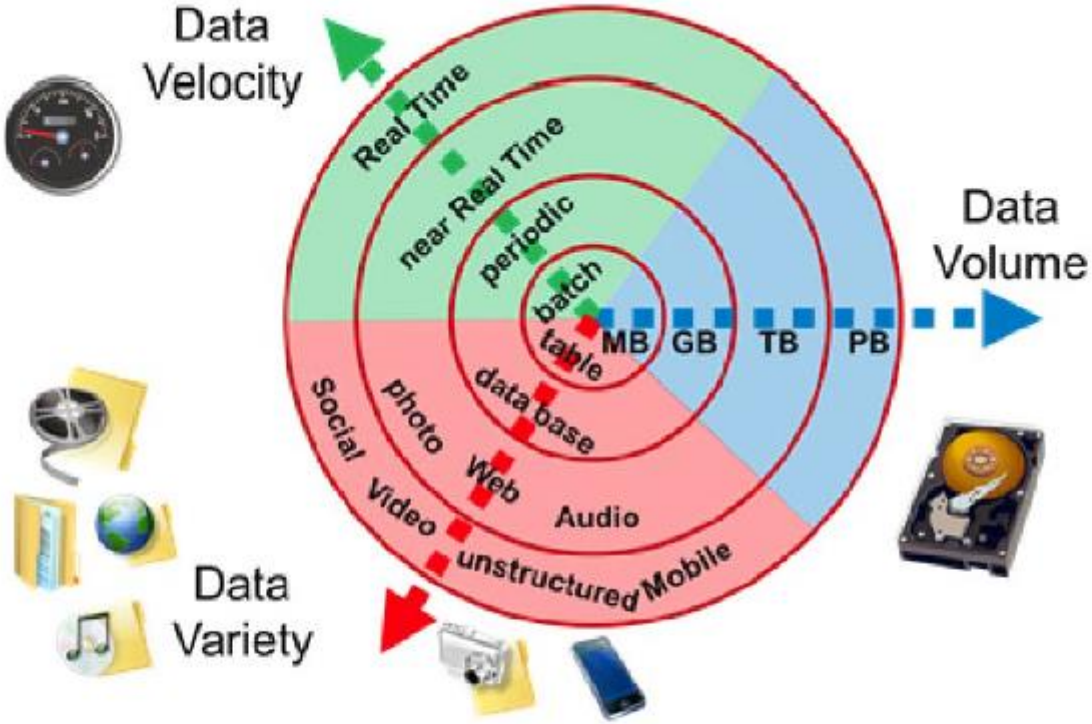
Introduction to Cloud, MapReduce, Hadoop, HDFS

Thanks to D. Tsoumakos for the slides
Material adapted from slides by **Jimmy Lin, The iSchool – University of Maryland**
And **from www.cloudcomputingchina.com**

IONIAN UNIVERSITY
DEPARTMENT OF INFORMATICS

The Big Data era

The 3 (or more) Vs



Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)

400B pages, 10+ PB (2/2014)

Hadoop: 365 PB, 330K nodes (6/2014)

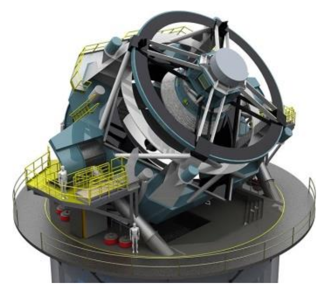
150 PB on 50k+ servers running 15k apps (6/2011)

Hadoop: 10K nodes, 150K cores, 150 PB (4/2014)

300 PB data in Hive + 600 TB/day (4/2014)

LHC: ~15 PB a year

S3: 2T objects, 1.1M request/second (4/2013)



LSST: 6-10 PB a year (~2020)

640K ought to be enough for anybody.

SKA: 0.3 – 1.5 EB per year (~2020)

How much data?

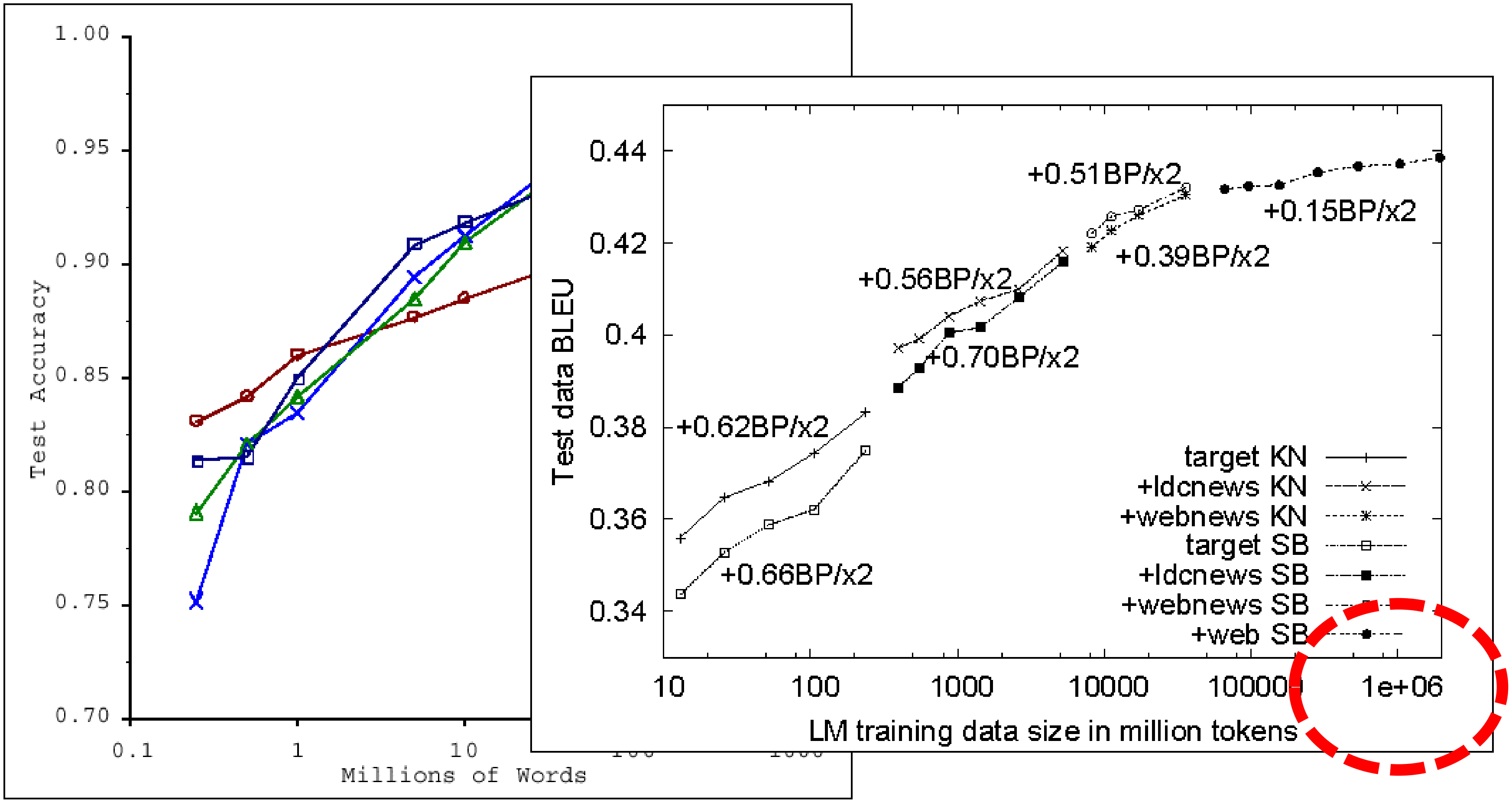
1 EB (Exabyte=10^{18}bytes) = 1000 PB (Petabyte=10^{15}bytes)
Κίνηση δεδομένων κινητής τηλεφωνίας στις ΗΠΑ για το 2010

1.2 ZB (Zettabyte) = 1200 EB
Σύνολο ψηφιακών δεδομένων το 2010

35 ZB (Zettabyte = 10^{21} bytes)
Εκτίμηση για σύνολο ψηφιακών δεδομένων το 2020

No data like more data!

s/knowledge/data/g;



How do we get here if we're not Google?

