

# **CS246: Mining Massive Data Sets**

## **Intro, MapReduce & Spark**

CS246: Mining Massive Datasets

Jure Leskovec, Stanford University

<http://cs246.stanford.edu>





# Data contains value and knowledge

# Data Mining

- But to extract the knowledge data needs to be
  - Stored (systems)
  - Managed (databases)
  - And ANALYZED ← this class

**Data Mining ≈ Big Data ≈  
Predictive Analytics ≈  
Data Science ≈ Machine Learning**

# What This Course Is About

- *Data mining* = extraction of actionable information from (usually) very large datasets, is the subject of extreme hype, fear, and interest
- It's not all about machine learning
- But some of it is
- Emphasis in CS246 on algorithms that **scale**
  - Parallelization often essential

# Data Mining Methods

## ■ Descriptive methods

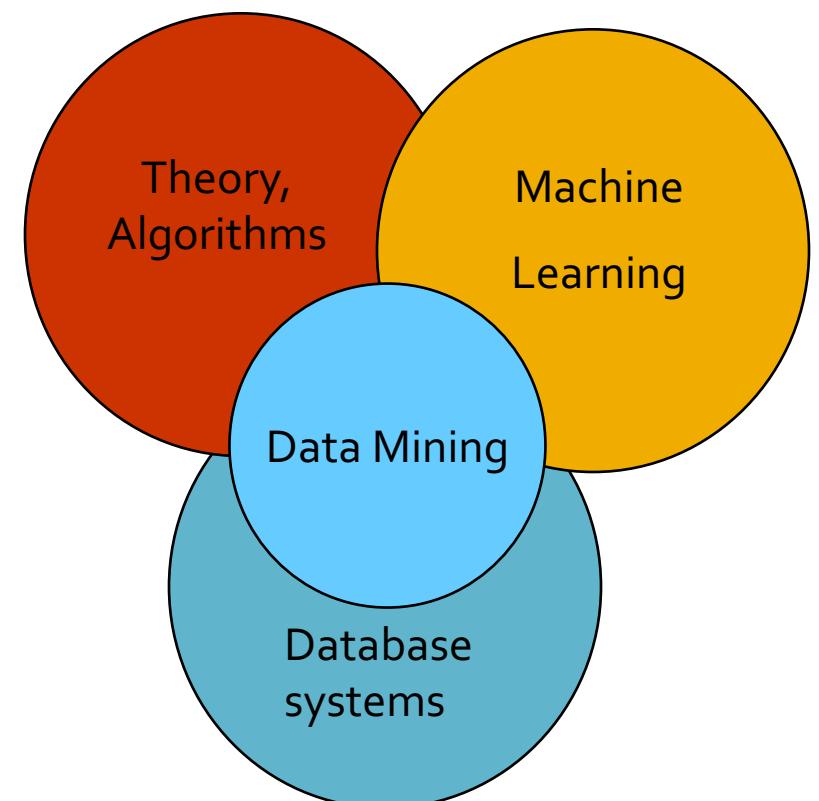
- Find human-interpretable patterns that describe the data
  - **Example:** Clustering

## ■ Predictive methods

- Use some variables to predict unknown or future values of other variables
  - **Example:** Recommender systems

# This Class: CS246

- This combines best of machine learning, statistics, artificial intelligence, databases but more stress on
  - Scalability (big data)
  - Algorithms
  - Computing architectures
  - Automation for handling large data



# What will we learn?

- We will learn to **mine different types of data**:
  - Data is high dimensional
  - Data is a graph
  - Data is infinite/never-ending
  - Data is labeled
- We will learn to **use different models of computation**:
  - MapReduce
  - Streams and online algorithms
  - Single machine in-memory

# What will we learn?

- We will learn to **solve real-world problems**:
  - Recommender systems
  - Market Basket Analysis
  - Spam detection
  - Duplicate document detection
- We will learn **various “tools”**:
  - Linear algebra (SVD, Rec. Sys., Communities)
  - Optimization (stochastic gradient descent)
  - Dynamic programming (frequent itemsets)
  - Hashing (LSH, Bloom filters)

# How the Class Fits Together

## High dim. data

Locality  
sensitive  
hashing

Clustering

Dimensional  
ity  
reduction

## Graph data

PageRank,  
SimRank

Network  
Analysis

Spam  
Detection

## Infinite data

Filtering  
data  
streams

Web  
advertising

Queries on  
streams

## Machine learning

SVM

Decision  
Trees

Perceptron,  
kNN

## Apps

Recommen  
der systems

Association  
Rules

Duplicate  
document  
detection



# How do you want that data?

# Course Logistics

# Course Staff



Stefanie Anna



Jayadev Bhaskaran



Jerry Jiang



Baige Liu



Lantao Mei



Eric Redondo



Shuyang Shi



Ansh Shukla



Hongtao Sun



Yang Wang



Shuiyi Yin



Wensi Yin

# CS246 Course Staff

- **Office hours:**

- See course website <http://cs246.stanford.edu> for TA office hours
  - **We start Office Hours next week**
- **Jure**: Tuesdays 9-10am, Gates 418
- **Michele**: Thursdays 5-7pm, Gates 452
- For SCPD students we will use **Google Hangout**
  - **Link posted on Piazza**

# Resources

- **Course website:** <http://cs246.stanford.edu>
  - Lecture slides (at least 30min before the lecture)
  - Homeworks, solutions, readings posted on Piazza
- **Class textbook:** **Mining of Massive Datasets** by A. Rajaraman, J. Ullman, and J. Leskovec
  - Sold by Cambridge Uni. Press but available for free at <http://mmds.org>
- **MOOC:** [www.youtube.com /channel/UC\\_Oao2FYkLAUIUVkBfze4jg/videos](https://www.youtube.com/channel/UC_Oao2FYkLAUIUVkBfze4jg/videos)

