

# CS246 Introduction

Essence of the Course

Administrivia

MapReduce

Other Parallel-Computing Systems

Jeffrey D. Ullman  
Stanford University



# 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 a lot of it is.
- Emphasis in CS246 on algorithms that **scale**.
  - Parallelization often essential.

# Modeling

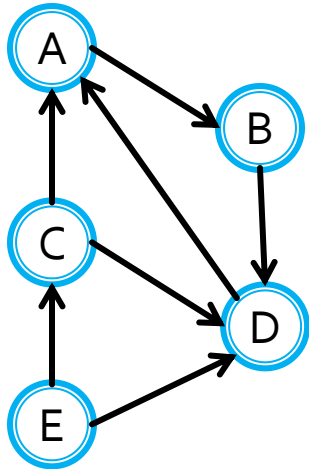
- Often, especially for ML-type algorithms, the result is a *model* = a simple representation of the data, typically used for prediction.
- **Example:** PageRank is a number Google assigns to each Web page, representing the “importance” of the page.
  - Calculated from the link structure of the Web.
  - Summarizes in one number, all the links leading to one page.
  - Used to help decide which pages Google shows you.

# Example: College Football Ranks

- **Problem**: Most teams don't play each other, and there are weaker and stronger leagues.
  - So which teams are the best? Who would win a game between teams that never played each other?
- **Model** (old style): a list of the teams from best to worst.
- **Algorithm** (old style):
  1. *Cost* of a list = # of teams out of order.
  2. Start with some list (e.g., order by % won).
  3. Make *incremental* cost improvements until none possible (e.g., swap adjacent teams)

# Example: Swap Adjacent Only

Tail = winner;  
head = loser.



A  
B  
C  
D  
E

Cost  
= 4

A  
B  
C  
E  
D

Cost  
= 3

A  
B  
E  
C  
D

Cost  
= 2

Note: best ordering , ECABD,  
with a cost of 1, is unreachable  
using only adjacent swaps.

# A More Modern Approach

- Model has one variable for each of  $n$  teams (“*power rating*”).
  - Variable  $x$  for team  $X$ .
- Cost function is an expression of variables.
- **Example:** *hinge loss* function  $h(\{x,y\}) =$ 
  - 0 if teams  $X$  and  $Y$  never played or  $X$  won and  $x > y$  or  $Y$  won and  $y > x$ .
  - $y - x$  if  $X$  won and  $y \geq x$ , and  $x - y$  if  $Y$  won and  $x \geq y$ .
- Cost function =  $\sum_{\{x,y\}} h(\{x,y\})$  – terms to prevent all variables from being equal, e.g.,  $c(\sum_{\{x,y\}} |x - y|)$ .

# Solution

- Start with some solution, e.g., all variables = 1.
- Find *gradient* of the cost for one or more variables (= derivative of cost function with respect to that variable).
- Make changes to variables in the direction of the gradient that lower cost, and repeat until improvements are “small.”

# Rules Versus Models

- In many applications, all we want is an algorithm that will say “yes” or “no.”
- **Example:** a model for email spam based on weighted occurrences of words or phrases.
  - Would give high weight to words like “Viagra” or phrases like “Nigerian prince.”
- **Problem:** when the weights are in favor of spam, there is no obvious reason why it is spam.
- Sometimes, no one cares; other times understanding is vital.



# Rules – (2)

- Rules like “Nigerian prince” -> spam are understandable and actionable.
- But the downside is that **every** email with that phrase will be considered spam.
- Next lecture will talk about these **association rules**, and how they are used in managing (brick and mortar) stores, where understanding the meaning of a rule is essential.

# Administrivia

**Prerequisites**

**Requirements**

**Staff**

**Resources**

**Honor Code**

**Lateness Policy**

# Prerequisites

- Basic Algorithms.
  - CS161 is surely sufficient, but may be more than needed.
- Probability, e.g., CS109 or Stat116.
  - 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.
- Programming (Java).
- Database systems (SQL, relational algebra).
  - CS145 is sufficient by not necessary.