(Banko and Brill, ACL 2001)
(Brants et al., EMNLP 2007)

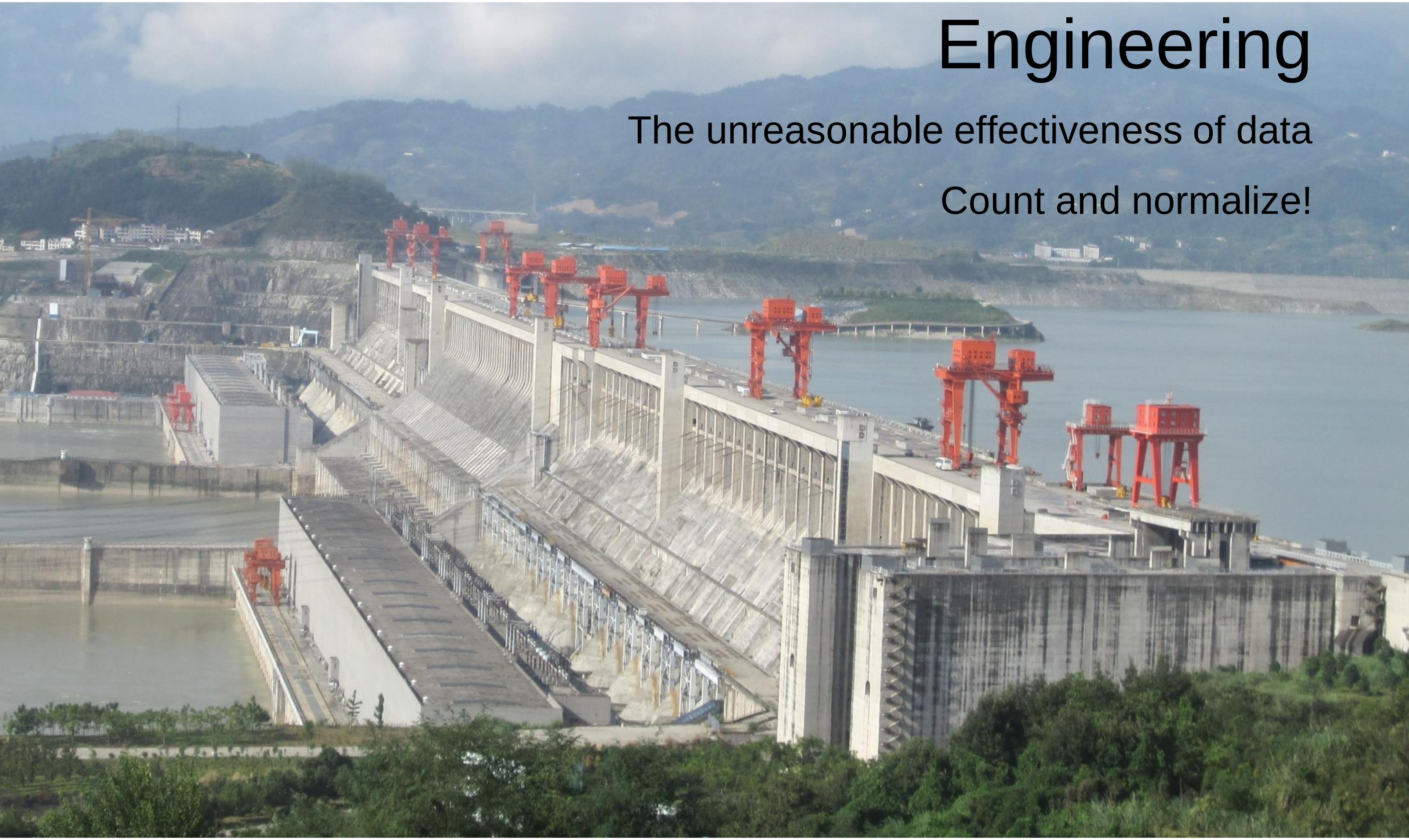
Science

Emergence of the 4th Paradigm

Engineering

The unreasonable effectiveness of data

Count and normalize!