# Special Tutorials

- **Spark tutorial and help session:**
  - Thursday, January 10, 4:30-5:50 PM, Location TBD
- **Review of basic probability and proof techniques**
  - Tuesday, January 15, 4:30-5:50 PM, Location TBD
- **Review of linear algebra:**
  - Thursday, January 17, 4:30-5:50 PM, Location TBD

# Logistics: Communication

- **Piazza Q&A website:**
  - <https://piazza.com/class/winter2019/cs246>
    - Use Piazza for all questions and public communication
      - Search the forum before asking a question
      - Please tag your posts and please no one-liners
- **For e-mailing course staff always use:**
  - [cs246-win1819-staff@lists.stanford.edu](mailto:cs246-win1819-staff@lists.stanford.edu)
- **We will post course announcements to Piazza (make sure you check it regularly)**

Auditors are welcome to sit-in & audit the class

# Work for the Course: Homeworks

## ■ 4 longer homeworks: 40%

- Four major assignments, involving programming, proofs, algorithm development.
- “Warmup” assignment, called “HW0,” to introduce everyone to Spark has just been posted
- Assignments take lots of time (+20h). **Start early!!**

## ■ How to submit?

### ■ Homework write-up:

- Submit via [Gradescope](#)
- Course code: MNPBKE

### ■ Everyone uploads code:

- Put all the code for 1 question into 1 file and submit at: <http://snap.stanford.edu/submit/>

# Homework Calendar

## ■ Homework schedule:

Date (23:59 PT)	Out	In
Today	HW0	
01/10, Thu	HW1	
01/24, Thu	HW2	HW0, HW1
02/07, Thu	HW3	HW2
02/21, Thu	HW4	HW3
03/07, Thu		HW4

- Two late periods for HWs for the quarter:
  - Late period expires on the following Monday 23:59 PT
  - Can use max 1 late period per HW

# Work for the Course: Gradiance

- **Short weekly Gradiance quizzes: 20%**
  - Quizzes are posted every **Tuesday**
  - Due 9 days later on **Thursday 23:59 PT. No late days!**
    - First quiz has already been posted
  - To register on Gradiance please use **SUNetID** or **<legal first name>\_<legal last name>** for username
    - We have to be able to match your Gradiance ID and SUNetId
    - Sign up at [www.gradiance.com/services](http://www.gradiance.com/services) and use code **3DBCAD12**
  - As many submissions as you like, your score is based on the **most recent** submission
  - After the due date, you can see the solutions to all problems by looking at one of your submissions, so you **must** try at least once

# Work for the Course: Gradiance

- You should work each of the implied problems before answering the multiple-choice questions.
- That way, if you have to repeat the work to get 100%, you will have available what you need for the questions you have solved correctly, and the process can go quickly.
- **Note:** There is a 10-minute delay between submissions, to protect against people who randomly fire off guesses.

# Work for the Course: Final Exam

- **Final exam: 40%**
  - **Tuesday, March 19** 3:30pm-6:30pm
  - There is no alternative final, but if you truly have a conflict, we can arrange for you to take the exam immediately after the regular final
- **Extra credit:** Up to 1% of your grade
  - For participating in Piazza discussions
    - Especially valuable are answers to questions posed by other students
  - Reporting bugs in course materials

# Prerequisites

- **Programming:** Python or Java
- **Basic Algorithms:** CS161 is surely sufficient
- **Probability:** e.g., CS109 or Stats116
  - There will be a review session and a review doc is linked from the class home page
- **Linear algebra:**
  - Another review doc + review session is available
- **Multivariable calculus**
- **Database systems (SQL, relational algebra):**
  - CS145 is sufficient but not necessary

# What If I Don't Know All This Stuff?

- **Each of the topics listed is important for a small part of the course:**
  - If you are missing an item of background, you could consider just-in-time learning of the needed material
- **The exception is programming:**
  - To do well in this course, you really need to be comfortable with writing code in Python or Java

# Honor Code

- We'll follow the standard CS Dept. approach:  
You can get help, but you **MUST** acknowledge  
the help on the work you hand in
- Failure to acknowledge your sources is a  
*violation of the Honor Code*
- We use MOSS to check the originality of your  
code

# Honor Code – (2)

- **You can talk to others about the algorithm(s) to be used to solve a homework problem;**
  - As long as you then mention their name(s) on the work you submit
- **You should not use code of others or be looking at code of others when you write your own:**
  - You can talk to people but have to write your own solution/code
  - If you fail to mention your sources, MOSS will catch you, and you will be charged with an HC violation

# CS246H: SparkLabs

- **CS246H covers practical aspects of Spark and other distributed computing architectures**
  - HDFS, Combiners, Partitioners, Hive, Pig, Hbase, ...
  - 1 unit course, optional homeworks
- **CS246H runs (somewhat) parallel to CS246**
  - CS246 discusses theory and algorithms
  - CS246H tells you how to implement them
- **Instructor:** Daniel Templeton (Cloudera)
  - CS246H lectures are recorded (available via SCPD)

# What's after the class

- **CS341: Project in Data Mining (Spring 2019)**
  - Research project on big data
  - Groups of 3 students
  - We provide interesting data, computing resources (Amazon EC2) and mentoring
- **My group has RA positions open:**
  - See <http://snap.stanford.edu/apply/>

