange packets

Allows applications to function on all networks

– Applications that can run on IP can use any network

Supports simultaneous innovations above and below IP

– But changing IP itself, i.e., **IPv6**, is very complex

Drawbacks of Internet Layering

Layer N may duplicate layer N-1 functionality

– E.g., error recovery to retransmit lost data

Layers may need same information

– E.g., timestamps, maximum transmission unit size

Layering can hurt performance

– E.g., hiding details about what is really going on

Some layers are not always cleanly separated

– Inter-layer dependencies for performance reasons

– Some dependencies in standards (header checksums)

2nd Design Idea: The End-To-End Argument

Hugely influential paper:

- "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)

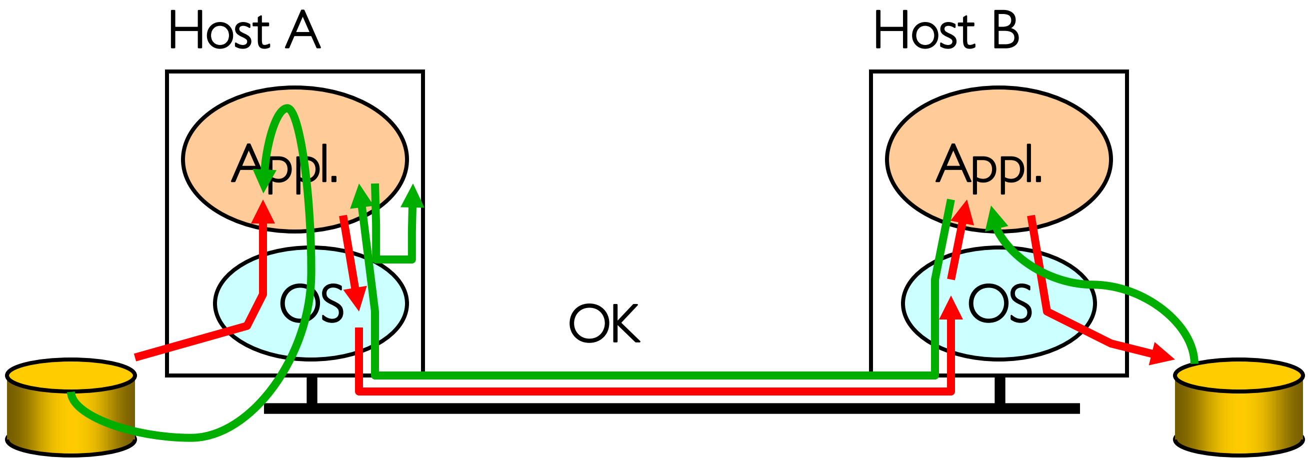
"Sacred Text" of the Internet

- Endless disputes about what it means
- Everyone cites it as supporting their position

Simple Message: Some types of network functionality can only be correctly implemented end-to-end

- Reliability, security, etc.
- Hosts can't rely on network to fully implement them, so must do it themselves

Example: Reliable File Transfer



Solution 1: make each step reliable, and then concatenate them

Solution 2: end-to-end check and try again if necessary

Discussion

Solution 1 is incomplete

- What happens if memory is corrupted?
- Receiver has to do the check anyway!

Solution 2 is complete

- Full functionality can be entirely implemented at application layer with no need for reliability from lower layers

Is there any need to implement reliability at lower layers?

- Well, it could be more efficient

End-to-End Principle

Implementing complex functionality in the network:

- Doesn't always reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality

However, implementing in network can enhance performance in some cases

- – e.g., very lossy link

How to Interpret E2E Argument?

Conservative interpretation: Don’t implement a function at the lower levels of the system unless it can be completely implemented at this level

Moderate interpretation: Think twice before implementing functionality in network

- If hosts can do the functionality E2E, do it in network only as perf enhancement
- Don’t put a burden on apps that don’t need the functionality

Topic Roadmap

Distributed File System: NFS

Peer-To-Peer System: The Internet

Distributed Data Processing
(MapReduce and Spark)

Coordination
(Atomic Commit and Consensus)

Case Study: Distributed Data Processing

The Big Data Problem

Data is growing faster than server speeds

Growing data sources
- » Web, mobile, scientific, …

Cheap storage
- » Doubling every ~18 months

Stalling CPU speeds

The Economist
Obama the warrior
Misgoverning Argentina
The economic shift from West to East
Genetically modified crops blossom
The right to eat cats and dogs
The data deluge
AND HOW TO HANDLE IT: A 14-PAGE SPECIAL REPORT
nature
THE BITER BIT
Viral infections for viruses
TROPICAL CYCLONES
The strong get stronger
BLACK HOLE PHYSICS
A new window on the Galactic Centre
BIG DATA
NATUREJOBS
Minnesota musings
SCIENCE IN THE PETABYTE ERA
POPULAR SCIENCE
THE FUTURE NOW
THE CONTROL CENTERS
Using Data to Feed the World, Solve Cold Cases, Battle Malware, Predict Our Fate
OFFICER ALGORITHM
Can a Crime Be Prevented Before It Begins?
NEW WAYS OF SEEING
A Gallery of Extraordinary Infographics
PLUS
Juan Enriquez Reprograms Life
James Gleick Unsplits the Bit
AND Lawrence Weschler Questions the Cloud
SPECIAL ISSUE
DATA IS POWER
HOW INFORMATION IS DRIVING THE FUTURE

Examples

1000 genomes project: 200 TB

Google web index: 100+ PB

Ingest per day at large Databricks customer: 1 PB

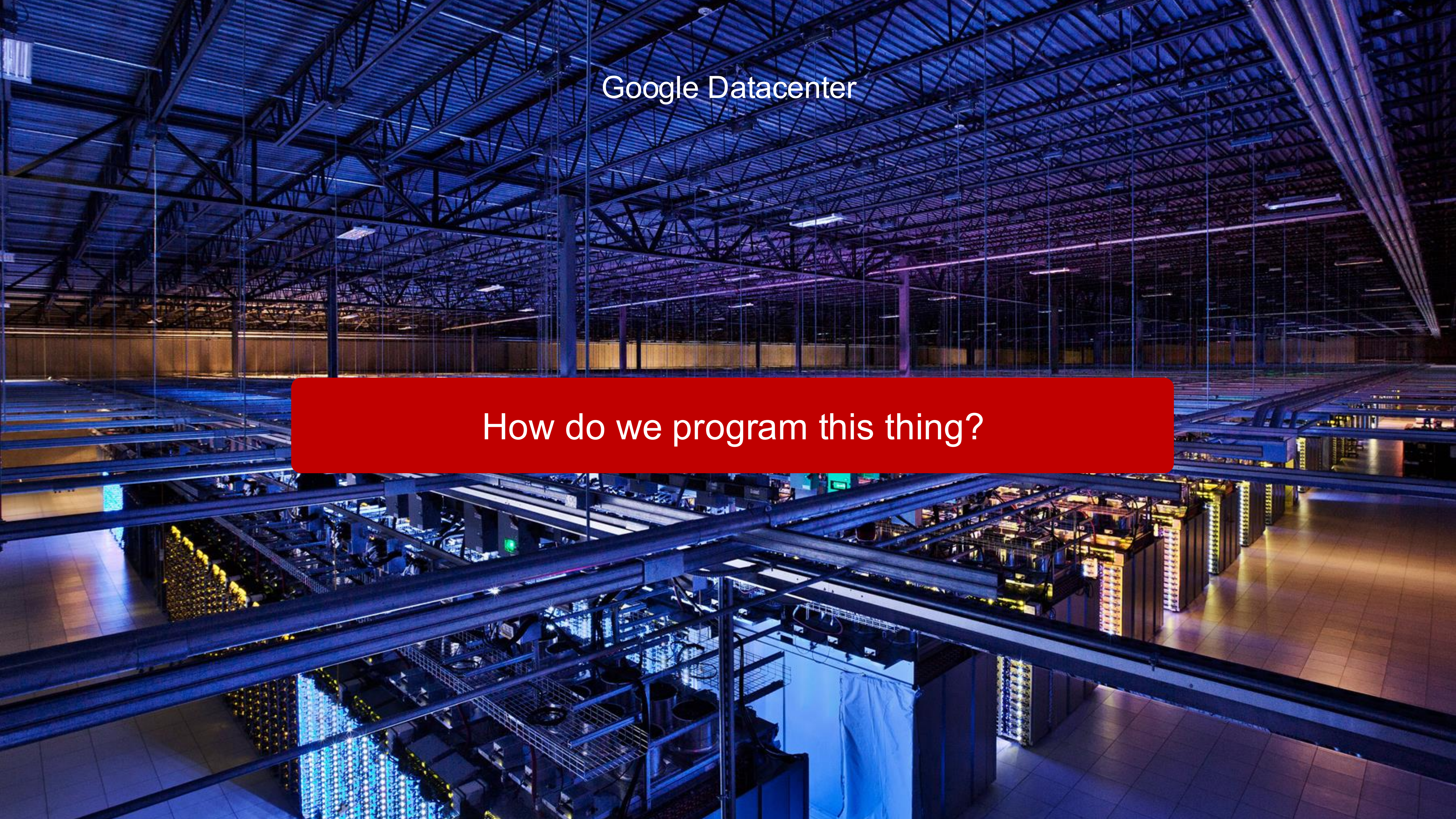
Cost of 1 TB of disk: $20

Time to read 1 TB from disk: 3 hours (100 MB/s)

The Big Data Problem

Single machine can no longer process or store all the data!

