CLOUD COMPUTING

INTRODUCTION TO DATA SCIENCE

TIM KRASKA

teachingdatascience.org

CLICKER

How was the celebration of knowledge?

A) **Very Easy**

B) **Just ok**

C) **Tough**

D) **Spring break made me forget about it**

E) **I do not want to talk about it**

BACKGROUND OF CLOUD COMPUTING

1980's and 1990's: 52% growth in performance per year!

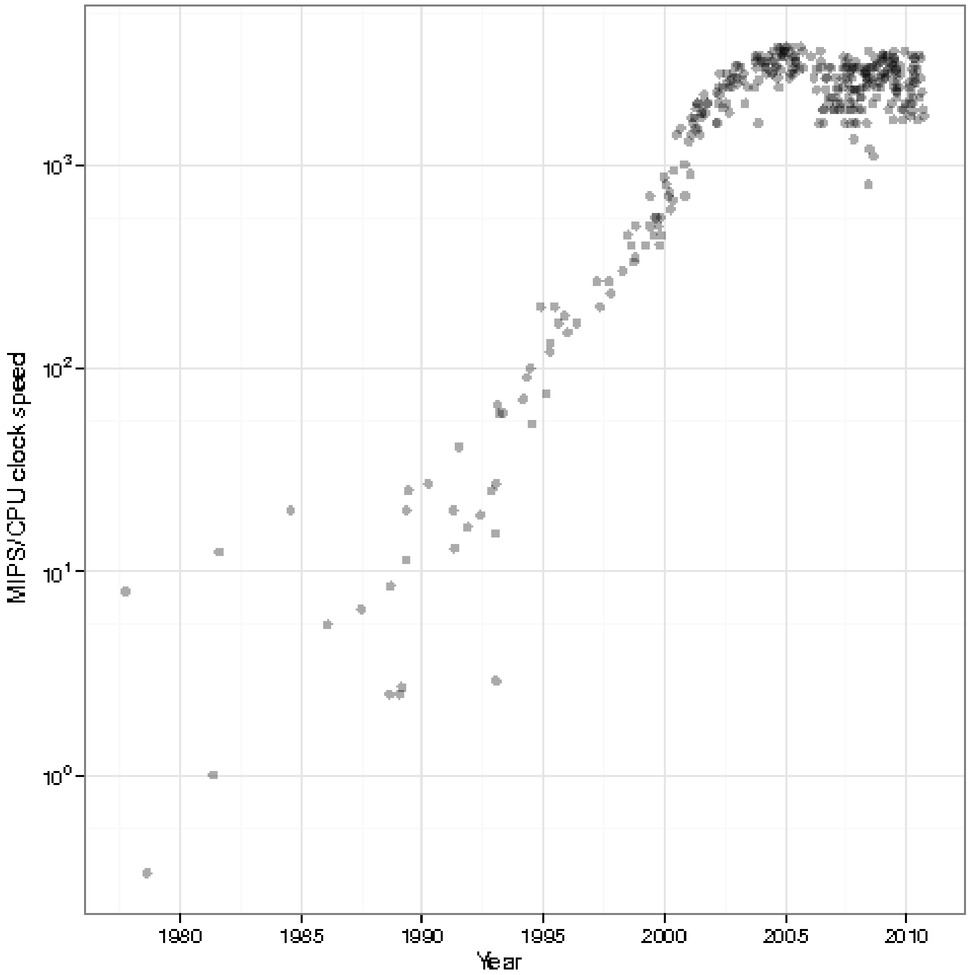
2002: The thermal wall