# Requirements: Final Exam

- Final Exam (40%). The exam will be held Weds., March 16, 8:30AM.
  - Place TBD.
  - Note: the on-line schedule is misaligned, and 8:30AM is my interpretation – more later if needed.
  - 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.

# Requirements: Gradiance

- Gradiance on-line homework (20%).
  - This automated system lets you try questions as many times as you like, and the goal is that everyone will work until they get all problems right.
  - Sign up at [www.gradiance.com/services](http://www.gradiance.com/services) and enter class 62B99A55
  - **Note:** your score is based on the most recent submission, not the max.
  - **Note:** 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.

# Gradiance – (2)

- 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.

# Requirements: Ordinary HW

- Written Homework (40%).
  - Four major assignments, involving programming, proofs, algorithm development.
  - Preceded by a “warmup” assignment, called “tutorial,” to introduce everyone to Hadoop.
  - Submission of work will be via gradescope.com and you need to use class code 92B7E9 for CS246.

# Staff

- There are 12 great TA's this year! Tim Althoff (chief TA), Sameep Bagadia, Ivaylo Bahtchevanov, Duyun Chen, Nihit Desai, Shubham Gupta, Jeff Hwang, Himabindu Lakkaraju, Caroline Suen, Jacky Wang, Leon Yao, You Zhou.
- See class page [www.stanford.edu/class/cs246](http://www.stanford.edu/class/cs246) for schedule of office hours.
- [cs246-win1516-staff@lists.stanford.edu](mailto:cs246-win1516-staff@lists.stanford.edu) will contact TA's + Ullman.



# Resources

- **Class textbook:** Mining of Massive Datasets by Jure Leskovec, Anand Rajaraman, and U.
  - Sold by Cambridge Univ. Press, but available for free download at [www.mmds.org](http://www.mmds.org)
- On-line review notes for Probability, Proofs, and Linear Algebra are available through the class home page.
- Coursera MOOC at [class.coursera.org/mmds-003](http://class.coursera.org/mmds-003)
- Piazza discussion group (please join) at [piazza.com/stanford/winter2016/cs246](http://piazza.com/stanford/winter2016/cs246)

# 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.

# Late Homeworks

- There is no possibility of submitting Gradiance homework late.
- For the five written homeworks, you have two *late periods*.
  - If a homework is due on a Thursday (Tuesday), then the late period expires 11:59PM on the following Tuesday (Thursday).
  - You can only use one late period per homework.