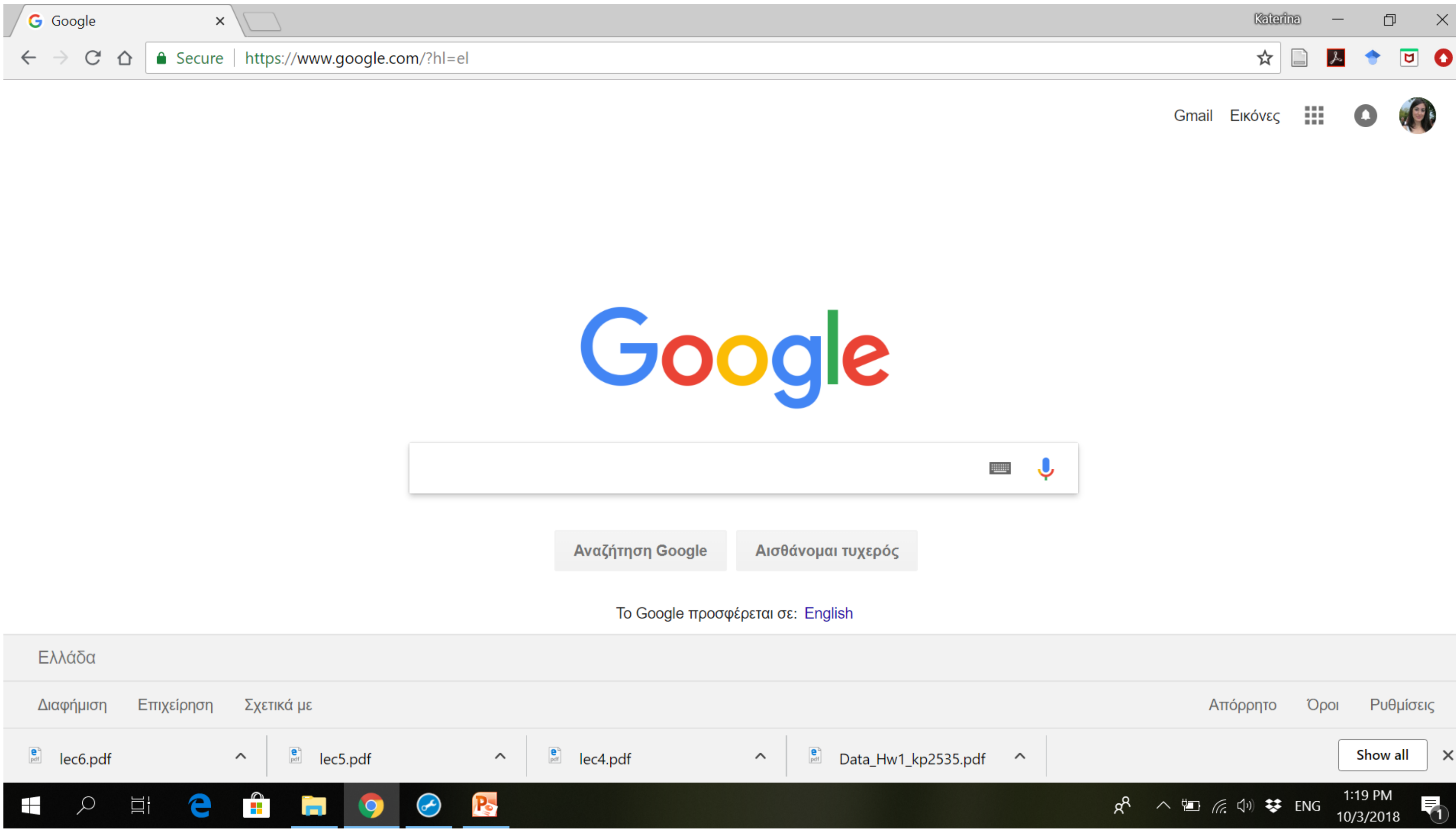


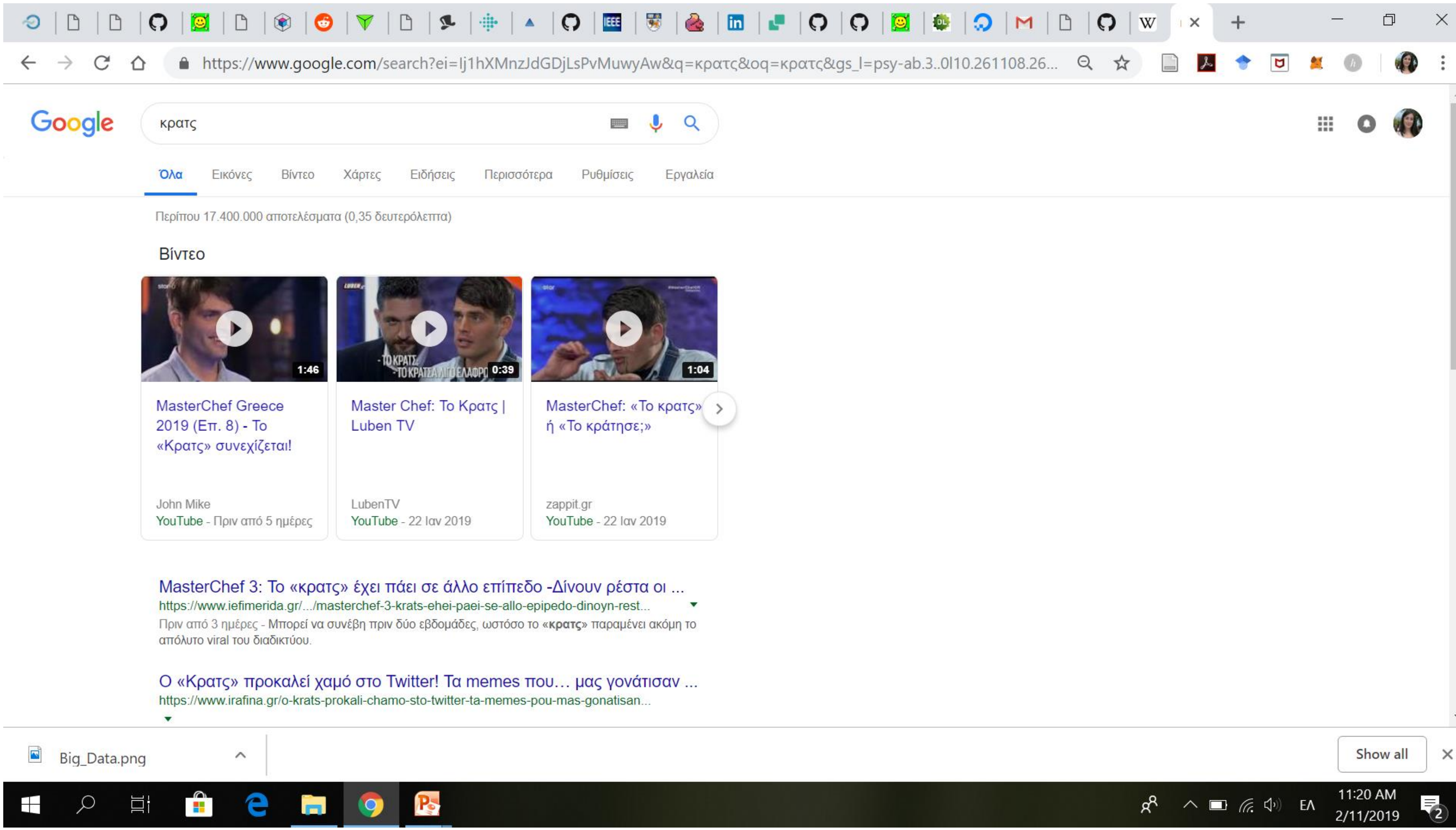
Know thy customers

Data → Insights → Competitive advantages

Commerce

How it all started...



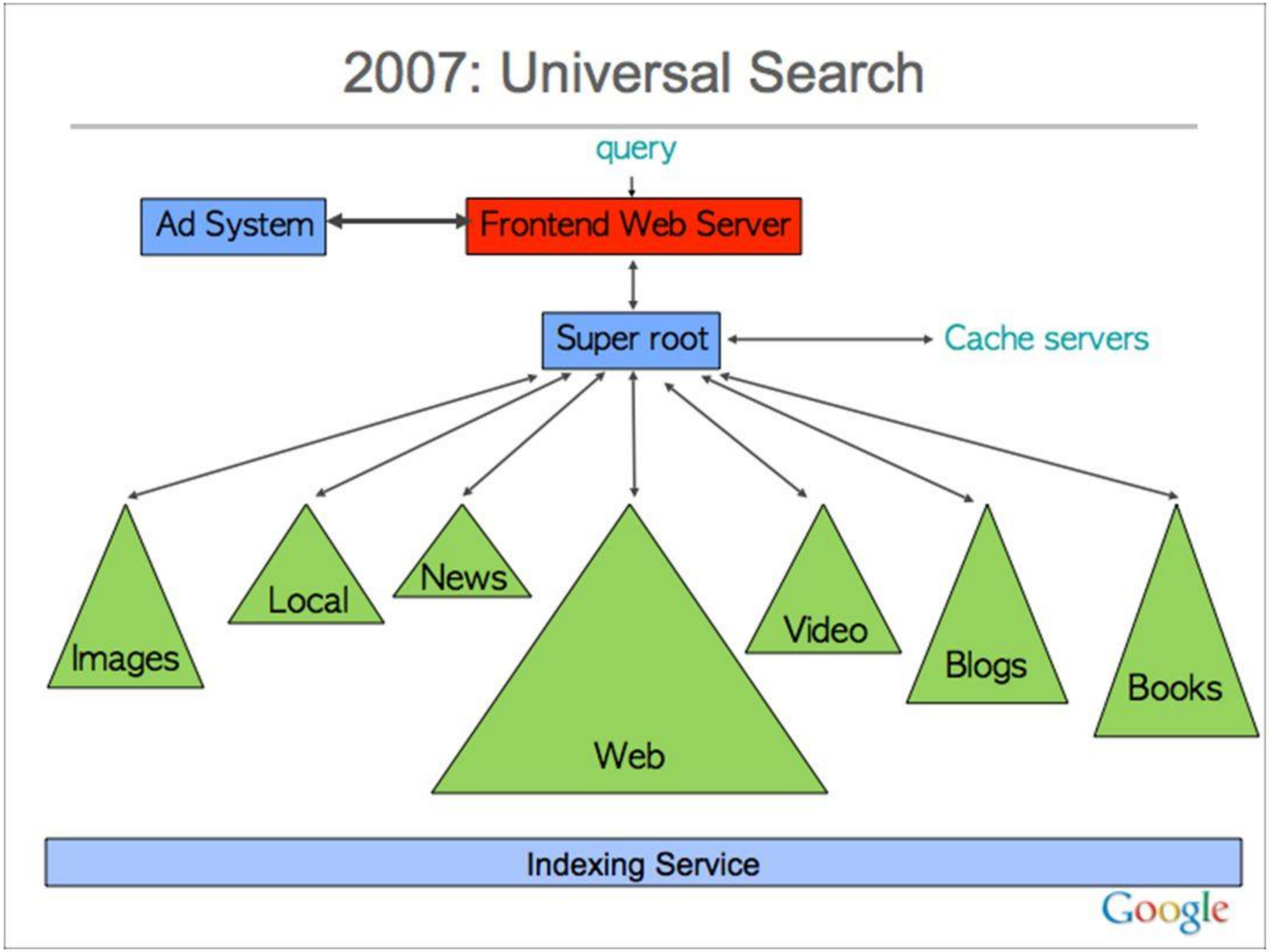

https://www.google.com/search?ei=lj1hXMnzJdGDjLsPvMuwyAw&q=κρατς&oq=κρατς&gs_l=psy-ab.3..0l10.261108.26...
Google
κρατς
Όλα
Εικόνες
Βίντεο
Χάρτες
Ειδήσεις
Περισσότερα
Ρυθμίσεις
Εργαλεία
Περίπου 17.400.000 αποτελέσματα (0,35 δευτερόλεπτα)
Βίντεο
1:46
0:39
1:04
MasterChef Greece 2019 (Επ. 8) - Το «Κρατς» συνεχίζεται!
John Mike
YouTube - Πριν από 5 ημέρες
Master Chef: Το Κρατς | Luben TV
LubenTV
YouTube - 22 Ιαν 2019
MasterChef: «Το κρατς» ή «Το κράτησε;»
zappit.gr
YouTube - 22 Ιαν 2019
MasterChef 3: Το «κρατς» έχει πάει σε άλλο επίπεδο -Δίνουν ρέστα οι ...
https://www.iefimerida.gr/.../masterchef-3-krats-ehei-paei-se-allo-epipedo-dinoyn-rest...
Πριν από 3 ημέρες - Μπορεί να συνέβη πριν δύο εβδομάδες, ωστόσο το «κρατς» παραμένει ακόμη το απόλυτο viral του διαδικτύου.
Ο «Κρατς» προκαλεί χαμό στο Twitter! Τα memes που… μας γονάτισαν ...
https://www.irafina.gr/o-krats-prokali-chamo-sto-twitter-ta-memes-pou-mas-gonatisan...
Big_Data.png
Show all
11:20 AM
2/11/2019
ΕΛ

Τι κάνει η Google;

1. crawling
2. indexing
3. $$$

Πώς δεικτοδοτείς το διαδίκτυο;

- Πάνω από 1 τρις μοναδικά URLs
- Δισεκατομμύρια μοναδικές ιστοσελίδες
- Exabytes κειμένου



slide from Jeff Dean, Google

Ένα Google Datacenter

...κι από μέσα

- Εκατομμύρια cores
- ~20k κόμβοι για ένα tasks

What is cloud computing?

Just a buzzword?