Only solution is to **distribute** over large clusters



Google Datacenter

How do we program this thing?

Traditional Network Programming

Message-passing between nodes

Really hard to do at scale:

- How to divide problem across nodes?
- How to deal with failures?
- Even worse: stragglers (node is not failed, but slow)

Almost nobody does this for distributed data processing

Distributed Data Processing Frameworks

Come up with a model for breaking large computations into smaller tasks, then build a framework that distributes those tasks to workers in a cluster

Framework handles scheduling, fault recovery, etc

Recent wave popularized with MapReduce (and open source Hadoop)

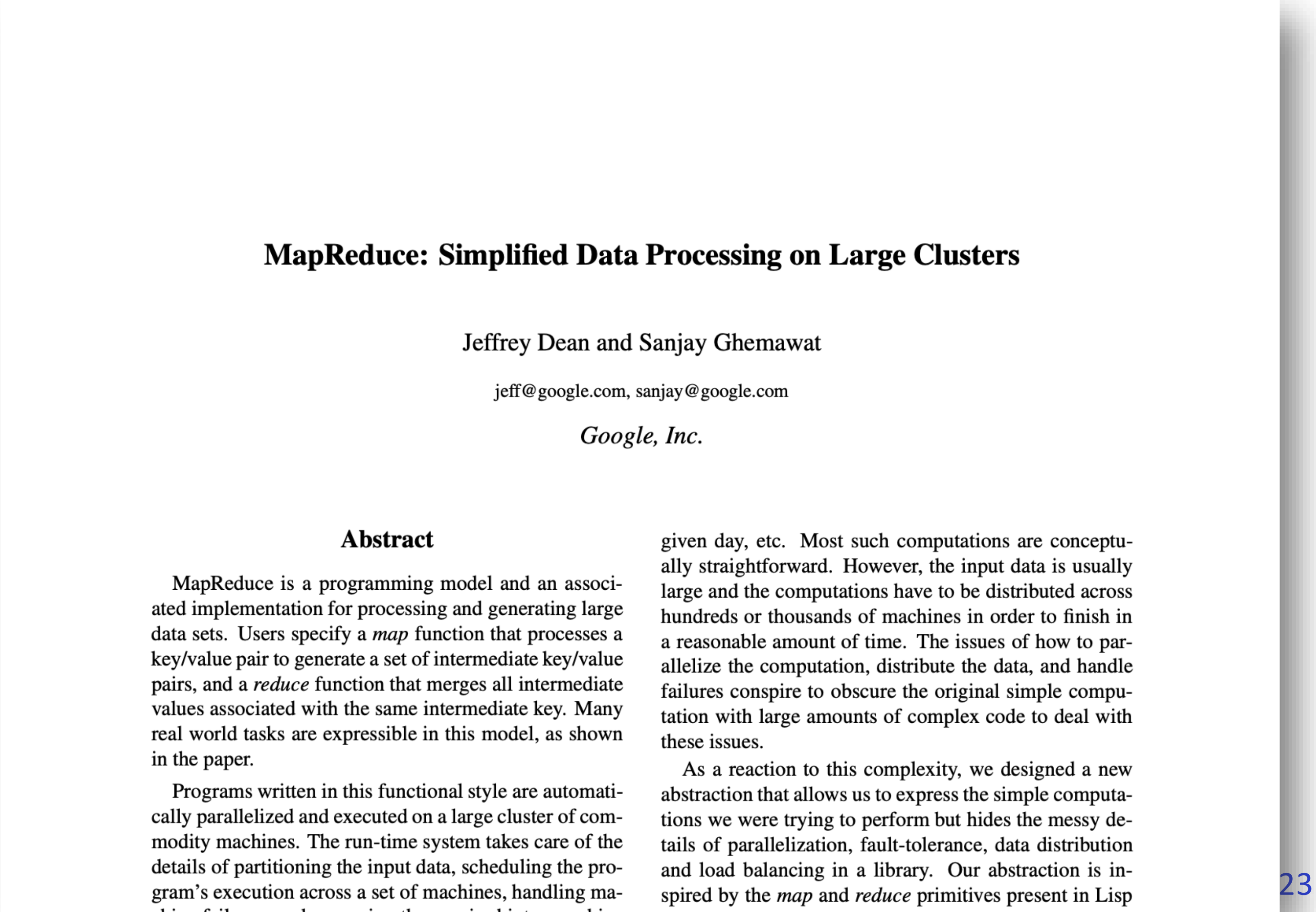
MapReduce History

Developed by Google, paper published in 2004

Google had lots of raw data:

- Crawled web pages
- Server logs
- Search data

Wanted to build many apps: web indexing, usage analysis, spam filtering, etc



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 ma-

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

MapReduce History

Environment: clusters of cheap commodity machines

- "Off-the-shelf" machines, i.e. hardware not custom-built for reliability

Many "one-off" solutions for parallelizing workloads

- Hard to maintain
- Hard to get right
- Time-consuming to implement

MapReduce Programming Model

Data type: key-value *records*

Map function:

$$(K_{in}, V_{in}) \Rightarrow \text{list}(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, \text{list}(V_{inter})) \Rightarrow \text{list}(K_{out}, V_{out})$$

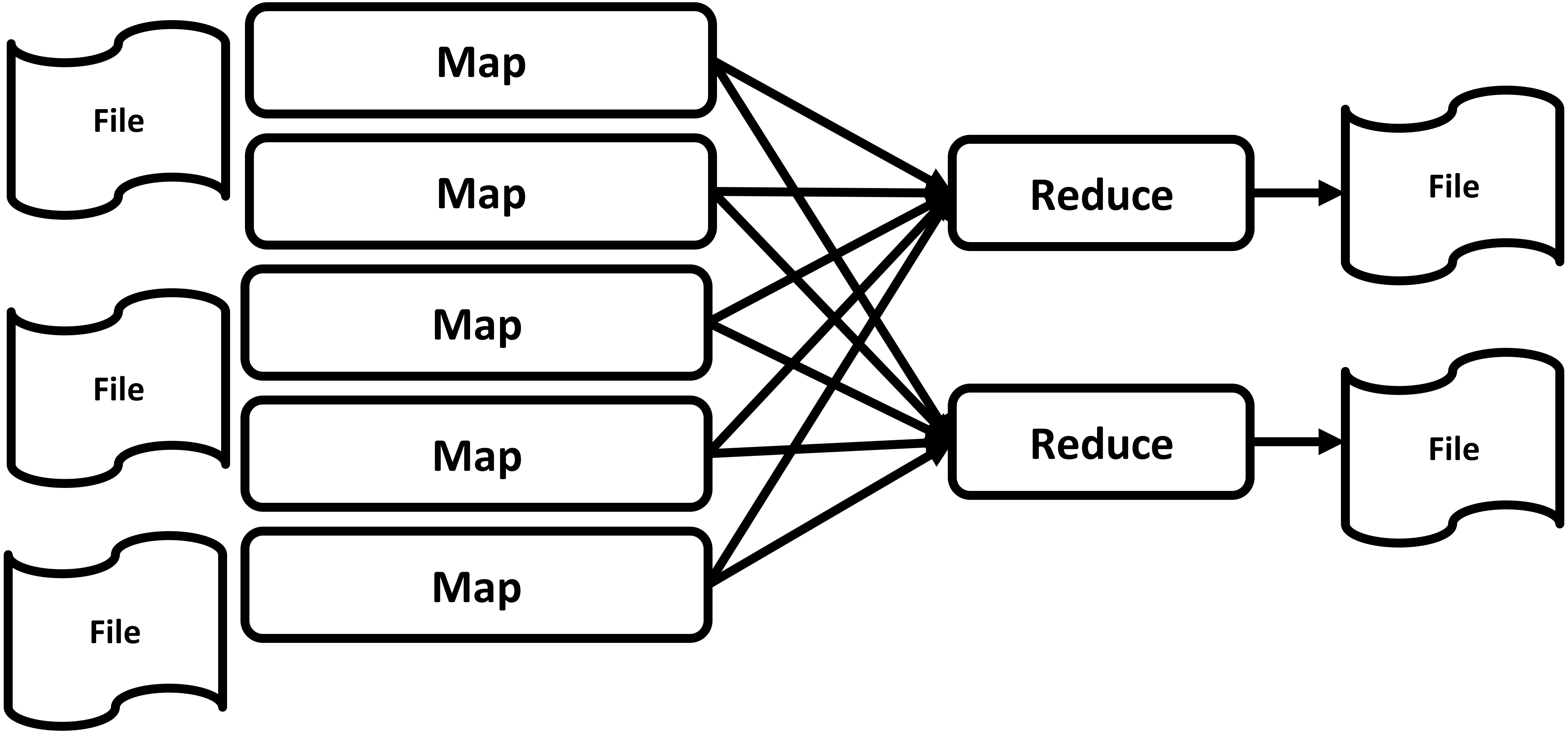
Example: Word Count

How can we count how many times each word occurs in a large dataset using only map and reduce?

