https://hao-ai-lab.github.io/dsc204a-f25/

# DSC 204A: Scalable Data Systems
# Fall 2025


Staff
Instructor: Hao Zhang
TAs: Mingjia Huo, Yuxuan Zhang

@haozhangml @haoailab
haozhang@ucsd.edu

# Where We Are

Motivations, Economics, Ecosystems, Trends




Networking

Storage

**Part3: Compute**

~~Datacenter networking~~

~~Collective communication~~

(Distributed) File Systems / Database

Cloud storage

**Distributed Computing**

**Big data processing**

# Let's Focus On: Multi-node Distributed Systems

We have two primary problems to solve in real systems

1. How to Distribute Data (we'll not cover, also not in exam)
   - Read DDIA
   - Read GFS paper
   - Read BigTable paper
2. How to Distribute Compute (we'll cover in lectures)
   - Batching Processing
   - Streaming Processing

# How to Analyze Distributed Systems

- Scalability
  - Data volume
  - Read/Write/Compute load

- Consistency and correctness
  - Read / Write sees consistency data
  - Compute produce correct results

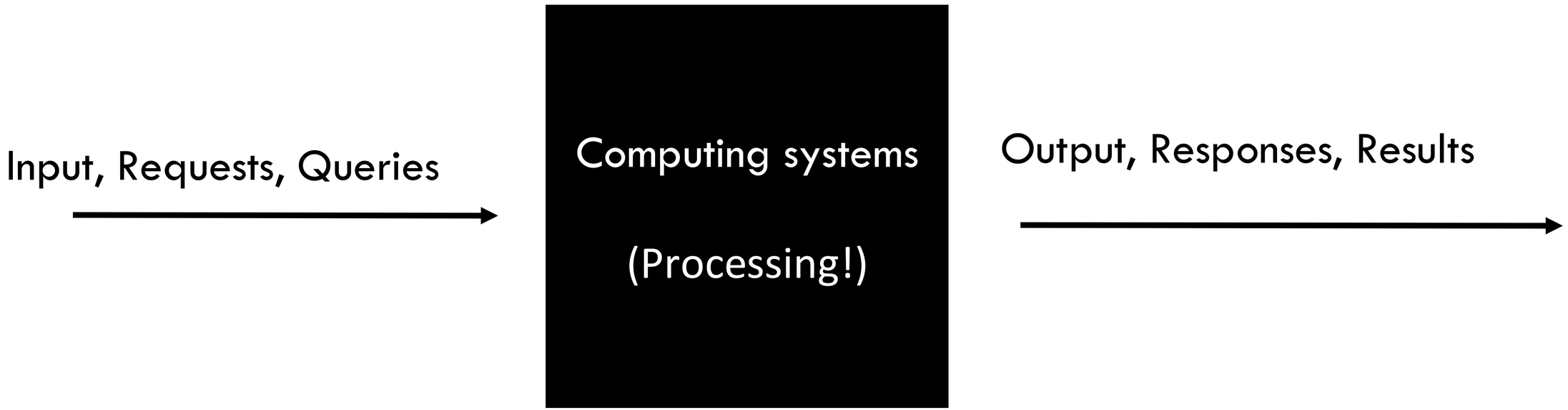
- Fault tolerance / high availability
  - When one fails, another can take over.

- Latency and throughput
  - Distribute machines worldwide.
  - Reduce network latency.

# Today's topic: Batch Processing

- Overview
- IO & Unix pipes
- MapReduce
- Beyond MapReduce

# Basic Computing System Paradigm



# Processing latency

feedback cycle time

**direct manipulation**
no visible lag → Interactive data science!

→ Online system: handle request ASAP

**turn-taking**
minutes to seconds → Stream processing systems (near-real-time systems)

**batch-processing**
hours or overnight → Batch processing systems (Offline systems)



Figure from https://publishup.uni-potsdam.de/opus4-ubp/frontdoor/deliver/index/docId/10090/file/mueller_diss.pdf

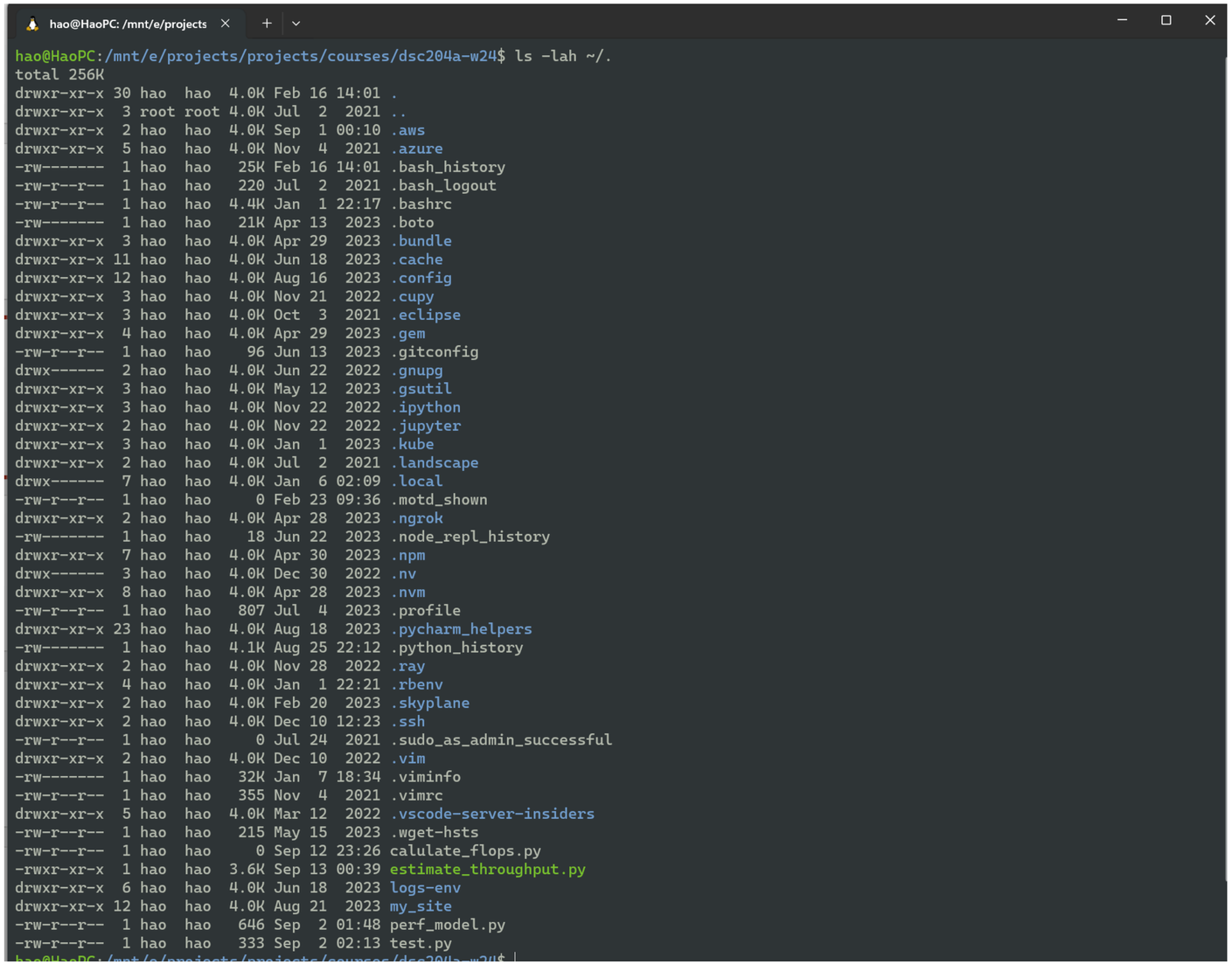
# Today's topic: Batch Processing

- Overview
- IO & Unix pipes: the first batching processing system
- MapReduce
- Beyond MapReduce

# Shell example

“.****rc”

Run commands



# Useful shell commands

- Shell already has a collection of rich commands
  - Some Useful commands
    - uptime, cut, date, cat, finger, hexdump, man, md5sum, quota,
    - mkdir, rmdir, rm, mv, du, df, find, cp, chmod, cd
    - uname, zip, unzip, gzip, tar
    - tr, sed, sort, uniq, ascii
    - Type “man command” to read about shell commands

# What do these shell commands do?

- cat dups.txt | sort | uniq
- cat dups.txt | sort -V | uniq
- cat dups.txt | sort -V | uniq > outfile.txt
- tr "a" "e" < z.txt
- cat z.txt | tr a e

# Batch processing with Unix Tools

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}' | ❷
  sort             | ❸
  uniq -c          | ❹
  sort -r -n       | ❺
  head -n 5          ❻
```

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
1369 /
 915 /css/typography.css
```

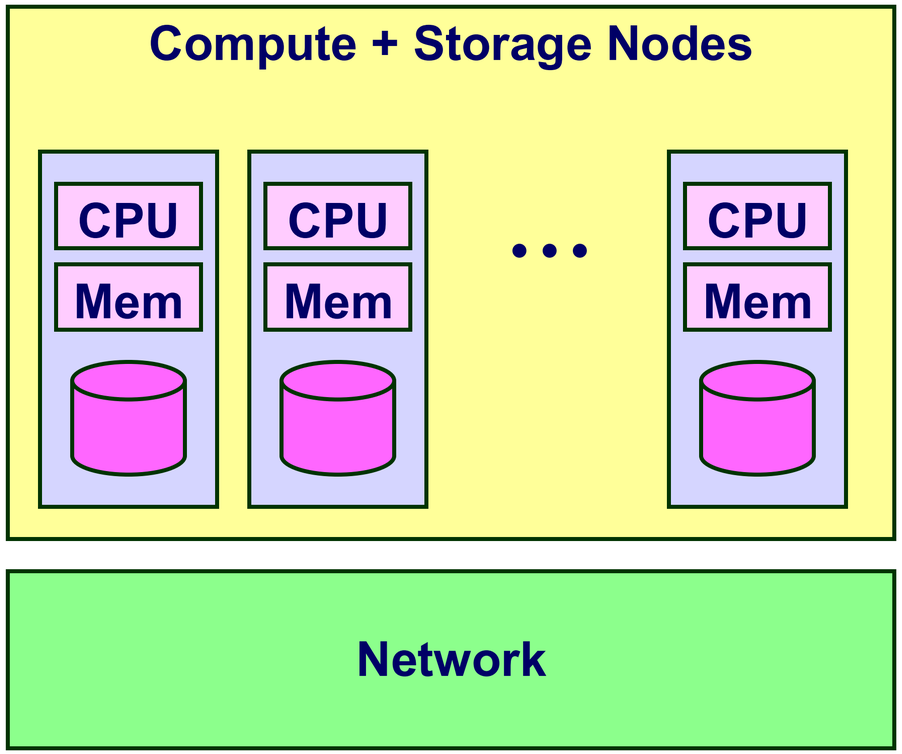
- Read the log file.
- Split each line into fields by white space, output only the 7th element (requested URL).
- Alphabetically sort
- Filter out repeated lines.
- Sort it again based on the line number (-n)
- Out put the first five lines.

The biggest limitation of Unix tools is that they run only on a single machine — and that's where tools like Hadoop come in.

# Today's topic: Batch Processing

- Overview
- IO & Unix pipes
- MapReduce
  - HDFS - infrastructure
  - Programming models (api)
  - Job execution
  - Workflow
- Beyond MapReduce

# History of MapReduce/Hadoop




Compute + Storage Nodes

- Medium-performance processors
- Modest memory
- 1-2 disks

Network

- Conventional Ethernet switches
  - 10 Gb/s within rack
  - 100 Gb/s across racks

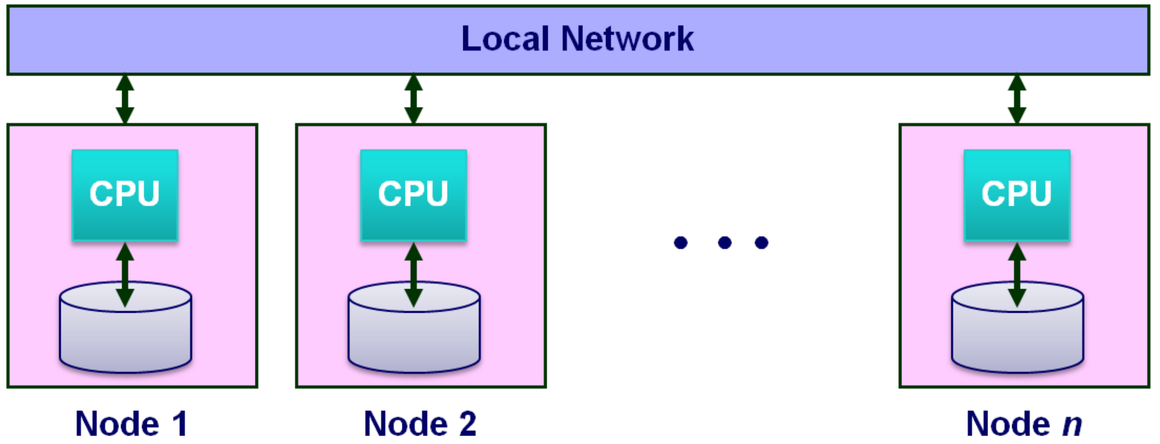
# Data-Intensive System Challenge

For Computation That Accesses 1 TB in 5 minutes

- Data distributed over 100+ disks
  - Assuming uniform data partitioning
- Compute using 100+ processors
- Connected by gigabit Ethernet (or equivalent)

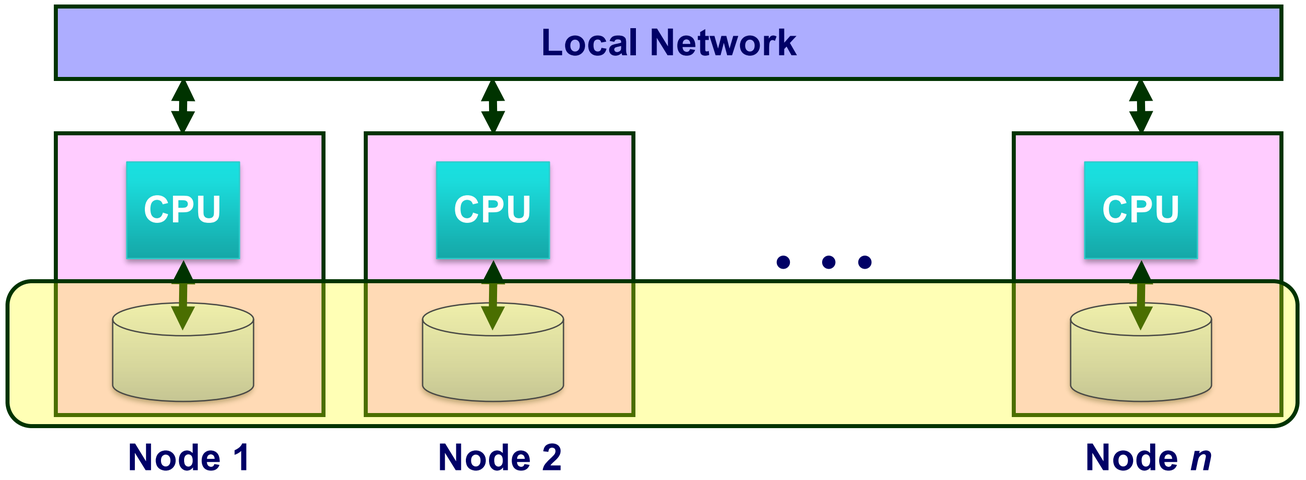
System Requirements

- Lots of disks
- Lots of processors
- Located in close proximity
  - Within reach of fast, local-area network



# Hadoop Project

File system with files distributed across nodes




- Store multiple (typically 3 copies of each file)
  - If one node fails, data still available
- Logically, any node has access to any file
  - May need to fetch across network (ideally, leverage locality for perf.)

Map / Reduce programming environment

- Software manages execution of tasks on nodes

# Today's topic: Batch Processing

- Overview
- IO & Unix pipes
- MapReduce
  - HDFS - infrastructure
  - Programming models (API)
  - Job execution (runtime)
  - Workflow
- Beyond MapReduce

# MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

## Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

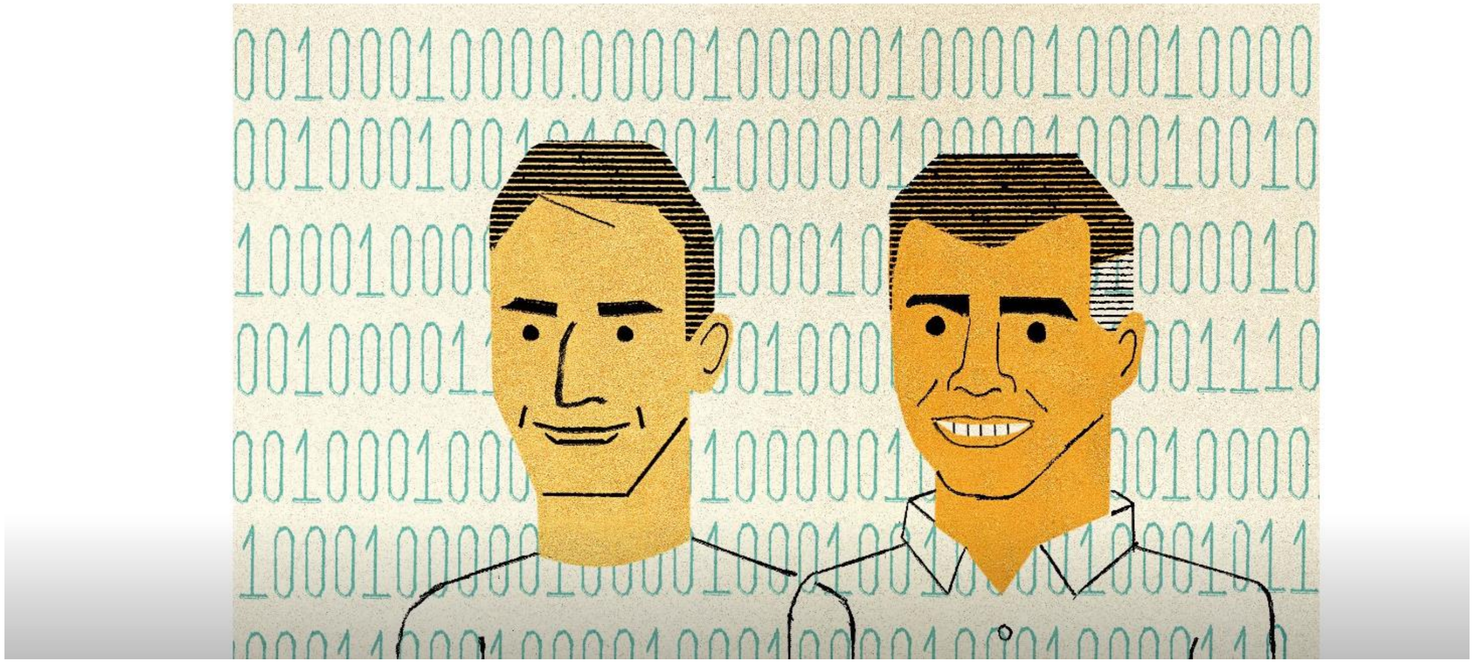
ANNALS OF TECHNOLOGY

# THE FRIENDSHIP THAT MADE GOOGLE HUGE

*Coding together at the same computer, Jeff Dean and Sanjay Ghemawat changed the course of the company—and the Internet.*

**By James Somers**

December 3, 2018



https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge

# TF-IDF

TF-IDF is a measure of originality of a word by comparing the number of times a word appears in a doc with the number of docs the word appears in.

$$\text{TF-IDF} = \text{TF}(t, d) \times \text{IDF}(t)$$

Term frequency

Number of times term t appears in a doc, d

Inverse document frequency

$$\log \frac{1 + n}{1 + \text{df}(d, t)} + 1$$

n ← # of documents

df(d, t) ← Document frequency of the term t

# Count the number of occurrences of word in a large collection of documents

```
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));
```

- Functional programming
- Functions are stateless
- They takes an input, processes and output a result.
- Pros and Cons?