# The MapReduce Environment

Distributed File Systems  
MapReduce and Hadoop  
Examples of MR Algorithms

Jeffrey D. Ullman  
Stanford University



# Distributed File Systems

Chunking

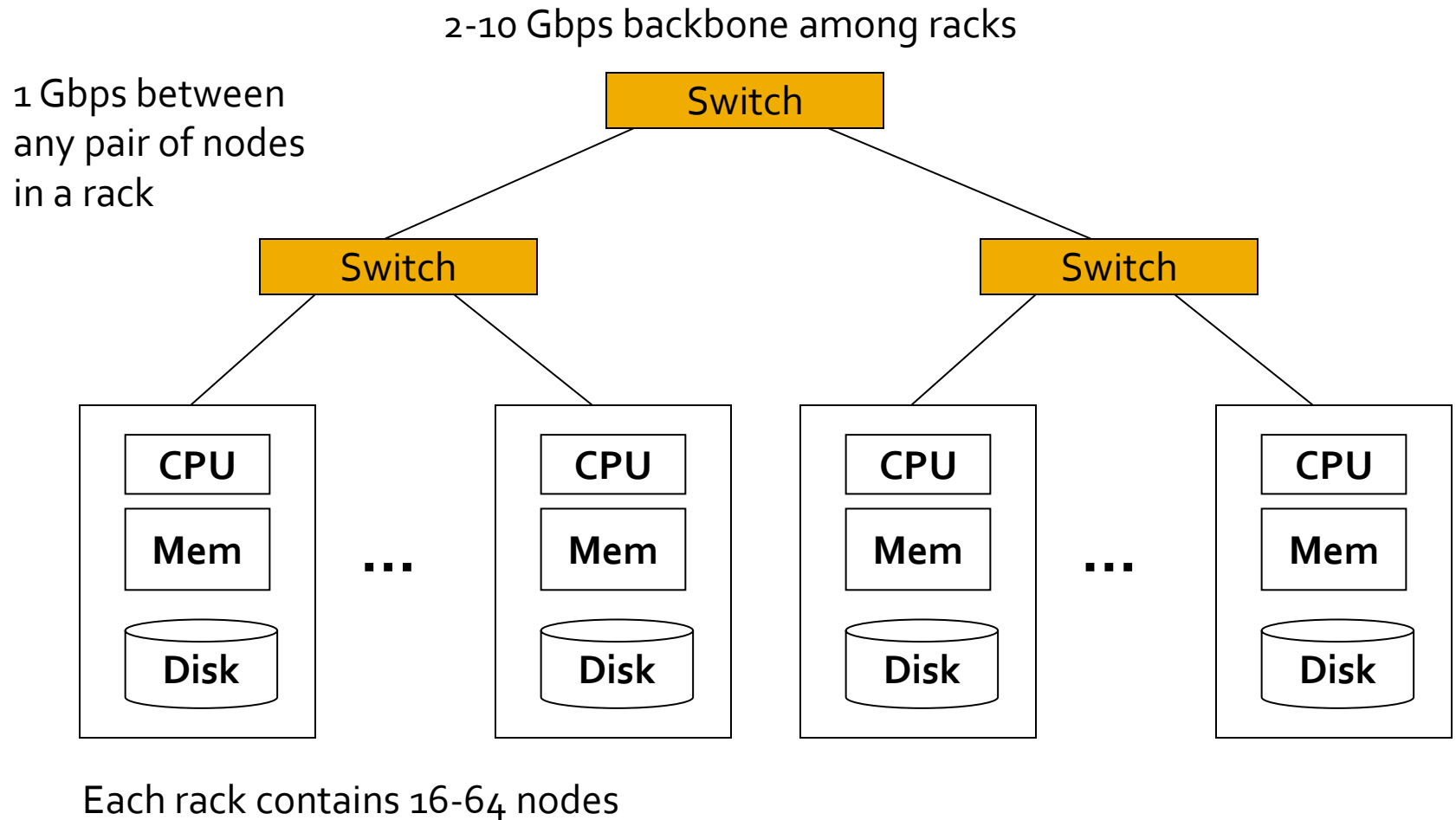
Replication

Distribution on Racks

# Commodity Clusters

- Standard architecture:
  1. Cluster of commodity Linux nodes (*compute nodes*).
    - = processor + main memory + disk.
  2. Gigabit Ethernet interconnect.