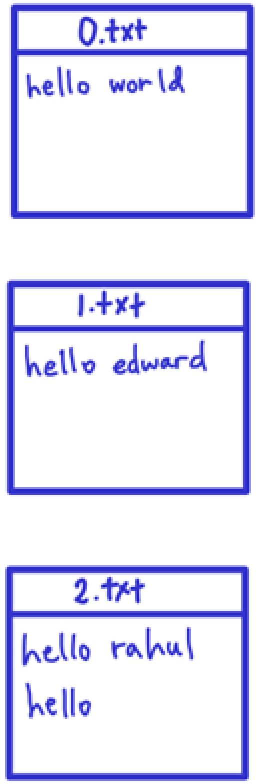
Four steps:

1) Convert files into pairs of (key, value)
2) Define a map function. Apply to all files.
3) Shuffle! All elements with same key go to same reduce.
4) Define a reduce function. Apply to result of the map function.

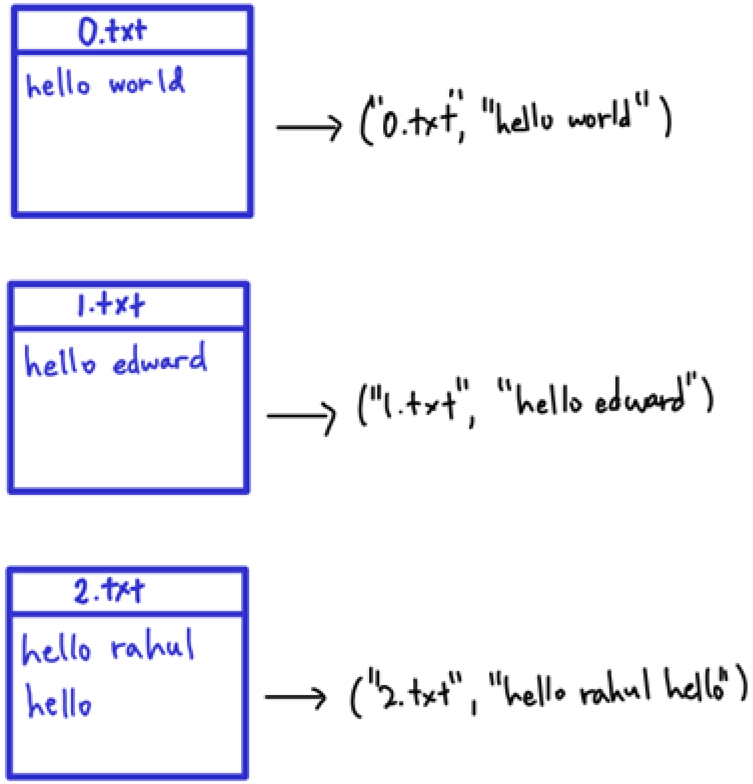
1000 ft view of Map Reduce



Word Count Map Reduce



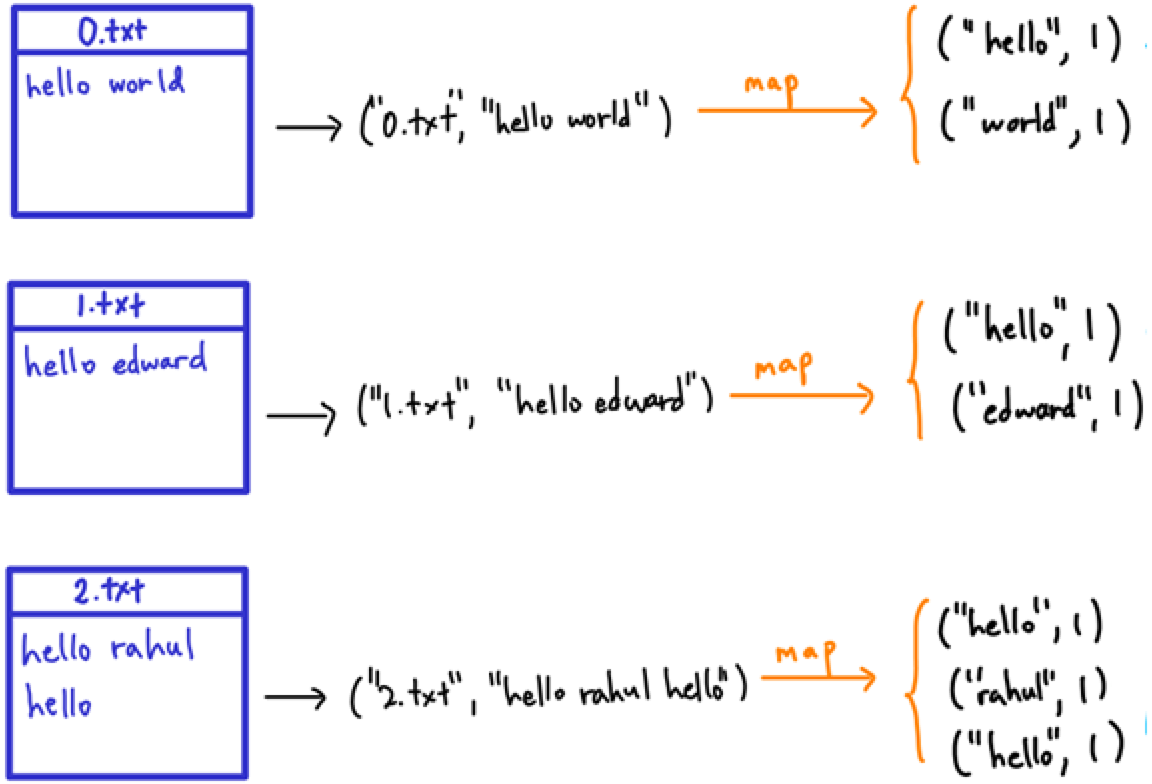
WC. Step 1: to (Key, Value)



Transform file into:
(File Name, List of words)

Map function takes (Key, List) and maps it to List (Key, Value).

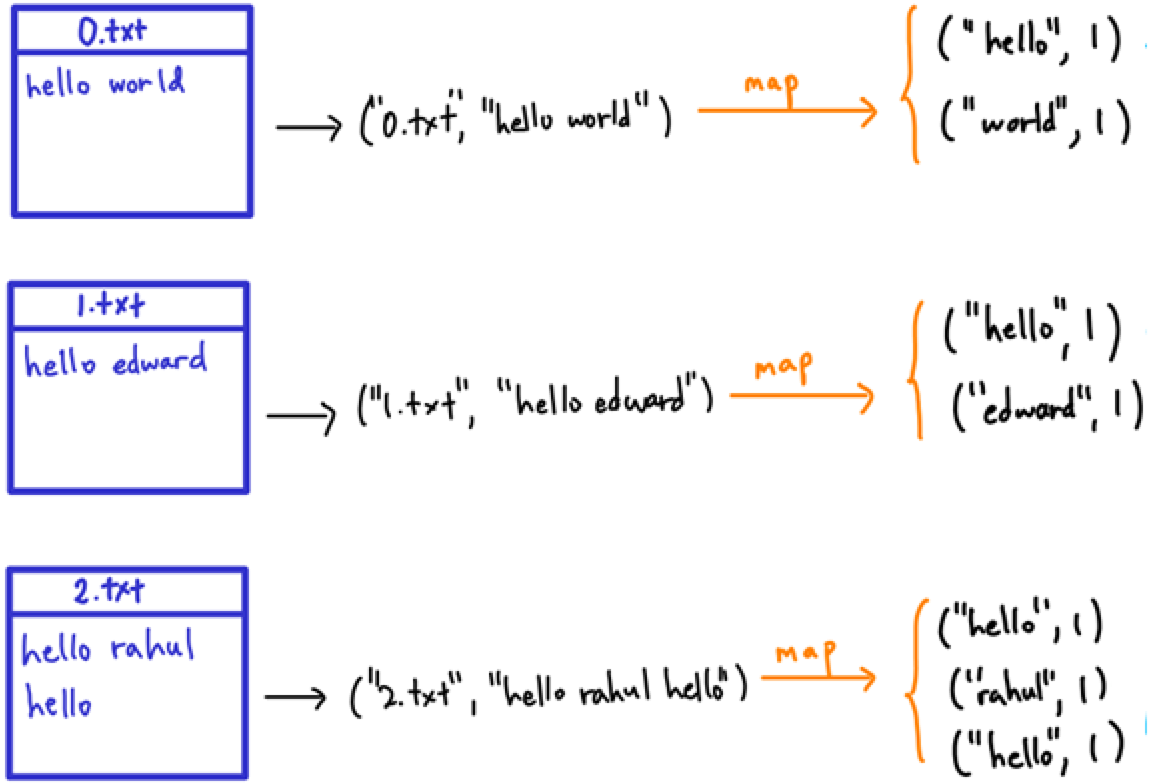
WC. Step 2: Map Function



Map function:

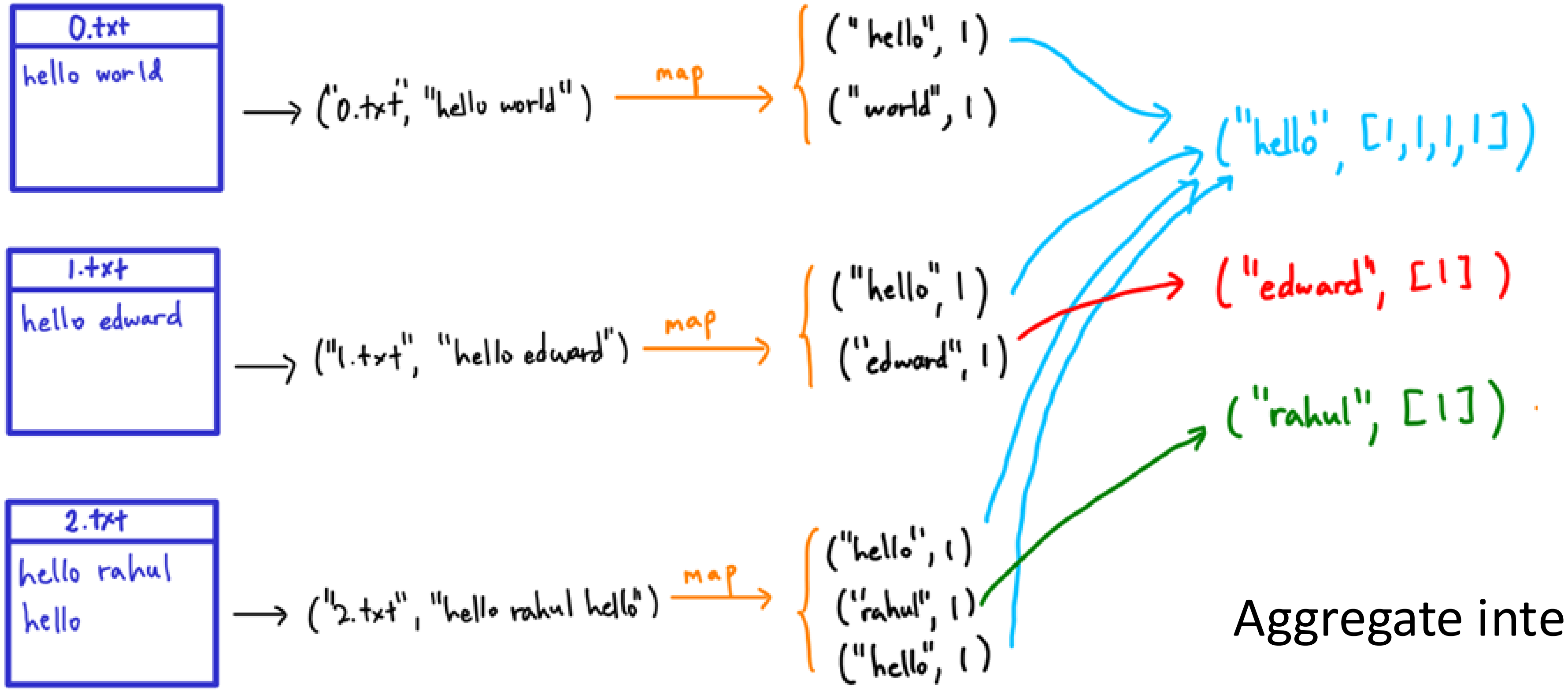
Associate each word with a count!

WC. Step 2: Map Function



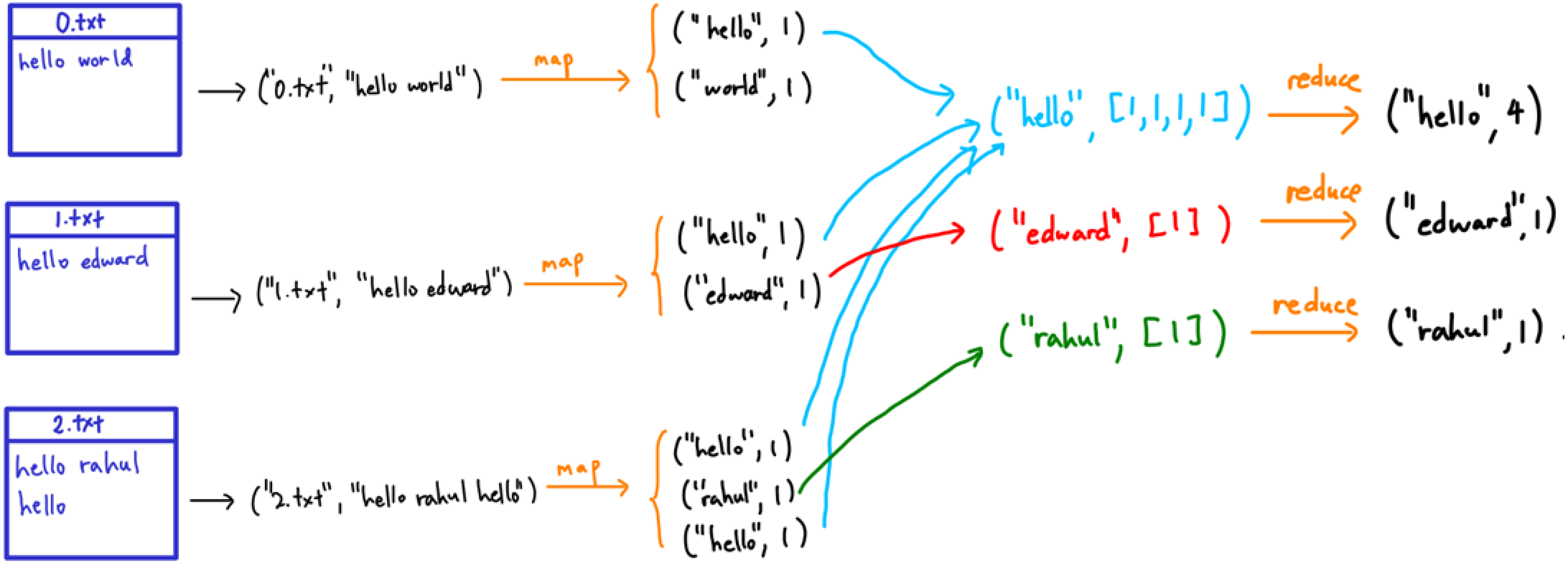
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

WC. Step 3: Shuffle!