- Speed (frequency) peaks, but transistors keep shrinking

2000's: Multicore revolution

- 15-20 years later than predicted, we have hit the performance wall

2010's: Rise of Big Data



SOURCES DRIVING BIG DATA

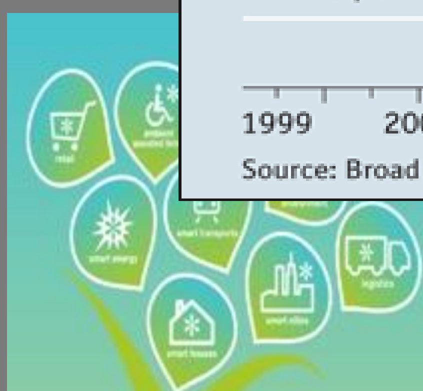
It's All Happening On-line

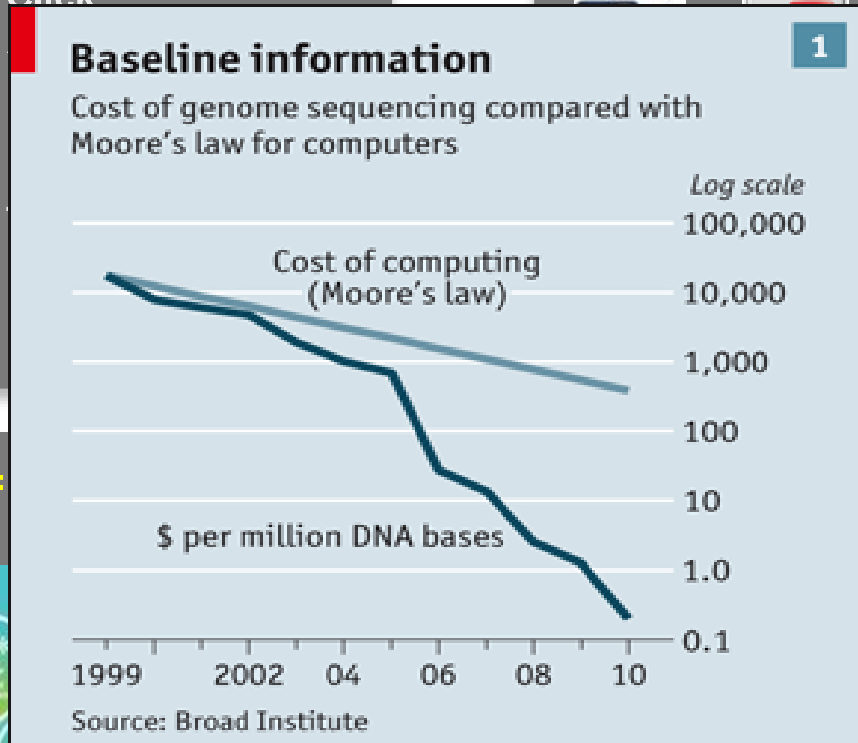
Every:
Click

User Generated (Web & Mobile)

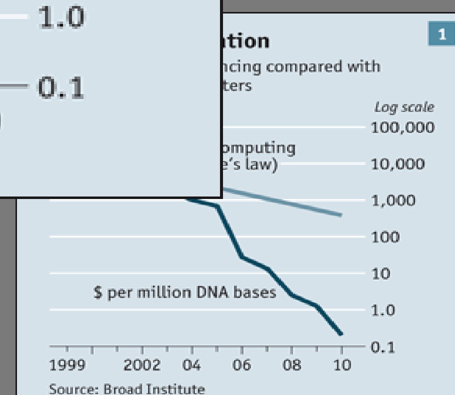
....