- Before clouds…
  - P2P computing
  - Grids
  - HPC
  - …
- Cloud computing means many different things:
  - Large-data processing
  - Rebranding of web 2.0
  - Utility computing
  - Everything as a service

Rebranding of web 2.0

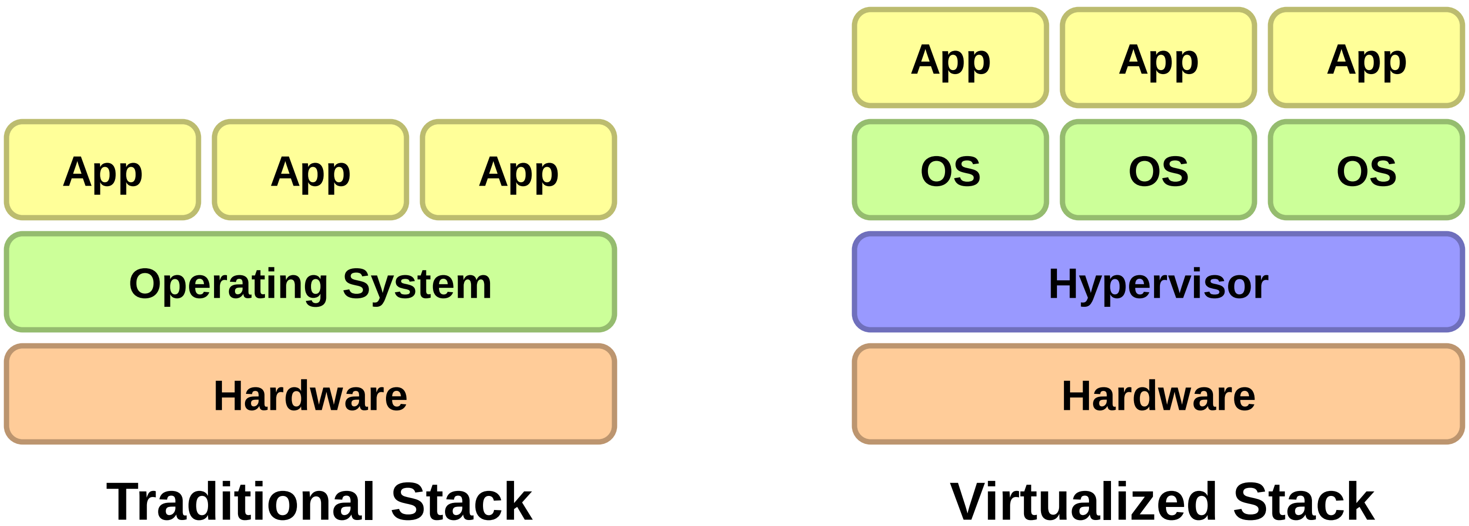
- Rich, interactive web applications
  - Clouds refer to the servers that run them
  - AJAX as the de facto standard (for better or worse)
  - Examples: Facebook, YouTube, Gmail, …
- “The network is the computer”: take two
  - User data is stored “in the clouds”
  - Rise of the netbook, smartphones, etc.
  - Browser *is* the OS

Utility Computing

- What?
  - Computing resources as a metered service ("pay as you go")
  - Ability to dynamically provision virtual machines
- Why?
  - Cost: capital vs. operating expenses
  - Scalability: "infinite" capacity
  - Elasticity: scale up or down on demand
- Does it make sense?
  - Benefits to cloud users
  - Business case for cloud providers

I think there is a world market for about five computers.

Enabling Technology: Virtualization



Cloud computing market

Software as a service

Everything is a service

Platform as a service

Infrastructure as a service

Cloud technology enabler

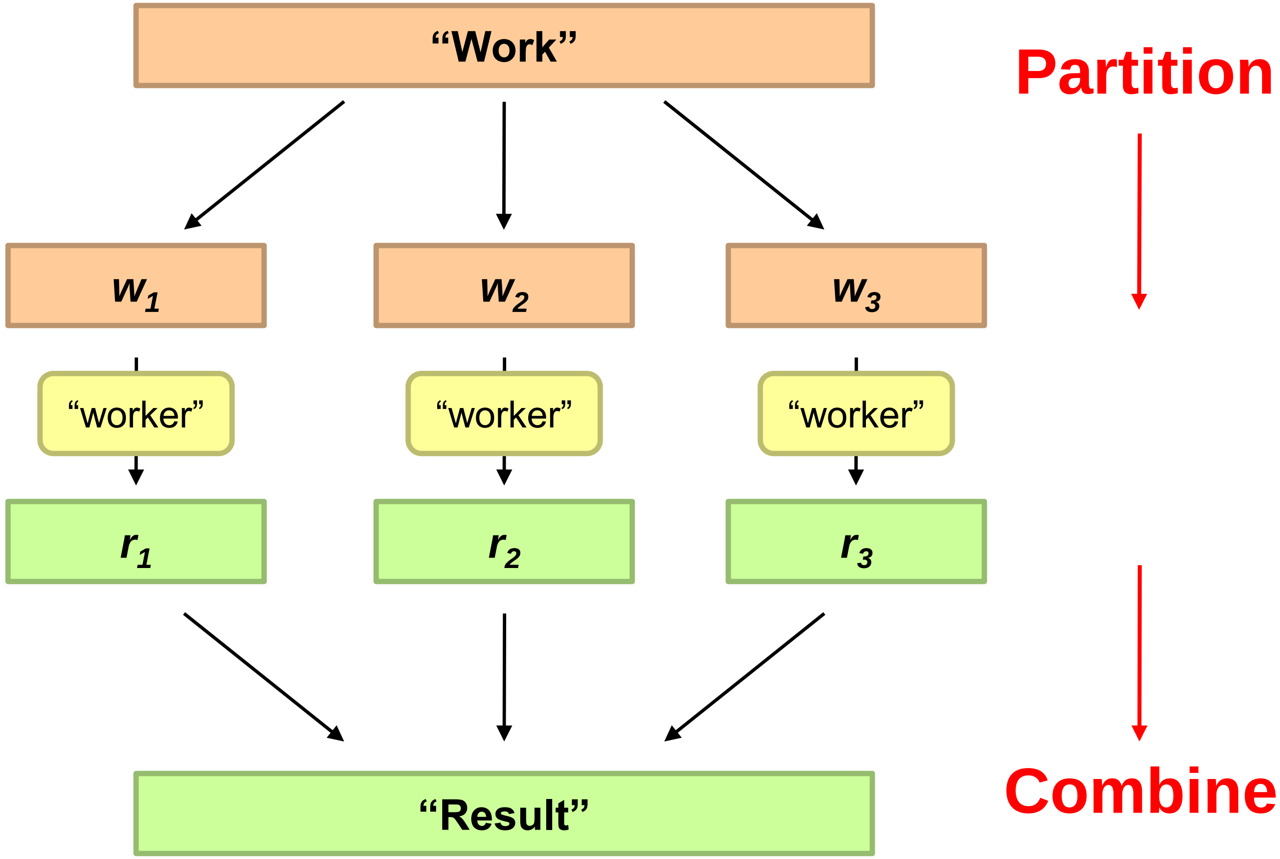
Hardware provider

Everything as a Service

- Utility computing = Infrastructure as a Service (IaaS)
  - Why buy machines when you can rent cycles?
  - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
  - Give me nice API and take care of the maintenance, upgrades, …
  - Example: Google App Engine
- Software as a Service (SaaS)
  - Just run it for me!
  - Example: Gmail, Salesforce

How do we scale up?

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

Synchronization!

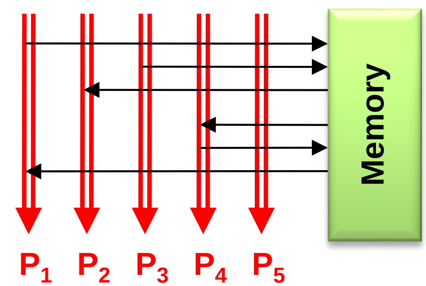
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

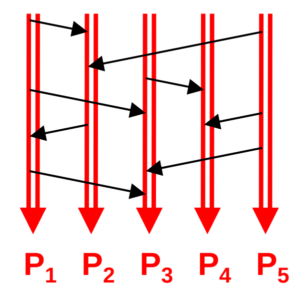
Managing Multiple Workers

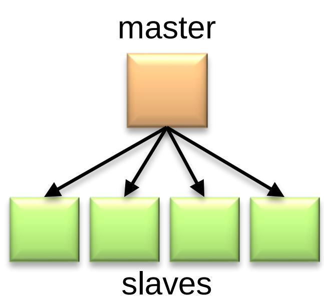
- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

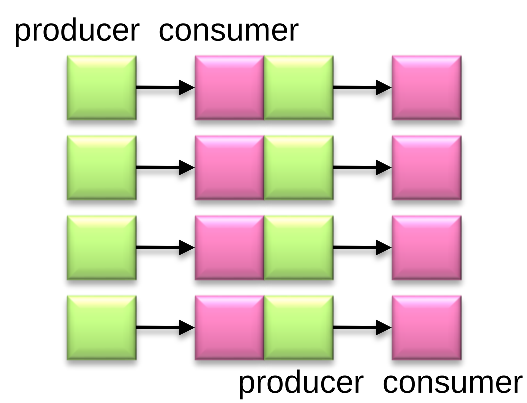
Current Tools

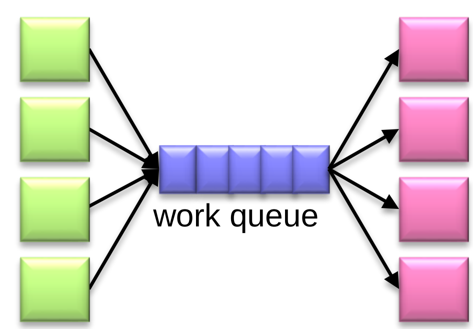
- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues











What's the point?