# Data models

```
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));
```

| | | |
|---|---|---|
| map | (k1,v1) | → list(k2,v2) |
| reduce | (k2,list(v2)) | → list(v2) |

# MapReduce Example

- Create a word index of set of documents

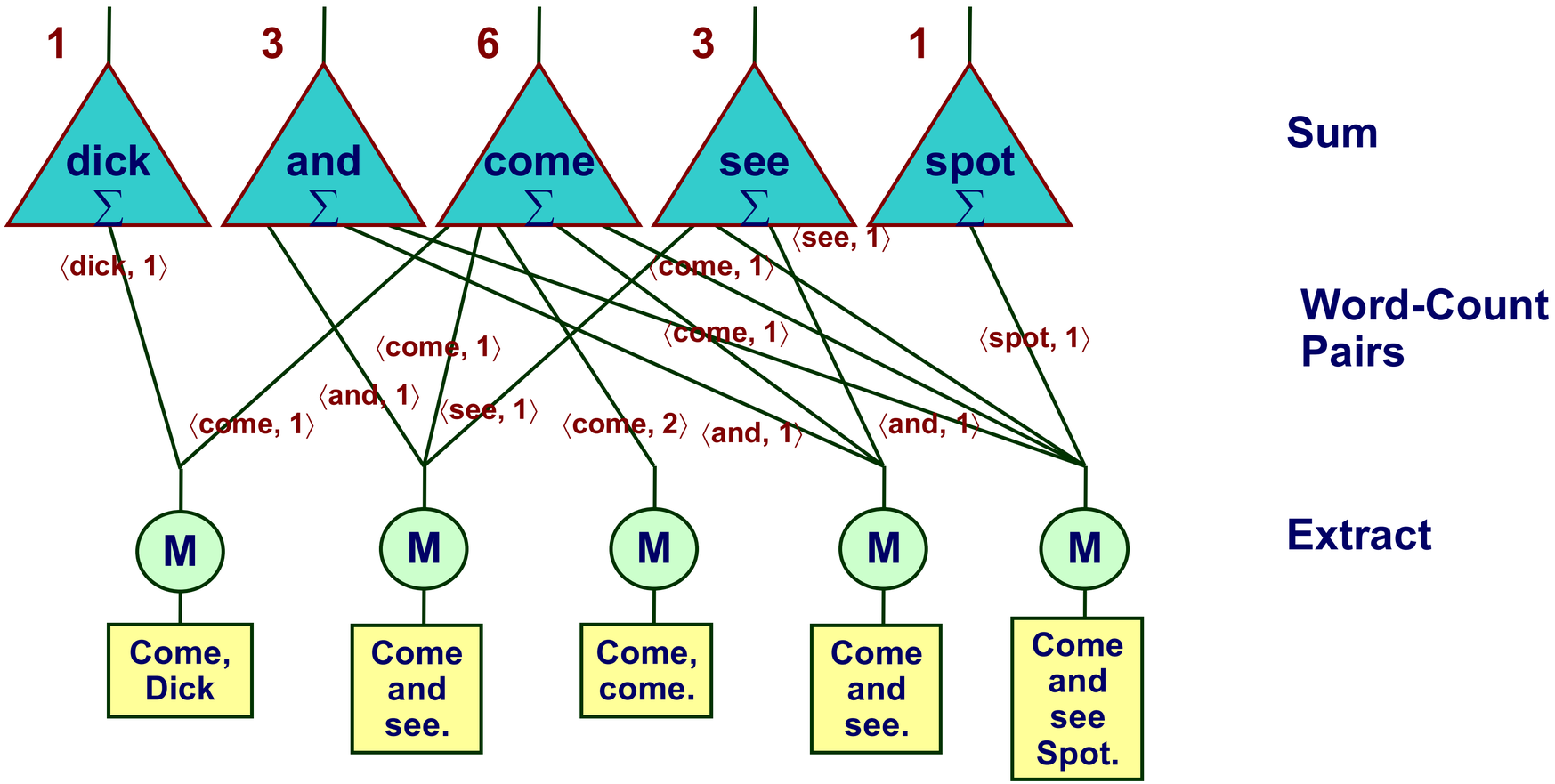

**Come, Dick**

**Come and see.**

**Come, come.**

**Come and see.**

**Come and see Spot.**




- Map: generate ⟨word, count⟩ pairs for all words in document
- Reduce: sum word counts across documents

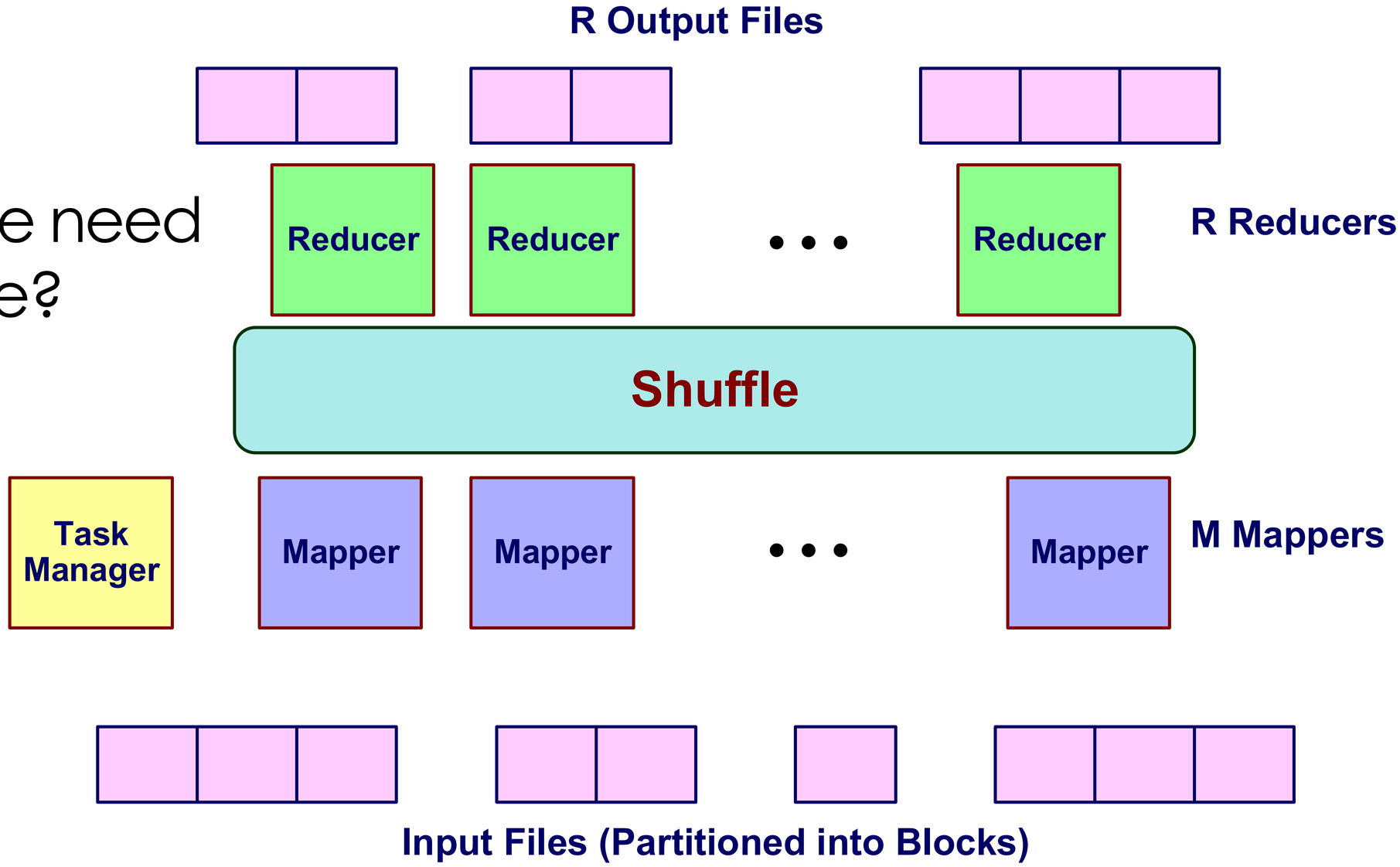
Discussion:
Other possible way to implement this using map-reduce?

# Today's topic: Batch Processing

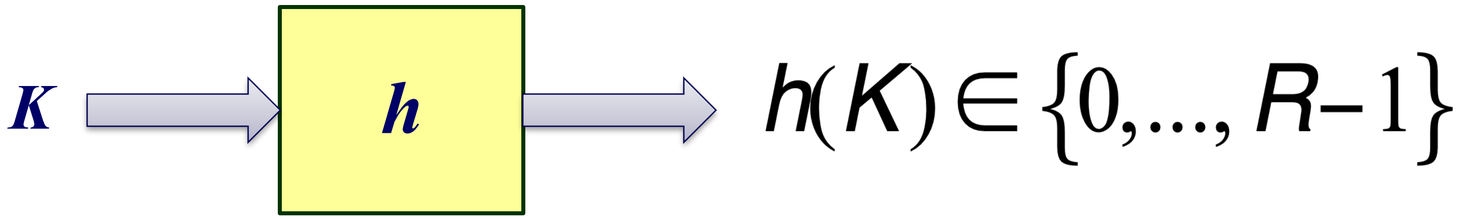
- Overview
- IO & Unix pipes
- MapReduce
  - HDFS - infrastructure
  - Programming models
  - Job execution
  - Workflow
- Beyond MapReduce

# MapReduce Execution (Runtime)

Why do we need shuffle?



# Single Mapper

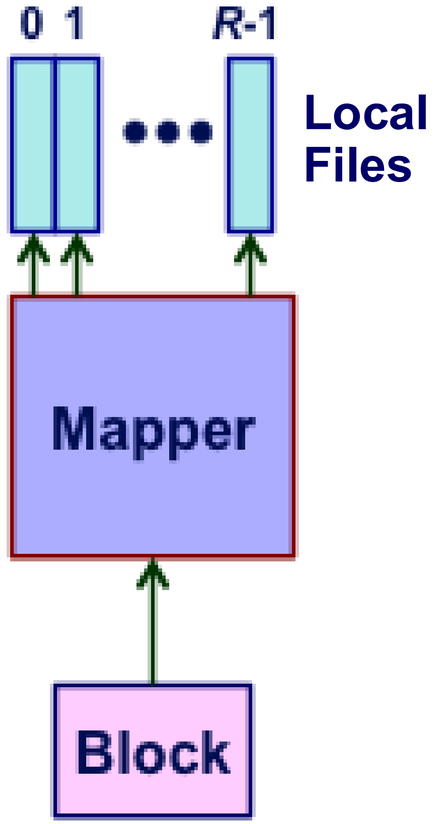



Hash Function *h*

- Maps each key *K* to integer *i* such that $0 \le i < R$

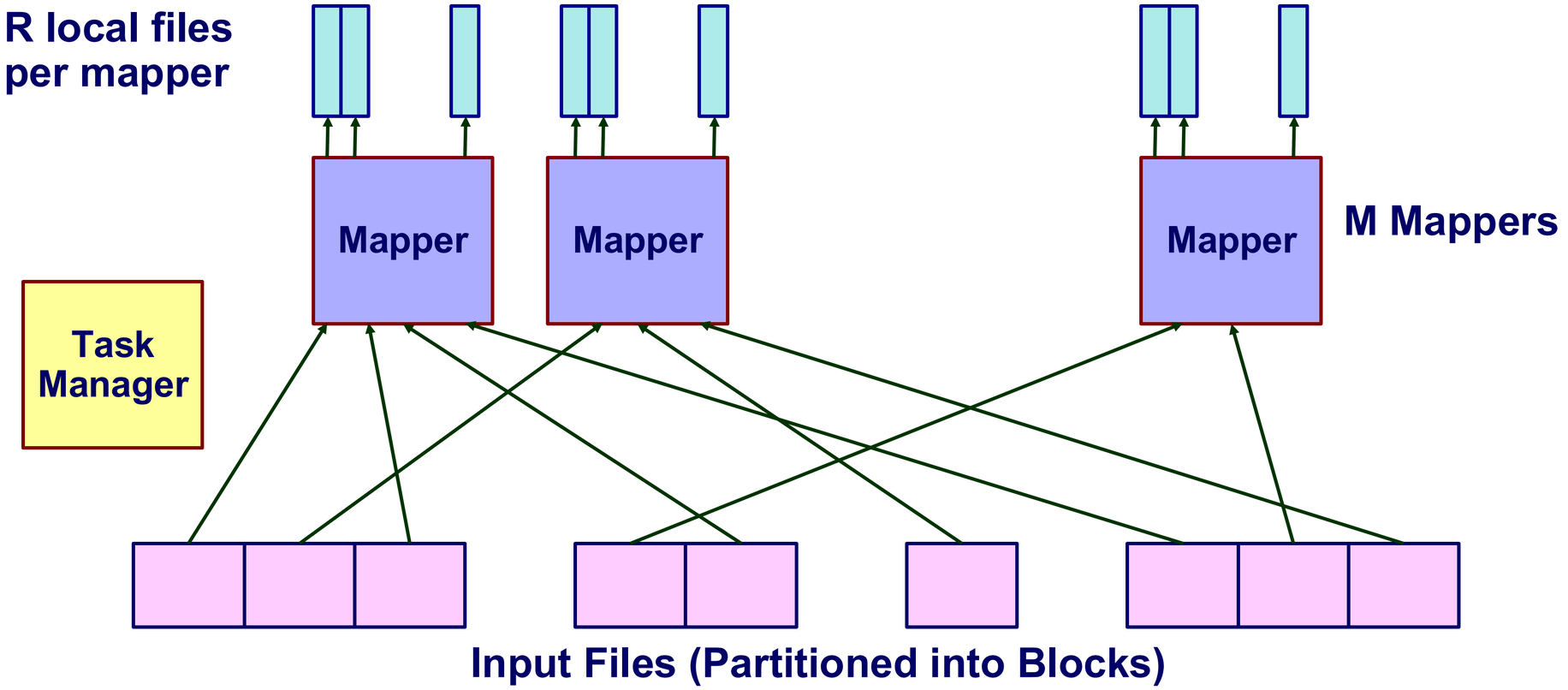
Mapper Operation

- Reads input file blocks
- Generates pairs ⟨*K*, *V*⟩
- Writes to local file *h*(*K*)



# Distributed Mapper

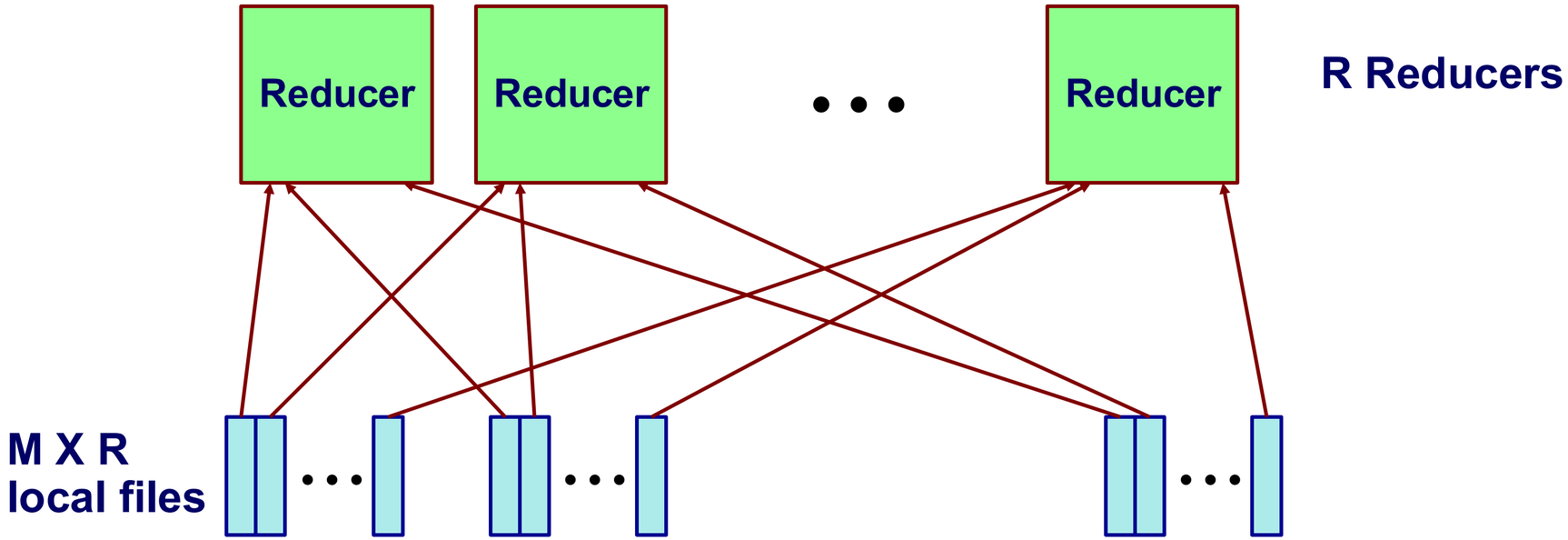
- Dynamically map input file blocks onto mappers
- Each generates key/value pairs from its blocks
- Each writes R files on local file system



# Shuffling

Each Reducer:

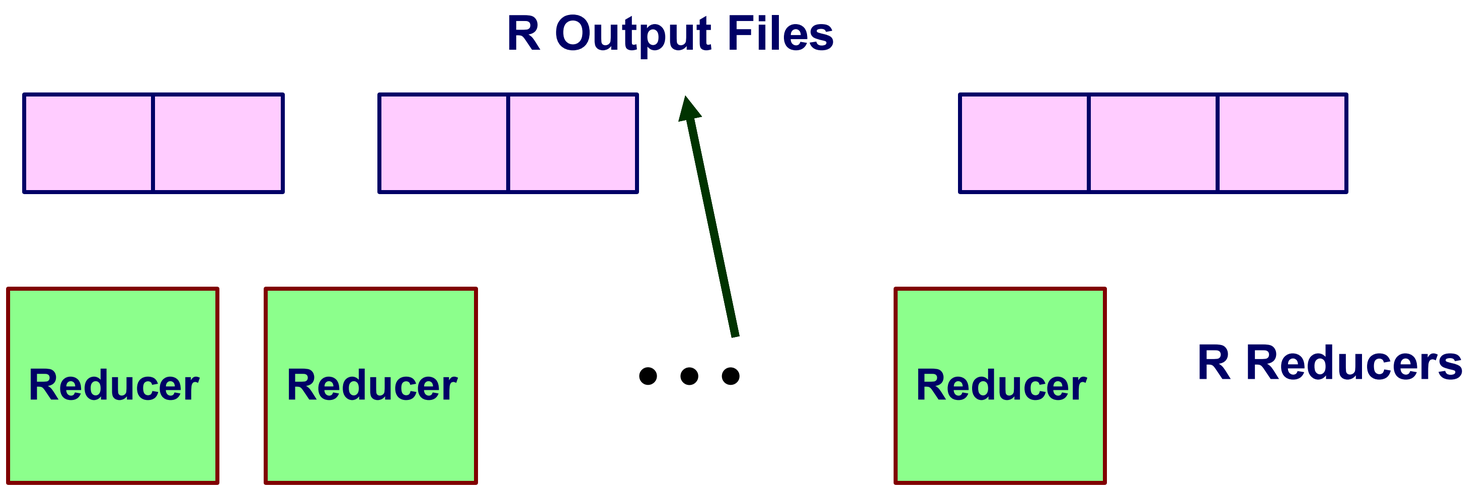
- Handles 1/R of the possible key values
- Most cases: just a hash function
- Need to handle fault tolerance



# Reducer

Each Reducer:

- Executes reducer function for each key
- Writes output values to parallel file system