# Cluster Architecture



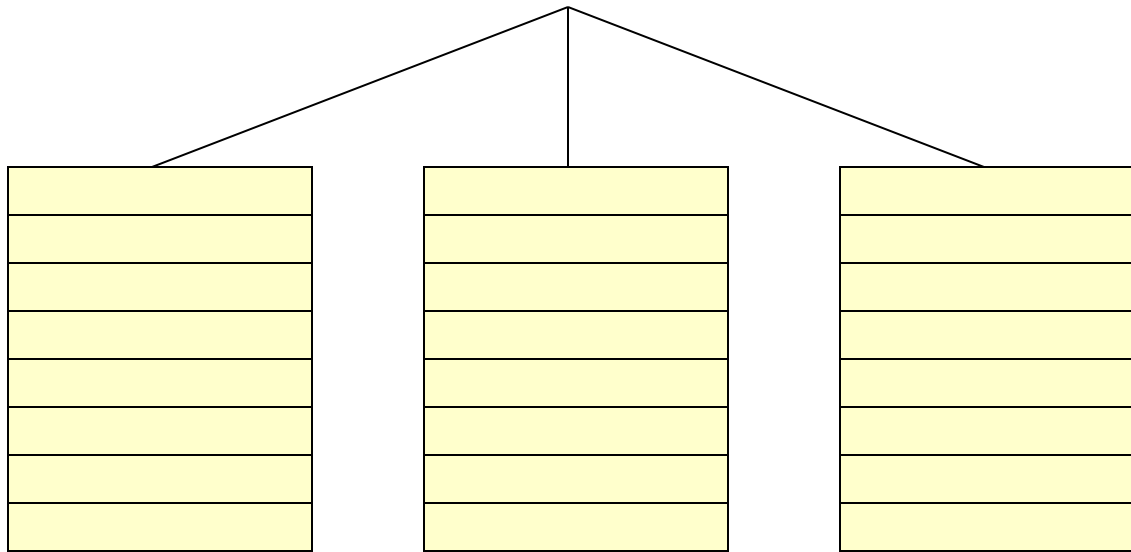
# Stable Storage

- **Problem:** if compute nodes can fail, how can we store data persistently?
- **Answer:** Distributed File System.
  - Provides global file namespace.
  - **Examples:** Google GFS, Colossus; Hadoop HDFS.



# Distributed File System

- Chunk Servers.
  - File is split into contiguous *chunks*, typically 64MB.
  - Each chunk replicated (usually 3x).
  - Try to keep replicas in different racks.
- Master Node for a file.
  - Stores metadata, location of all chunks.
  - Possibly replicated.

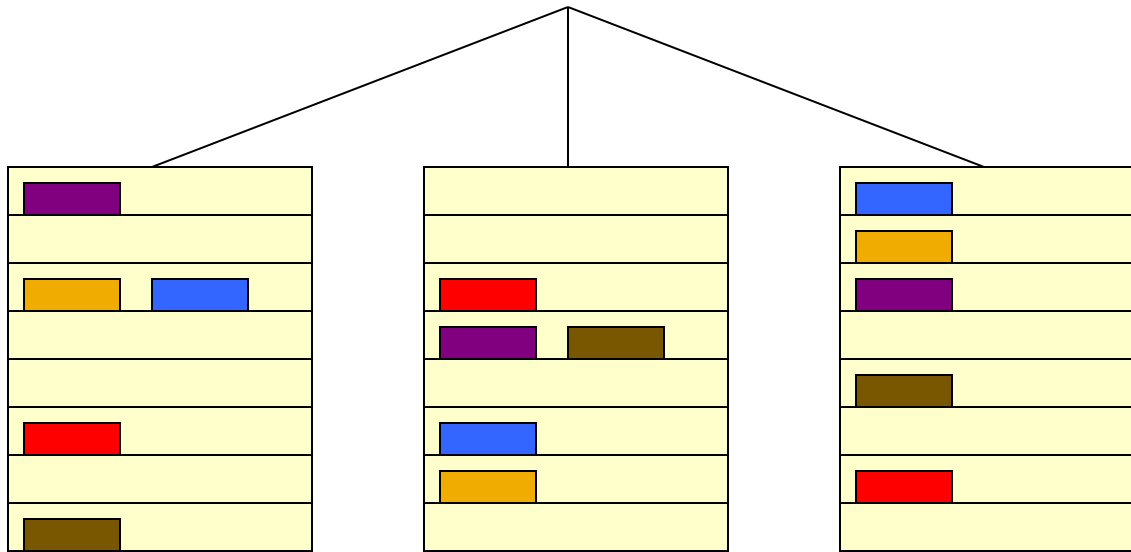


Racks of Compute Nodes

File



Chunks



3-way replication of  
files, with copies on  
different racks.