# Final Thoughts

- **CS246 is fast paced!**
  - Requires programming maturity
  - Strong math skills
    - SCPD students tend to be rusty on math/theory
- **Course time commitment:**
  - Homeworks take +20h
  - Gradiance quizzes take about 1-2h
- Form study groups
- **It's going to be fun and hard work.** ☺

# 4 To-do items

- **4 to-do items for you:**
  - **Register to Piazza**
  - **Register to Gradescope**
  - **Register to Gradiance and complete the first quiz**
    - **Use your SUNet ID to register!** (so we can match grading records)
    - Complete the first quiz (we will announce when it is posted)
  - **Complete HW0**
    - HW0 should take you about 1-2 hours to complete  
(Note this is a “toy” homework to get you started. Real homeworks will be much more challenging and longer)
- **Additional details/instructions at**  
**<http://cs246.stanford.edu>**

# **Distributed Computing for Data Mining**



# Large-scale Computing

- Large-scale computing for data mining problems on commodity hardware
- Challenges:
  - How do you distribute computation?
  - How can we make it easy to write distributed programs?
  - Machines fail:
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to lose 1/day
    - With 1M machines 1,000 machines fail every day!

# An Idea and a Solution

- **Issue:**

**Copying data over a network takes time**

- **Idea:**

- Bring computation to data
- Store files multiple times for reliability

- **Spark/Hadoop address these problems**

- **Storage Infrastructure – File system**

- Google: GFS. Hadoop: HDFS

- **Programming model**

- MapReduce
  - Spark

# Storage Infrastructure

- **Problem:**

- If nodes fail, how to store data persistently?

- **Answer:**

- **Distributed File System**

- Provides global file namespace

- **Typical usage pattern:**

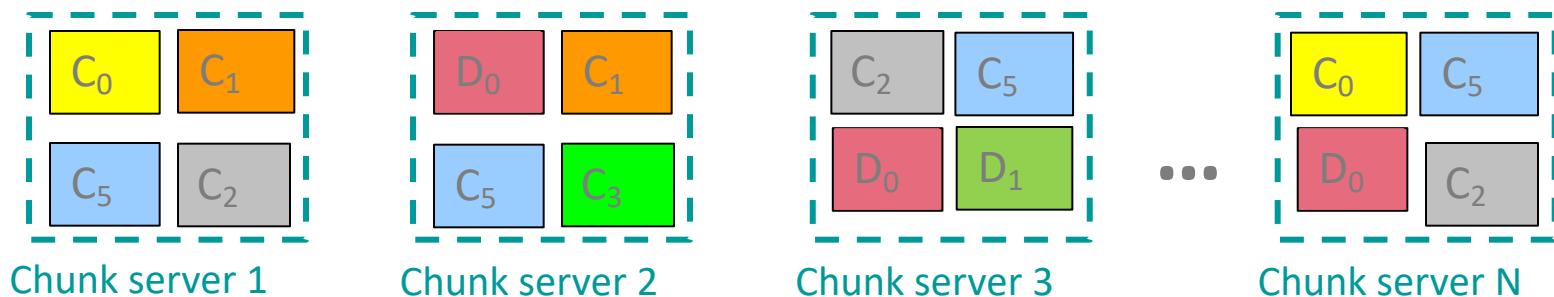
- Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

# Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

# Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

# Programming Model

- MapReduce is a **style of programming** designed for:
  1. Easy parallel programming
  2. Invisible management of hardware and software failures
  3. Easy management of very-large-scale data
- It has several **implementations**, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called “MapReduce”

# MapReduce: Overview

## 3 steps of MapReduce

### ■ **Map:**

- Apply a user-written *Map function* to each input element
  - *Mapper* applies the Map function to a single element
  - Many mappers grouped in a *Map task* (the unit of parallelism)
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.