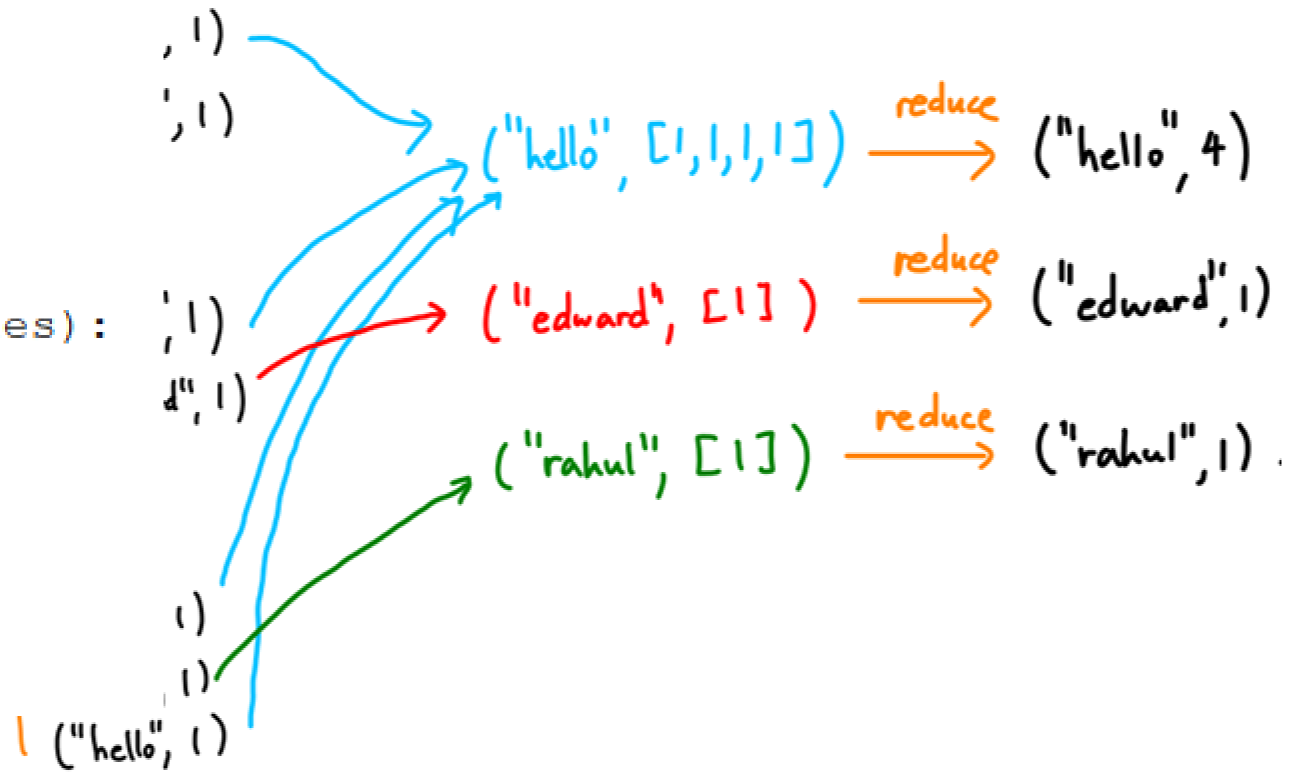
Aggregate intermediate results by key

WC. Step 4: Reduce

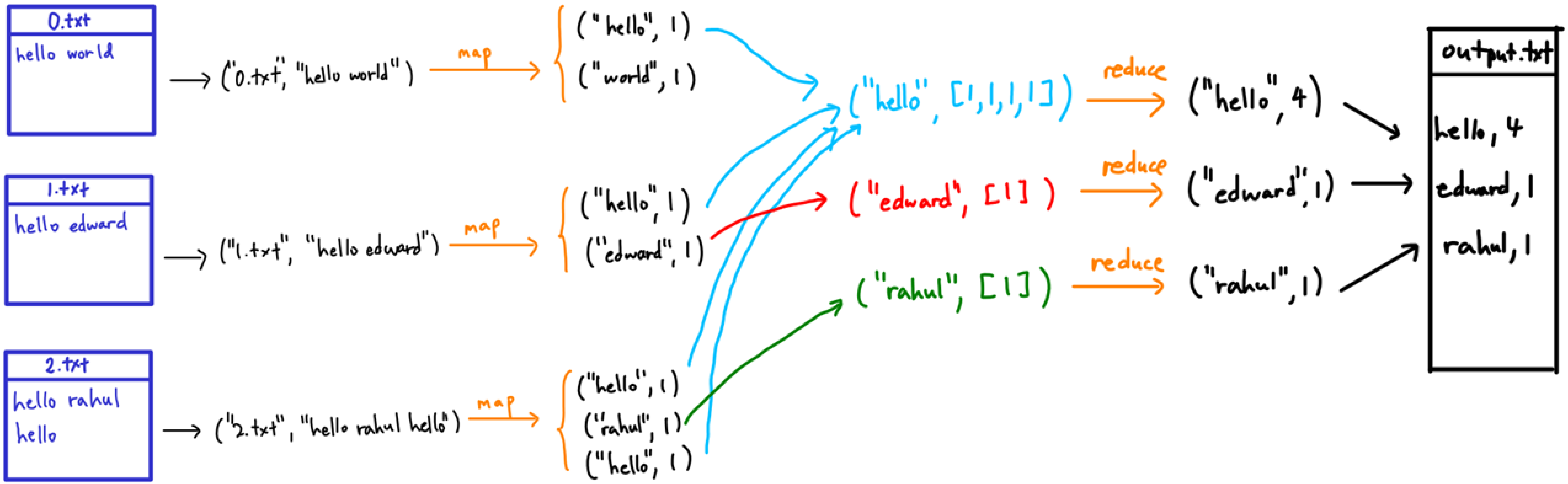


WC. Step 4: Reduce

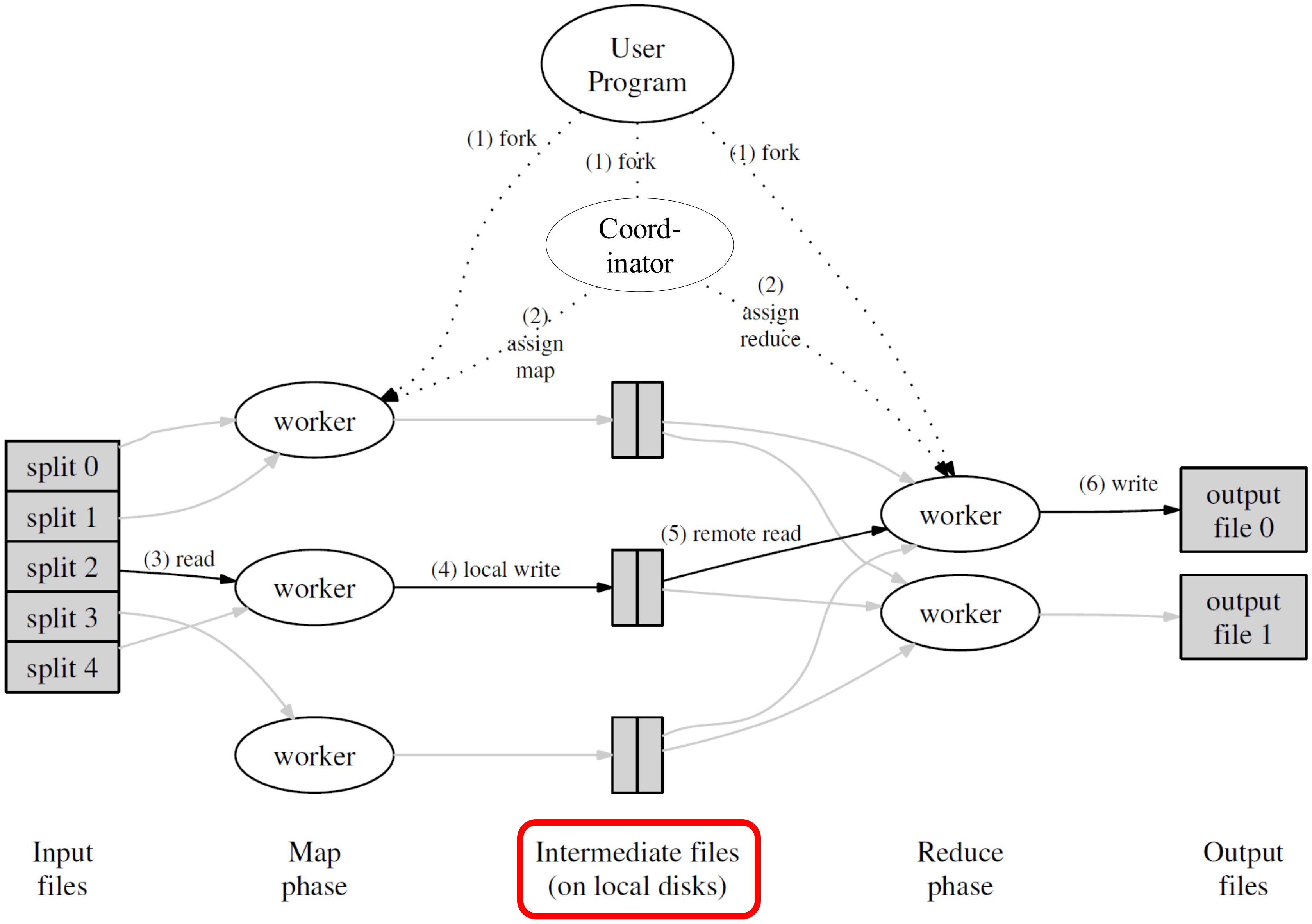
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



WC. Final Step, Generate Output



Map Reduce System Architecture



Fault Tolerance

MapReduce assumes that:

Any *individual* machine is unlikely to crash
But large cluster of machines is likely to experience failures

MapReduce does not attempt to handle coordinator crashes

MapReduce does try to handle worker failures

Fault Tolerance in MapReduce

1. If a map or reduce task crashes:
   - – Retry on another node
     - » OK for a map because it had no dependencies
     - » OK for reduce because map outputs are on disk
   - – If the same task repeatedly fails, fail the job or ignore that input block

Note: For the fault tolerance to work, **user tasks must be idempotent**

- Deterministic and side-effect-free (e.g. not based on time, randomness, etc)

Fault Tolerance in MapReduce

2. If a worker node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - » Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):

   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

Critical for performance in large clusters (many possible causes of stragglers)