- It's all about the right level of abstraction
  - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!

MapReduce

What is MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
- Open-source implementation called Hadoop

Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Challenges

1. **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

2. **Commodity network = low bandwidth**
   - Solution: Push computation to the data

3. **Programming distributed systems is hard**
   - Solution: Data-parallel programming model: users write “map” & “reduce” functions, system distributes work and handles faults

MapReduce

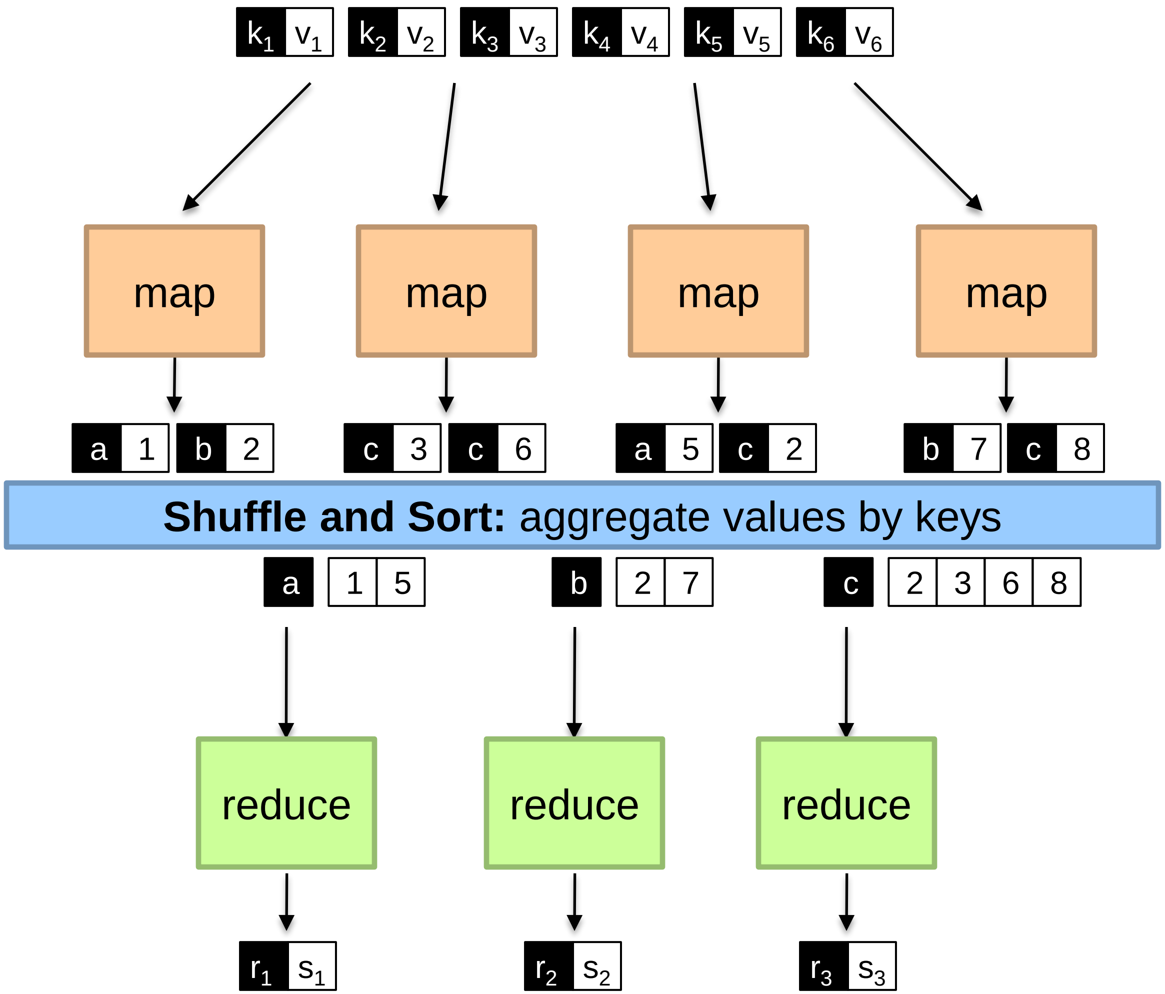
- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’) → <k’’, v’’>*

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What’s “everything else”?


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

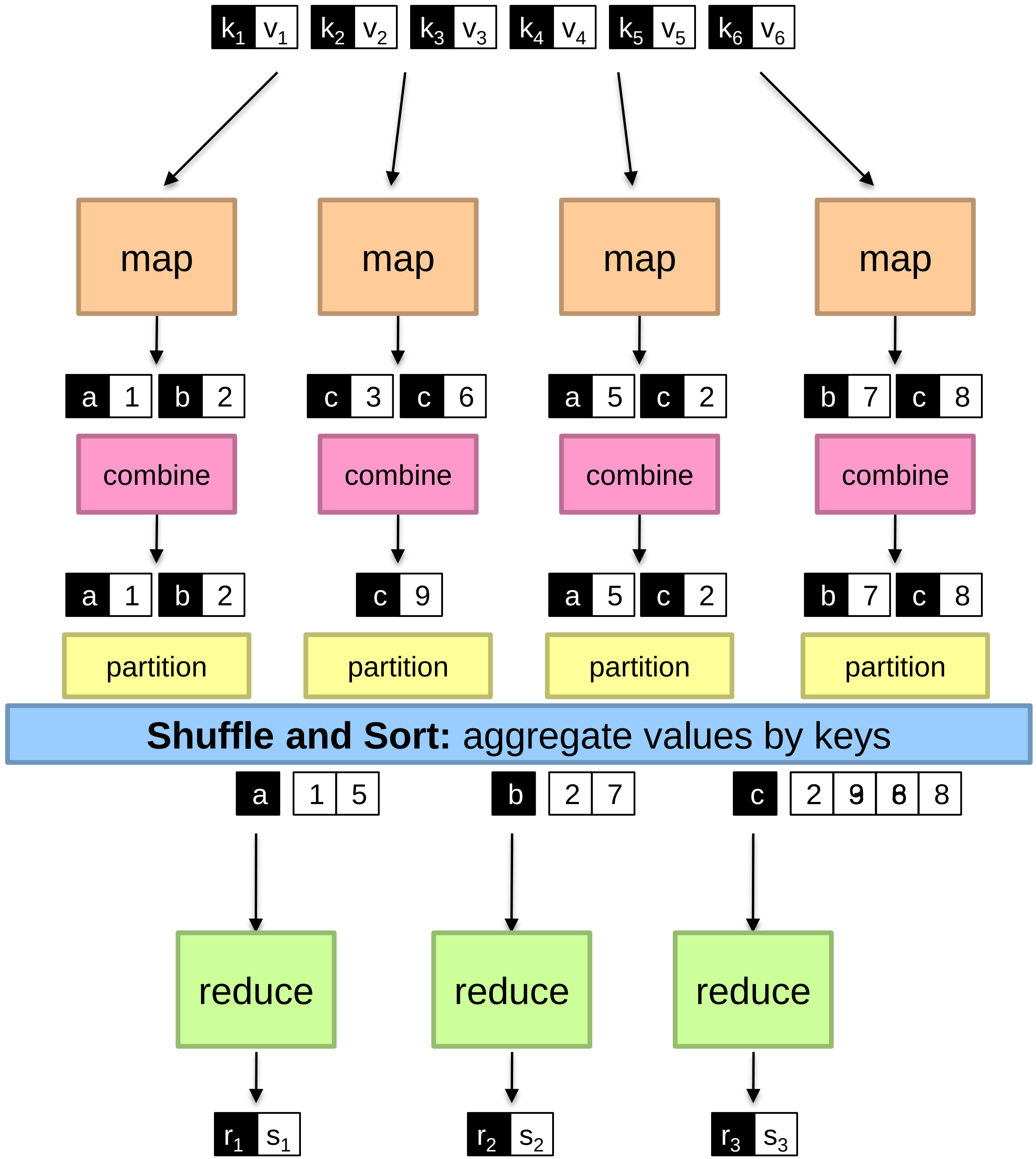
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic



k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2
8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