### ■ **Group by key:** Sort and shuffle

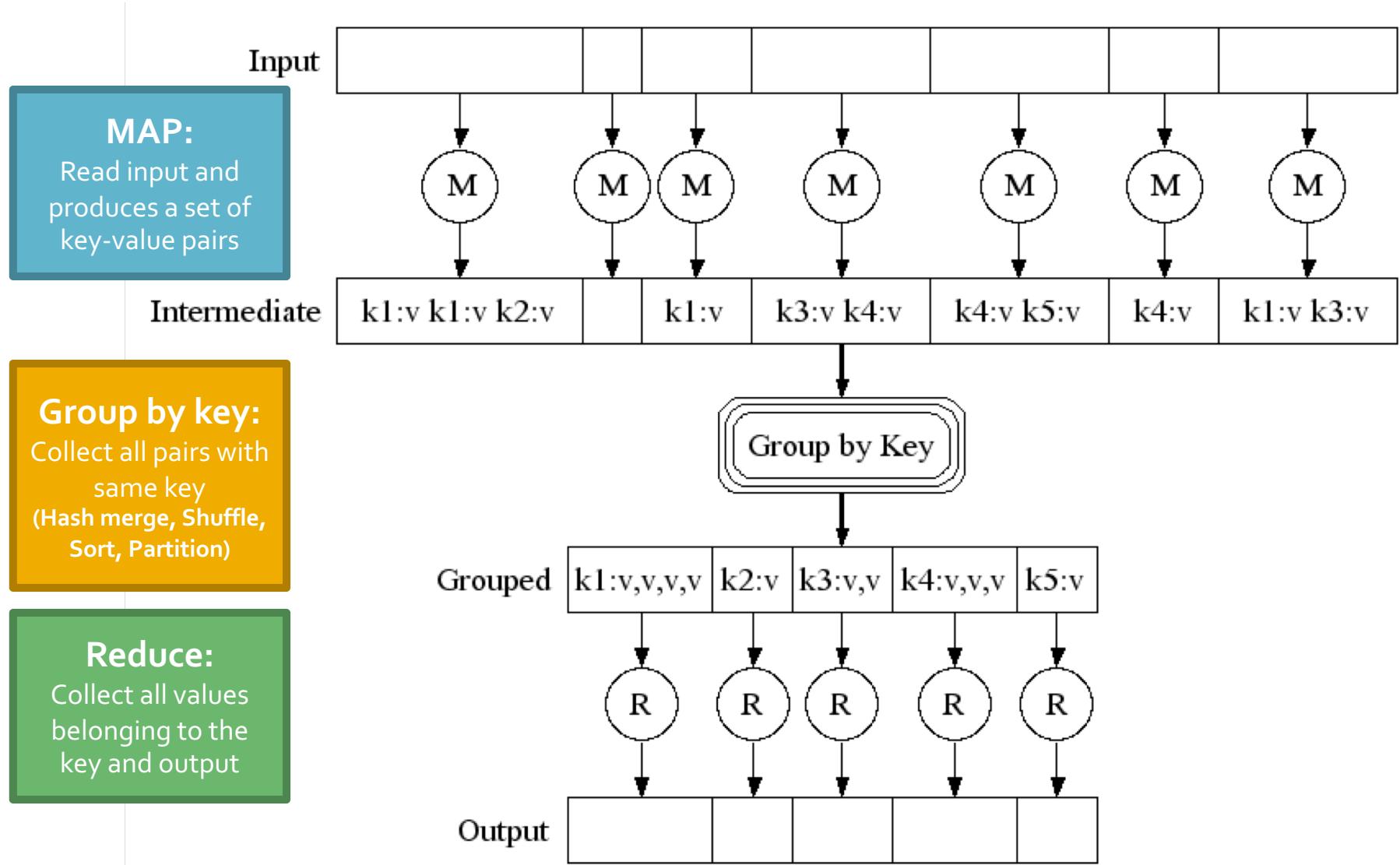
- System sorts all the key-value pairs by key, and outputs key-(list of values) pairs

### ■ **Reduce:**

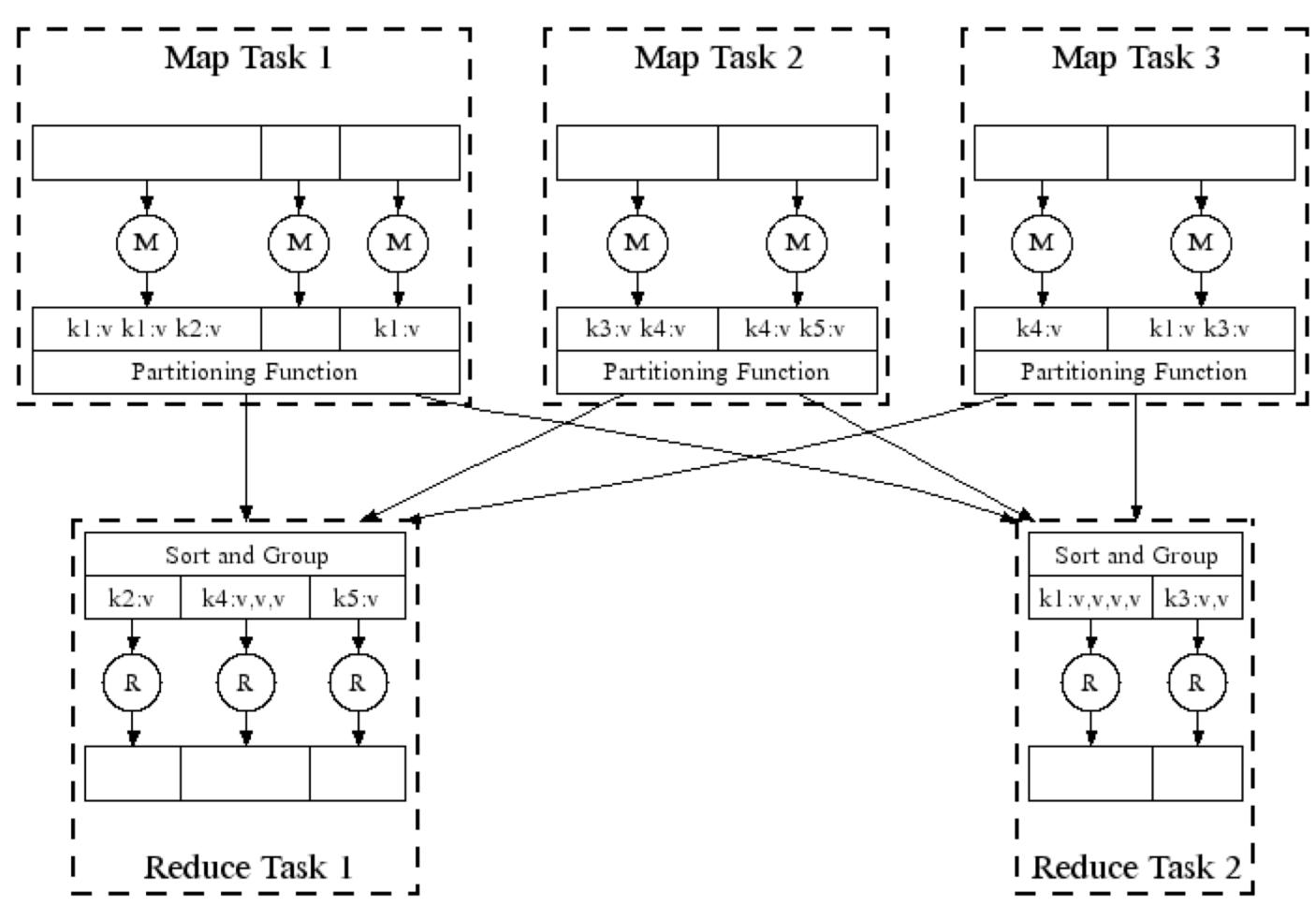
- User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, **Map** and **Reduce** change to fit the problem