Internet of





Computing



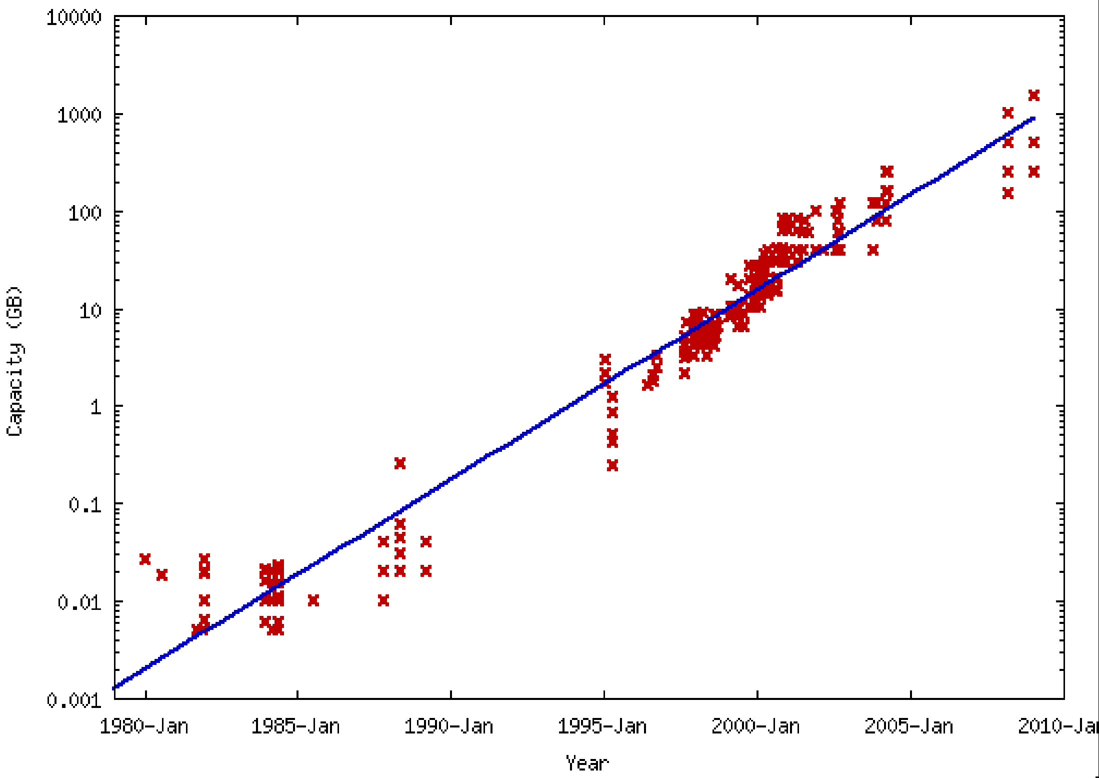
DATA DELUGE

Billions of users connected through the net

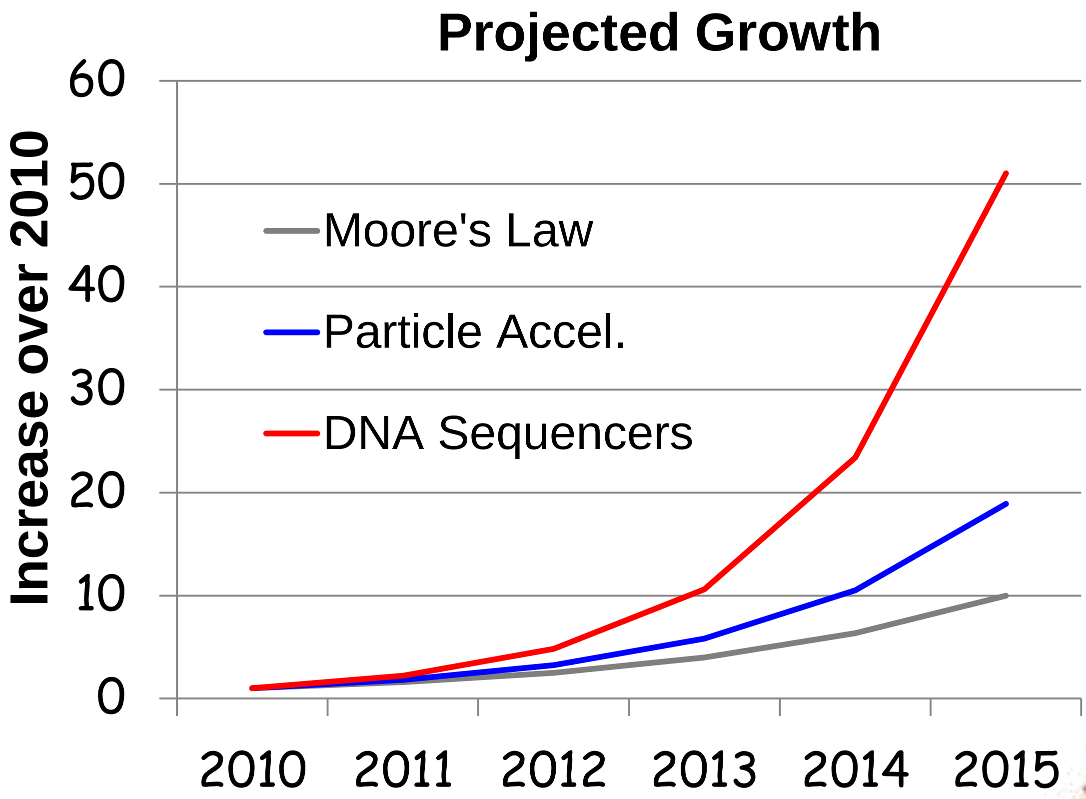
- WWW, FB, twitter, cell phones, …
- 80% of the data on FB was produced last year