Beyond MapReduce

Not all applications can easily be expressed with maps & reduces

MapReduce stores all intermediate state on disk, which is slow

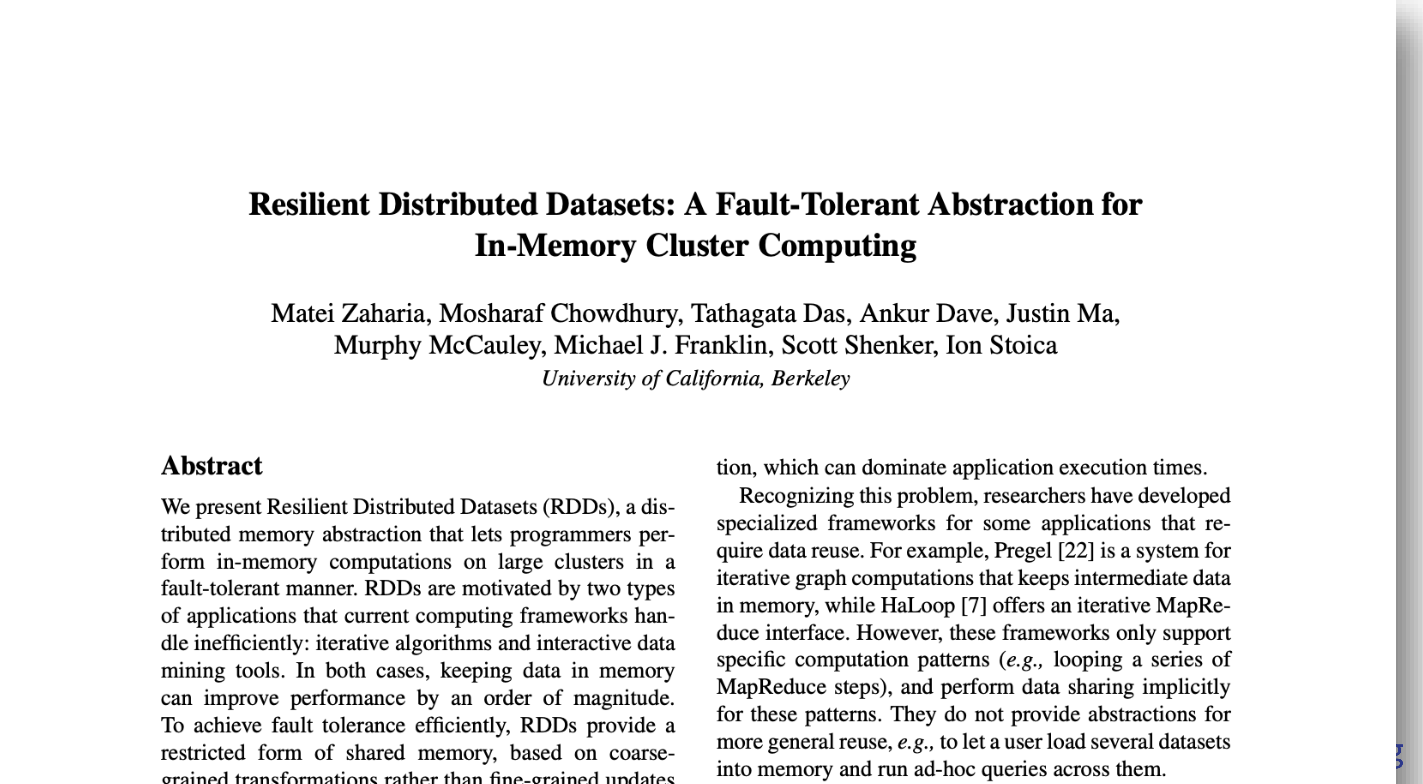
A lot of other distributed programming frameworks:

in-memory processing (Spark), distributed SQL (Apache Hive), graph (PowerGraph), incremental processing (Naiad), streaming (Flink), and many others

Example Apache Spark:

Open source engine that generalizes the MapReduce model with

- Higher-level operators: not just map & reduce, but join, filter, SQL, streaming operators, and many built-in libraries
- Efficient handling of *intermediate datasets* (e.g. reliable storage in memory)



Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

Key Idea in Spark

Resilient Distributed Datasets (RDDs)

- Immutable collections of objects that can be stored in memory or disk across a cluster
- Built with parallel transformation operators (map, filter, …)
- Automatically rebuilt on failure

Think of these as a form of distributed, reliable virtual memory!

Example: Log Mining

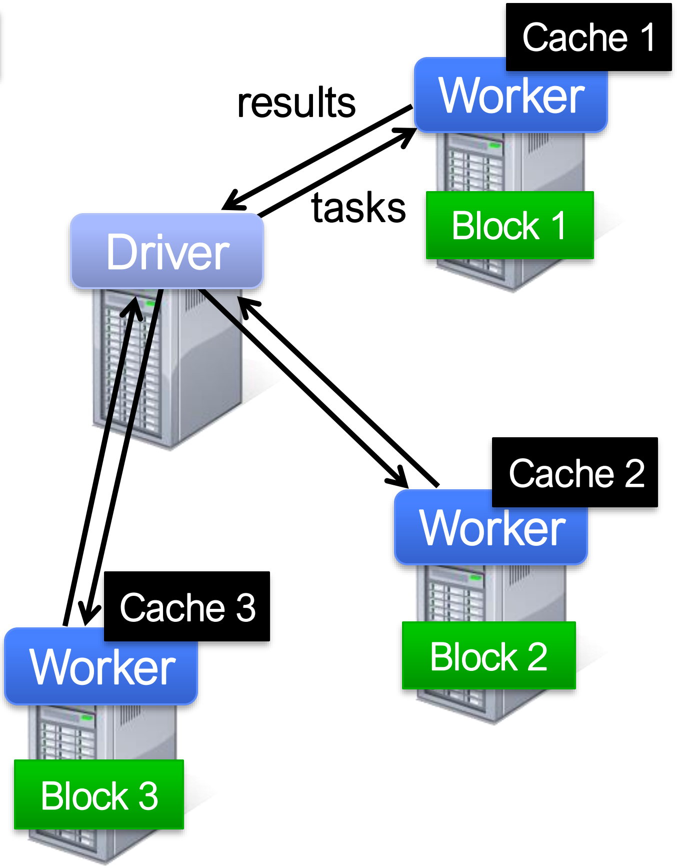
Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')(2))
messages.cache()