Two more details…

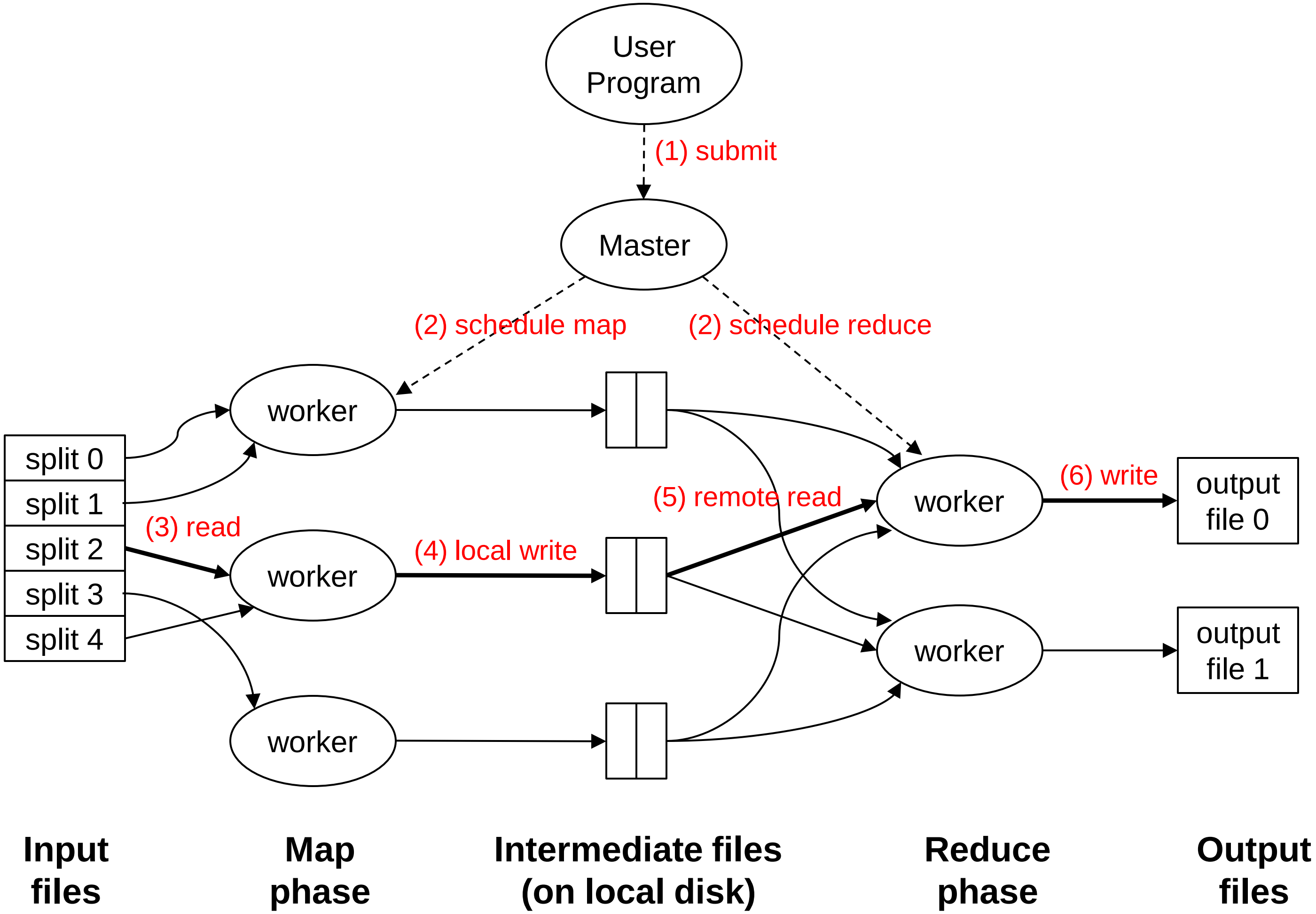
- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce Execution

- 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



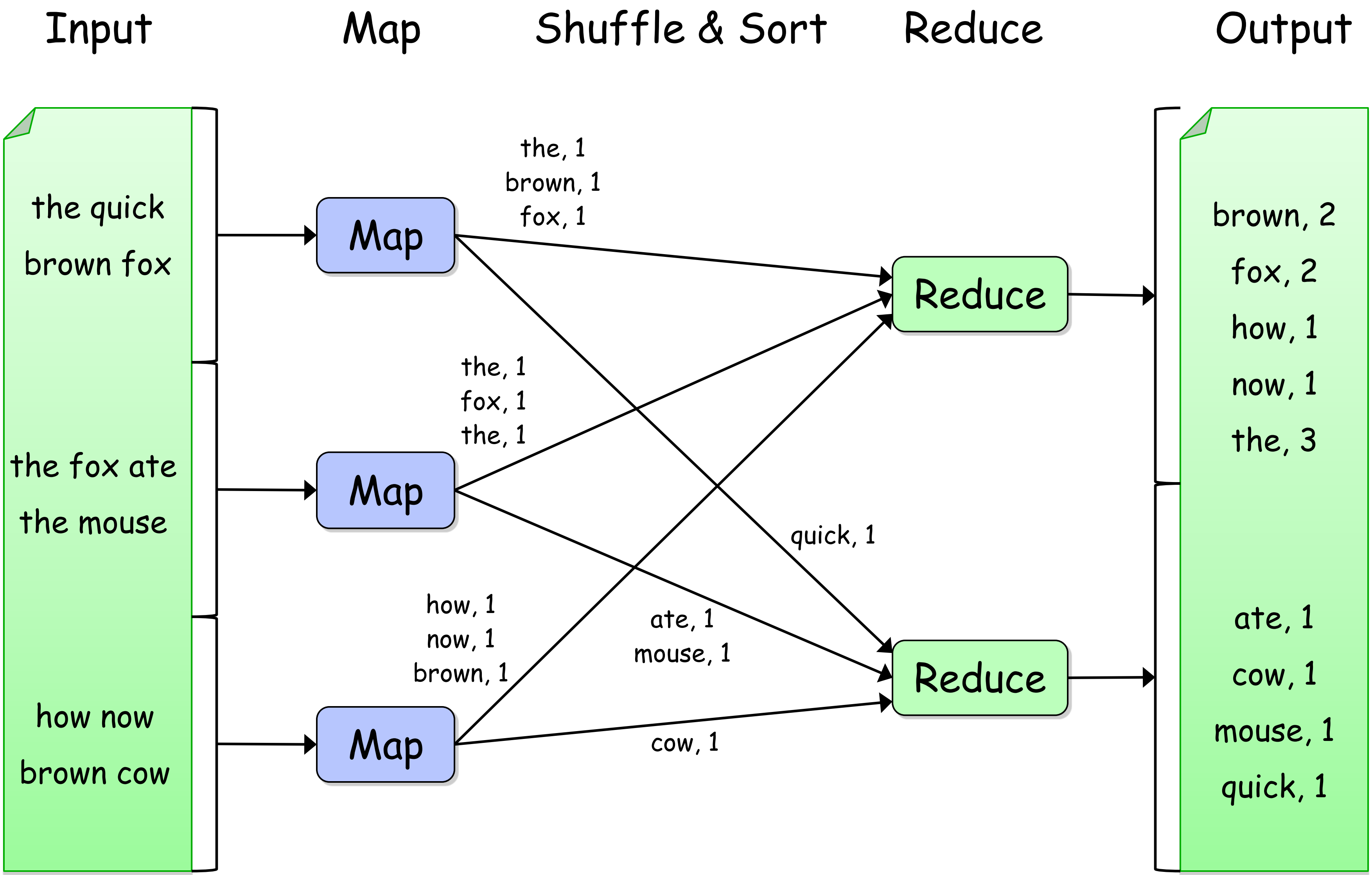
Adapted from (Dean and Ghemawat, OSDI 2004)