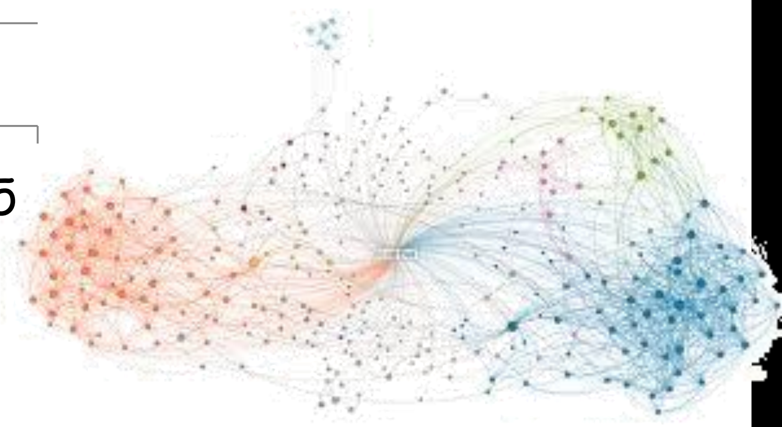
Storage getting cheaper

- Store more data!



DATA GROWS FASTER THAN MOORE'S LAW





SOLVING THE IMPEDANCE MISMATCH

Computers not getting faster, and we are drowning in data

- How to resolve the dilemma?

Solution adopted by web-scale companies

- Go massively *distributed* and *parallel*

ENTER THE WORLD OF DISTRIBUTED SYSTEMS

Distributed Systems/Computing

- *Loosely coupled* set of computers, communicating through message passing, solving a common goal
- Tools: Msg passing, Distributed shared memory, RPC

Distributed computing is *challenging*

- Dealing with *partial failures* (examples?)
- Dealing with *asynchrony* (examples?)
- Dealing with *scale* (examples?)
- Dealing with *consistency* (examples?)

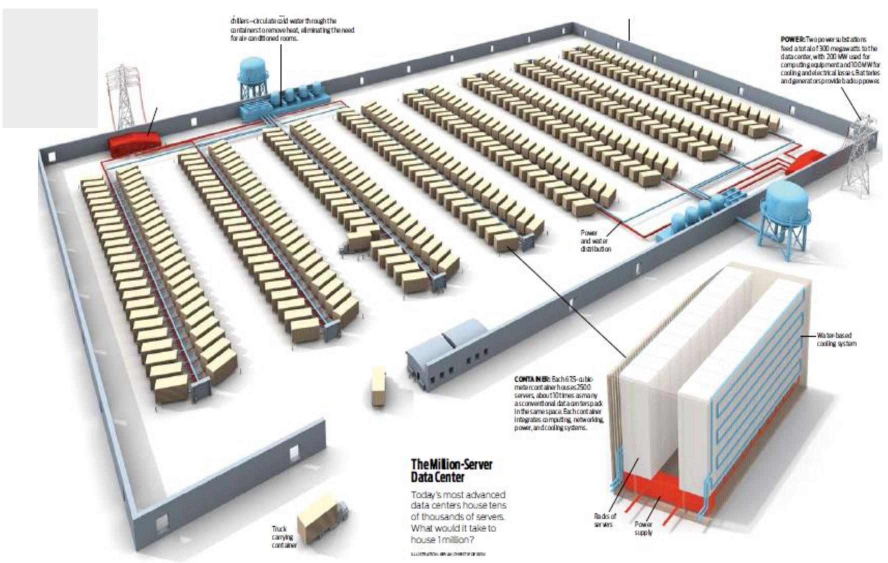
Distributed Computing versus Parallel Computing?

- distributed computing=parallel computing + partial failures

THE DATACENTER IS THE NEW COMPUTER

“The datacenter as a computer” still in its infancy

- Special purpose clusters, e.g., Hadoop cluster
- Built from less reliable components
- Highly variable performance
- Complex concepts are hard to program (low-level primitives)



=?

DATA CENTER

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf


HP Performance
Optimized Datacenters

DATACENTER/CLOUD COMPUTING OS

If the datacenter/cloud is the new computer

- What is its **Operating System**?
- Note that we are not talking about a host OS

Could be equivalent in benefit as the LAMP stack was to the .com boom – every startup *secretly* implementing the same functionality!