messages.filter(lambda s: s.contains("foo")).count()
messages.filter(lambda s: s.contains("bar")).count()
. . .
```

Base

Transformed RDD

Action

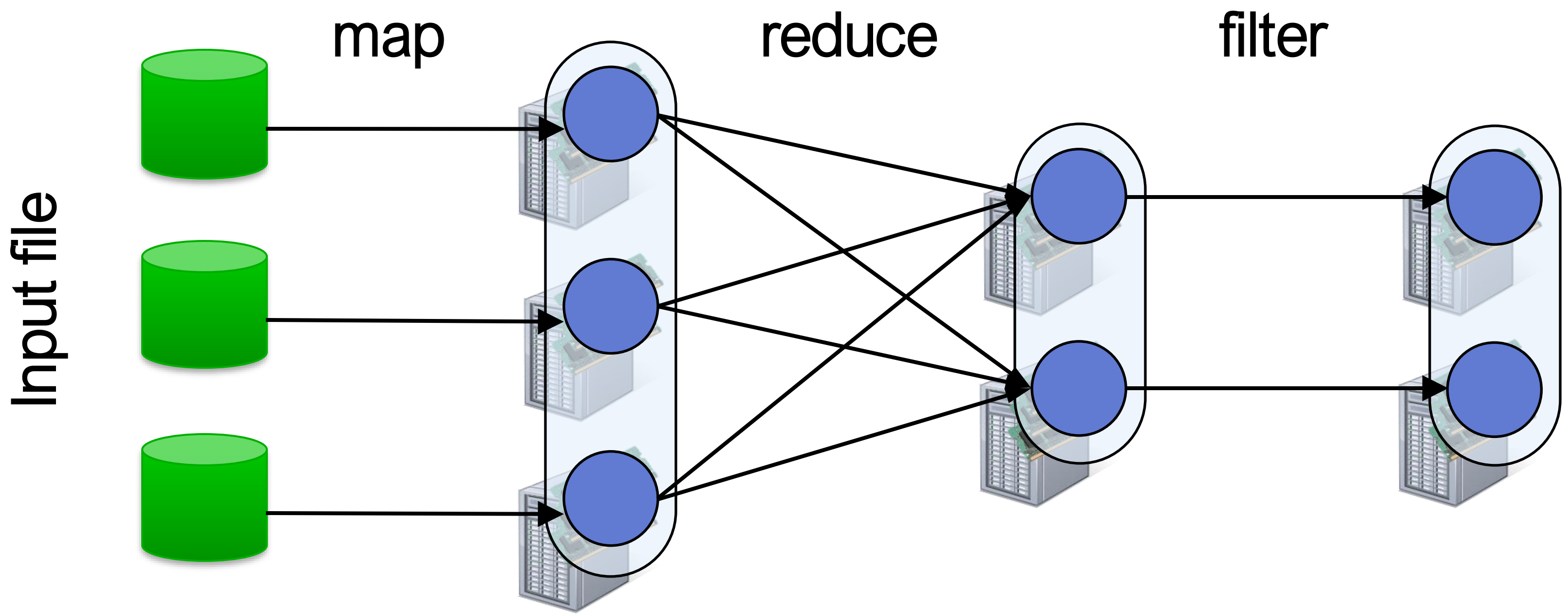


Performance: full-text search of Wikipedia in 1 sec (vs 40 s for on-disk data)

RDDs: Fault Tolerance

RDDs track *lineage* info to rebuild lost data

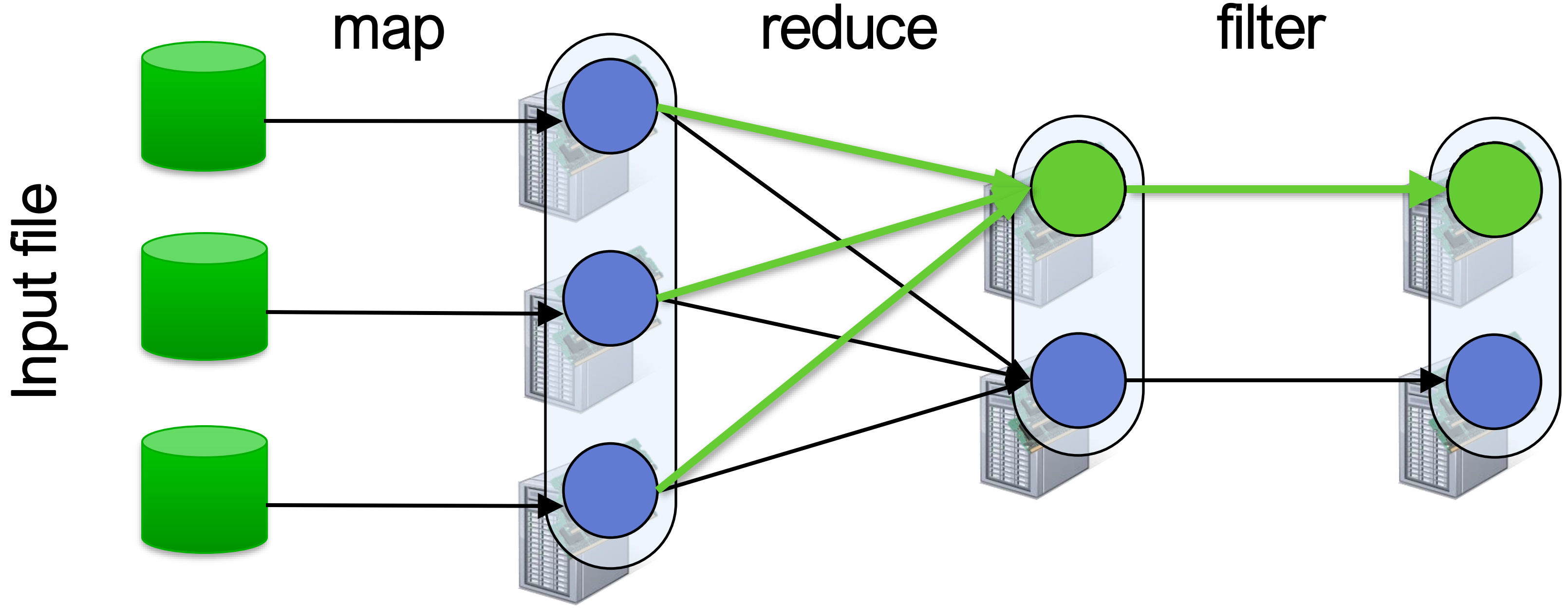
```
file.map(record => (record.type, 1))
   .reduceByKey((x, y) => x + y)
   .filter((type, count) => count > 10)
```



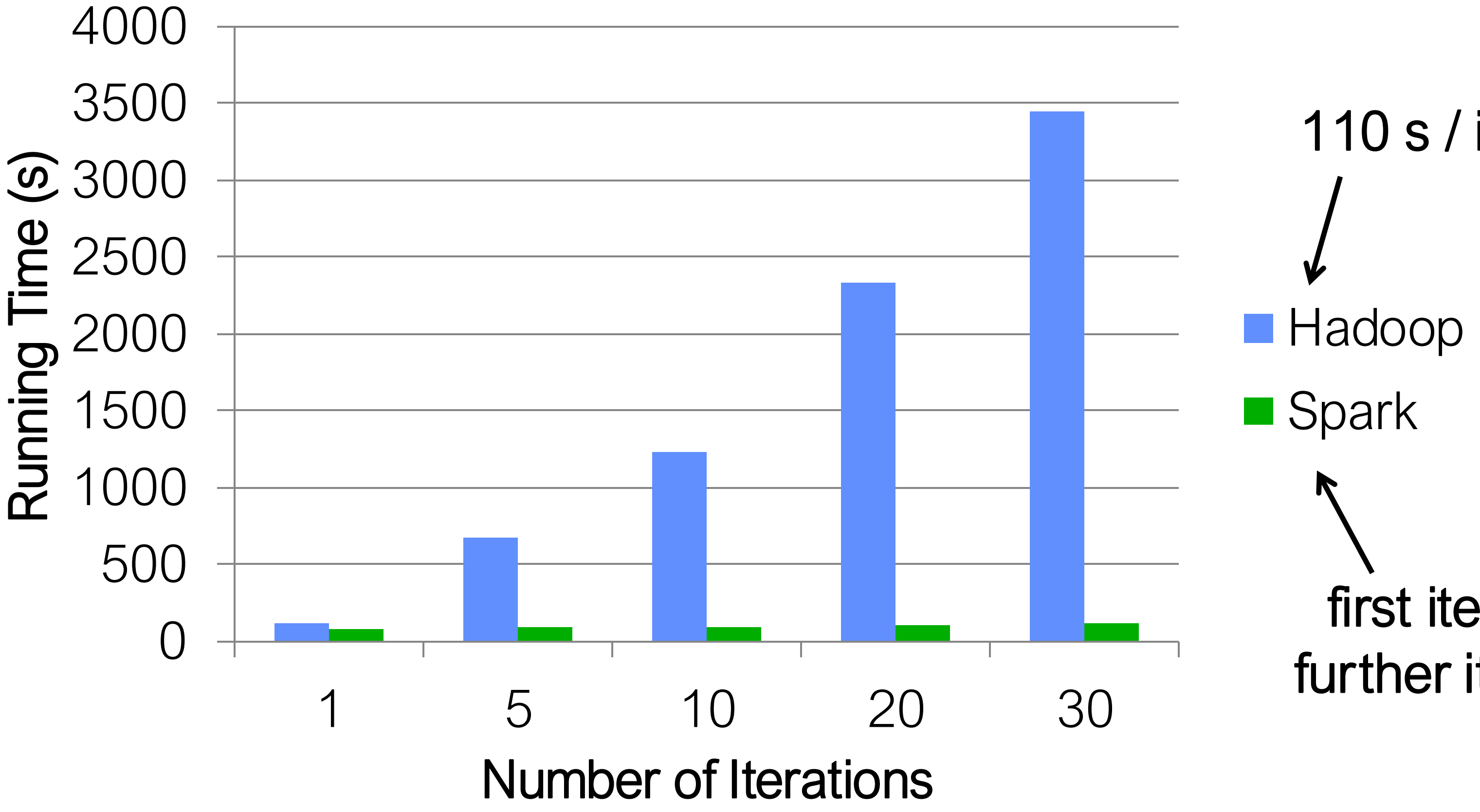
RDDs: Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(record => (record.type, 1))
   .reduceByKey((x, y) => x + y)
   .filter((type, count) => count > 10)
```