# Alternative: Erasure Coding

- More recent approach to stable file storage.
- Allows reconstruction of a lost chunk.
- **Advantage**: less redundancy for a given probability of loss.
- **Disadvantage**: no choice regarding where to obtain a given chunk.

# MapReduce

**A Quick Introduction**  
**Word-Count Example**  
**Fault-Tolerance**

# What Does MapReduce Give You?

1. Easy parallel programming.
2. Invisible management of hardware and software failures.
3. Easy management of very-large-scale data.

# MapReduce in a Nutshell

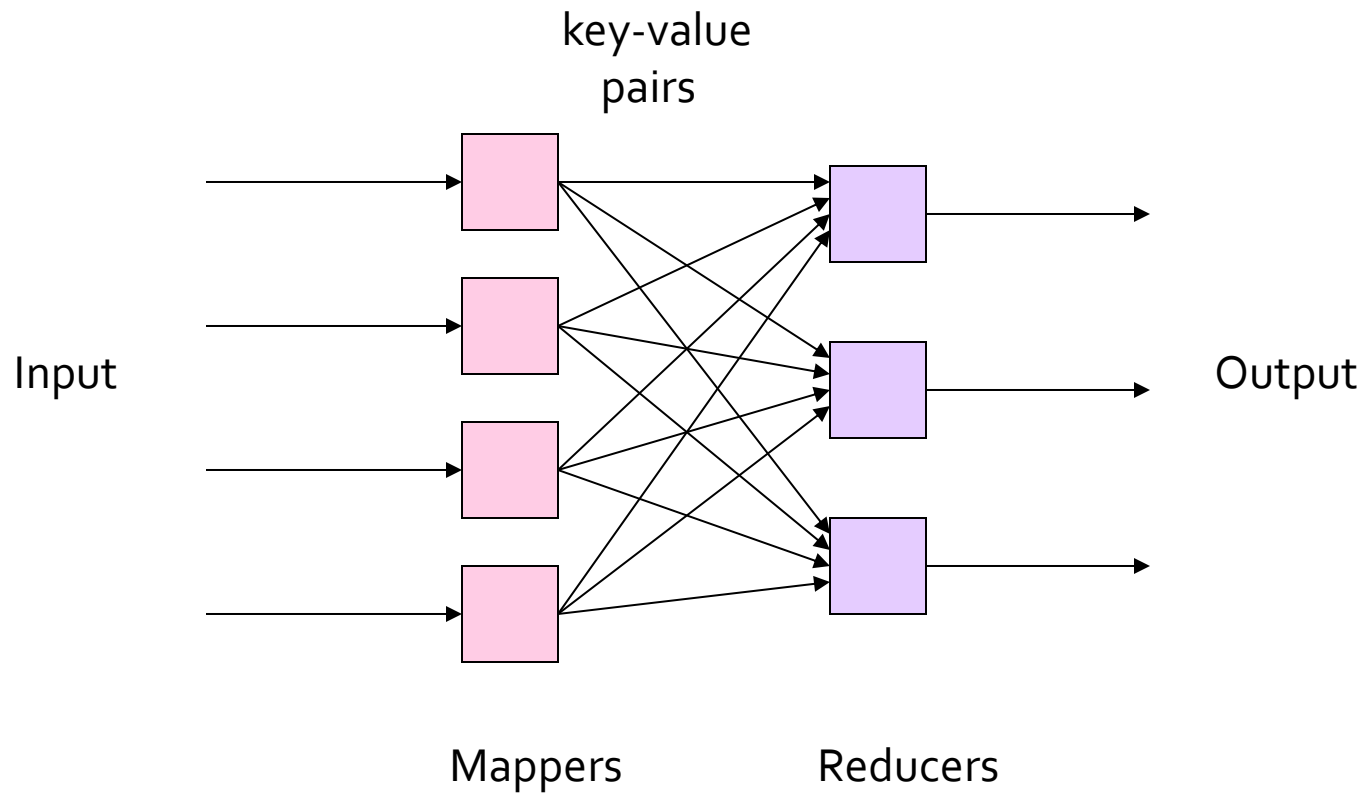
- A MapReduce job starts with a collection of input elements of a single type.
  - Technically, all types are key-value pairs.
- Apply a user-written *Map function* to each input element, in parallel.
  - *Mapper* applies the Map function to a single element.
    - Many mappers grouped in a *Map task* (the unit of parallelism).
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- The system sorts all the key-value pairs by key, forming key-(list of values) pairs.