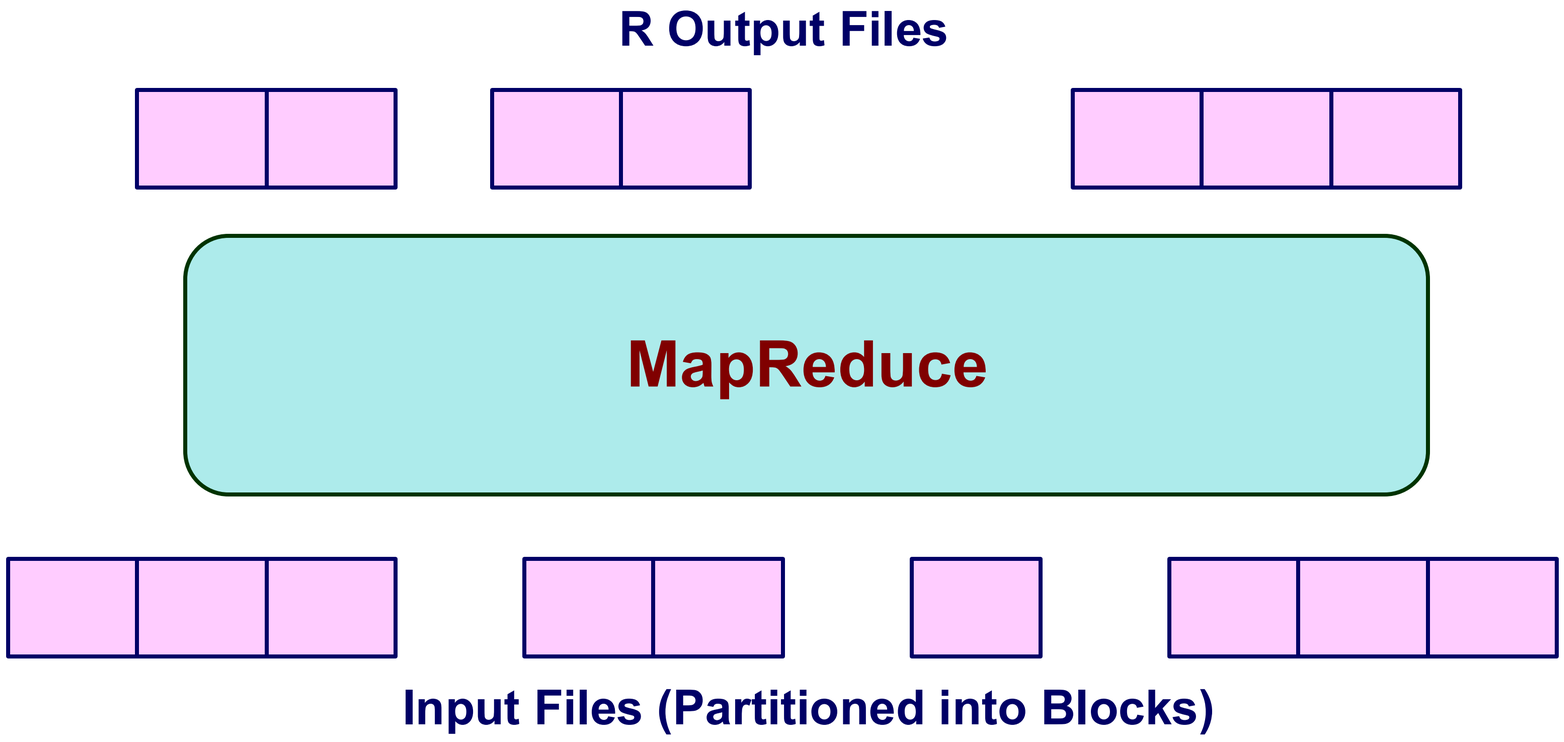


# MapReduce Effect

MapReduce Step

- Reads set of files from file system
- Generates new set of files

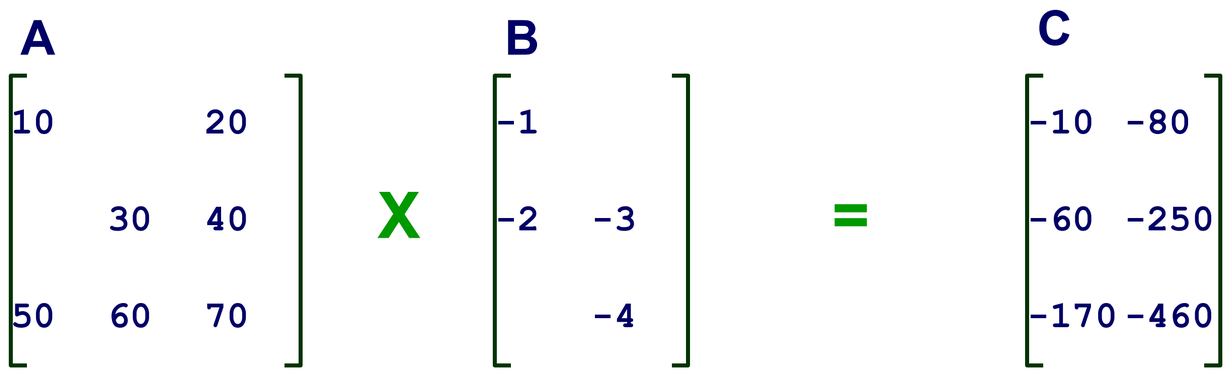
Can iterate to do more complex processing



# Today's topic: Batch Processing

- Overview
- IO & Unix pipes
- MapReduce
  - HDFS - infrastructure
  - Programming models (API)
  - Job execution (runtime)
  - MapReduce dataflow
- Beyond MapReduce

# Example: Sparse Matrices with Map/Reduce

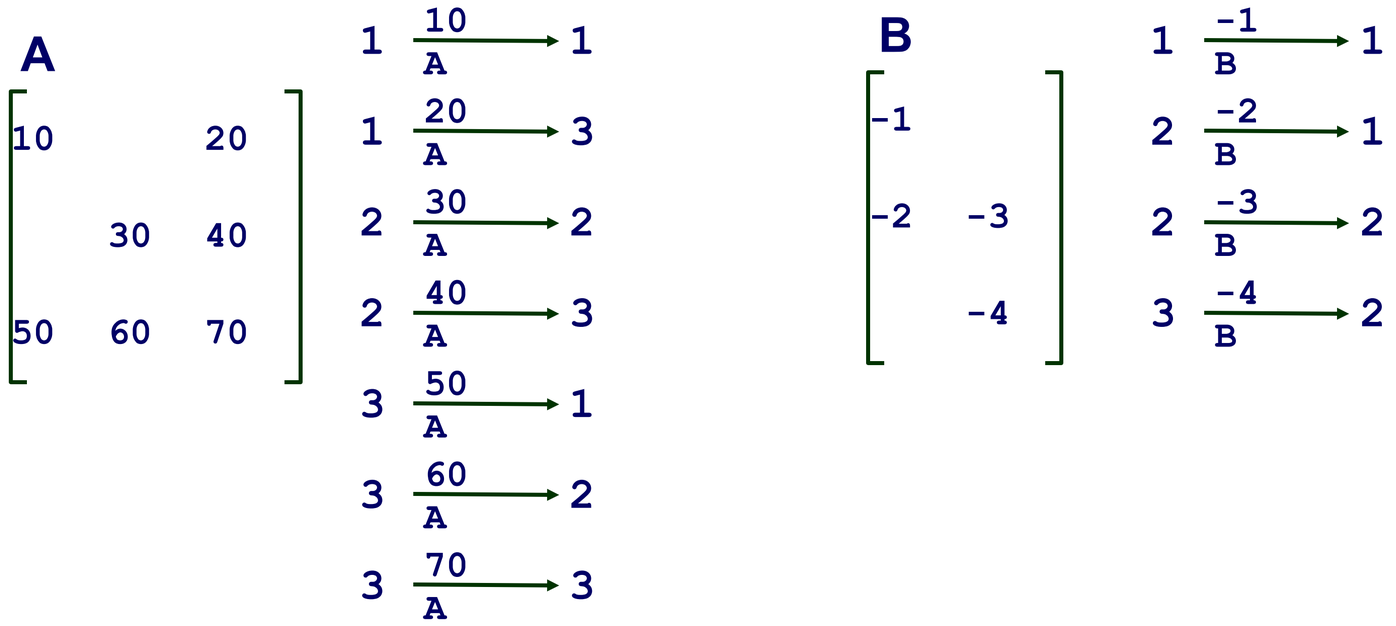



- Task: Compute product C = A·B
- Assume most matrix entries are 0

Motivation

- Core problem in scientific computing
- Challenging for parallel execution
- Demonstrate expressiveness of Map/Reduce

# Computing Sparse Matrix Product

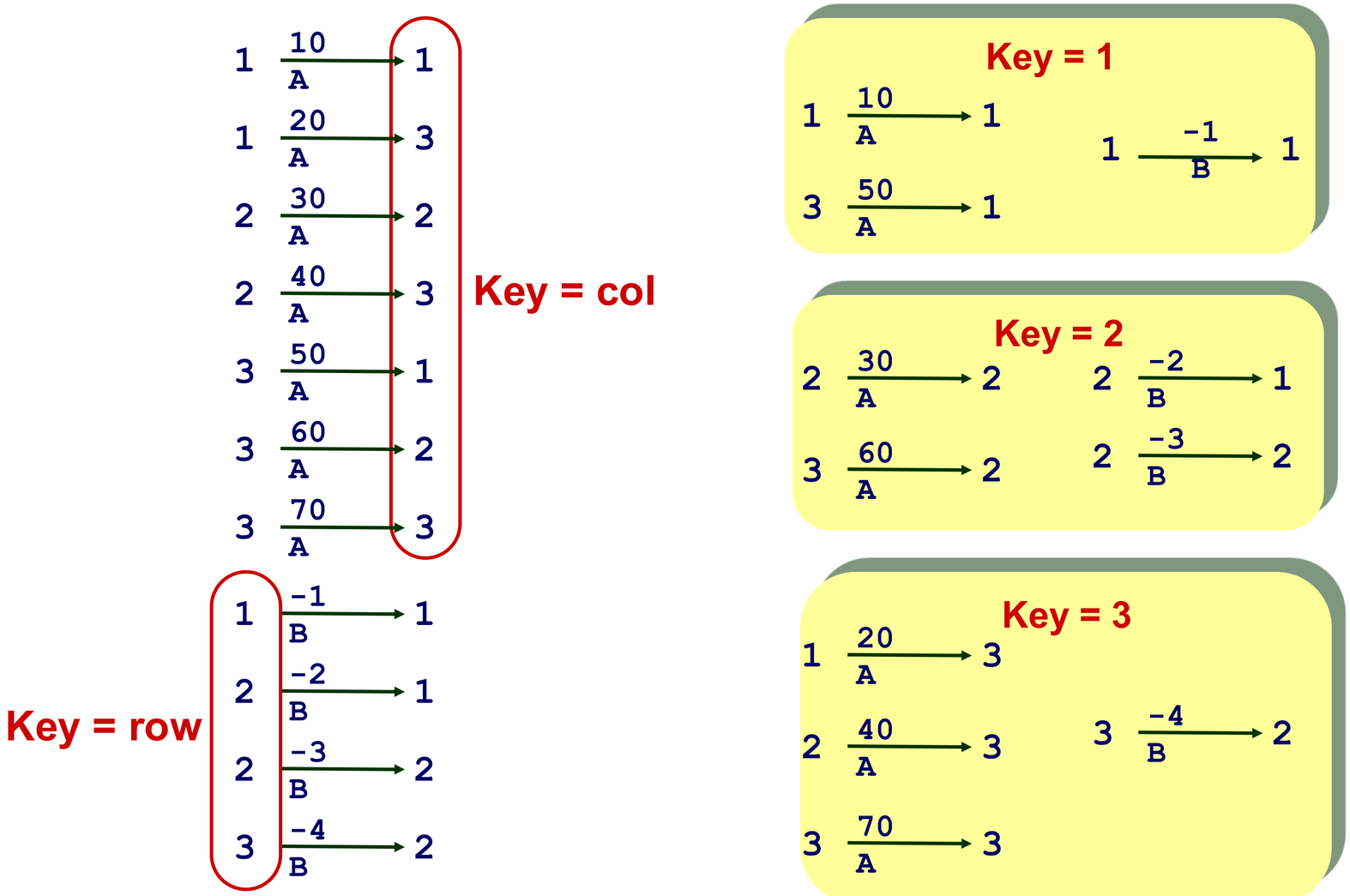



- Represent matrix as list of nonzero entries

  <row, col, value, matrixID>

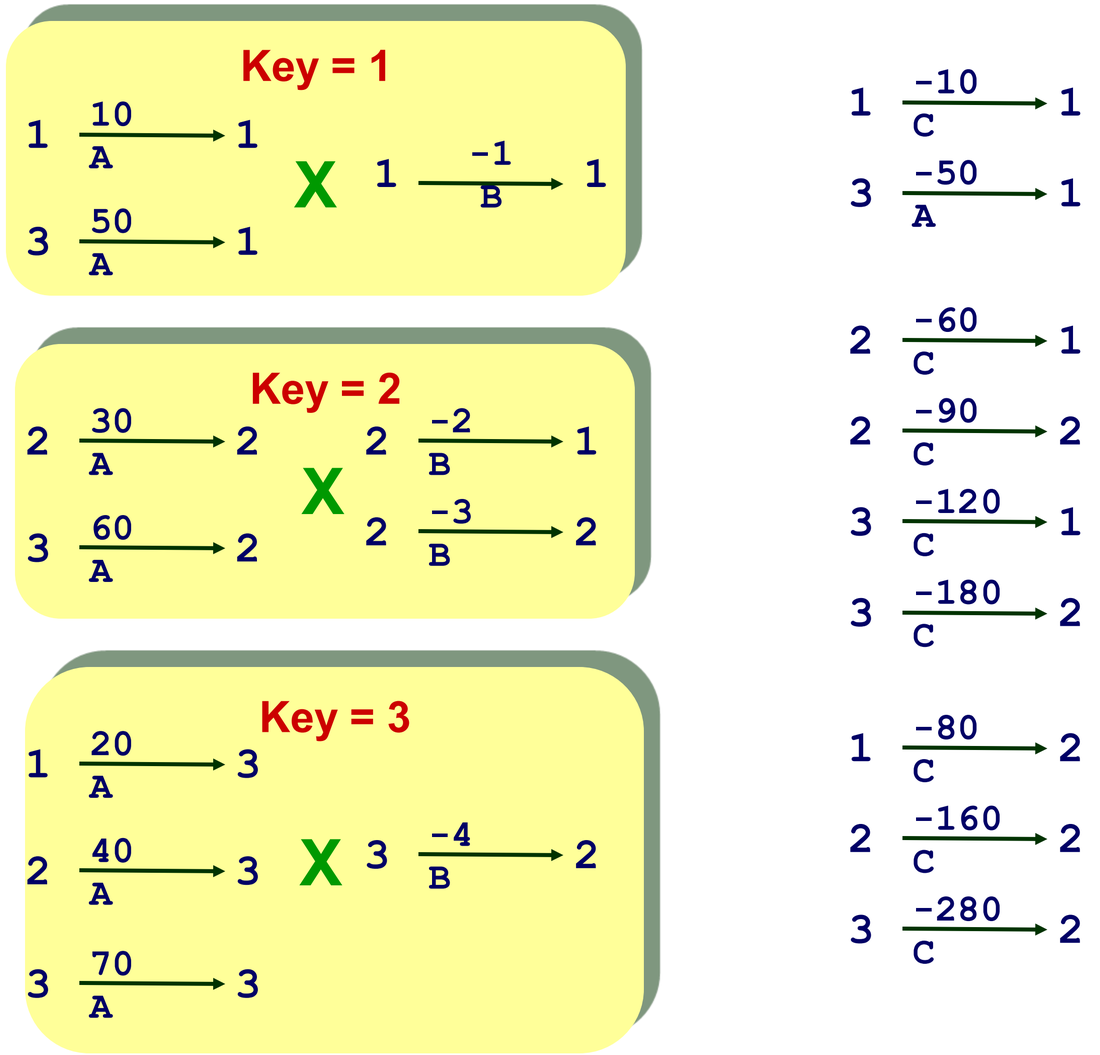
- How to represent the computation as map-reduce?
  - Phase 1: Compute all products $a_{i,k} \cdot b_{k,j}$
  - Phase 2: Sum products for each entry i,j
  - Each phase involves a Map/Reduce

# Phase 1 Map of Matrix Multiply



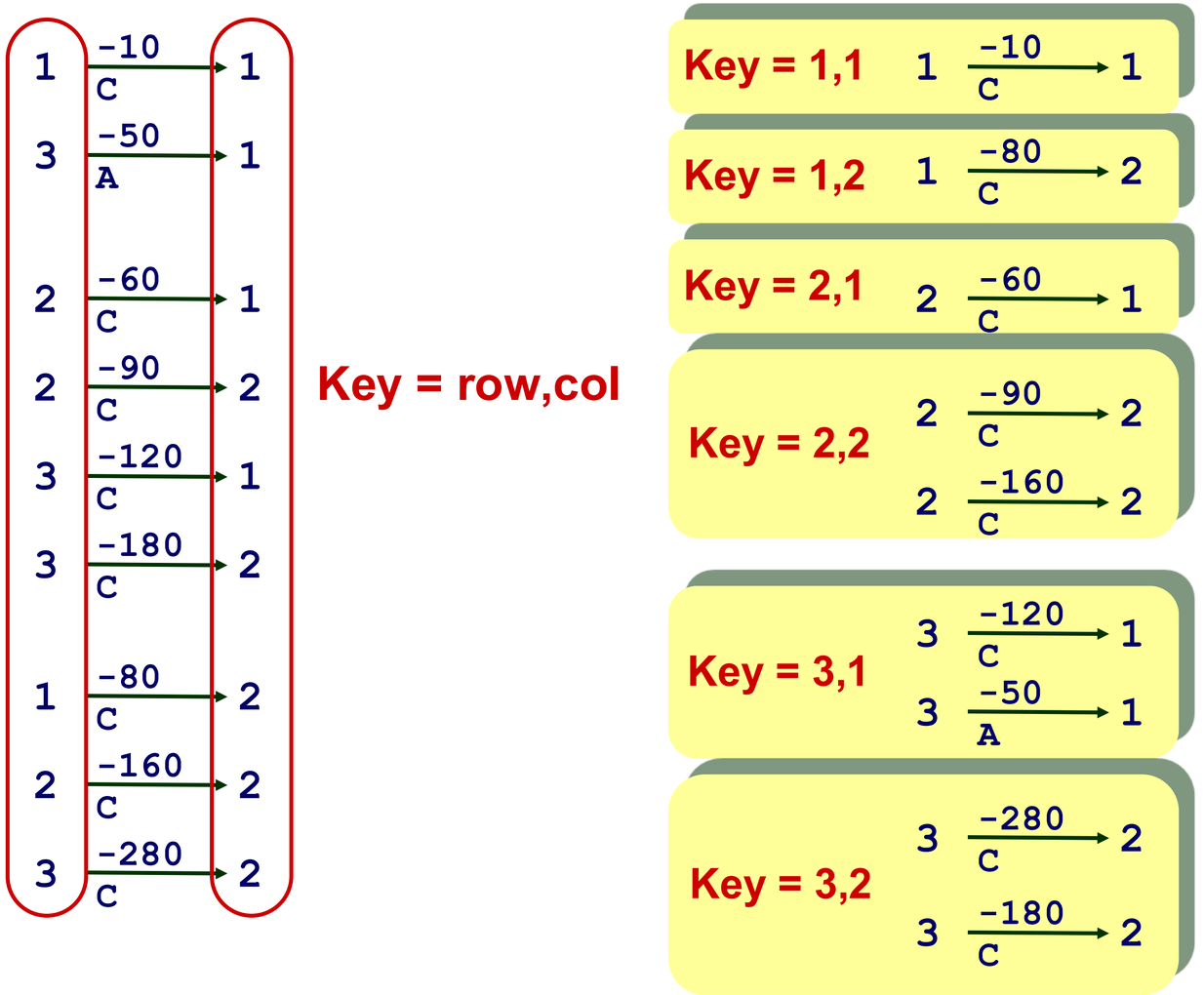

- Group values $a_{i,k}$ and $b_{k,j}$ according to key k

# Phase 1 "Reduce" of Matrix Multiply



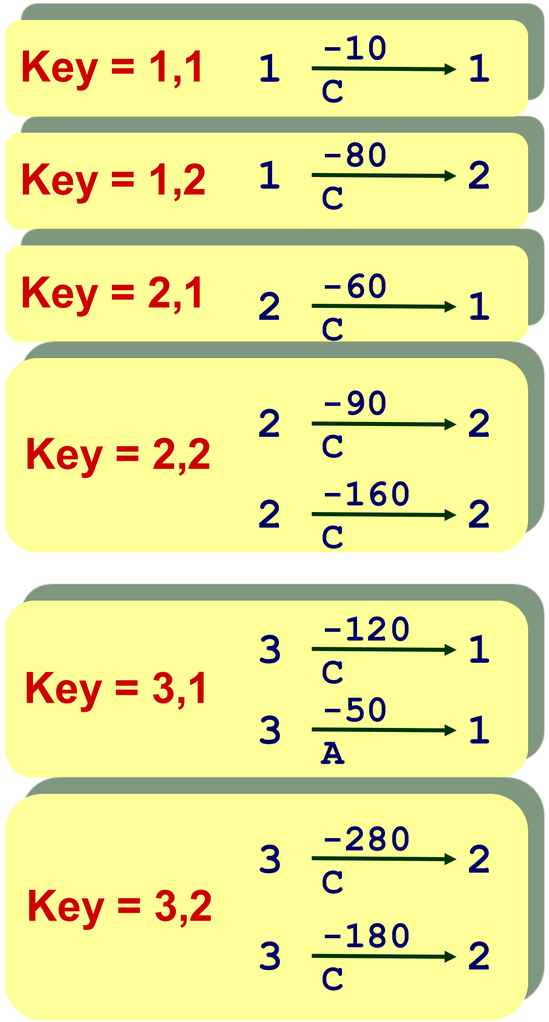

- Generate all products $a_{i,k} \cdot b_{k,j}$

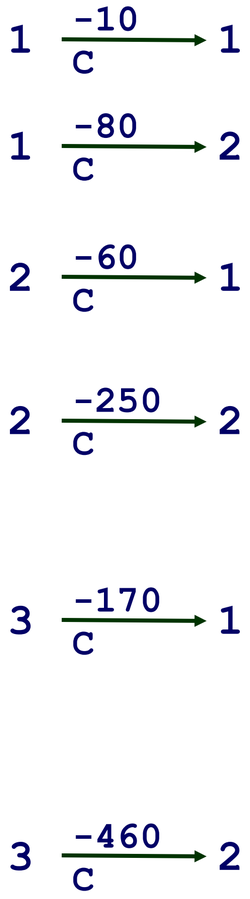
# Phase 2 Map of Matrix Multiply




- Group products $a_{i,k} \cdot b_{k,j}$ with matching values of i and j

# Phase 2 Reduce of Matrix Multiply




C

$$\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

- Sum products to get final entries

# Recap: MapReduce Implementation

Built on Top of Parallel File System

- Google: GFS, Hadoop: HDFS
- Provides global naming
- Reliability via replication (typically 3 copies)