Open source stack for a Web 2.0 company:

- **L**inux OS
- **A**pache web server
- **M**ySQL, MariaDB or MongoDB DBMS
- **P**HP, Perl, or Python languages for dynamic web pages

CLASSICAL OPERATING SYSTEMS

Data sharing

- Inter-Process Communication, RPC, files, pipes, …

Programming Abstractions

- Libraries (libc), system calls, …

Multiplexing of resources

- Scheduling, virtual memory, file allocation/protection, …

DATACENTER/CLOUD OPERATING SYSTEM

Data sharing

- Google File System, key/value stores
- Apache project: Hadoop Distributed File System

Programming Abstractions

- Google MapReduce
- Apache projects: Hadoop, Pig, Hive, Spark

Multiplexing of resources

- Apache projects: Mesos, YARN (MapReduce v2), ZooKeeper, BookKeeper, …

GOOGLE CLOUD INFRASTRUCTURE

Google File System (GFS), 2003

- Distributed File System for entire cluster
- Single namespace

Google MapReduce (MR), 2004

- Runs queries/jobs on data
- Manages work distribution & fault-tolerance
- Colocated with file system

The Google File System

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

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive com-

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 pro-

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 in-

Apache open source versions: Hadoop DFS and Hadoop MR

HADOOP DISTRIBUTED FILE SYSTEM

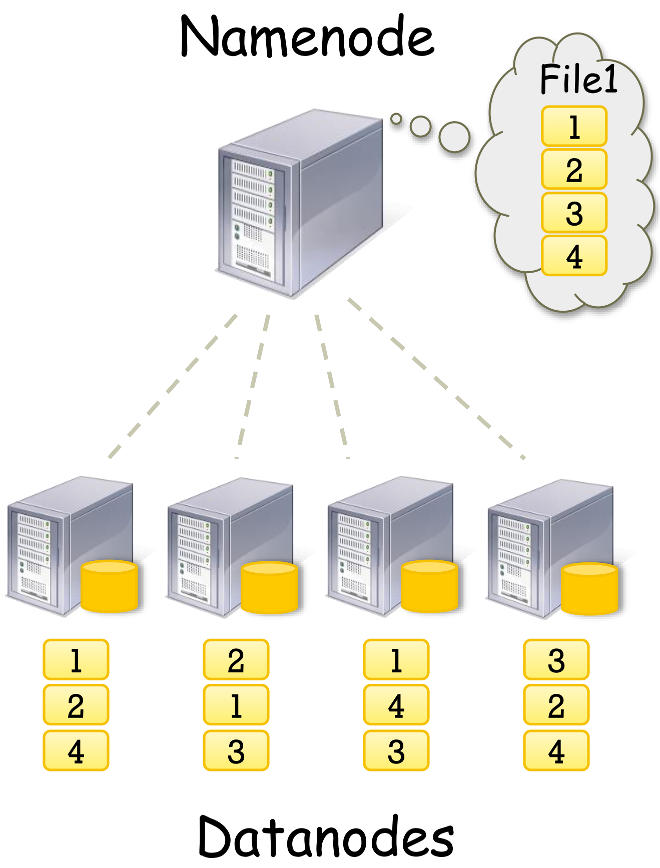
Files split into 128MB *blocks*

Blocks replicated across several *datanodes* (usually 3)

Single *namenode* stores metadata (file names, block locations, etc)

Optimized for large files, sequential reads

Files are append-only



Data *striped* on hundreds/thousands of servers

- Scan 100 TB on 1 node @ 50 MB/s = 24 days
- Scan on 1000-node cluster = 35 minutes

CLICKER

The chance of a machine failing in 24h is 0.1%

What is the likelihood that one machine in a cluster of 1000 machines fails in 24h?

a) 0.1%

b) 10%

c) 63%

d) 99.999%

GFS/HDFS INSIGHTS (2)

***Failures* will be the norm**

Mean time between failures for 1 node = 3 years

Mean time between failures for 1000 nodes = **1 day**

Use *commodity* hardware

Failures are the norm anyway, buy cheaper hardware

No complicated consistency models

Single writer, append-only data

WHAT IS MAPREDUCE?