“Hello World”: Word Count

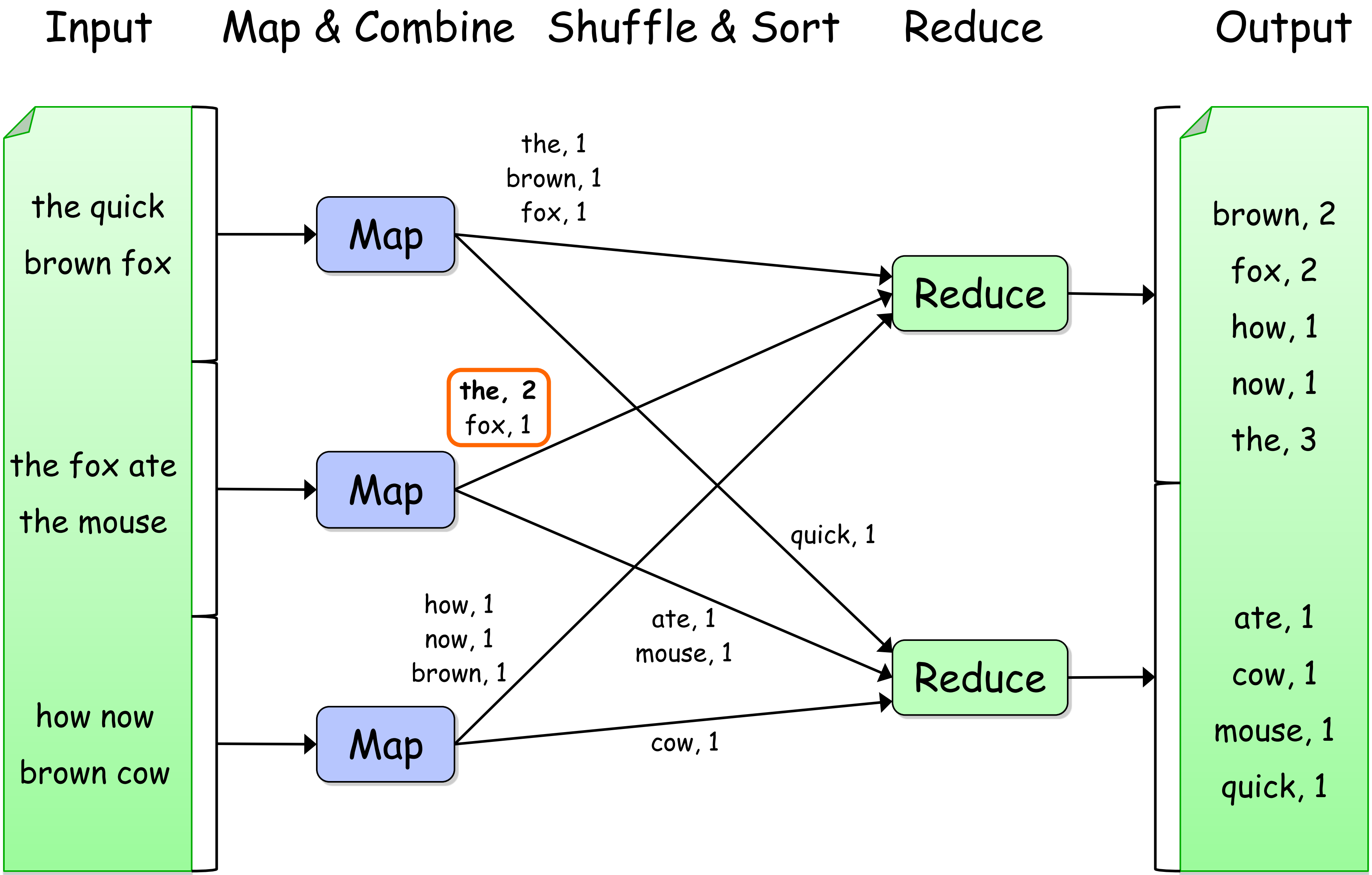
```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
```

Word Count Execution



Word Count with Combiner



Search Example

- **Input:** (lineNumber, line) records
- **Output:** lines matching a given pattern

- **Map:**

```
if(line matches pattern):
    output(line)
```

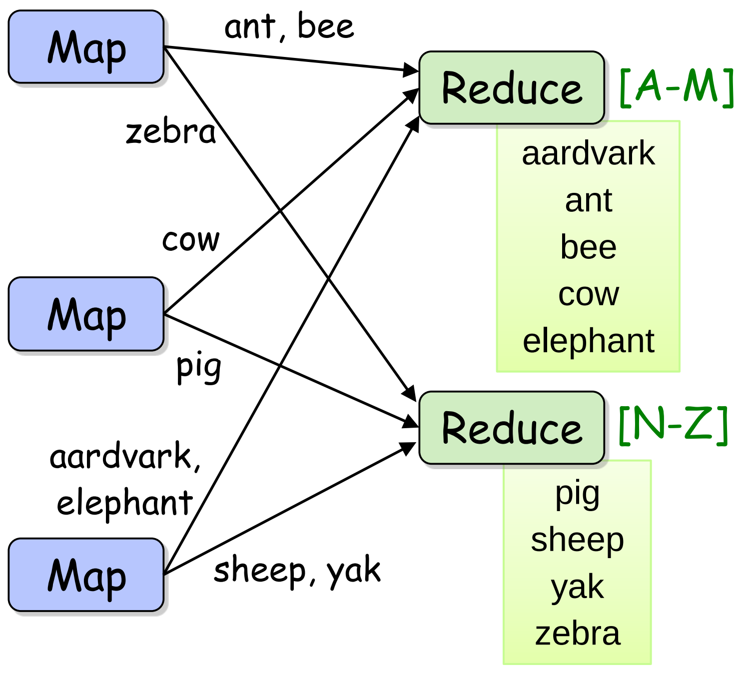
- **Reduce:** identify function
  - Alternative: no reducer (map-only job)

Sort Example

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function h such that $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$



Inverted Index Example

- **Input:** (filename, text) records
- **Output:** list of files containing each word

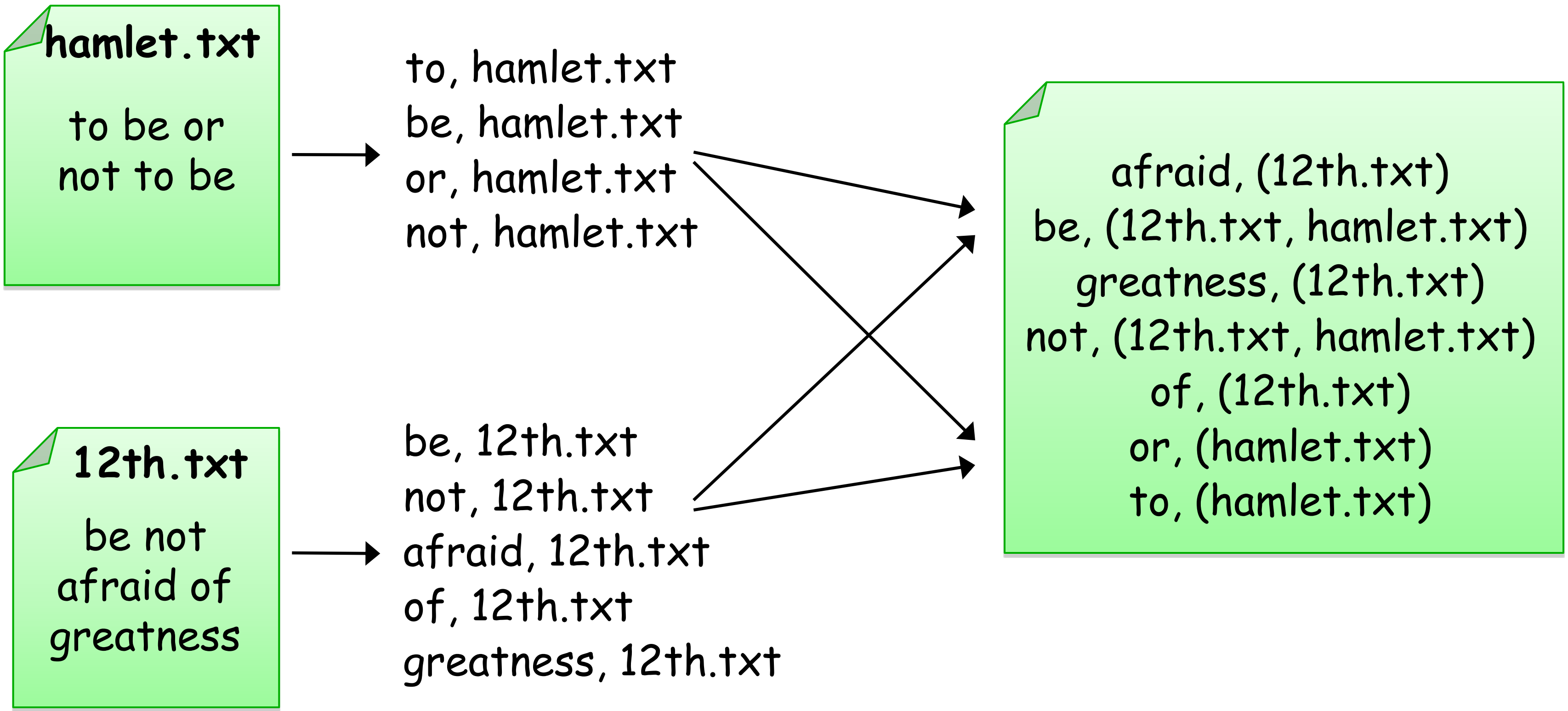
- **Map:**

```
foreach word in text.split():
    output(word, filename)
```

- **Combine:** uniquify filenames for each word
- **Reduce:**

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

Inverted Index Example



Most Popular Words Example

- **Input:** (filename, text) records
- **Output:** top 100 words occurring in the most files

- Two-stage solution:
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

- Optimizations:
  - Map to (word, 1) instead of (word, file) in Job 1
  - Count files in job 1's reducer rather than job 2's mapper
  - Estimate count distribution in advance and drop rare words