# In a Nutshell – (2)

- Another user-written function, the *Reduce function*, is applied to each key-(list of values).
  - Application of the Reduce function to one key and its list of values is a *reducer*.
    - Often, many reducers are grouped into a *Reduce task*.
- Each reducer produces some output, and the output of the entire job is the union of what is produced by each reducer.



# MapReduce Pattern



# Example: Word Count

- We have a large file of documents (the input elements).
- Documents are sequences of words.
- Count the number of times each distinct word appears in the file.

# Word Count Using MapReduce

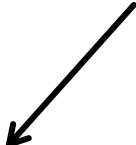
```
map(key, value):
```

```
// key: document ID; value: text of document
```

```
  FOR (each word w IN value)
```

```
    emit(w, 1);
```

Expect to be all 1's,  
but “combiners” allow  
local summing of  
integers with the same  
key before passing  
to reducers.



```
reduce(key, value-list):
```

```
// key: a word; value-list: a list of integers
```

```
  result = 0;
```

```
  FOR (each integer v on value-list)
```

```
    result += v;
```

```
  emit(key, result);
```

# Coping With Failures

- MapReduce is designed to deal with compute nodes failing to execute a Map task or Reduce task.
- Re-execute failed tasks, not whole jobs.
- **Key point:** MapReduce tasks have the *blocking property*: no output is used until task is complete.
- Thus, we can restart a Map task that failed without fear that a Reduce task has already used some output of the failed Map task.

# Some MapReduce Algorithms

**Relational Join**

**Matrix Multiplication in One or Two  
Rounds**

# Relational Join

- Stick the tuples of two relations together when they agree on common attributes (column names).
- **Example:**  $R(A,B) \text{ JOIN } S(B,C) = \{abc \mid ab \text{ is in } R \text{ and } bc \text{ is in } S\}.$

A	B
1	2
3	2
4	5

JOIN

B	C
2	6
5	7
5	8
9	10

=

A	B	C
1	2	6
3	2	6
4	5	7
4	5	8

# The Map Function

- Each tuple  $(a,b)$  in  $R$  is mapped to key =  $b$ , value =  $(R,a)$ .
  - **Note:** “ $R$ ” in the value is just a bit that means “this value represents a tuple in  $R$ , not  $S$ .”
- Each tuple  $(b,c)$  in  $S$  is mapped to key =  $b$ , value =  $(S,c)$ .
- After grouping by keys, each reducer gets a key-list that looks like
$$(b, [(R,a_1), (R,a_2), \dots, (S,c_1), (S,c_2), \dots]).$$

# The Reduce Function

- For each pair  $(R,a)$  and  $(S,c)$  on the list for key  $b$ , emit  $(a,b,c)$ .
  - Note this process can produce a quadratic number of outputs as a function of the list length.
  - If you took CS245, you may recognize this algorithm as essentially a “parallel hash join.”
  - It’s a really efficient way to join relations, as long as you don’t have too many tuples with a common shared value.



# Two-Pass Matrix Multiplication

- Multiply matrix  $M = [m_{ij}]$  by  $N = [n_{jk}]$ .
  - **Want:**  $P = [p_{ik}]$ , where  $p_{ik} = \sum_j m_{ij} * n_{jk}$ .
- First pass is similar to relational join; second pass is a group+aggregate operation.
- Typically, large relations are *sparse* (mostly 0's), so we can assume the nonzero element  $m_{ij}$  is really a tuple of a relation  $(i, j, m_{ij})$ .
  - Similarly for  $n_{jk}$ .
  - 0 elements are not represented at all.