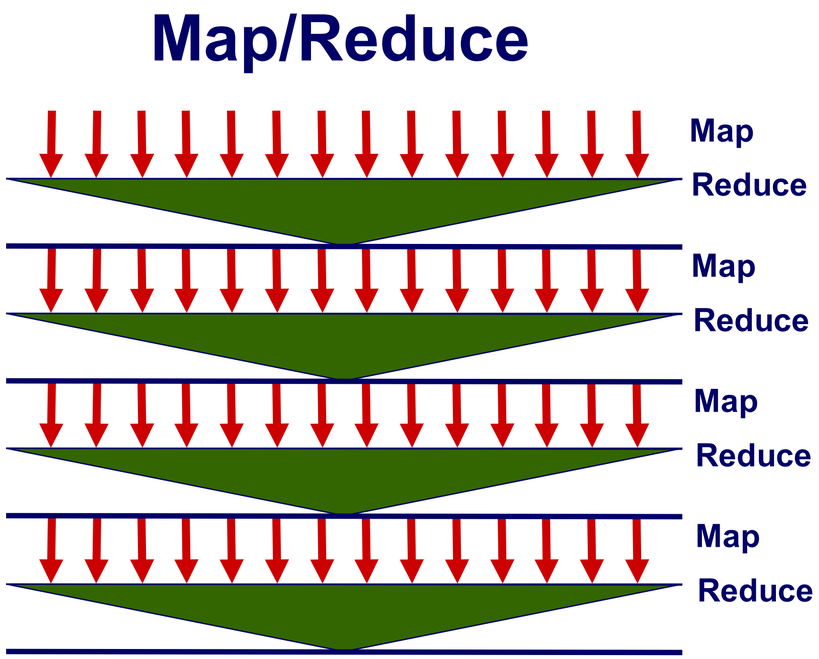
Breaks work into tasks

- Master schedules tasks on workers dynamically
- Typically #tasks >> #processors

Net Effect

- Input: Set of files in reliable file system
- Output: Set of files in reliable file system

# Analyzing Pros and Cons of Map/Reduce




## Characteristics

- Computation broken into many, short-lived tasks
  - Mapping, reducing
- Use disk storage to hold intermediate results

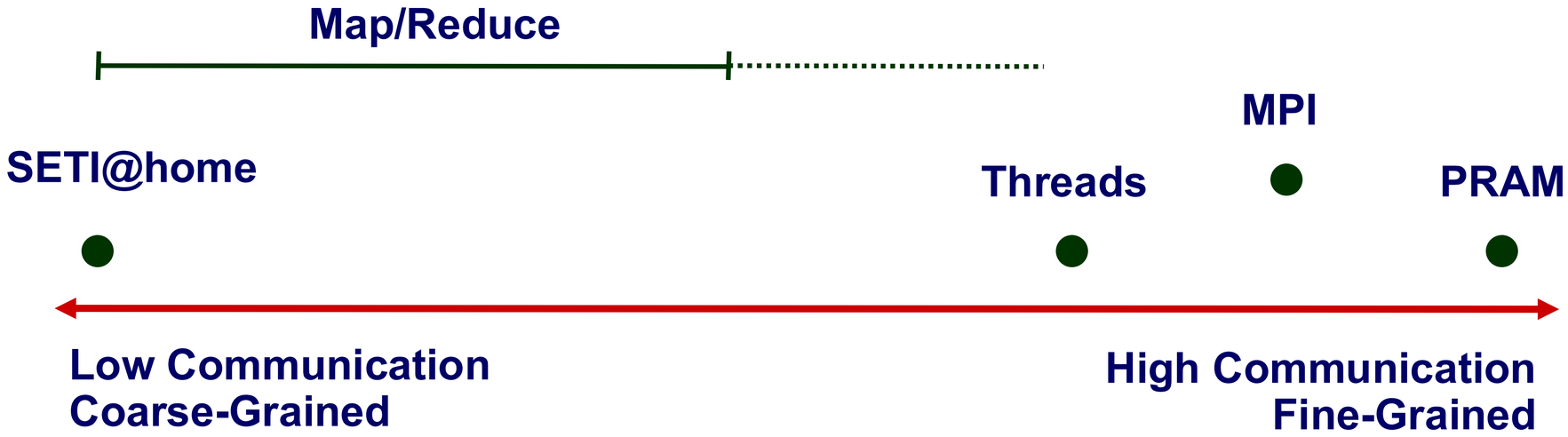
## Strengths

- Great flexibility in placement, scheduling, and load balancing
- Can access large data sets

## Weaknesses

- Higher overhead due to disk read/write
- Lower raw performance (each map / reduce task takes long to invoke)
- Learning Functional programming is non-trivial!

# Exploring Parallel Computation Models




Map/Reduce Provides Coarse-Grained Parallelism

- Computation done by independent processes
- File-based communication

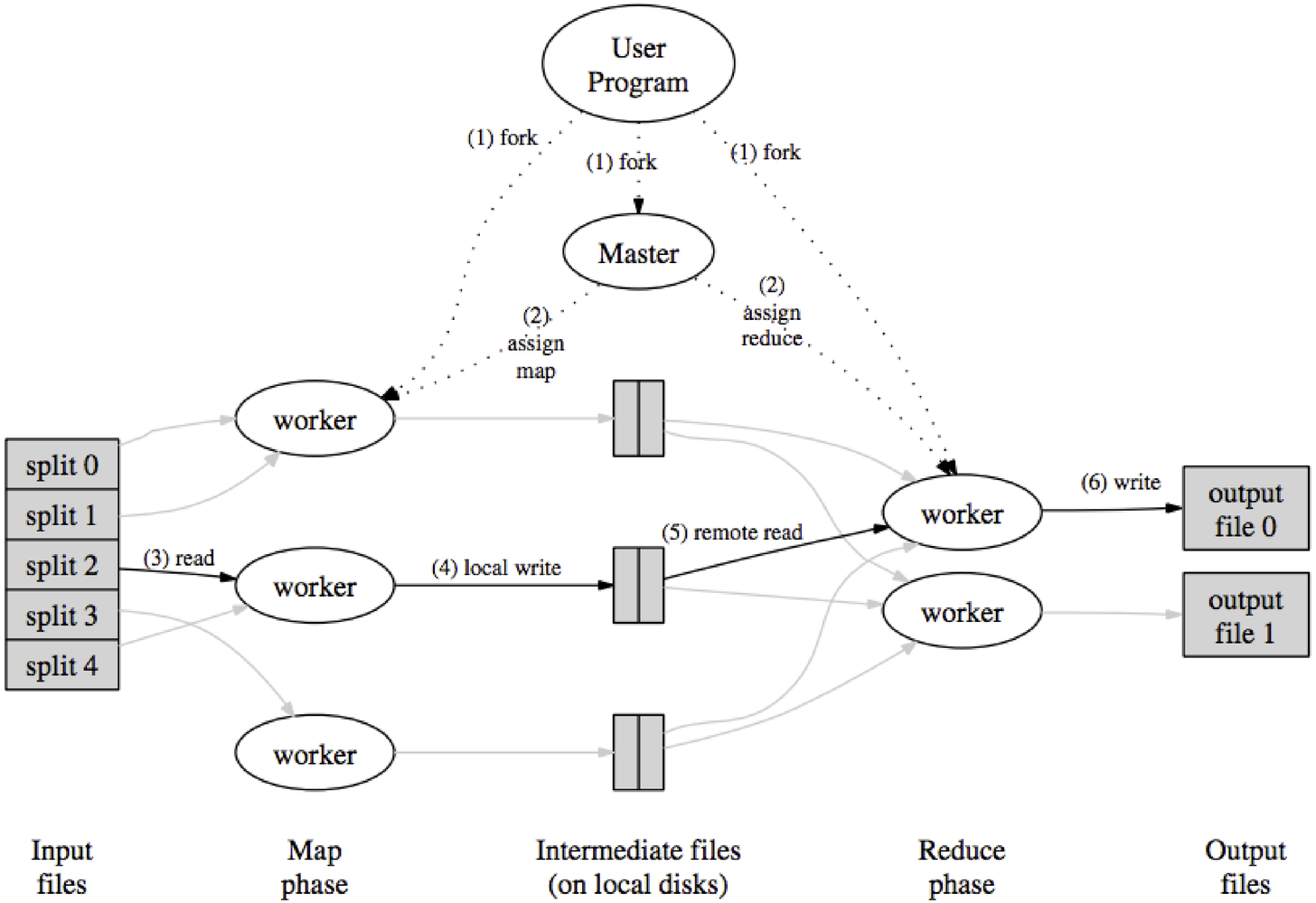
Observations

- Relatively "natural" programming model
- Research issue to explore full potential and limits

# Today's topic: Batch Processing

- Overview
- IO & Unix pipes
- MapReduce
  - HDFS - infrastructure
  - Programming models (API)
  - Job execution (runtime)
  - Workflow
  - MapReduce Recap
- Beyond MapReduce

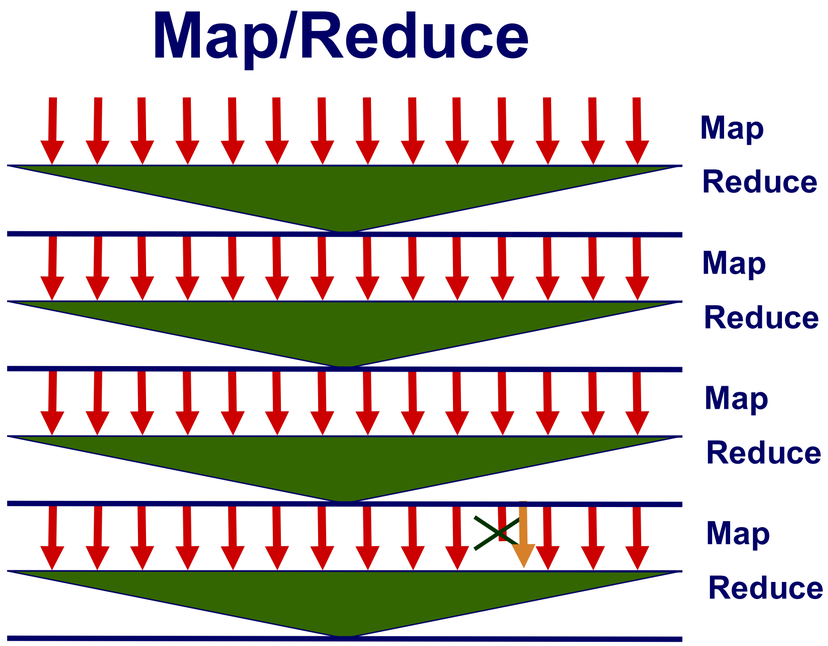
# MapReduce System architecture (Paper)



# Fault Tolerance and Straggler Mitigation

- Fault Tolerance
  - Assume reliable file system
  - Detect failed worker
    - Heartbeat mechanism
  - Reschedule failed task
- Dealing with Stragglers
  - Tasks that take long time to execute
  - Might be bug, flaky hardware, or poor partitioning
  - When done with most tasks, reschedule any remaining executing tasks
    - Keep track of redundant executions
    - Significantly reduces overall run time

# Fault Tolerance




## Data Integrity

- Store multiple copies of each file
- Including intermediate results of each Map / Reduc
  - Continuous checkpointing

## Recovering from Failure

- Simply recompute lost result
  - Localized effect
- Dynamic scheduler keeps all processors busy

# Map/Reduce Summary

## Typical Map/Reduce Applications

- Sequence of steps, each requiring map & reduce
- Series of data transformations

## Strengths of Map/Reduce

- User writes simple functions, system manages complexities of mapping, synchronization, fault tolerance
- Very general
- Good for large-scale data analysis

# Map Reduce Summary: Cons

- Disk I/O overhead is super high
- Not flexible enough: Each map/reduce step must complete before next begins
- Not suitable for workloads:
  - Iterative processing
  - Real-time processing
- Map-reduce is still difficult to program with

# PageRank Computation

Initially

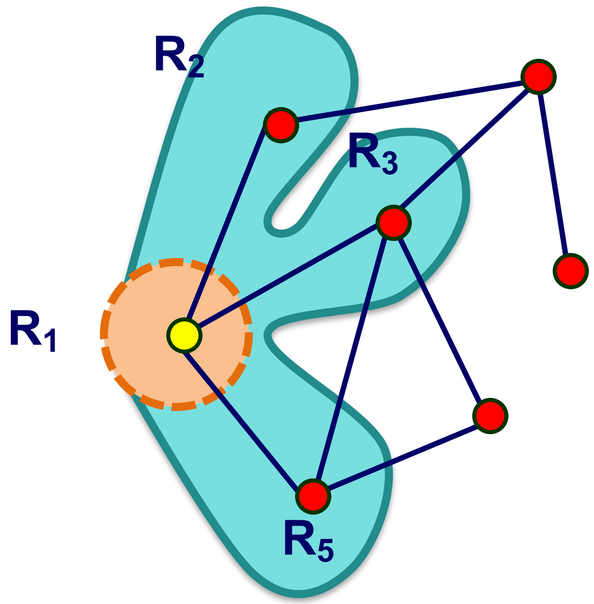
- Assign weight 1.0 to each page

Iteratively

- Select arbitrary node and update its value

Convergence

- Results unique, regardless of selection ordering




$$R_1 \leftarrow \quad 0.1 + 0.9 * (\tfrac{1}{2} R_2 + \tfrac{1}{4} R_3 + \tfrac{1}{3} R_5)$$

Q: how to express pagerank using map-reduce?

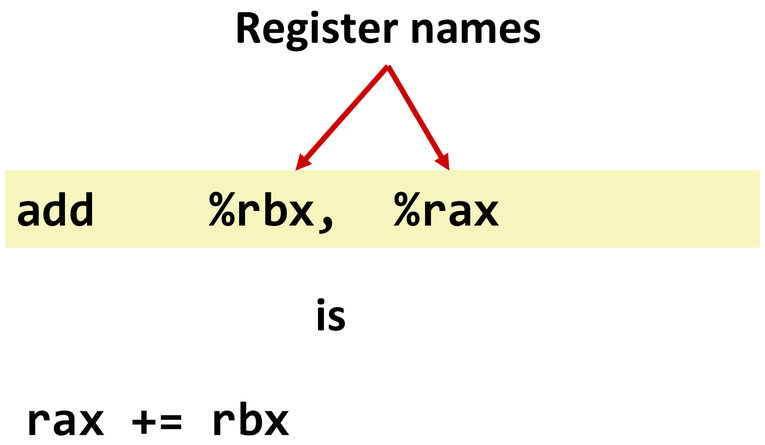
# Summary: Batch Processing

- Batch Processing
  - Suitable for latency insensitive tasks
  - Map-reduce prog model: mapper, reducer, (combiner, partitioner)
  - Many Map-reduce jobs to compose dataflows
    - They communicate via disk I/O
  - Pros and Cons
    - Pros: expressive, scalable, and fault tolerant
    - Cons: low performance due to disk I/O

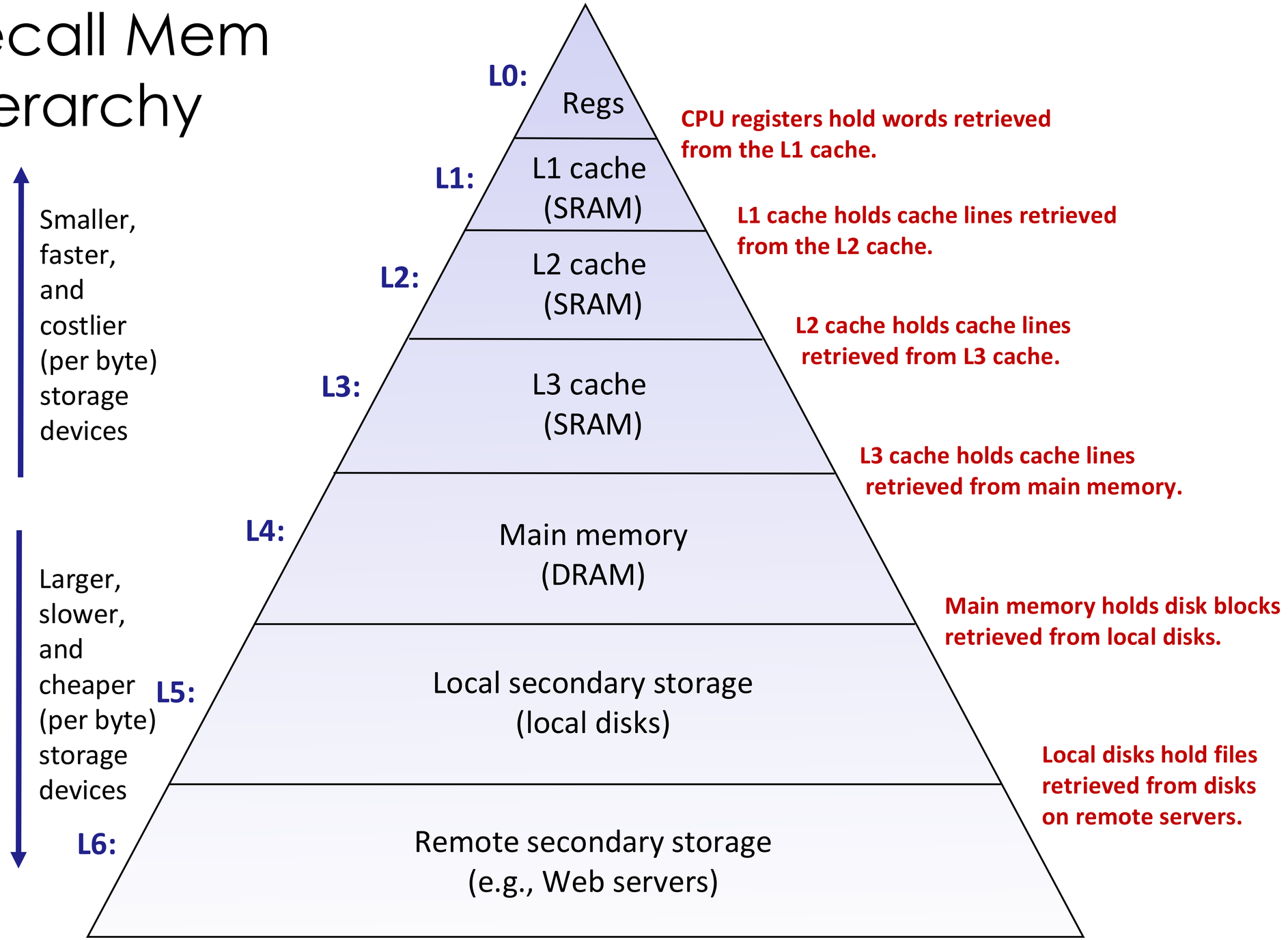
# Next: Stream Processing

- Computation vs. I/O: Arithmetic intensity
  - Loop fusion
- When MapReduce fails
- Spark and RDD
- Why Spark succeeded

# Recall: Instruction



# Recall Mem Hierarchy



# How to measure the impact of I/O

- I/O is the primary enemy of computer engineers/scientists: it will always slow down computation in every levels of the memory hierarchy
  - Processor reads/writes cache or memory
  - Map-reduce save and load results from distributed storage
- Q: how we measure such slowdown?
  - Arithmetic intensity

# Arithmetic Intensity

$$AI = \frac{\#Compute\ Op}{\#\mathrm{I/O}\ op}$$

# Arithmetic intensity

```
void add(int n, float*   A, float*   B, float*   C){
   for   (int   i=0; i<n; i++)
      C[i]  = A[i]  + B[i];
}
```