# Map-Reduce: A diagram

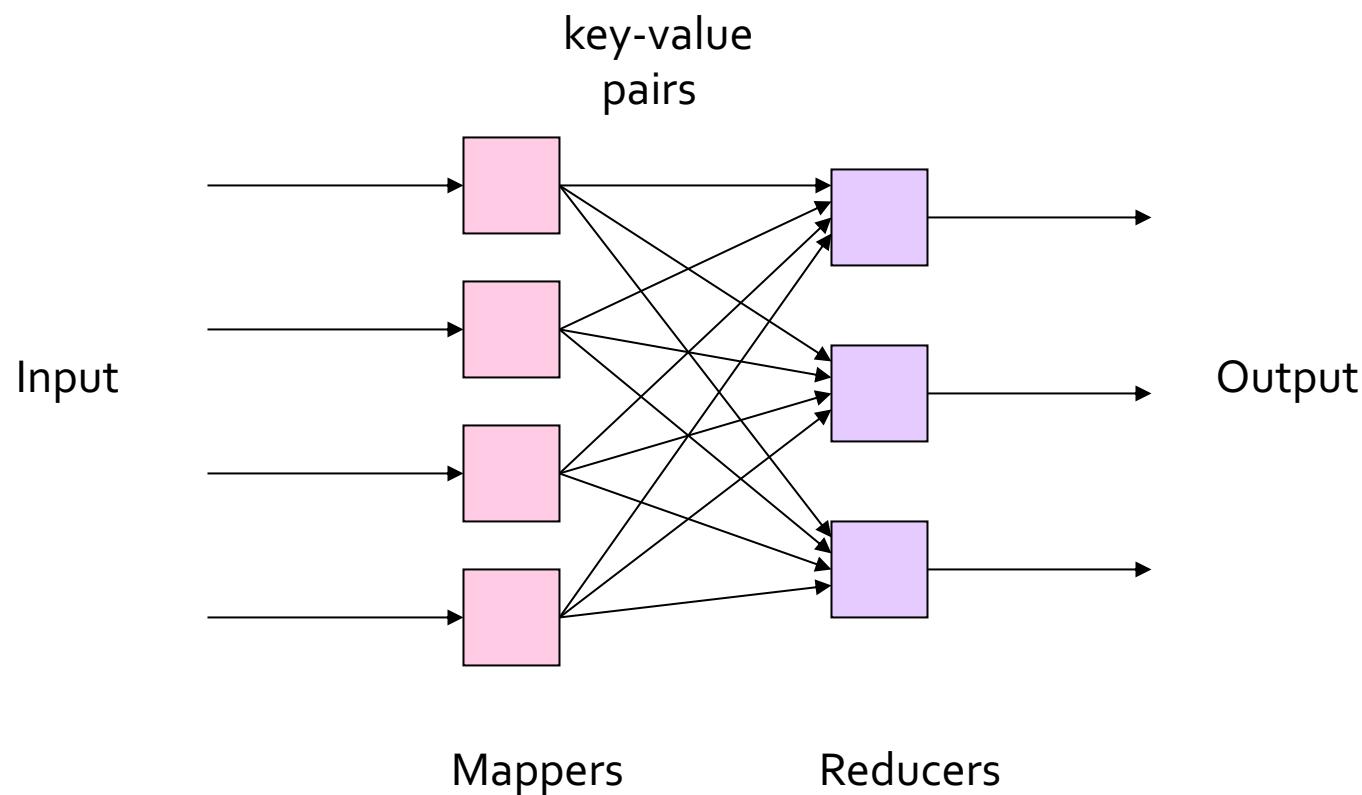


# Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

# MapReduce Pattern

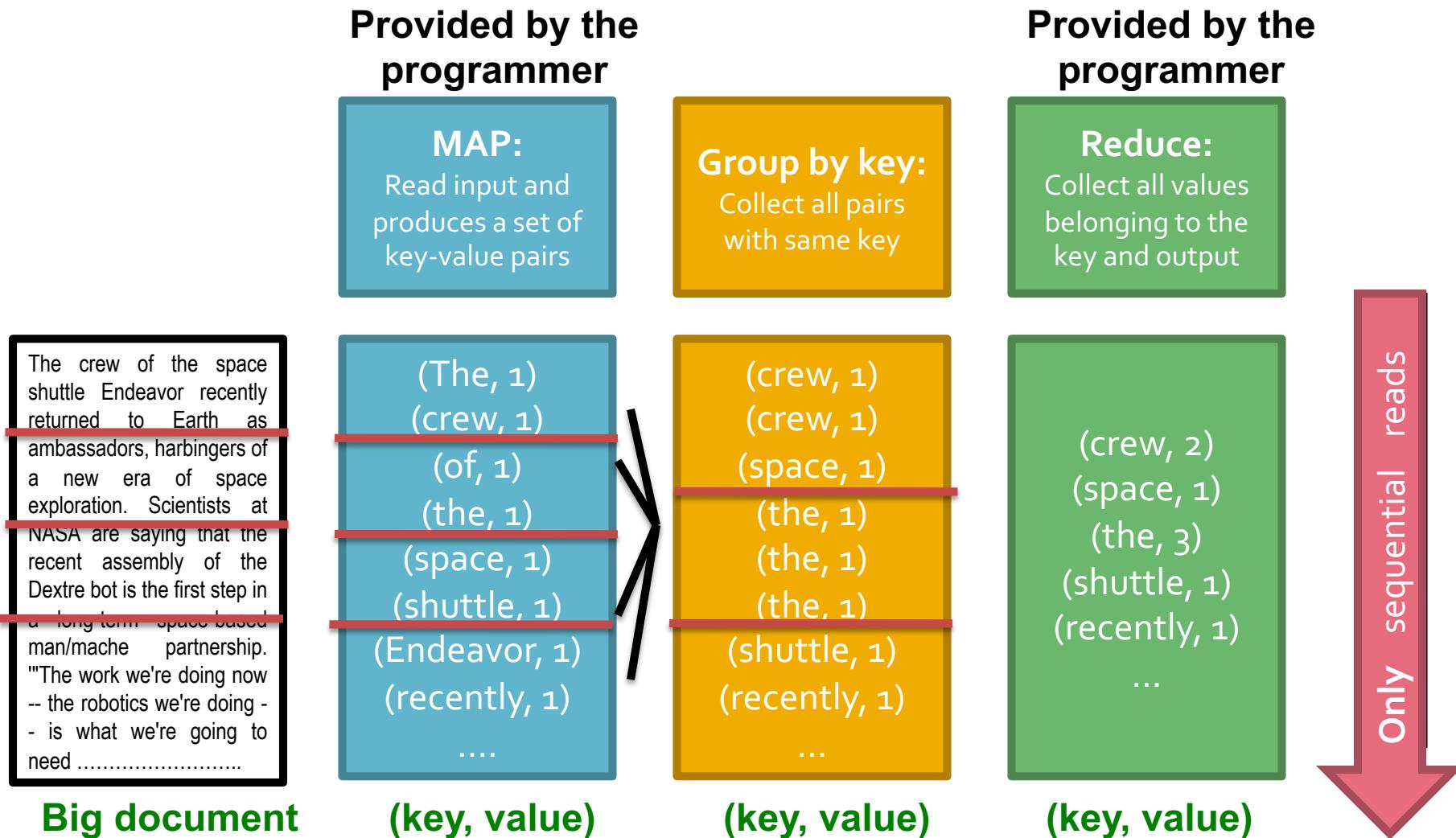


# Example: Word Counting

## Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Many applications of this:**
  - Analyze web server logs to find popular URLs
  - Statistical machine translation:
    - Need to count number of times every 5-word sequence occurs in a large corpus of documents

# MapReduce: Word Counting



# Word Count Using MapReduce

```
map(key, value):
# key: document name; value: text of the document
for each word w in value:
    emit(w, 1)

reduce(key, values):
# key: a word; value: an iterator over counts
result = 0
for each count v in values:
    result += v
emit(key, result)
```

# MapReduce: Environment

**MapReduce environment takes care of:**

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
  - In practice this is the bottleneck
- Handling machine **failures**
- Managing required inter-machine **communication**

# Dealing with Failures

## ■ Map worker failure

- Map tasks completed or in-progress at worker are reset to idle and rescheduled
- Reduce workers are notified when map task is rescheduled on another worker

## ■ Reduce worker failure

- Only in-progress tasks are reset to idle and the reduce task is restarted

# Spark

# Problems with MapReduce

- **Two major limitations of MapReduce:**
  - Difficulty of programming directly in MR
    - Many problems aren't easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work
- **In short, MR doesn't compose well for large applications**
  - Many times one needs to chain multiple map-reduce steps

# Data-Flow Systems

- **MapReduce uses two “ranks” of tasks:**  
One for Map the second for Reduce
  - Data flows from the first rank to the second
- **Data-Flow Systems generalize this in two ways:**
  1. Allow any number of tasks/ranks
  2. Allow functions other than Map and Reduce
    - As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs

# Spark: Most Popular Data-Flow System

- Expressive computing system, not limited to the map-reduce model
- Additions to MapReduce model:
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce
- Compatible with Hadoop

# Spark: Overview

- Open source software (Apache Foundation)
- Supports **Java, Scala and Python**
- **Key construct/idea:** Resilient Distributed Dataset (RDD)
- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support