Iterative App Performance (Logistic Regression) with RDDs

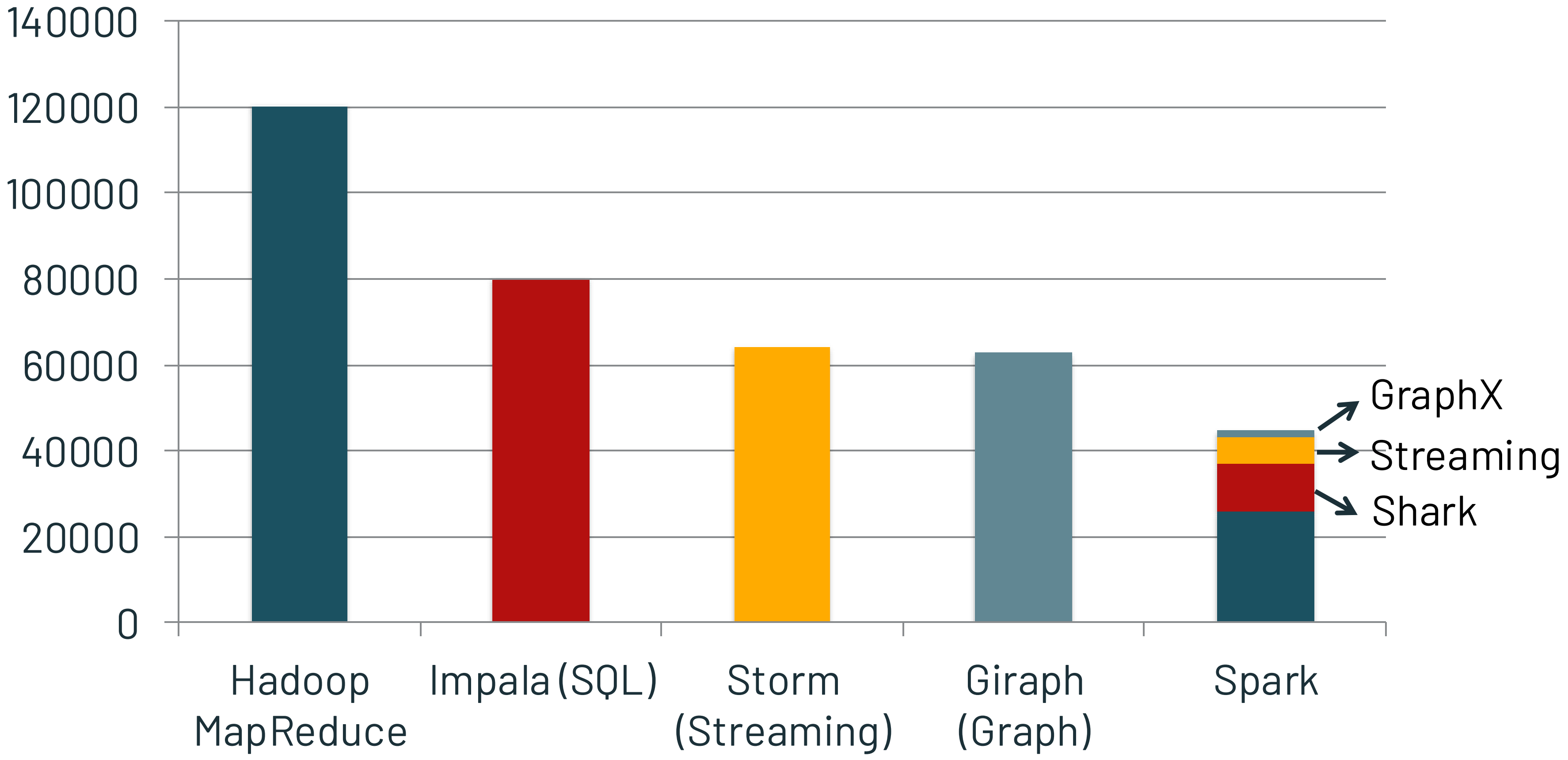


110 s / iteration

first iteration 80 s
further iterations 1 s

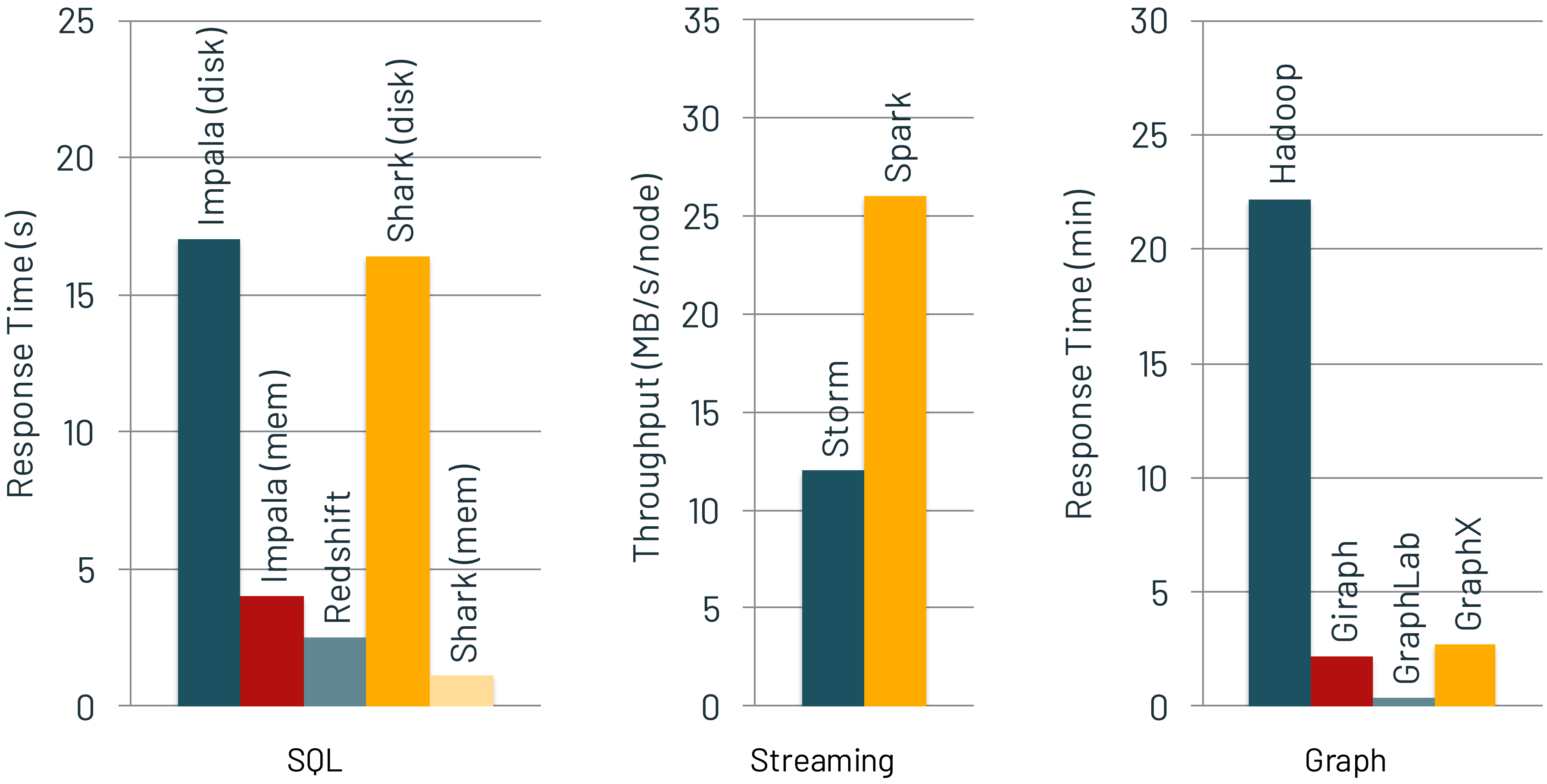
Early Spark Meetups

2013 Slide: Libraries Built on Spark



non-test, non-example source lines

2013 Slide: Performance Comparison



Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System

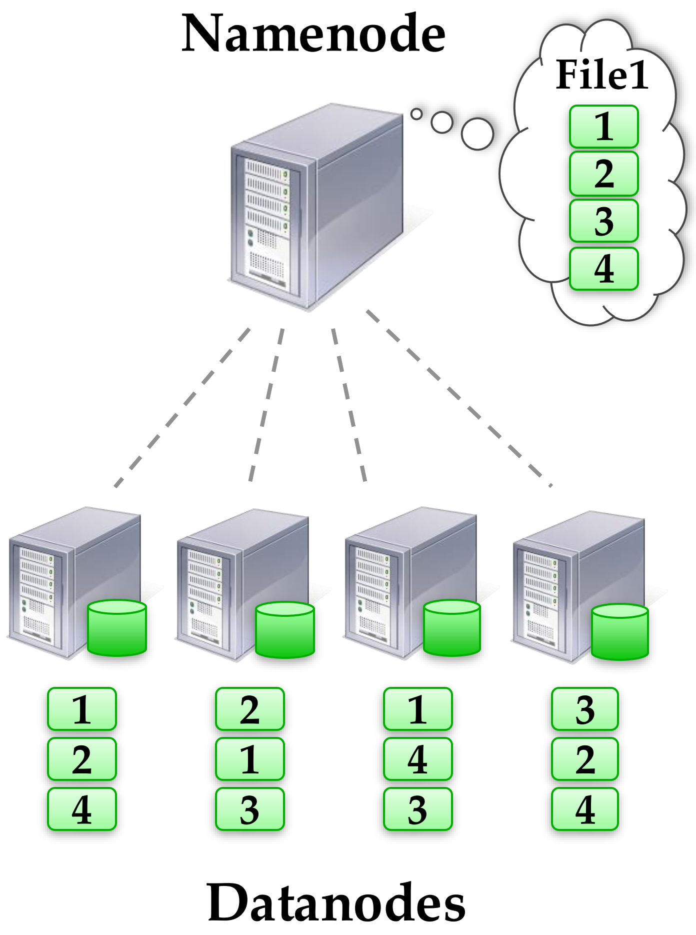
Files split into 128MB blocks

Blocks replicated across several datanodes (often 3)

Namenode stores metadata (file names, locations, etc)

Optimized for large files, sequential reads

Files are append-only


Namenode
File1
1
2
3
4
1
2
4
2
1
3
1
4
3
3
2
4
Datanodes