Two loads, one store per math op

1. Read A[i]
2. Read B[i]
3. Add A[i]+B[i]
4. Store C[i]

# Which program performs better? Program 1

```
void add(int n, float* A, float* B, float* C){
   for (int i=0; i<n; i++)
      C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
   for (int i=0; i<n; i++)
      C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;
//  assume arrays are allocated here
//  compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

# Which program performs better? Program 2

```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);



void fused(int n, float* A, float* B, float* C, float* D,
   float* E) {
   for (int i=0; i<n; i++)
      E[i] = D[i] + (A[i] + B[i]) * C[i];
}
// compute E = D + (A + B) * C
fused(n, A, B,C, D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5

**computation fusion!**

Core Problem of Map-reduce

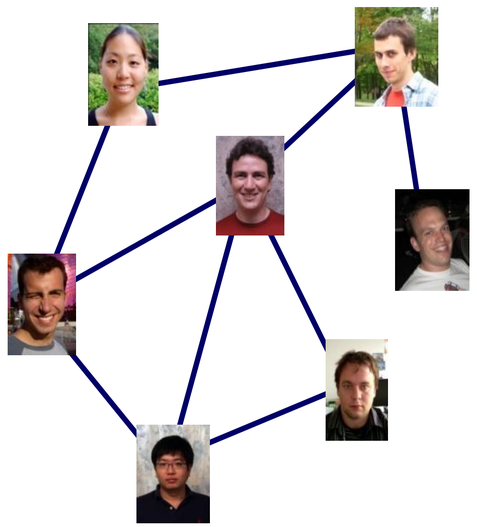
# Low arithmetic intensity due to Disk I/O

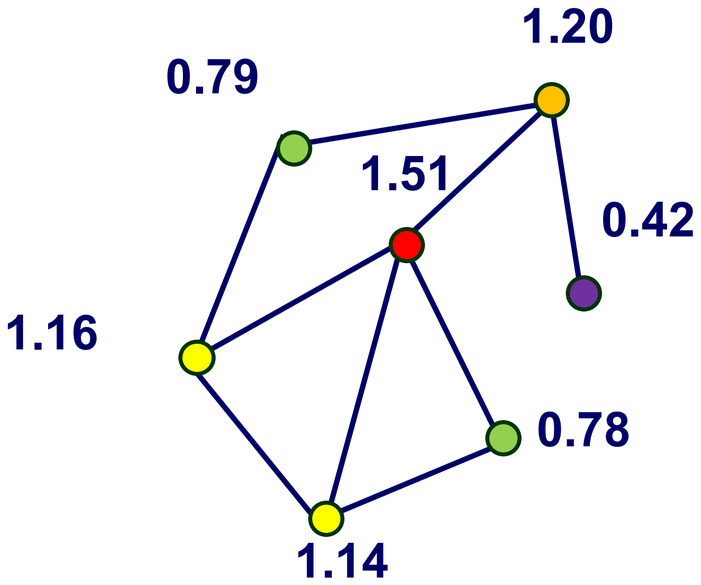
# PageRank

## PageRank Computation

- Larry Page & Sergey Brinn, 1998

## Rank “Importance” of Web Pages





# PageRank Computation

Initially

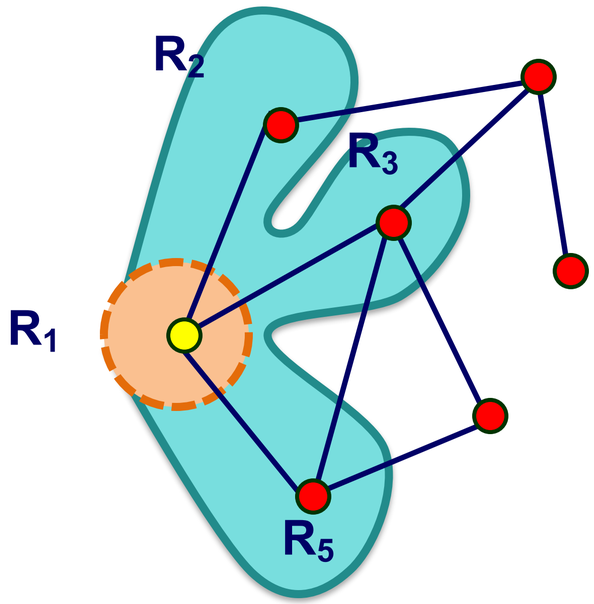
- Assign weight 1.0 to each page

Iteratively

- Select arbitrary node and update its value

Convergence

- Results unique, regardless of selection ordering




$R_1 \leftarrow \quad 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$

# PageRank with Map/Reduce

$R_1 \leftarrow 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$

## Each Iteration: Update all nodes

- Map: Generate values to pass along each edge
  - Key value 1: (1, ½ $R_2$)  (1, ¼ $R_3$)  (1, ⅓ $R_5$)
  - Similar for all other keys
- Reduce: Combine edge values to get new rank
  - $R_1 \leftarrow 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$
  - Similar for all other nodes

# Iterative algorithms must load from disk each iteration

```
void pagerank_mapper(graphnode n, map<string,string> results) {
   float val = compute update value for n
   for (dst in outgoing links from n)
     results.add(dst.node, val);
}

void pagerank_reducer(graphnode n, list<float> values, float& result) {
    float sum = 0.0;
    for (v in values)
       sum += v;
    result = sum;
}

for (i = 0 to NUM_ITERATIONS) {
   input = load graph from last iteration
   output = file for this iteration output
   runMapReduceJob(pagerank_mapper, pagerank_reducer, result[i-1], result[i]);
}
```

Low
Arithmetic Intensity!


in-memory, fault-tolerant distributed computing
http://spark.apache.org/

# Goals

- This guy felt UC Grad student salary too low so he decided to make some money (roughly 1M) via the Netflix challenge.

## Netflix Prize

2 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Netflix Prize** was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films, i.e. without the users being identified except by numbers assigned for the contest.

The competition was held by Netflix, a video streaming service, and was open to anyone who is neither connected with Netflix (current and former employees, agents, close relatives of Netflix employees, etc.) nor a resident of certain blocked countries (such as Cuba or North Korea).[1] On September 21, 2009, the grand prize of US$1,000,000 was given to the BellKor's Pragmatic Chaos team which bested Netflix's own algorithm for predicting ratings by 10.06%.[2]

**Recommender systems**

**Concepts**
Collective intelligence · Relevance · Star ratings · Long tail

**Methods and challenges**
Cold start · Collaborative filtering · Dimensionality reduction · Implicit data collection · Item-item collaborative filtering · Matrix factorization · Preference elicitation · Similarity search

## Matei Zaharia

Associate Professor, Computer Science
matei@berkeley.edu
Google Scholar | LinkedIn | Twitter

I'm an associate professor at UC Berkeley (previously Stanford), where I work on computer systems and machine learning. I'm also co-founder and CTO of Databricks.

**Interests:** I'm interested in computer systems for large-scale workloads such as AI, data analytics

# Goals

- Programming model for cluster-scale computations where there is **significant reuse** of intermediate datasets
  - Iterative machine learning and graph algorithms
  - Interactive data mining: load large dataset into aggregate memory of cluster and then perform multiple ad-hoc queries

- Don't want incur inefficiency of writing intermediates to persistent distributed file system (want to keep it in memory)
  - Challenge: efficiently implementing fault tolerance for large-scale distributed in-memory computations.

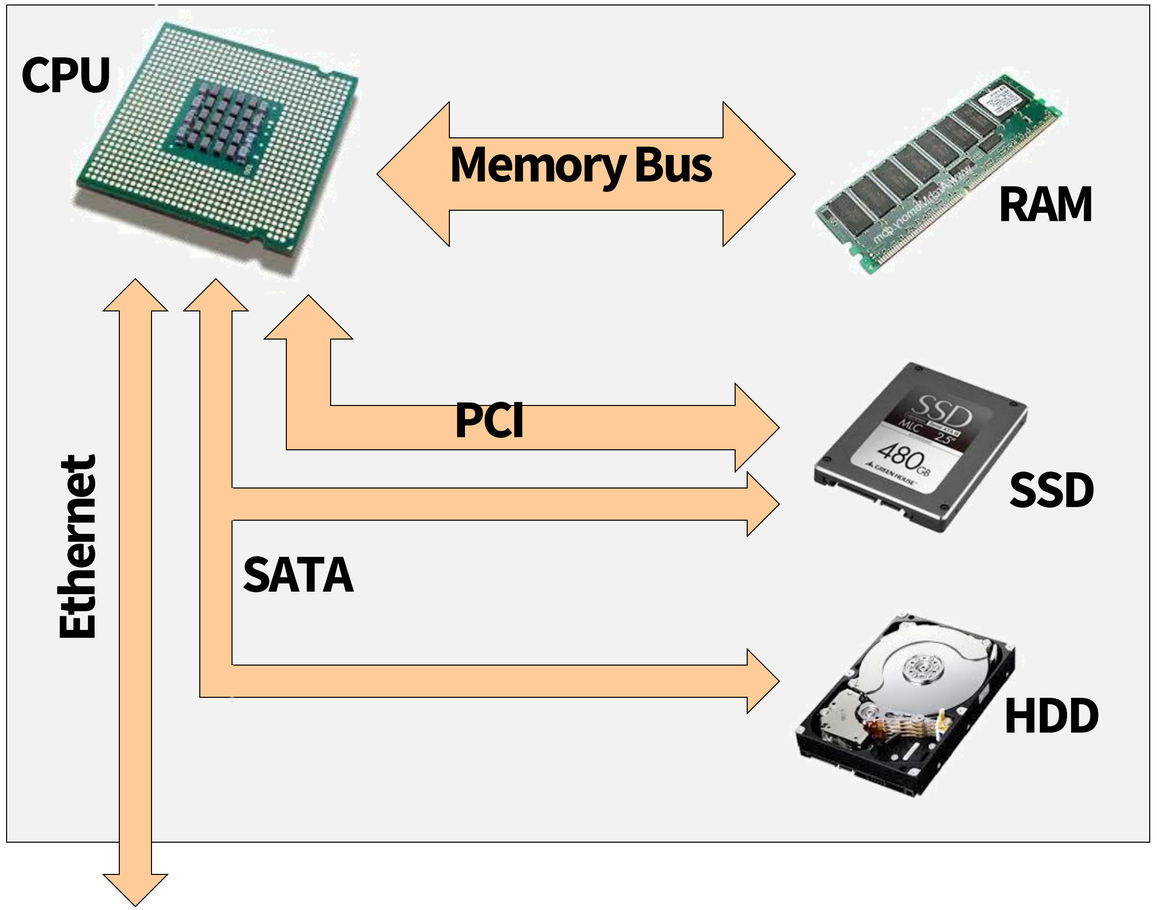
# Recall map-reduce solutions

- Checkpoints after each map/reduce step by writing results to fle system
- Scheduler's list of outstanding (but not yet complete) jobs is a log
- Functional structure of programs allows for restart at granularity of a single mapper or reducer invocation
  - (don't have to restart entire program)

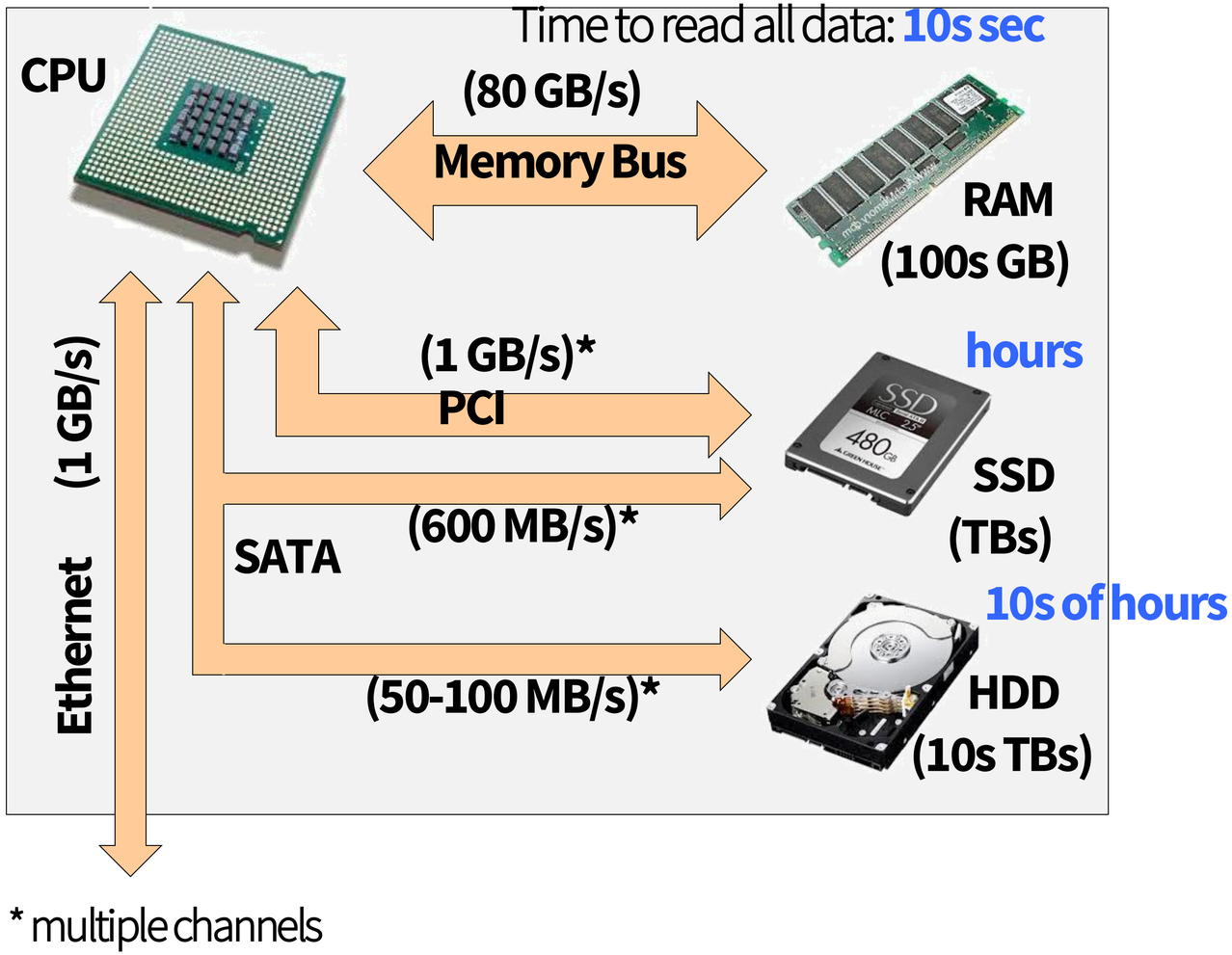
# Three Necessary Conditions

- Memory: large (cheap) enough
- Network: fast (cheap) enough
- fault tolerance: at least as good as map-reduce