# Spark: RDD

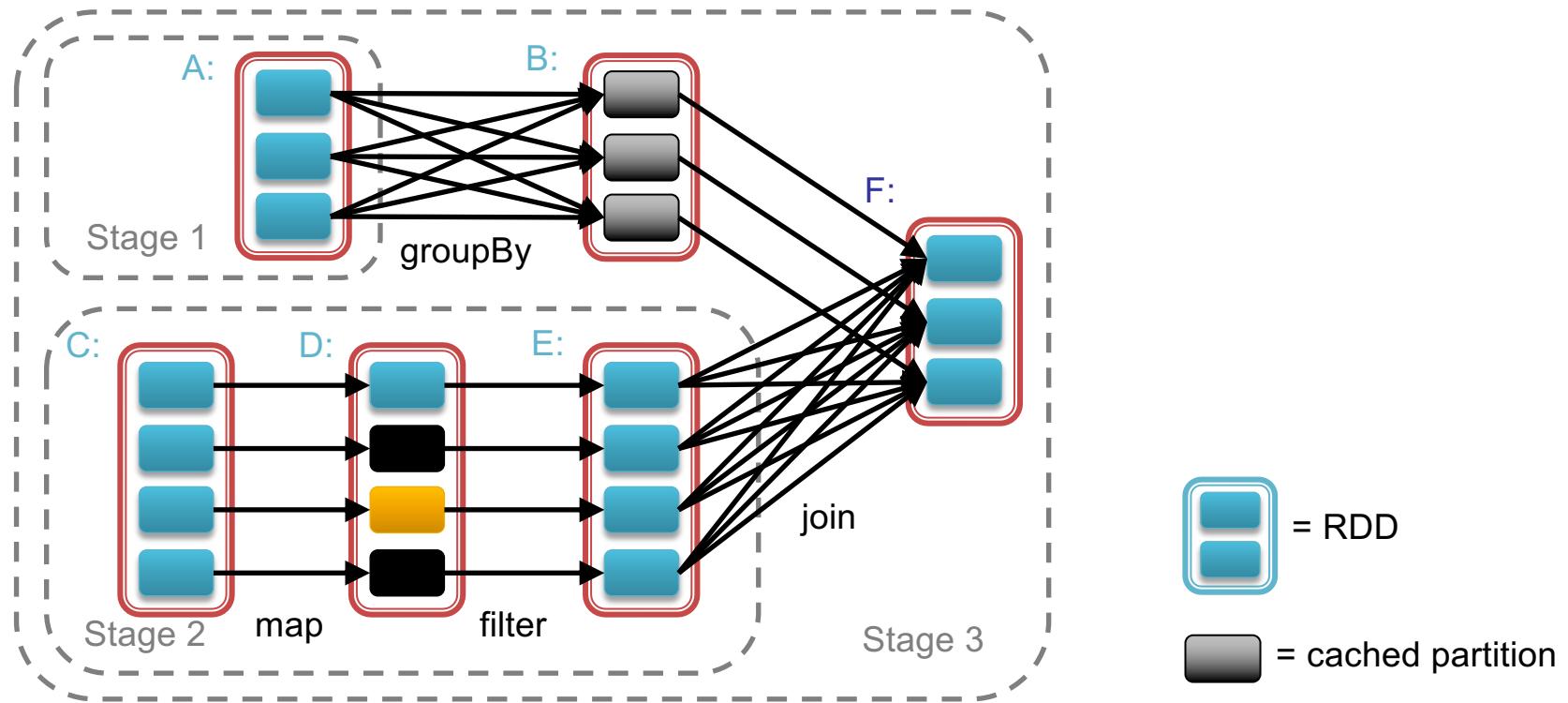
## Key concept *Resilient Distributed Dataset* (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible
- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset

# Spark RDD Operations

- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include *map*, *filter*, *join*, *union*, *intersection*, *distinct*
  - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
  - Actions include *count*, *collect*, *reduce*, *save*
  - Actions can be applied to RDDs; actions force calculations and return values

# Task Scheduler: General DAGs



- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles

# DataFrame & Dataset

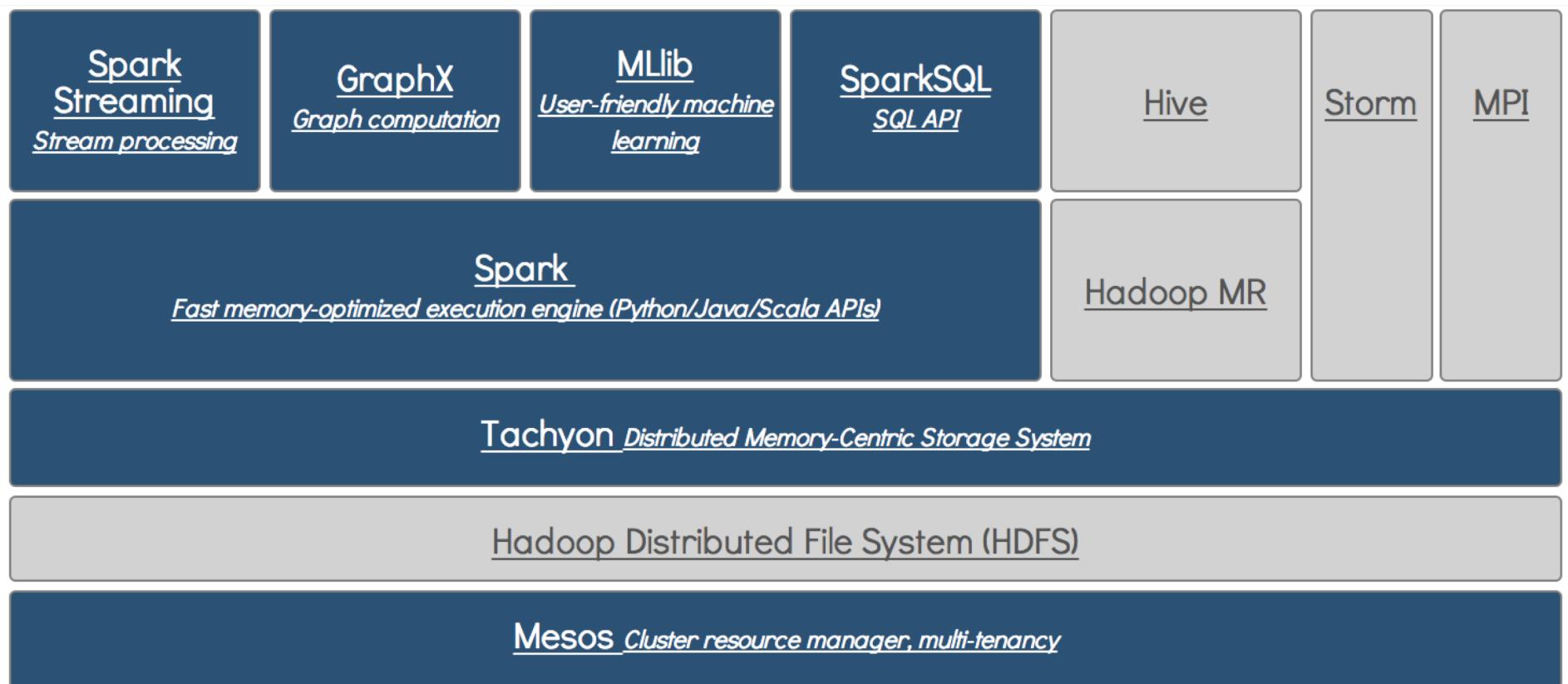
- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection)