Simple data-parallel programming model designed for scalability and fault-tolerance

Pioneered by Google

- Processes 20 petabytes of data per day

Popularized by open-source Hadoop project

- Used at Yahoo!, Facebook, Amazon, …

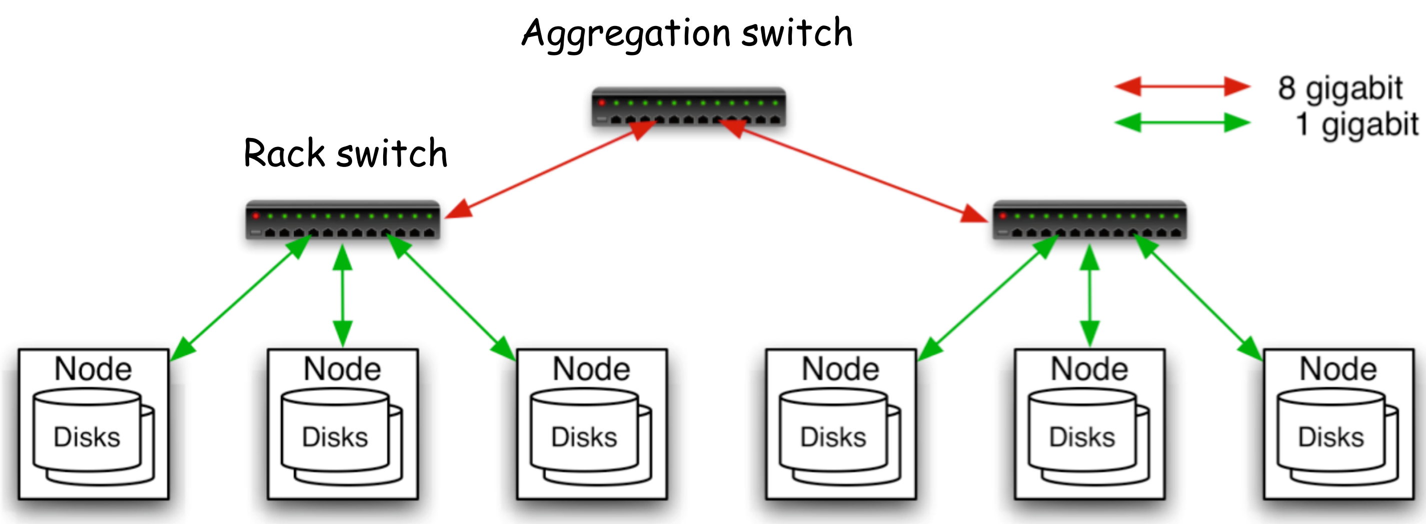
WHAT IS MAPREDUCE USED FOR?

- **At Google:**
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation

- **At Yahoo!:**
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail

- **At Facebook:**
  - Data mining
  - Ad optimization
  - Spam detection

TYPICAL HADOOP CLUSTER



40 nodes/rack, 1000-4000 nodes in cluster

1 Gbps bandwidth within rack, 8 Gbps out of rack

Node specs (Yahoo terasort):

8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

CHALLENGES

Cheap nodes fail, especially if you have many

Mean time between failures for 1 node = 3 years

Mean time between failures for 1000 nodes = 1 day

Solution: Build fault-tolerance into system

Commodity network = low bandwidth

Solution: Push computation to the data

Programming distributed systems is hard

Solution: Data-parallel programming model: users write "map" & "reduce" functions, system distributes work and handles faults

MAPREDUCE PROGRAMMING MODEL

Data type: key-value *records*

Map function:

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

Reduce function:

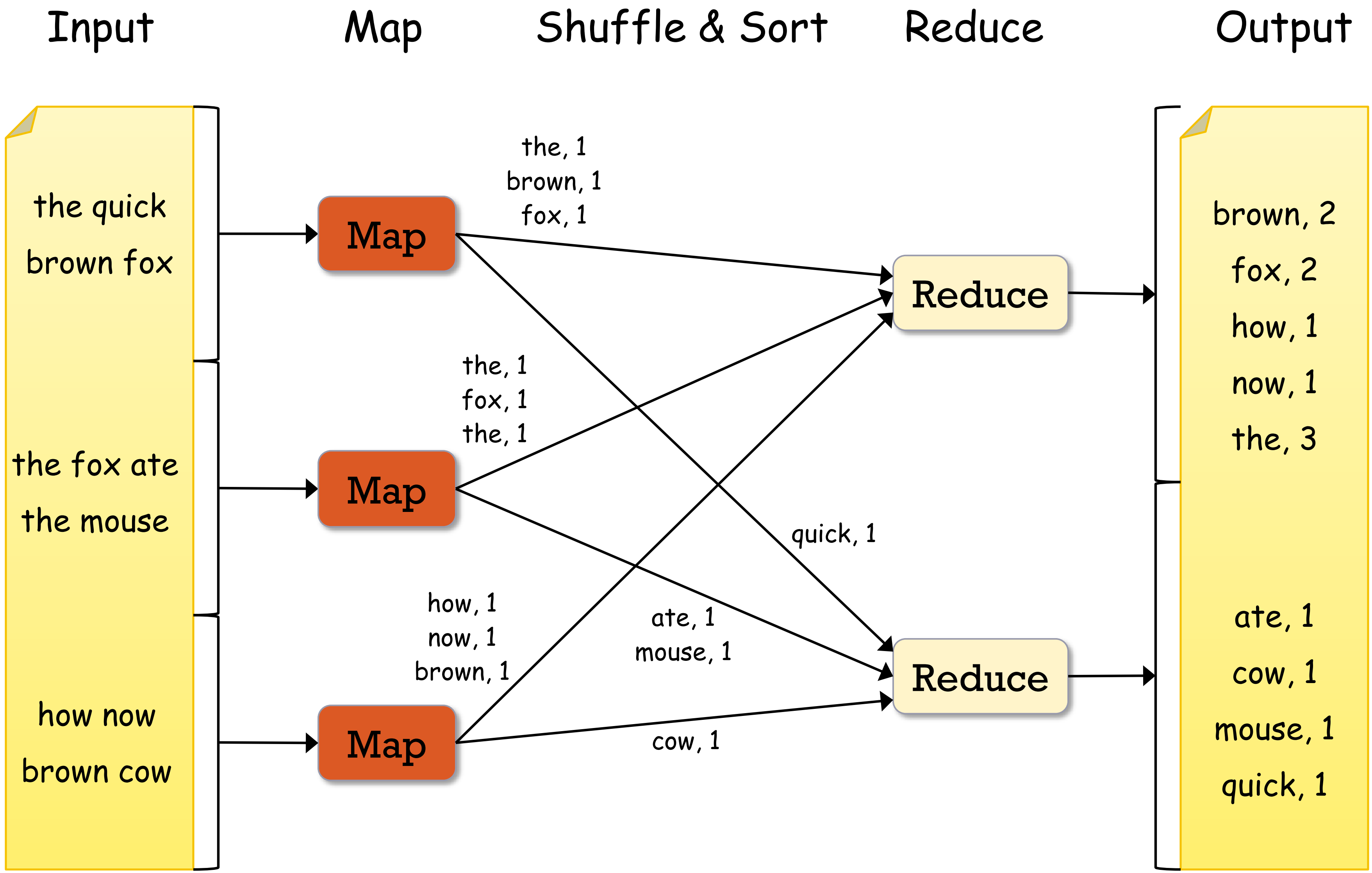
$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

EXAMPLE: WORD COUNT

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

WORD COUNT EXECUTION



MAPREDUCE EXECUTION DETAILS

Single *master* controls job execution on multiple *slaves*

Mappers preferentially placed on same node or same rack as their input block

- Minimizes network usage

Mappers save outputs to local disk before serving them to reducers

- Allows recovery if a reducer crashes
- Allows having more reducers than nodes

AN OPTIMIZATION: THE COMBINER

A combiner is a local aggregation function for repeated keys produced by same map