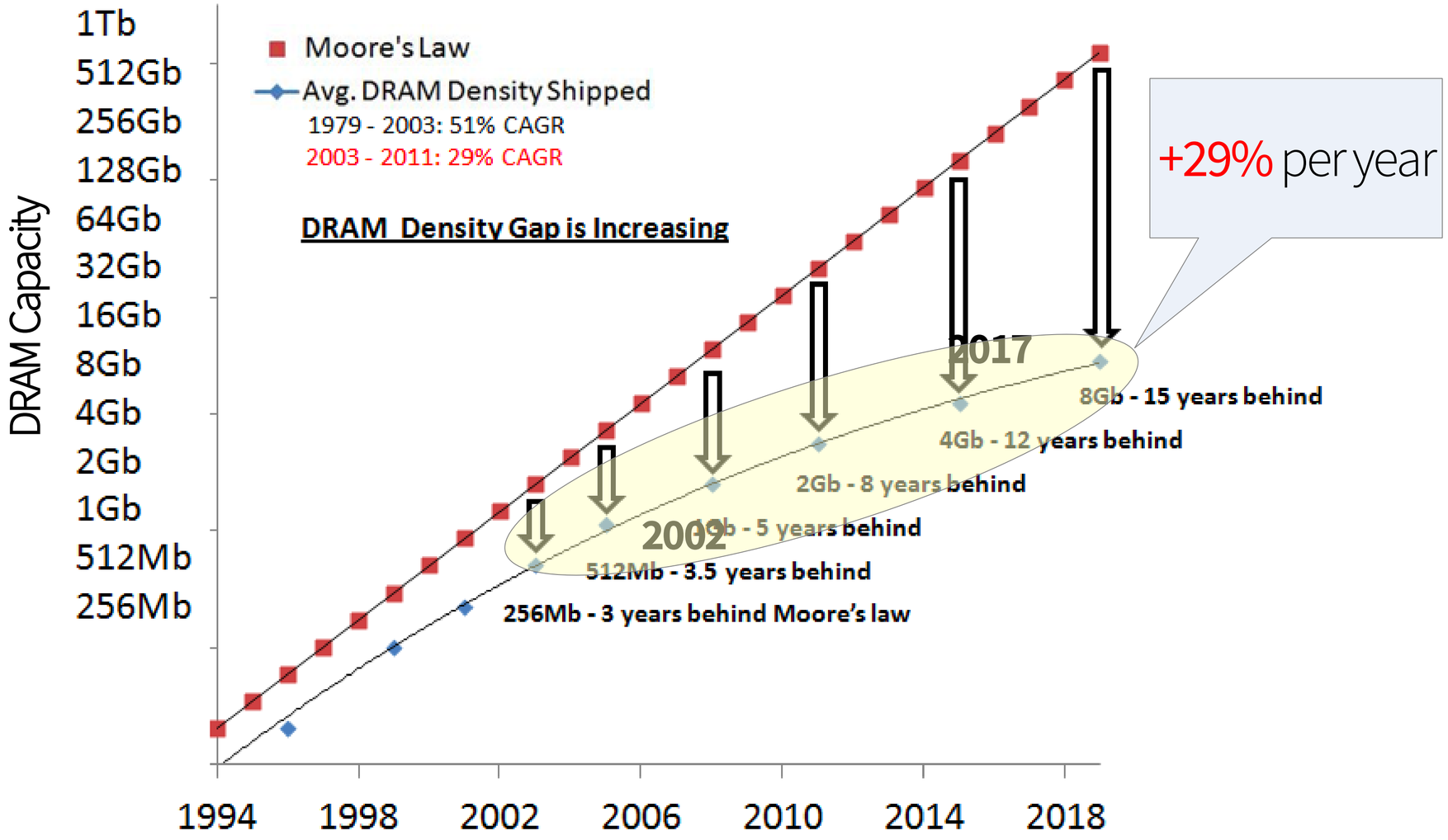
# Typical Server Node



# Typical Server Node



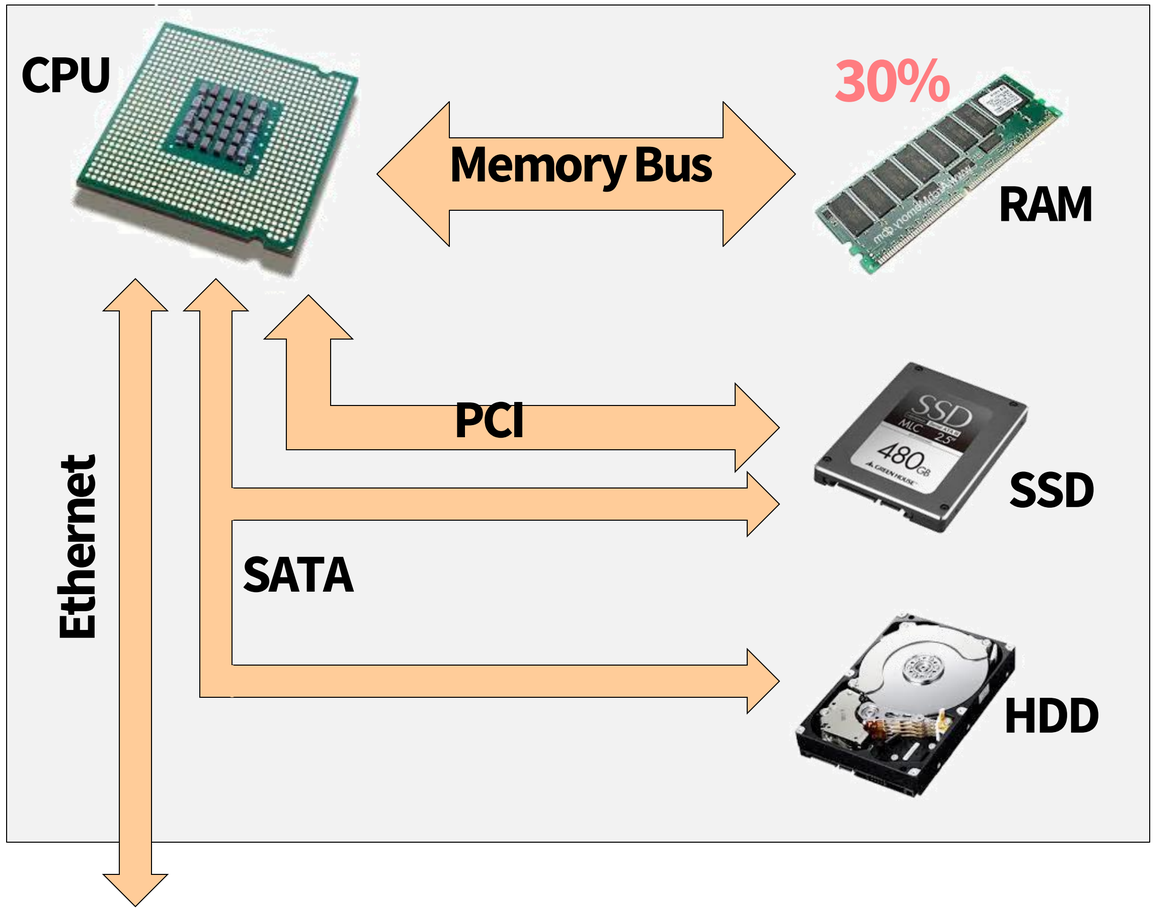
# Memory Capacity



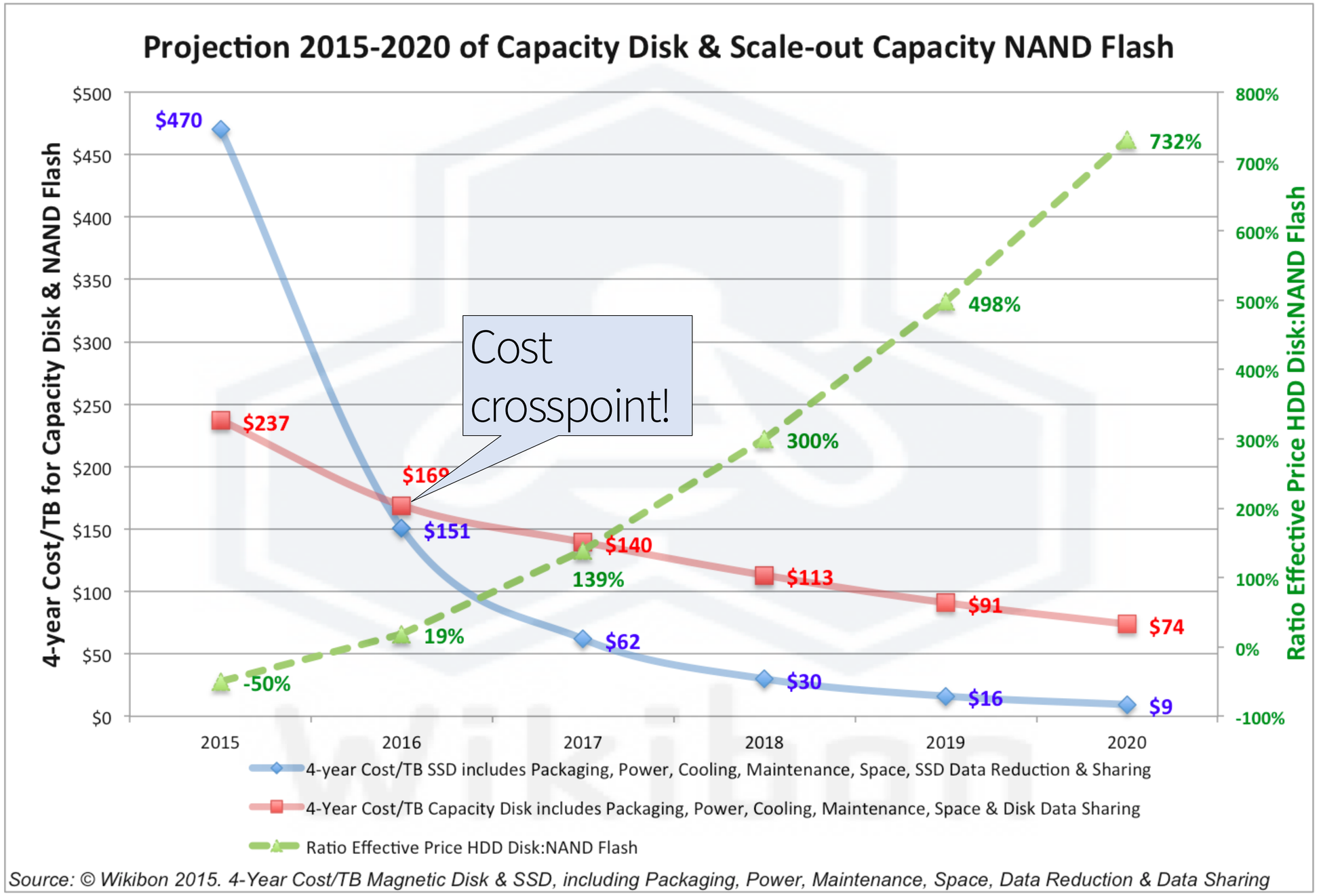
# Memory Price/Byte Evolution

- 1990-2000: -54% per year
- 2000-2010: -51% per year
- 2010-2015: -32% per year
- (http://www.jcmit.com/memoryprice.htm)

# Typical Server Node



d




*Source: © Wikibon 2015. 4-Year Cost/TB Magnetic Disk & SSD, including Packaging, Power, Maintenance, Space, Data Reduction & Data Sharing*

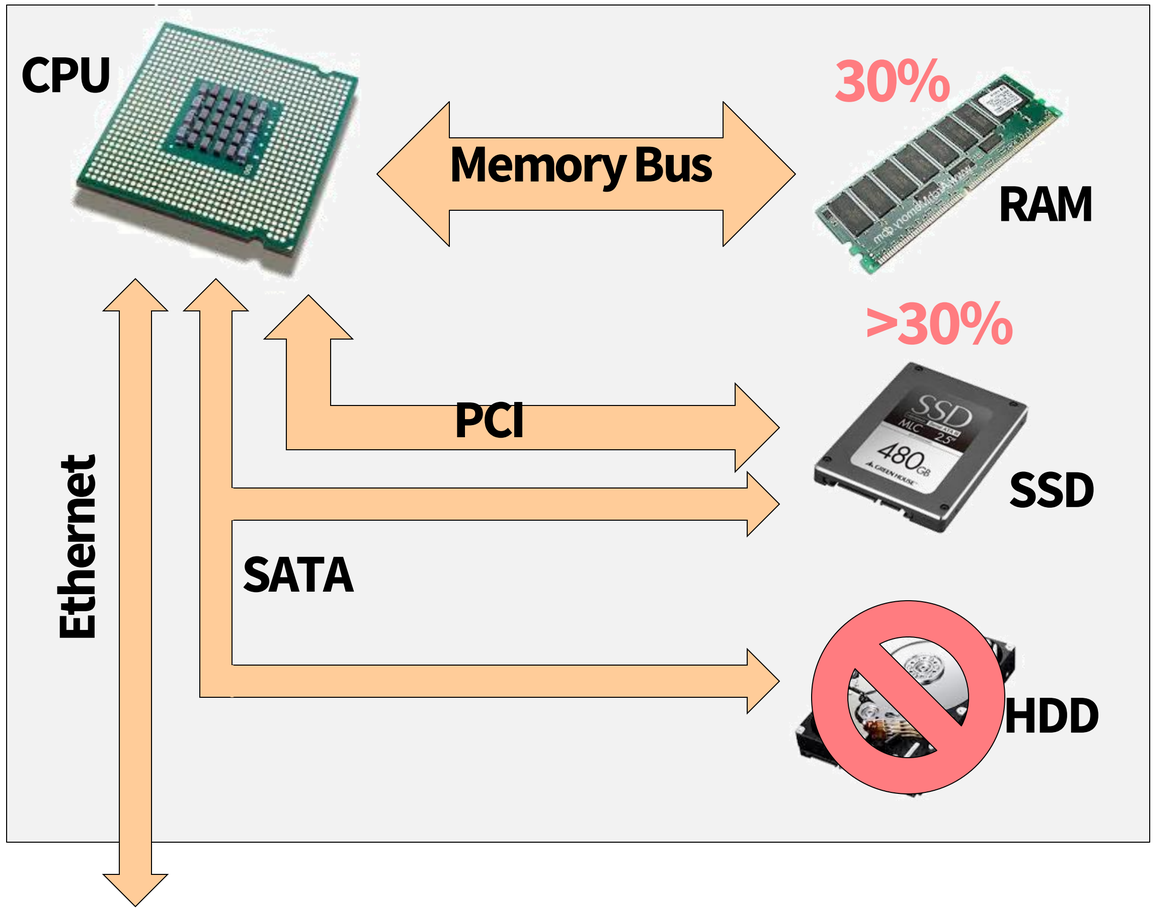
# SSDs vs. HDDs

- SSDs has become cheaper than (or as cheap as to) HDDs

- Transition from HDDs to SSDs has accelerate
  - Already most instances in AWS have SSDs
  - Digital Ocean instances are SSD only

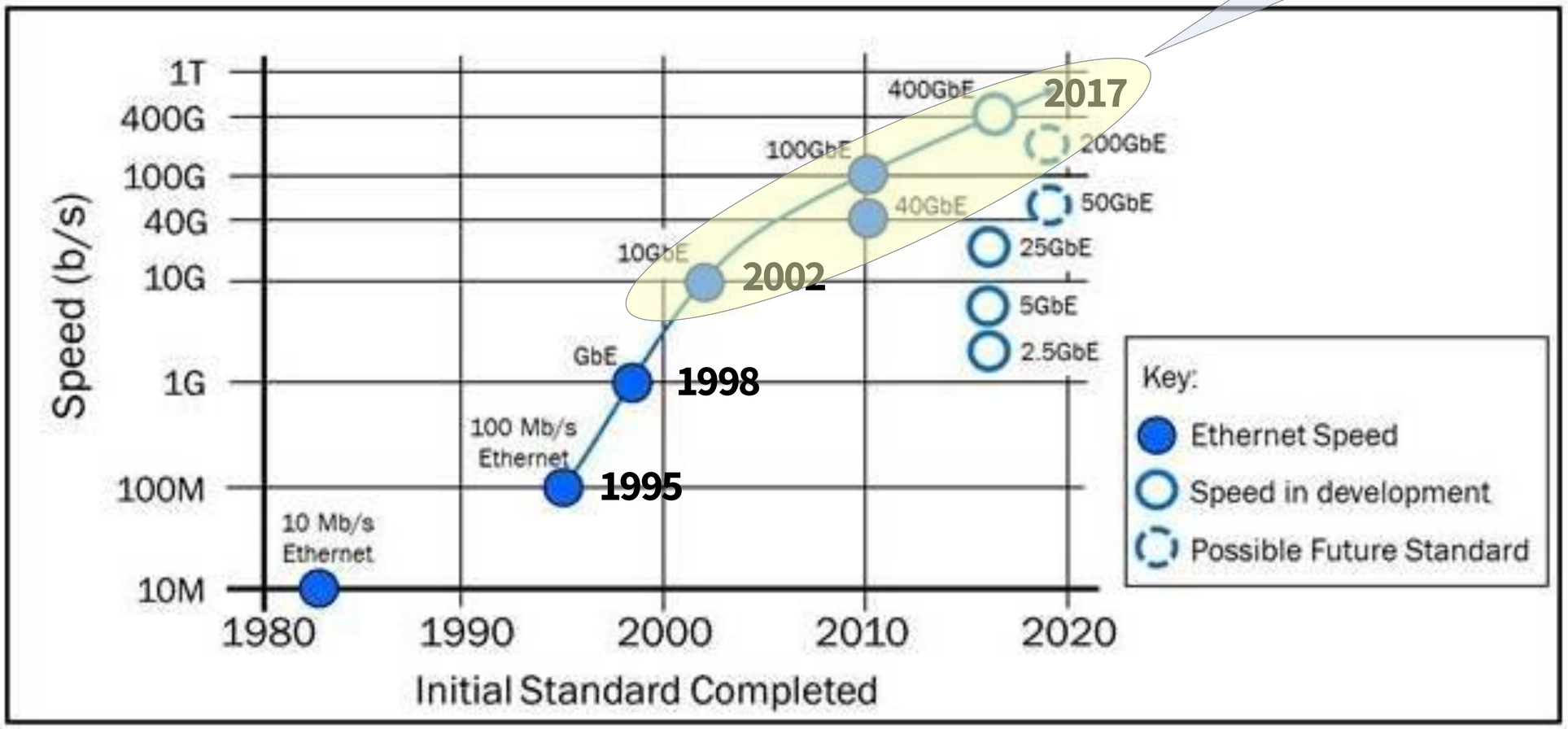
- Going forward we can assume SSD only clusters

# Typical Server Node

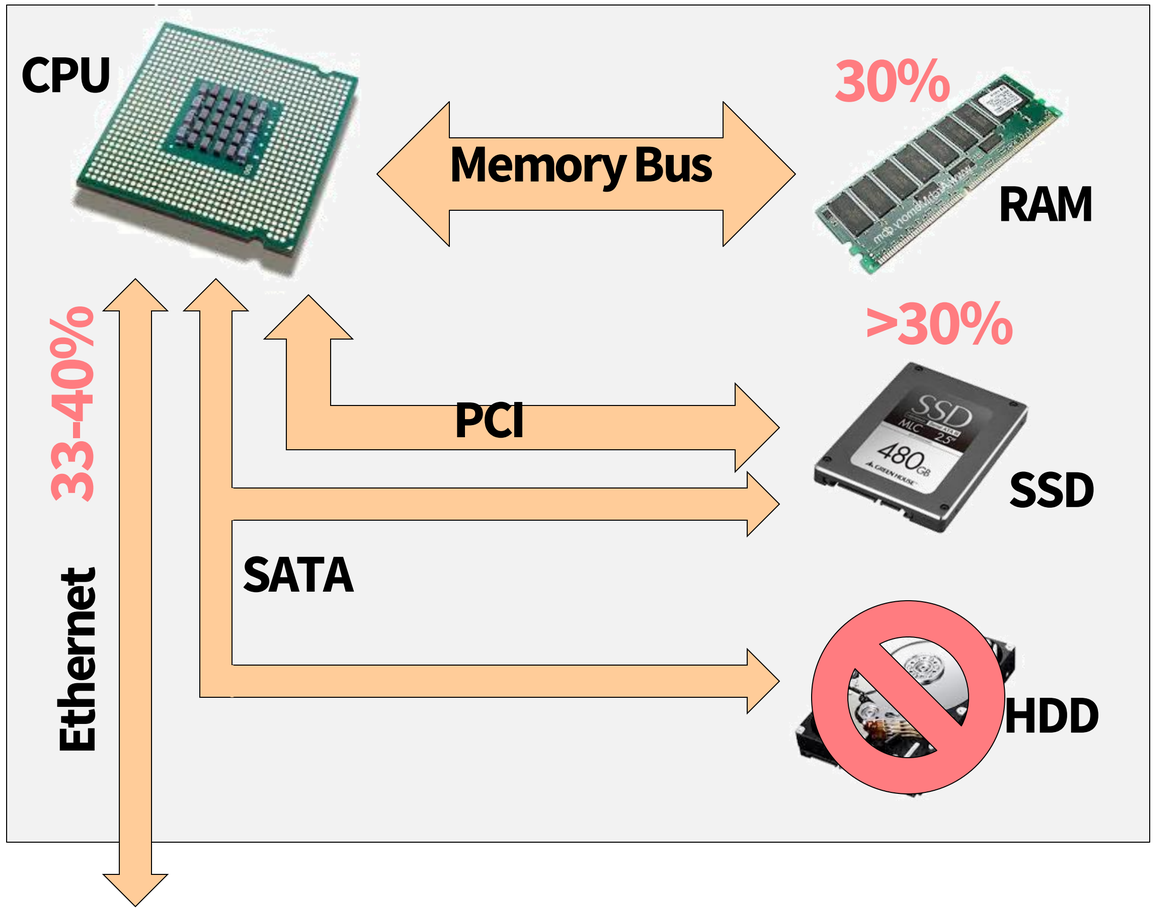


# Ethernet Bandwidth

33-40% per year



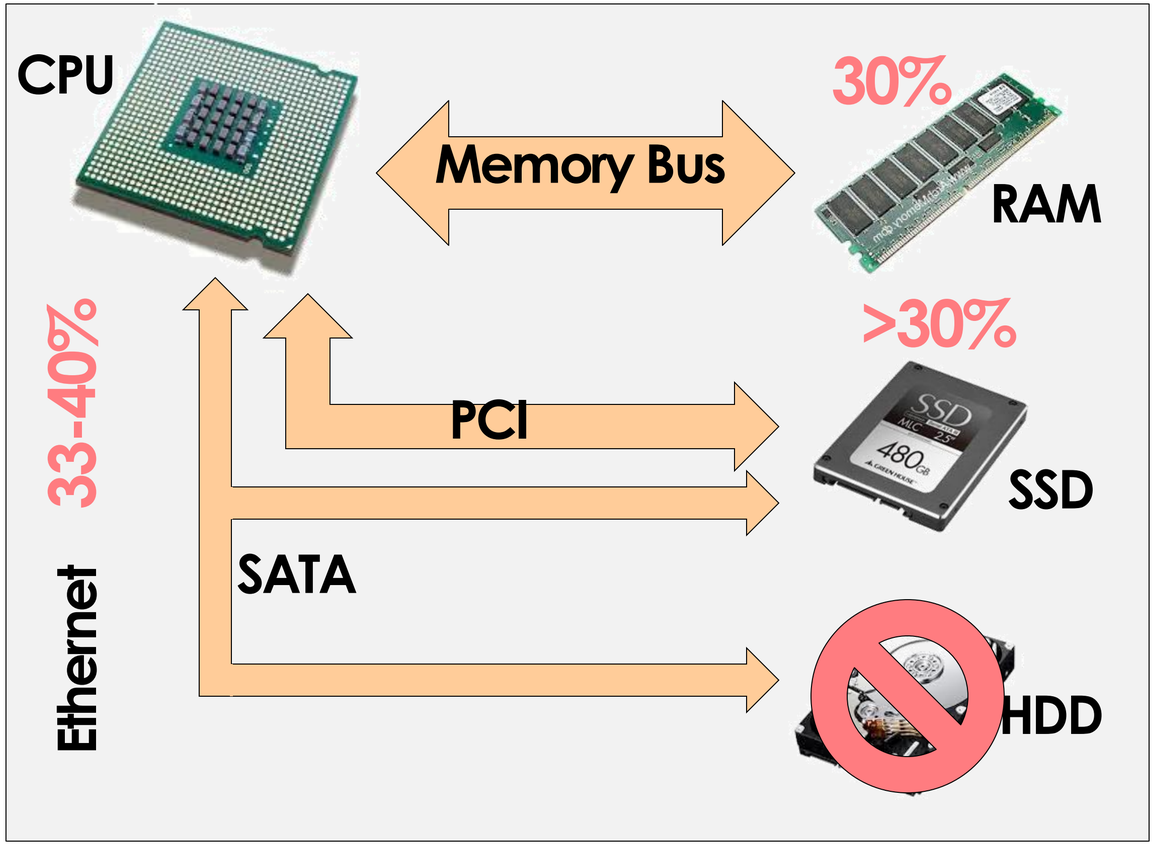
# Typical Server Node



# What Does This Mean?

- Memory hierarchy has shift one layer up

- HDD is virtually dead
- We have unlimited space of SSD
- Today's RAM space = yesterday's SSD space
- Today's SSD space = yesterday's HDD space
- Ethernet may become faster than PCI/SATA bandwidth



# Three Necessary Conditions

- Memory: large (cheap) enough ✅
- Network: fast (cheap) enough ✅
- Fault tolerance: at least as good as map-reduce

# Fault tolerance for in-memory calculations

- Replicate all computations
  - Expensive solution: decreases peak throughput
- Checkpoint and rollback
  - Periodically save state of program to persistent storage
  - Restart from last checkpoint on node failure
- Maintain log of updates (commands and data)
  - High overhead for maintaining logs

# Resilient distributed dataset (RDD)

**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 to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

## 1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.,* between two MapReduce jobs) is to write it to an external stable storage system, 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.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.,* map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

# Stream Processing

- Computation vs. I/O: Arithmetic intensity
  - Loop fusion
- When MapReduce fails
- **Spark and RDD**
- Spark Ecosystem and Beyond
- Early ML systems: parameter server