# The Map and Reduce Functions

- **The Map function:**  $m_{ij} \rightarrow \text{key} = j, \text{value} = (M, i, m_{ij})$ ;  $n_{jk} \rightarrow \text{key} = j, \text{value} = (N, k, n_{jk})$ .
  - As for join, M and N here are bits indicating which relation the value comes from.
- **The Reduce function:** for key j, pair each  $(M, i, m_{ij})$  on its list with each  $(N, k, n_{jk})$  and produce  $\text{key} = (i, k), \text{value} = m_{ij} * n_{jk}$ .

# The Second Pass

- **The Map function:** The identity function.
- Result is that each key  $(i,k)$  is paired with the list of products  $m_{ij} * n_{jk}$  for all  $j$ .
- **The Reduce function:** sum all the elements on the list, and produce key =  $(i,k)$ , value = that sum.
  - I.e., each output element  $((i,k),s)$  says that the element  $p_{ik}$  of the product matrix  $P$  is  $s$ .

# Single-Pass Matrix Multiplication

- We can use a single pass if:
  1. Keys (reducers) correspond to output elements  $(i,k)$ .
  2. We send input elements to more than one reducer.
- **The Map function:**  $m_{ij} \rightarrow$  for all  $k$ : key =  $(i,k)$ , value =  $(M,j,m_{ij})$ ;  $n_{jk} \rightarrow$  for all  $i$ : key =  $(i,k)$ , value =  $(N,j,n_{jk})$ .
- **The Reduce function:** for each  $(M,j,m_{ij})$  on the list for key  $(i,k)$  find the  $(N,j,n_{jk})$  with the same  $j$ . Multiply  $m_{ij}$  by  $n_{jk}$  and then sum the products.

# Extensions to MapReduce

**Data-Flow Systems**

**Bulk-Synchronous Systems**

**Tyranny of Communication**

# 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.
- Generalize in two ways:
  1. Allow any number of 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 and Flink

- Two recent data-flow systems from Apache.
- Incorporate group+aggregate (**GA**) as an explicit operator.
- Allow any acyclic data flow involving these primitives.
- Are tuned to operate in main memory.
  - Big performance improvement over Hadoop in many cases.

# Example: 2-Pass MatMult

- The 2-pass MapReduce algorithm had a second Map function that didn't really do anything.
- We could think of it as a five-rank data-flow algorithm of the form Map-GA-Reduce-GA-Reduce, where the output forms are:
  1.  $(j, (M, i, m_{ij}))$  and  $(j, (N, k, n_{jk}))$ .
  2.  $j$  with list of  $(M, i, m_{ij})$ 's and  $(N, k, n_{jk})$ 's.
  3.  $((i, k), m_{ij} * n_{jk})$ .
  4.  $(i, k)$  with list of  $m_{ij} * n_{jk}$ 's.
  5.  $((i, k), p_{ik})$ .