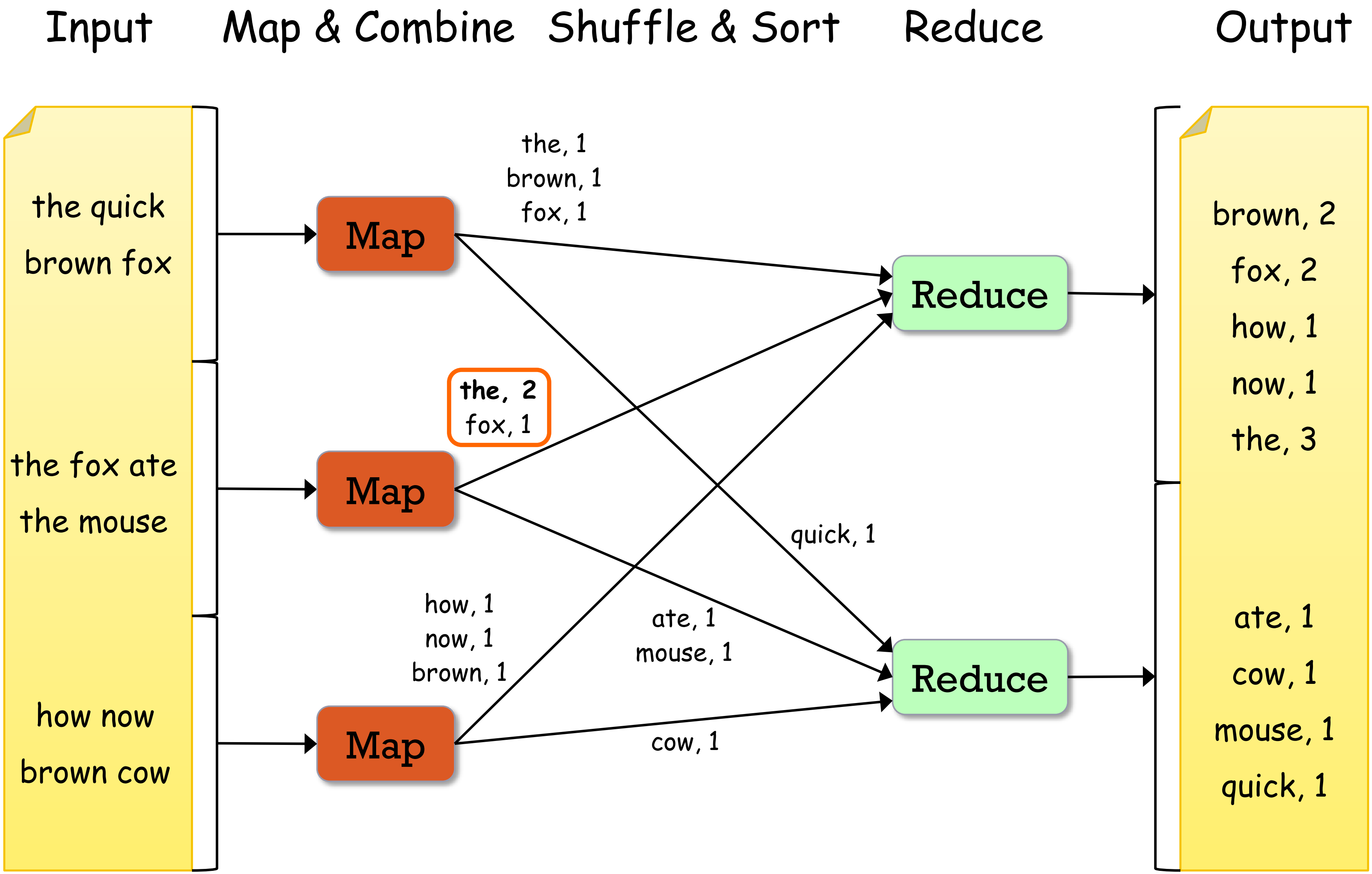
Works for associative functions like sum, count, max

Decreases size of intermediate data

Example: map-side aggregation for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

WORD COUNT WITH COMBINER



FAULT TOLERANCE IN MAPREDUCE

1. If a task crashes:

- Retry on another node
  - OK for a map because it has no dependencies
  - OK for reduce because map outputs are on disk
- If the same task fails repeatedly, fail the job or ignore that input block (user-controlled)

➢ Note: For these fault tolerance features to work, *your map and reduce tasks must be side-effect-free*

FAULT TOLERANCE IN MAPREDUCE

2. If a node crashes:

- Re-launch its current tasks on other nodes
- Re-run 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 ("speculative execution")
- Take the output of whichever copy finishes first, and kill the other

➢ **Surprisingly important in large clusters**

- Stragglers occur frequently due to failing hardware, software bugs, misconfiguration, etc
- Single straggler may noticeably slow down a job

TAKEAWAYS

By providing a data-parallel programming model, MapReduce can control job execution in useful ways:

- Automatic division of job into tasks
- Automatic placement of computation near data
- Automatic load balancing
- Recovery from failures & stragglers

User focuses on application, not on complexities of distributed computing

MAPREDUCE PROS

Distribution is completely transparent

- Not a single line of distributed programming (ease, correctness)

Automatic fault-tolerance

- Determinism enables running failed tasks somewhere else again
- Saved intermediate data enables just re-running failed reducers

Automatic scaling

- As operations as side-effect free, they can be distributed to any number of machines dynamically

Automatic load-balancing

- Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers*)