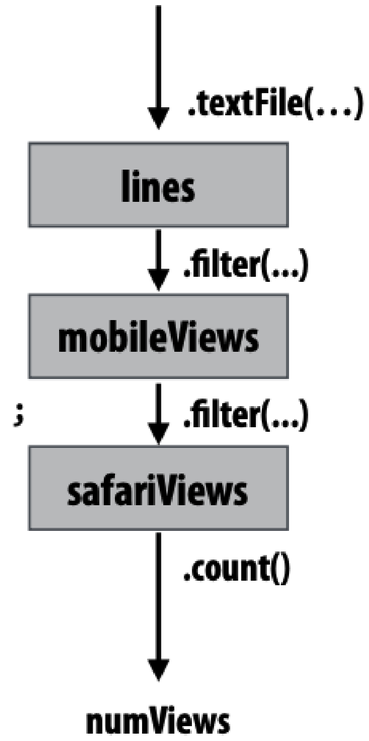
# RDD: Spark's key programming abstraction:

- Read-only collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs

RDDs

```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```

int



Predefined Set of Operators

Transformation

Action

# RDD transformations and actions

**Transformations: (data parallel operators taking an input RDD to a new RDD)**

$$
\begin{aligned}
map(f : \mathrm{T} \Rightarrow \mathrm{U}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]} \\
filter(f : \mathrm{T} \Rightarrow \mathrm{Bool}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]} \\
flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]} \\
sample(fraction : \mathrm{Float}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]} \text{ (Deterministic sampling)} \\
groupByKey() &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]} \\
reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]} \\
union() &: (\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]} \\
join() &: (\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]} \\
cogroup() &: (\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]} \\
crossProduct() &: (\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]} \\
mapValues(f : \mathrm{V} \Rightarrow \mathrm{W}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]} \text{ (Preserves partitioning)} \\
sort(c : \mathrm{Comparator[K]}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]} \\
partitionBy(p : \mathrm{Partitioner[K]}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}
\end{aligned}
$$

**Actions: (provide data back to the "host" application)**

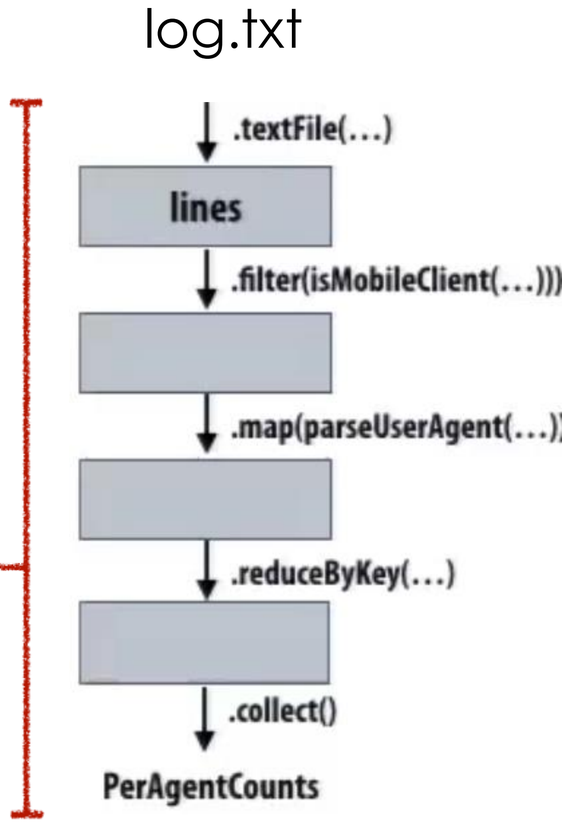
$$
\begin{aligned}
count() &: \mathrm{RDD[T]} \Rightarrow \mathrm{Long} \\
collect() &: \mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]} \\
reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{T} \\
lookup(k : \mathrm{K}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]} \text{ (On hash/range partitioned RDDs)} \\
save(path : \mathrm{String}) &: \text{Outputs RDD to a storage system, } e.g., \text{ HDFS}
\end{aligned}
$$

# Repeating the map-reduce example

```
// 1. create RDD from file system data
// 2. create RDD with only lines from mobile clients
// 3. create RDD with elements of type (String,Int) from line string
// 4. group elements by key
// 5. call provided reduction function on all keys to count views
var perAgentCounts = spark.textFile("hdfs://log.txt")
                          .filter(x => isMobileClient(x))
                          .map(x => (parseUserAgent(x),1));
                          .reduceByKey((x,y) => x+y)
                          .collect();
```

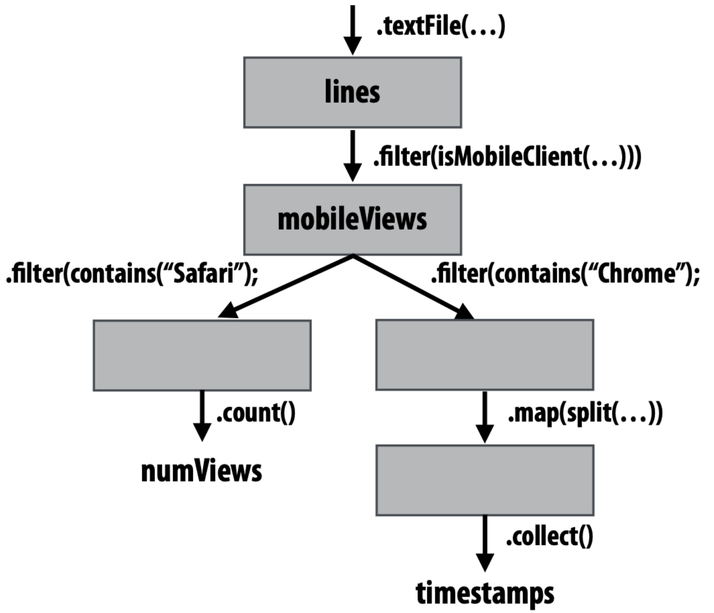
Array [String,int]

"Lineage": Sequence of RDD operations needed to compute output



# Another Spark program

```
// create RDD from file system data
var lines = spark.textFile("hdfs://log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// instruct Spark runtime to try to keep mobileViews in memory
mobileViews.persist();

// create a new RDD by filtering mobileViews
// then count number of elements in new RDD via count() action
var numViews = mobileViews.filter(_.contains("Safari")).count();

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of // page view
// 3. convert RDD to a scalar sequence (collect() action)
var timestamps = mobileViews.filter(_.contains("Chrome"))
                            .map(_.split(" ")(0))
                            .collect();
```



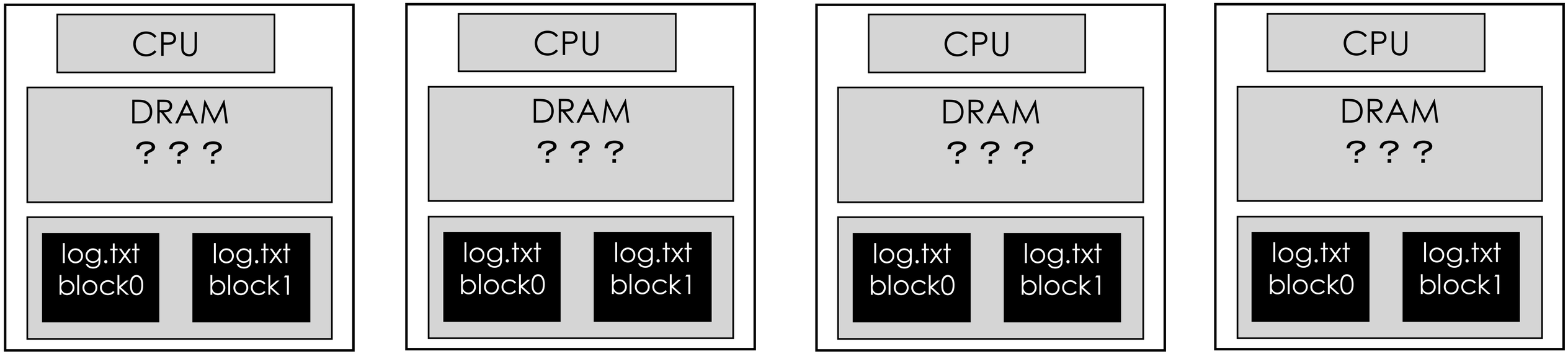
# Discussion

- How do you like this programming model?
- v.s. map reduce
  - Flexibility and Expressiveness?
  - Simplicity?
  - Scalability?
  - Fault tolerance?

# How do we implement RDDs?

- In particular, how should they be stored?
  - var lines = spark.textFile("hdfs://log.txt");
  - var lower = lines.map(_.toLower());
  - var mobileViews = lower.filter(x => isMobileClient(x));
  - var howMany = mobileViews.count();

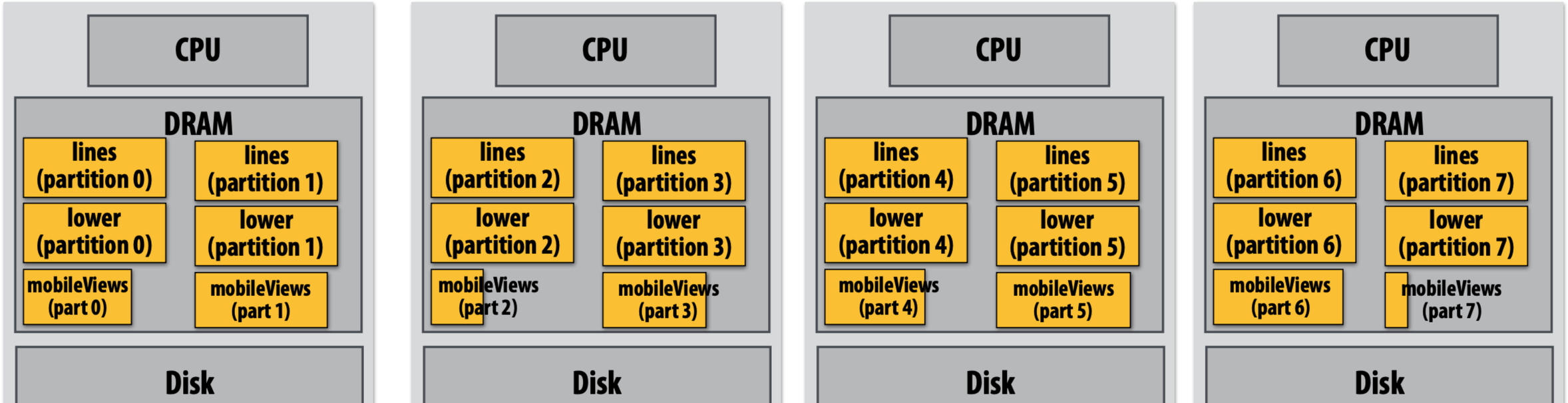
Question: should we think of RDD's like arrays?



# How do we implement RDDs?

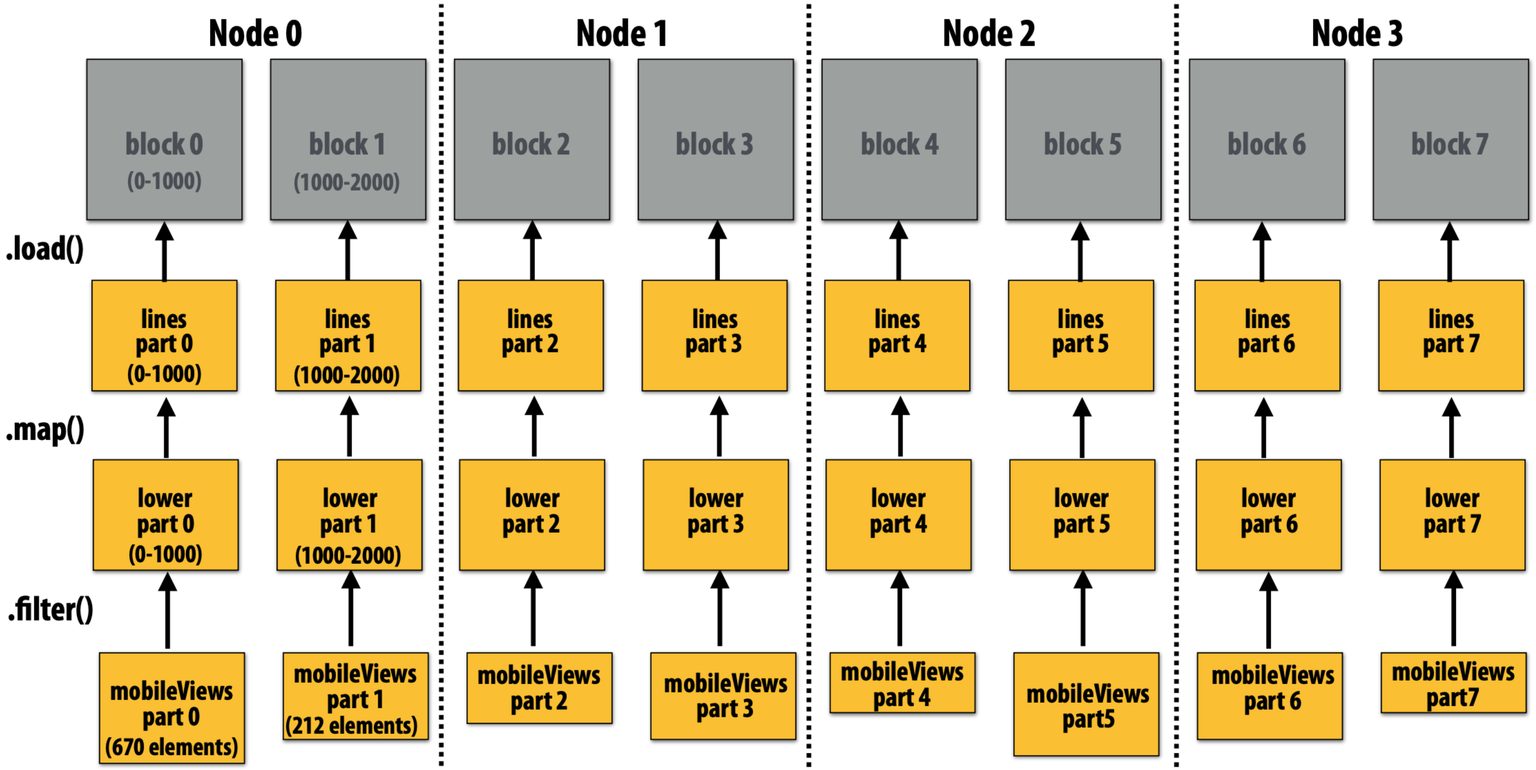
- In particular, how should they be stored?
  - var lines = spark.textFile("hdfs://log.txt");
  - var lower = lines.map(_.toLower());
  - var mobileViews = lower.filter(x => isMobileClient(x));
  - var howMany = mobileViews.count();

Question: Array -> In-memory representation would be huge! (larger than original flie on disk)



# RDD partitioning and dependencies

```
var lines = spark.textFile("hdfs://log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```




**Black lines show dependencies between RDD partitions.**

# Implementing sequence of RDD ops efficiently

```
var lines = spark.textFile(“hdfs://log.txt”);
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

- Recall “loop fusion” from start of lecture
- The following code stores only a line of the log file in memory, and only reads input data from disk once (“streaming” solution)

```
int count = 0;
while (inputFile.eof()) {
   string line = inputFile.readLine();
   string lower = line.toLower;
   if (isMobileClient(lower))
      count++;
}
```

# A simple interface for RDDs

```
var lines = spark.textFile("hdfs://log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

```
// create RDD by mapping fun onto input (parent) RDD
RDD::map(RDD parent, func) {
   return new RDDFromMap(parent, func);
}

// create RDD from text file on disk
RDD::textFile(string filename) {
   return new RDDFromTextFile(open(filename));
}

// count action (forces evaluation of RDD)
RDD::count() {
   int count = 0;
   while (hasMoreElements()) {
      var el = next();
      count++;
   }
}
```

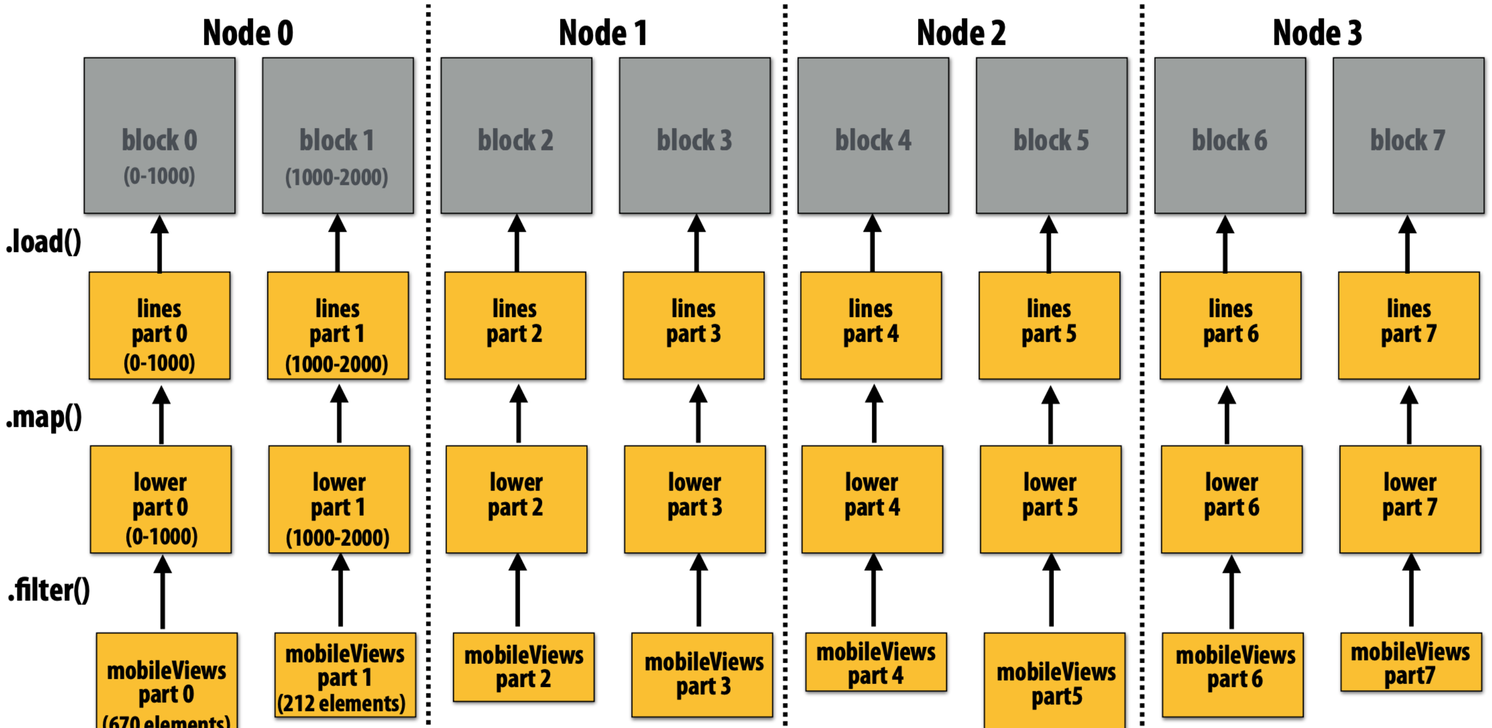
```
RDD::hasMoreElements() {
   parent.hasMoreElements();
}

// overloaded since no parent exists
RDDFromTextFile::hasMoreElements() {
   return !inputFile.eof();
}

RDDFromTextFile::next() {
   return inputFile.readLine();
}

RDDFromMap::next() {
   var el = parent.next();
   return el.toLower();
}

RDDFromFilter::next() {
  while (parent.hasMoreElements()) {
    var el = parent.next();
    if (isMobileClient(el))
      return el;
}
```