# Bulk-Synchronous Systems

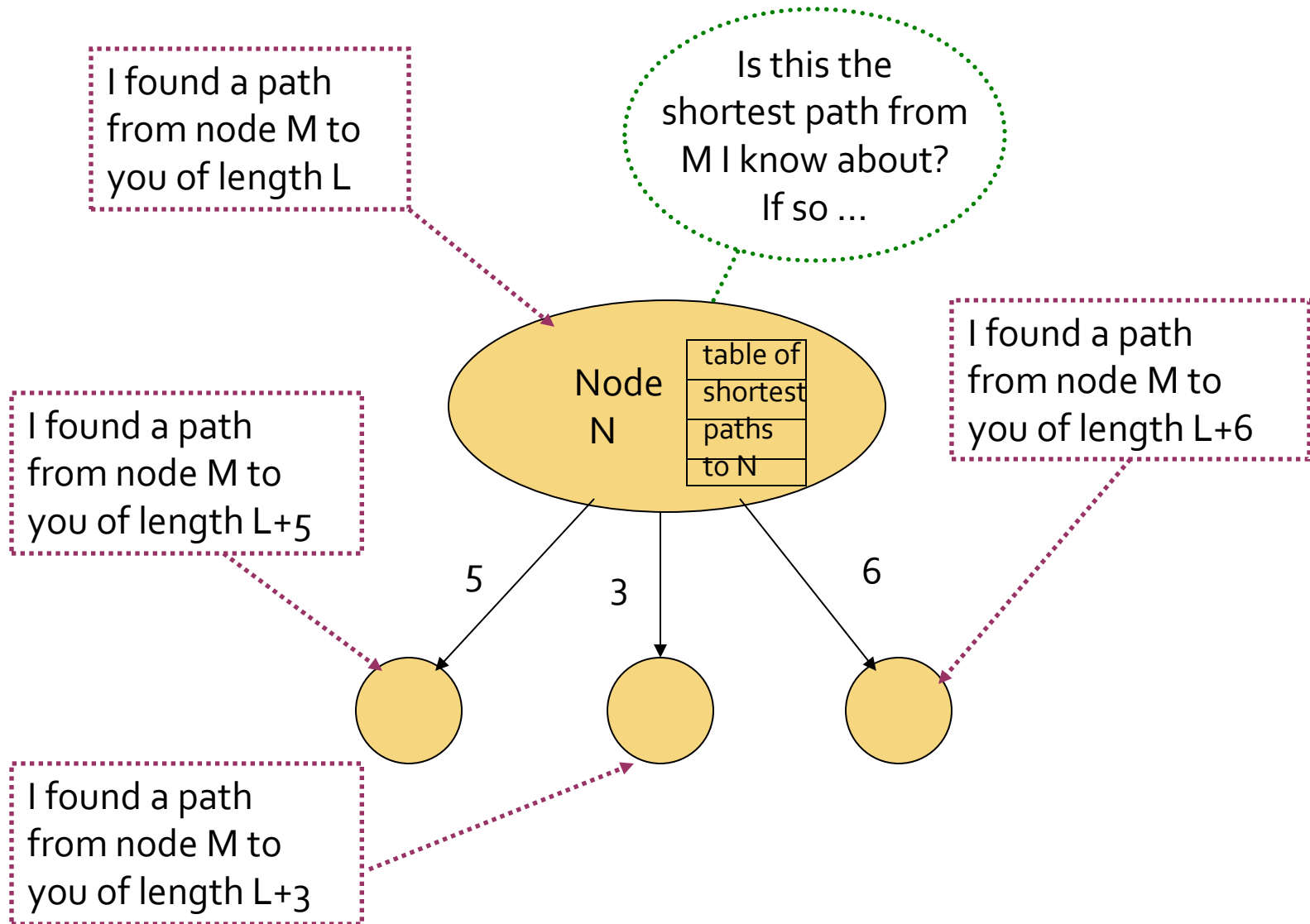
**Graph Model of Data**

**Some Systems Using This Model**

# The Graph Model

- Views all computation as a recursion on some graph.
- Graph nodes send messages to one another.
  - Messages bunched into *supersteps*, where each graph node processes all data received.
  - Sending individual messages would result in far too much overhead.
- Checkpoint all compute nodes after some fixed number of supersteps.
- On failure, rolls all tasks back to previous checkpoint.

# Example: Shortest Paths



# Some Systems

- *Pregel*: the original, from Google.
- *Giraph*: open-source (Apache) Pregel.
  - Built on Hadoop.
- *GraphX*: a similar front end for Spark.
- *GraphLab*: similar system that deals more effectively with nodes of high degree.
  - Will split the work for such a graph node among several compute nodes.