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

Hadoop Components

- **Distributed file system (HDFS)**
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- **MapReduce framework**
  - Executes user jobs specified as “map” and “reduce” functions
  - Manages work distribution & fault-tolerance

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Large and expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

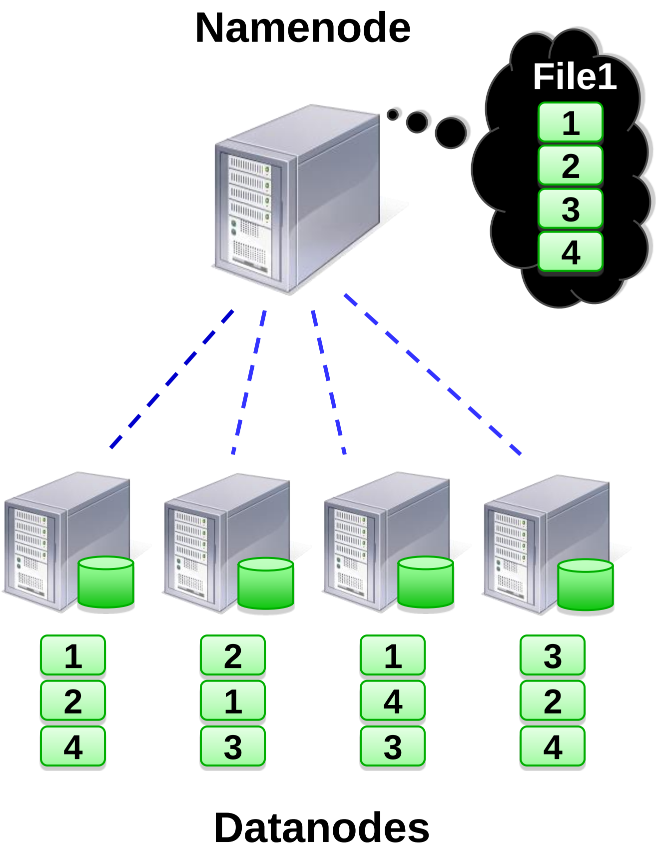
From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology…

Hadoop Distributed File System

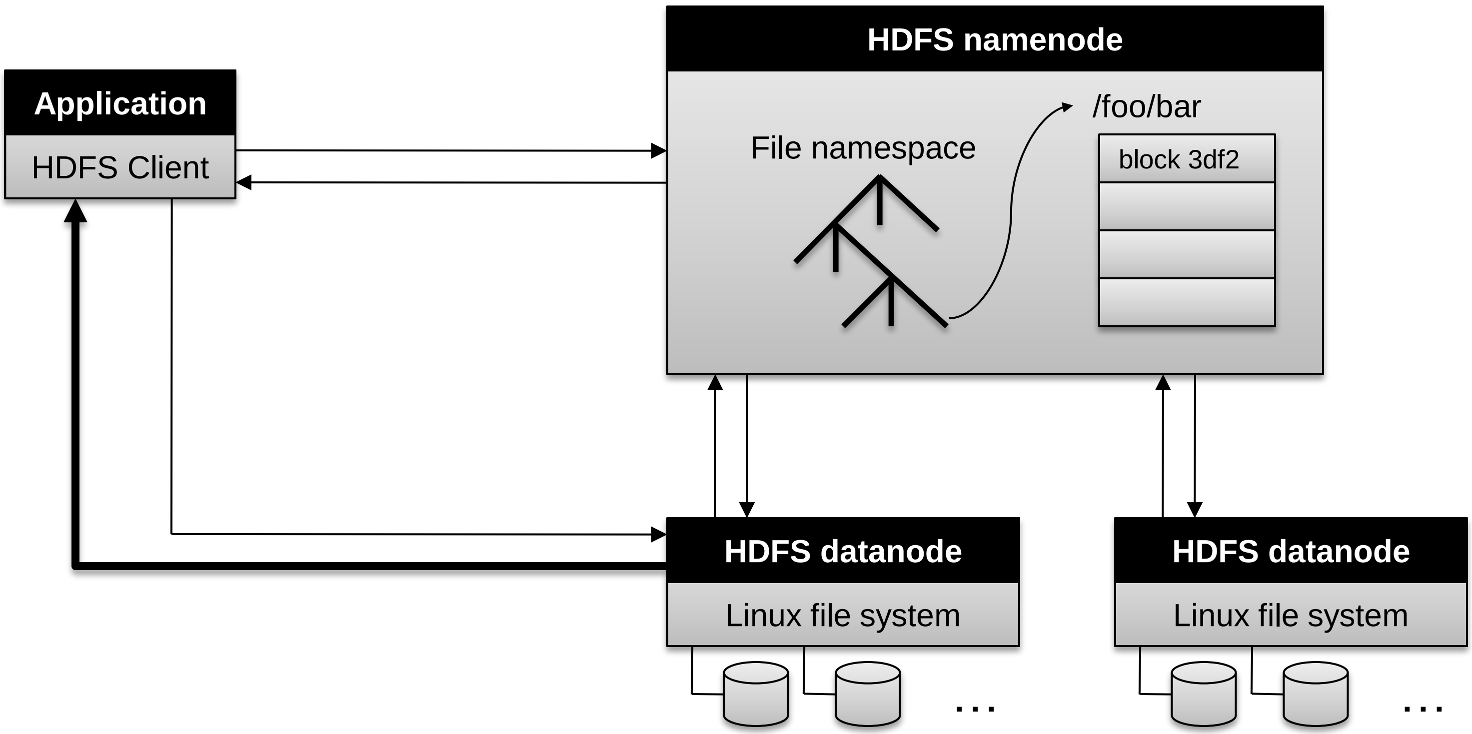
- Files split into 64MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



Namenode Responsibilities

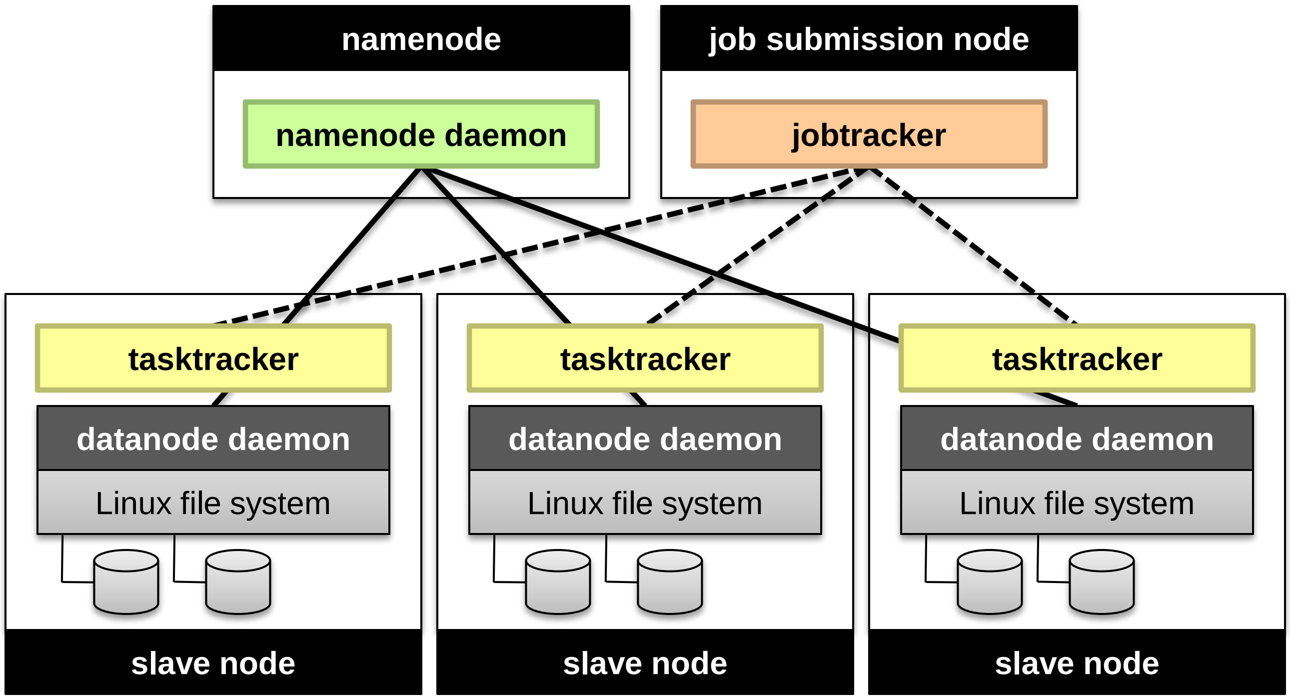
- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

HDFS Architecture



Adapted from (Ghemawat et al., SOSP 2003)

Putting everything together…



Typical Hadoop Cluster

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