# Narrow dependencies

```
var lines = spark.textFile("hdfs://log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

"Narrow dependencies" = each partition of parent RDD referenced by at most one child RDD partition

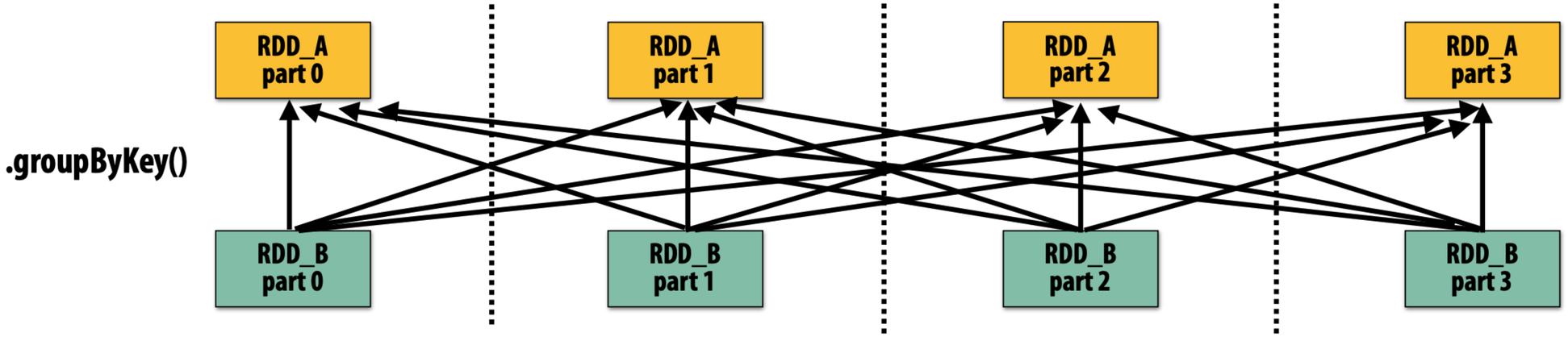
- Allows for fusing of operations
  (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster
  (communication of one int at end to perform count() reduction)



# Wide dependencies

groupByKey: RDD[(K,V)] → RDD[(K,Seq[V])]

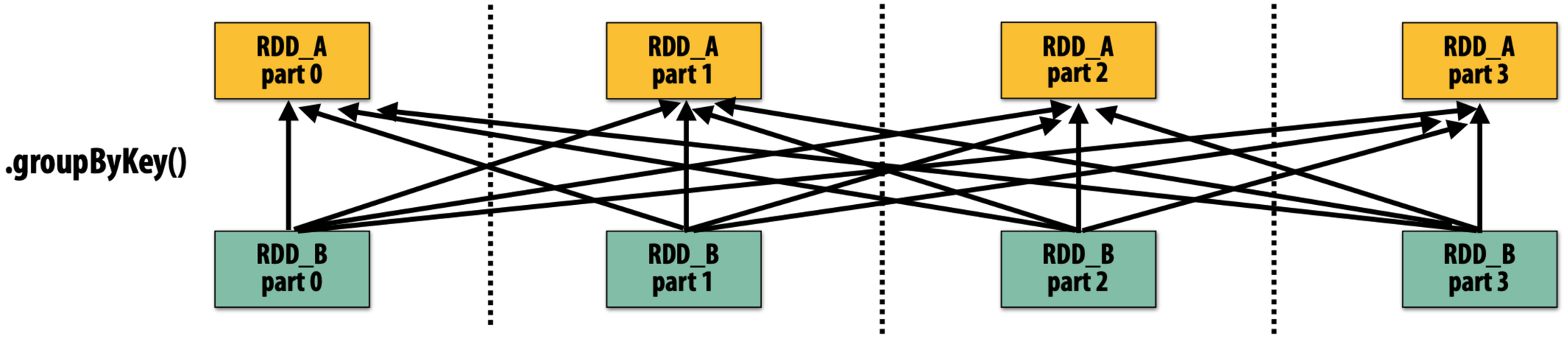
"Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key."




Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

# Wide dependencies

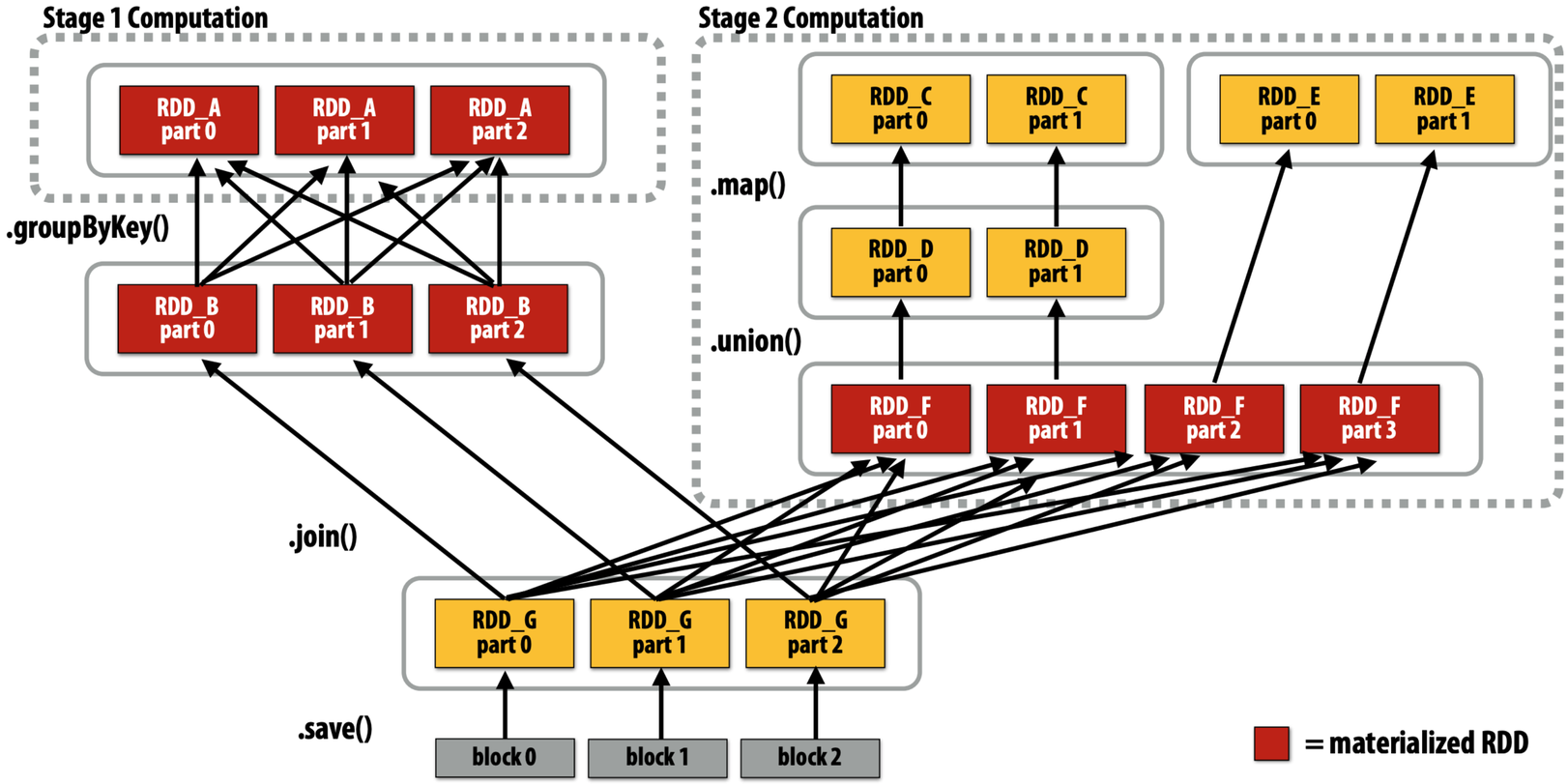
Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions




Challenges:

- Must compute all of RDD_A before computing RDD_B
  - Example: groupByKey() may induce all-to-all communication as shown above
- May trigger significant recompilation of ancestor lineage upon node failure (will address resilience in a few slides)

# Scheduling Spark computations

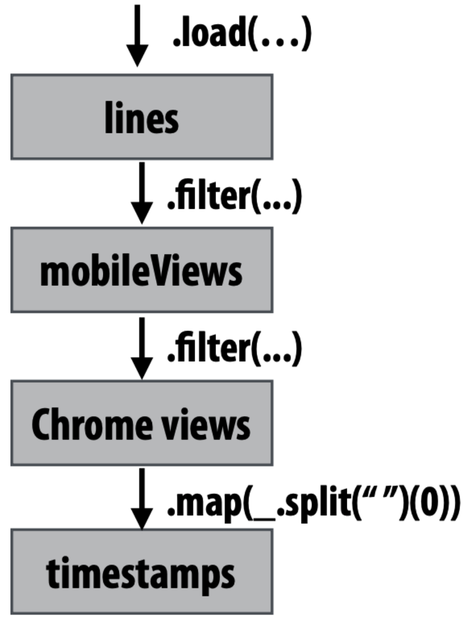



- Actions (e.g., save()) trigger evaluation of Spark lineage graph.
  - Stage 1 Computation: do nothing since input already materialized in memory
  - Stage 2 Computation: evaluate map in fused manner, only actually materialize RDD F
  - Stage 3 Computation: execute join (could stream the operation to disk, do not need to materialize )

# Implementing resilience via lineage

- RDD transformations are bulk, deterministic, and functional
  - Implication: runtime can always reconstruct contents of RDD from its lineage (the sequence of transformations used to create it)
  - Lineage is a log of transformations
  - Efficient: since log records bulk data-parallel operations, overhead of logging is low (compared to logging fine-grained operations, like in a database)

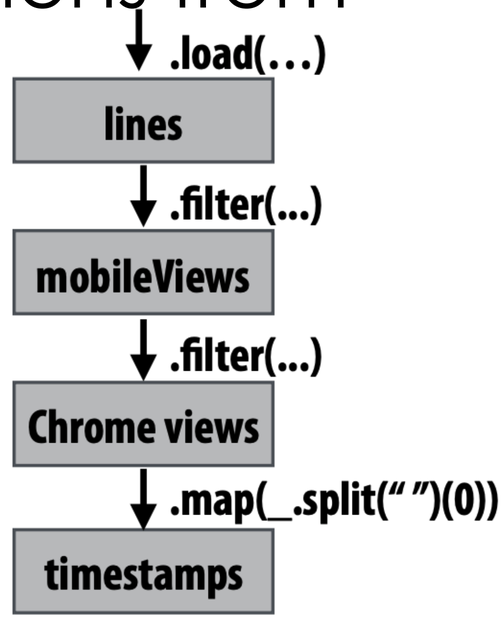
```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");
// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));
// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of // page view (first element)
// 3. convert RDD To a scalar sequence (collect() action)
var timestamps = mobileView.filter(_.contains("Chrome")) .map(_.split(" ")(0));
```

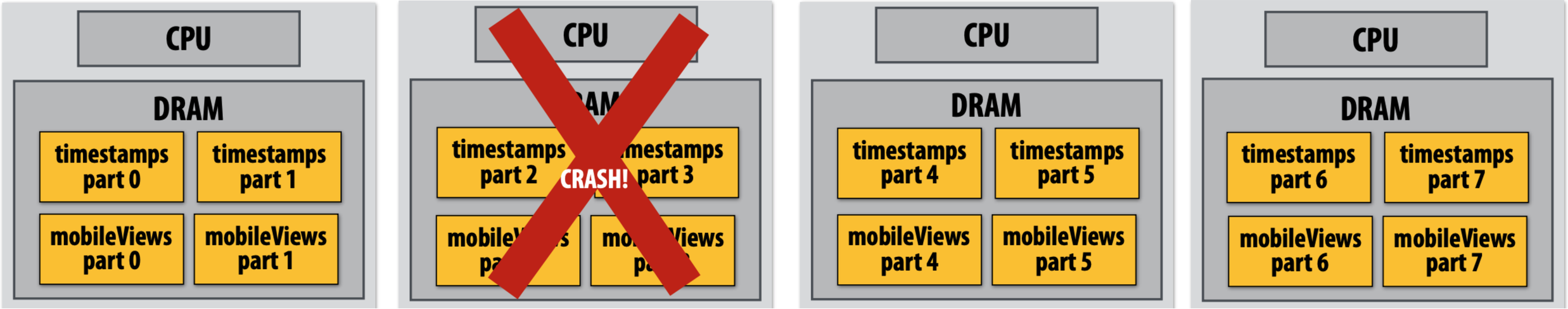


# Upon node failure: recompute lost RDD partitions from lineage

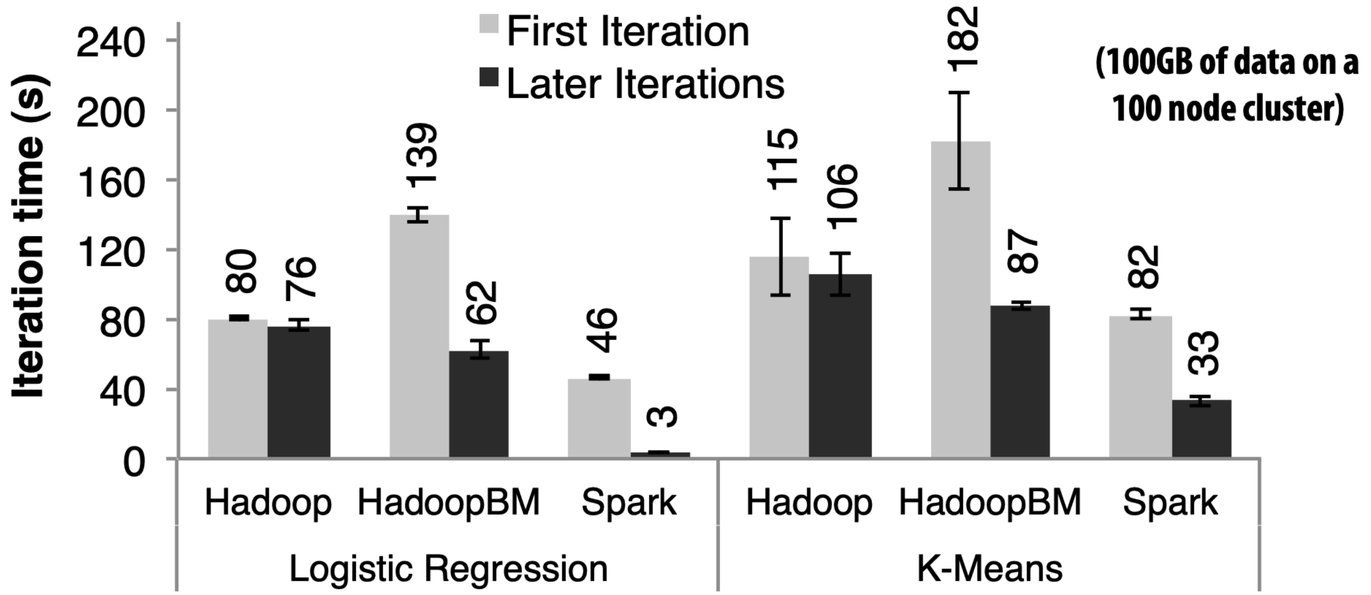
Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes



# Spark Performance



# Spark Improves MapReduce Over

- Easy for programmers because you express your computation by chaining atomic operators
- Much fewer I/O -> very improved AI

# Spark Cons?

- Debuggability
- Bulky
  - Map-reduce is not bulky as it works well if you only have one worker. That’s why now every PL has a “map” function

# Caution: "scale out" is not the entire story

- Distributed systems designed for cloud execution address many difficult challenges, and have been instrumental in the explosion of "big-data" computing and large-scale analytics
  - Scale-out parallelism to many machines
  - Resiliency in the face of failures
  - Complexity of managing clusters of machines
- But scale out is not the whole story:

**20 Iterations of Page Rank**

| scalable system | cores | twitter | uk-2007-05 |
|---|---|---|---|
| GraphChi [10] | 2 | 3160s | 6972s |
| Stratosphere [6] | 16 | 2250s | - |
| X-Stream [17] | 16 | 1488s | - |
| Spark [8] | 128 | 857s | 1759s |
| Giraph [8] | 128 | 596s | 1235s |
| GraphLab [8] | 128 | 249s | 833s |
| GraphX [8] | 128 | 419s | 462s |
| Single thread (SSD) | 1 | 300s | 651s |
| Single thread (RAM) | 1 | 275s | - |
| Vertex order (SSD) | 1 | 300s | 651s |
| Vertex order (RAM) | 1 | 275s | - |
| Hilbert order (SSD) | 1 | 242s | 256s |
| Hilbert order (RAM) | 1 | 110s | - |

| name | twitter_rv [11] | uk-2007-05 [4] |
|---|---|---|
| nodes | 41,652,230 | 105,896,555 |
| edges | 1,468,365,182 | 3,738,733,648 |
| size | 5.76GB | 14.72GB |

Further optimization of the baseline brought time down to 110s



["Scalability! At what COST?" McSherry et al. HotOS 20

# Modern Spark ecosystem

**Compelling feature: enables integration/composition of multiple domain-specific frameworks (since all collections implemented under the hood with RDDs and scheduled using Spark scheduler)**


```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

**Interleave computation and database query**
**Can apply transformations to RDDs produced by SQL queries**


**Machine learning library build on top of Spark abstractions.**

```
points = spark.textFile("hdfs://...")
              .map(parsePoint)

model = KMeans.train(points, k=10)
```


**GraphLab-like library built on top of Spark abstractions.**

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

# Story time: Spark and Databricks

- Initially just an open-source project by a few students
- The community grows because of advantages over Hadoop and Map-reduce
- Students were about to graduate and could not commit time to those projects, what's next?
- "We asked Hortonworks if they wanted to take over Spark…They were not willing… We started Databricks."
- Hortonworks -> later merged with Cloudera at 2019

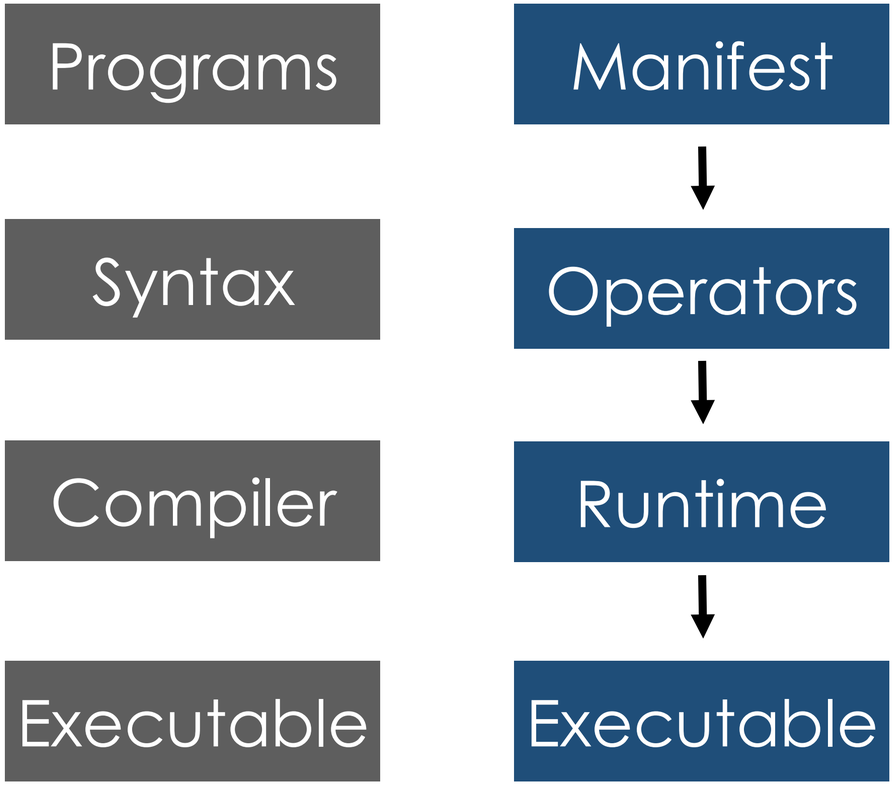
# Spark and Databricks

- Cloudera: data platform company, founded by Hadoop authors
  - Used to be a unicorn / high-profile / high-tech company
  - Was beat hard by Databricks / Snowflake
  - Went to public 2017, stock price keeps declining…, merged with Hortonworks in 2018, went to private in 2021 after being acquired by investment companies.
- Databricks: 7 cofounders, Initial CEO is Prof. Ion Stoica.
  - They tried to sell Spark but were unsuccessful
  - Switched to Ali Ghodsi: Iranian-Swedish, visitor to UC Berkeley, no US-born nor US-educated

# Spark and Databricks

- Databricks struggled for quite a few years
  - Raised up to Series I (Seed, A, B, C, D, E, F, G, H, I)
  - Almost failed during 2018 – 2020
  - Data warehousing and OLAP gradually become a business, why?
    - Competitors all failed
    - Customer Education
    - Data indeed bigger and bigger
  - Intended to go public in 2022, but hit covid
  - Valued at 43B today (is there any bubble? No one knows)
  - Create 3 billionaires
  - Competitions with Snowflake are intense

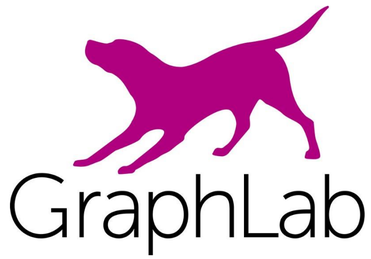
# After Spark:
# All Modern Data/ML Systems follow a similar architecture




A fixed set of operators

A trusted runtime with a small set of pre-loaded implementations

# After Spark: Many new systems


Naiad