Both built on Spark SQL engine. both can be converted back to an RDD

# Useful Libraries for Spark

- Spark SQL
- Spark Streaming – **stream processing of live datastreams**
- MLlib – **scalable machine learning**
- GraphX – **graph manipulation**
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge

# Data Analytics Software Stack



# Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to perform well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program (higher-level APIs)
- Data processing: Spark more general

# Problems Suited for MapReduce

# Example: Host size

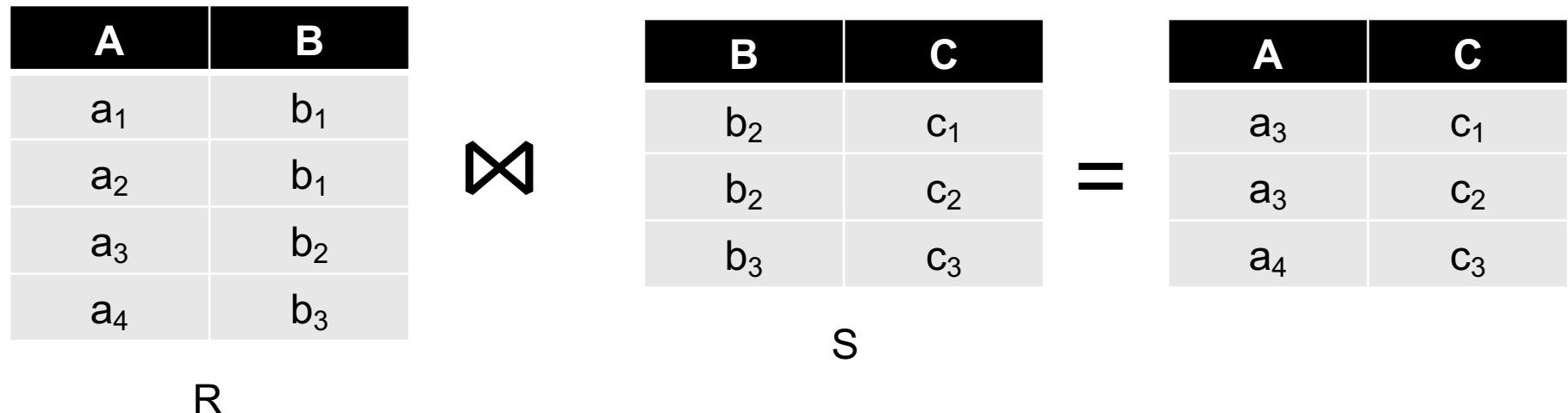
- Suppose we have a large web corpus
- Look at the metadata file
  - Lines of the form: (URL, size, date, ...)
- For each host, find the total number of bytes
  - That is, the sum of the page sizes for all URLs from that particular host
- Other examples:
  - Link analysis and graph processing
  - Machine Learning algorithms

# Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

# Example: Join By Map-Reduce

- Compute the natural join  $R(A,B) \bowtie S(B,C)$
- $R$  and  $S$  are each stored in files
- Tuples are pairs  $(a,b)$  or  $(b,c)$



# Map-Reduce Join

- Use a hash function  $h$  from B-values to  $1\dots k$
- A Map process turns:
  - Each input tuple  $R(a,b)$  into key-value pair  $(b,(a,R))$
  - Each input tuple  $S(b,c)$  into  $(b,(c,S))$
- Map processes send each key-value pair with key  $b$  to Reduce process  $h(b)$ 
  - Hadoop does this automatically; just tell it what  $k$  is.
- Each Reduce process matches all the pairs  $(b,(a,R))$  with all  $(b,(c,S))$  and outputs  $(a,b,c)$ .

# Problems NOT suitable for MapReduce

- **MapReduce is great for:**
  - Problems that require sequential data access
  - Large batch jobs (**not** interactive, real-time)
- **MapReduce is inefficient for problems where random (or irregular) access to data required:**
  - **Graphs**
  - Interdependent data
    - Machine learning
    - Comparisons of many pairs of items

# Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using
  1. *Communication cost* = total I/O of all processes
  2. *Elapsed communication cost* = max of I/O along any path
  3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful  
(adding more machines is always an option)

# Example: Cost Measures

- For a map-reduce algorithm:
  - **Communication cost** = input file size +  $2 \times$  (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

# What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

# Cost of Map-Reduce Join

- **Total communication cost**  
 $= O(|R| + |S| + |R \bowtie S|)$
- **Elapsed communication cost** =  $O(s)$ 
  - We're going to pick  $k$  and the number of Map processes so that the I/O limit  $s$  is respected
  - We put a limit  $s$  on the amount of input or output that any one process can have.  $s$  could be:
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost

# CS246: Mining massive datasets

Grab a handout  
at the back of  
the room